ЩАСЛИВСЬКИЙ НАВЧАЛЬНО-ВИХОВНИЙ КОМПЛЕКС

КИЇВСЬКИЙ ОБЛАСНИЙ ІНСТИТУТ
ПІСЛЯДИПЛОМНОЇ ОСВІТИ ПЕДАГОГІЧНИХ КАДРІВ
НАУКОВА ЛАБОРАТОРІЯ
ІНТЕРАКТИВНИХ ТЕХНОЛОГІЙ НАВЧАННЯ

В. С. ОСТАПЕЦЬ

ЕКСПРЕС–КУРС З ОСНОВ АЛГОРИТМІЗАЦІЇ ТА ПРОГРАМУВАННЯ

Щасливе
2009

ЩАСЛИВСЬКИЙ НАВЧАЛЬНО-ВИХОВНИЙ КОМПЛЕКС

КИЇВСЬКИЙ ОБЛАСНИЙ ІНСТИТУТ
ПІСЛЯДИПЛОМНОЇ ОСВІТИ ПЕДАГОГІЧНИХ КАДРІВ
НАУКОВА ЛАБОРАТОРІЯ
ІНТЕРАКТИВНИХ ТЕХНОЛОГІЙ НАВЧАННЯ

В.С. ОСТАПЕЦЬ

Е К С П Р Е С – К У Р С
З ОСНОВ
АЛГОРИТМІЗАЦІЇ ТА ПРОГРАМУВАННЯ

Щасливе

2009

Рекомендовано до друку
Вченою радою
Київського обласного інституту післядипломної освіти педагогічних кадрів
(Протокол № 1 від 18 лютого 2009 року)

Рецензенти: **Федорчук В.А.**,
Завідуючий
центром інформаційних технологій навчання
Київського обласного інституту
післядипломної освіти педагогічних кадрів

Скляр І.В.,
викладач УФМЛ КПІ ім. Т. Шевченка,
редактор газети "Інформатика",
Заслужений учитель України

Остапець В.С.
ЕКСПРЕС–КУРС З ОСНОВ АЛГОРИТМІЗАЦІЇ ТА ПРОГРАМУВАННЯ
Навчально-методичний збірник – Щасливе: Щасливський НВК, 2009 – 108 с.

Анотація: У збірнику пропонується шлях вирішення проблеми забезпечення необхідного рівня теоретичних знань і практичних навичок при викладанні теми курсу основ інформатики "Комп'ютерне моделювання. Основи алгоритмізації та програмування" згідно діючої програми (універсальний профіль) в умовах обмеженої кількості відведених навчальних годин та при недостатній якості діючих підручників для учнів. Перша частина забезпечує методичне обґрунтування експрес-курсу згідно введених оригінальних робочих ідей. Друга частина містить 12 тем-уроків експрес-курсу з прикладами, запитаннями та завданнями. Третя частина – це практикум, що складається із шести практичних робіт на застосування програмування. Тут детально розглянуто відповідні шкільному курсу математики та фізики прикладні задачі на програмування. Остання частина містить збірник задач підвищеної складності з програмування із вказівками по математичній постановці та опису алгоритмів. Усі задачі пропонувались протягом останніх 5-и років на ІІІ та IV етапах Всеукраїнської олімпіади з інформатики у Київській області. Збірник буде корисним учителям основ інформатики та учням старших класів загальноосвітніх шкіл, які цікавляться програмування, може бути використаний у якості навчального посібника при вивченні розділу основ алгоритмізації та програмування і в позакласній роботі.

Остапець В.С.,
Щасливський НВК, 2009

ЗМІСТ:

I Вивчення теми “Комп’ютерне моделювання. Основи алгоритмізації та програмування“ у формі експрес-курсу 4

I.1 Проблеми вивчення теми “основи алгоритмізації та програмування “ у школі 4

I.2 Методичне обгрунтування експрес-курсу з алгоритмізації та програмування 5

I.3 Висновки 11

II Екап з програмування мовою Паскаль 16

II.1 Поняття про алгоритми і найпростіші програми 16

Урок I. Поняття інформаційної та математичної моделі. Етапи розв’язування задач з допомогою комп’ютера 16

Урок II. Алгоритми, їх властивості та виконавці. Система команд виконавця. Форми подання алгоритмів. Поняття програми. 19

Урок III. Структура програми. Опис величин мовою програмування. Лінійні програми опрацювання величин. Команди вводу, виводу та присвоєння. 22

Урок IV. Виконання програми та аналіз її правильності. Покрокове виконання і відлагодження програм. 26

II.2 Структуровані алгоритми та їх програмна реалізація 30

Урок V. Діалогові програми. Структури повторення та їх графічні схеми. 30

Урок VI. Структури розгалуження та їх графічні схеми. 33

Урок VII. Конструювання алгоритмів методами покрокової деталізації. Підпрограми і вказівки їх виклику. Формальні та фактичні параметри. 36

Урок VIII. Структуровані алгоритми. 41

II.3 Структуровані величини та їх опрацювання 45

Урок IX. Структуровані типи величин. Табличні величини (масиви). Лінійні масиви та алгоритми їх опрацювання. 45

Урок X. Алгоритми пошуку елементів у таблиці. 49

Урок XI. Алгоритми упорядкування лінійних таблиць. 52

Урок XII. Рядкові величини та алгоритми їх опрацювання. 56

III Практикум з програмування 60

Практична робота 1. 60
Обчислення значення многочлена за схемою Горнера.

Практична робота 2. 61
Табулювання функції.

Практична робота 3. 63
Уточнення кореня рівняння.

Практична робота 4. 66
Розв'язування системи двох лінійних рівнянь з двома невідомими.

Практична робота 5. 68
Наближене обчислення площі криволінійної трапеції.

Практична робота 6. 69
Розподіл температур в однорідному стержні.

IV Розв’язування задач з програмування підвищеної складності. 71

IV.1 Письмовий опис розв’язку задачі. 71

IV.2 Олімпіадні задачі. 73

IV.3 Від задачі до задачі. 97

Література 108

I. ВИВЧЕННЯ ТЕМИ "КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ. ОСНОВИ АЛГОРИТМІЗАЦІЇ ТА ПРОГРАМУВАННЯ" У ФОРМІ ЕКСПРЕС-КУРСУ

Єдине і справжнє чудо – це ***мислення****, адже саме завдяки йому люди змогли створити усі інші, так звані, чудеса світу.*

I.1 Проблеми вивчення теми "Основи алгоритмізації та програмування" у школі

До сучасних шкільних програм з основ інформатики [1][1] входить тема "Комп'ютерне моделювання. Основи алгоритмізації та програмування". Цілком слушно вважати названу тему *розділом*, адже протягом багатьох років весь курс інформатики майже виключно зводився до вивчення алгоритмів і програм. Крім того, *темою* прийнято називати частину деякого розділу певного предмету чи курсу, тоді, як "Алгоритмізація і програмування" логічно не належить до жодної з частин курсу інформатики, які прийнято скорочено називати АС, СПЗ чи ППЗ[2] тощо, а враховуючи логічну відособленість від решти навчального матеріалу шкільної інформатики, цю тему доцільно вважати навіть *окремим курсом*. Щоправда, курс, як правило, має значно більшу кількість навчальних годин.

Вже в назві цього розділу чи курсу, що поєднує терміни *комп'ютерне моделювання*, *основи*, *алгоритмізація* та *програмування* натякається на стислість і оглядовість. Це цілком підтверджується відведеною у програмі кількістю навчальних годин (12). Якщо врахувати, що у переважній більшості загальноосвітніх навчальних закладів інформатика вивчається у обсязі універсального профілю, при його викладанні практично перед усіма вчителями виникає багато особливо гострих питань і проблем, які дуже важко вирішити. Розглянемо деякі з них, що, на наш погляд, цілком доречно поставити:

- Який сенс вивчати у школі такий досить абстрактний і формалізований розділ *оглядово*?
- Як "*втиснути*" у 12 навчальних годин названий розділ?
- *Чи можливо* при цьому *досягти* поставленої мети та засвоєння учнями вказаного рівня теоретичних знань і практичних навичок?

Виходячи з того, що учителі не в змозі змінювати програму, ці питання можна вважати риторичними. Але тоді проблема, пов'язана з ними залишиться. Її можна сформулювати так: ***Проблема забезпечення необхідного рівня теоретичних знань і практичних навичок при викладанні розділу курсу основ інформатики "Комп'ютерне моделювання. Основи алгоритмізації та програмування" згідно діючої програми в умовах недостатньої кількості відведених навчальних годин***.

I.2 Методичне обґрунтування експрес-курсу алгоритмізації та програмування

[1] Тут і далі посилання на використану літературу будемо вказувати за її номером у списку, взятим у квадратні дужки.

[2] Маються на увазі частини обчислювальної системи: АС – апаратна складова, СПЗ – системне програмне забезпечення, ППЗ – прикладне програмне забезпечення.

Спробуємо вказати на досить, на нашу думку, ефективний шлях вирішення поставленої проблеми. Безперечно, її розв’язати можна лише шляхом *максимального ущільнення* навчального матеріалу, *універсалізації* задач, запропонованих для його закріплення, забезпечення *логічної єдності* всього навчального матеріалу та при умові *обов’язкового виконання* учнями *теоретичних і практичних завдань у позаурочний час*. Крім того, практично неможливо обійтись без збільшення кількості відведених для вивчення розділу навчальних годин. Їх можна взяти за рахунок ущільнення інших тем курсу, яке, виходячи з місцевих умов, допускається авторами програми[3].

Зрозуміло, що при застосуванні до розділу таких умов його справді можна називати окремим *експрес-курсом з основ алгоритмізації та програмування* (ЕКАП) для загальноосвітніх шкіл згідно діючих навчальних планів та програм для універсального профілю[4]. Термін “експрес“ тут вжито як в традиційному розумінні, тобто “прискорений“, адже програмування за 12-15 навчальних годин – це справді надзвичайно швидко і стисло, так і в розумінні “особливий“, “надзвичайний“ та “ефективний“.

Вважаємо за необхідне наголосити, що нижче буде описано не стислий курс алгоритмізації та програмування для учнів загальноосвітніх шкіл, а методичні поради щодо вивчення окремої теми шкільної інформатики у формі ЕКАПу. Кожен учитель при потребі та бажанні матиме можливість, врахувавши наші методичні поради, самостійно створити курс лекцій і практичних занять, більш чи менш стислий і компактний.

Створити такий ЕКАП не можливо без застосування певних *робочих ідей*, або *принципів*, які покликані допомогти у досягненні ущільнення, універсалізації, логічної єдності і т.д. Тому дуже доречно згадати думку Р.Декарта: “Кожна розв’язана мною задача ставала образом, який застосовувався надалі при розв’язуванні інших задач“. Вона висловлює загальновідому, навіть банальну в наш час дидактичну істину, але все геніальне справді просте. Дійсно, курс обов’язково передбачає розгляду певної кількості задач, розв’язання кожної з яких перш за все потребує математичної постановки. Учням, що в процесі вивчення розділу алгоритмізації та програмування ще тільки знайомляться з новими різновидами мислення, зокрема *алгоритмічним* та *структурним*, при цьому буде необхідно не тільки багато додаткового часу, а й значних інтелектуальних зусиль, спрямованих на подолання психологічних ефектів постійної новизни і невідомості та розпорошення уваги, яка приводить до втрати почуття цільності курсу. Тому ми намагатимемось розглядати таку *серію задач, де кожна наступна - в певному розумінні буде розширенням чи аналогією попередньої*, іншими словами, з метою економії часу та інтелектуальної напруги спробуємо реалізувати ідею *викладу курсу програмування на одній великій задачі*, або за ***принципом нитки*** (умовно позначимо його літерою **А**).

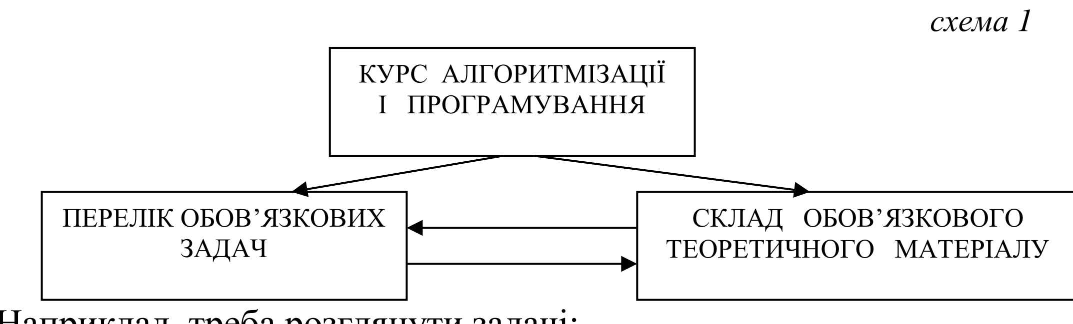
---

[3] Рекомендуємо доповнити вивчення теми хоча б трьома додатковими навчальними годинами, які слід поставити в кінці кожного з трьох блоків уроків експрес-курсу.

[4] Описані тут ідеї та методи можна адаптувати і до інших варіантів діючих програм з інформатики для загальноосвітніх навчальних закладів.

Але будь-який курс повинен охоплювати певне завершене коло основних задач, матимемо це на увазі й при створенні нашого ЕКАПу. Перелік задач повинен відповідати діючій програмі. Крім того, виконуючи її, ми повинні забезпечити необхідний рівень засвоєння мінімального обсягу теоретичного матеріалу, тобто необхідно скласти взаємопов'язані *повний перелік обов'язкових задач* і *мінімальний обсяг теоретичного матеріалу* (схема1).

схема 1



Наприклад, треба розглянути задачі:

список 1

номер та назва обов'язкові задачі		*номер відповідного задачі теоретичного матеріалу*
1	Написати лінійну програму обчислювального характеру	1,2,3,4,6
2	Написати діалогову програму	~~1,2~~,3,4,5,6
3	Написати програму із розгалуженням	~~1,2,3,4~~,5,6,8
4	Написати циклічну програму	~~1,2,3,4,5~~,6,7
5	Написати програму визначення суми заданих формулою чисел	~~1,2,3,4,5~~,6,7
6	Написати програму визначення суми даних чисел, заданих масивом	~~1,2,3,4,5~~,6,7,9,10
7	Написати програму пошуку в лінійних неупорядкованих числових масивах	~~1,2,3,4,5,6,7~~,8,9,10)
8	Написати програму пошуку в лінійних упорядкованих числових масивах	~~1,2,3,4,5,6,7~~,8,~~9~~,10)
9	Написати програму сортування масивів	~~1,2,3,4,5,6,7,8,9,10~~,11

У якості відповідного цим задачам обов'язкового теоретичного матеріалу можна взяти теми:

список 2

1. Поняття величини в інформатиці, оголошення і опис величин;
2. Математичні операції та функції над числовими типами та запис математичних виразів у лінійній формі;
3. Команда присвоєння;
4. Процедури вводу/виводу;
5. Символьний та рядковий типи;
6. Коментарі в програмі;
7. Поняття циклу;
8. Поняття команди розгалуження;
9. Поняття масиву;
10. Ввод масиву;
11. Вивод массиву.

Як бачимо, пункти першого списку повинні бути узгоджені при вивченні із деякими пунктами другого списку, причому, з кожним наступним ідуть повтори та наростання обсягу обов'язкового теоретичного матеріалу (у першому

списку відповідні пункти другого списку наведені справа в дужках). Таким чином, що цілком природньо, застосування однієї і тієї ж порції теоретичного матеріалу при розгляді різних задач часто повторюється у різних комбінаціях з іншими порціями, а не рідко дублюється (звернемо увагу хоча б на порції теоретичного матеріалу 1-4, які дублюються в усіх дев'яти наведених задачах). Скориставшись щойно поміченим, можна досягти значного ущільнення як навчального матеріалу так і часу, відведеного для вивчення. Як видно із списку 1, порції теоретичного матеріалу з меншими номерами на кожному наступному кроці викреслюються, наприклад, починаючи із задачі 3, викреслені п.1-4, починаючи із задачі 7 викреслені п.1-7 і т.д. Це слід розуміти так: викреслені у списку 1 порції теоретичного матеріалу з моменту викреслення більше не вивчаються і не повторюються, наприклад, перед розглядом задачі 8, можна обійтись вивченням тільки двох порцій теоретичного матеріалу: поняття команди розгалуження (№8) та ввод масиву (№10), а з рештою тем (№№1-7 та №9) ознайомлення відбулось при розв'язуванні попередніх семи задач.

Але чи можна обійтись у цій ситуації навіть без повторення тем 1-7 та 9? Відповімо: *не тільки можна*, а при вивченні розділу у вигляді ЕКАПу і *потрібно*. Тут ми свідомо йдемо проти вікової дидактичної, але, на жаль, в дійсності *не зовсім педагогічної традиції* "повторення – мати навчання". Чому практично на всіх уроках організовується повторення попереднього навчального матеріалу? Щоб створити максимум комфорту для учнів. Але погодимось, що безкінечне повторення прописних істин – це зовсім не цікаво для тих учнів, які давно засвоїли їх на пам'ять. Скільки таких учнів у класі з n осіб? Якщо цю величину позначити k, то ось строга відповідь: залежно від ситуації $0 \le k \le n$. Напрошується цікавий висновок: *у навчальному процесі не доцільно зловживати повторенням раніше вивченого*, інакше в учнів можуть виробитись неуважність, байдужість, бездіяльність, безініціативність і т.д., замість повсякденного планування повторення *слід постійно слідкувати, щоб k було якомога ближче до n, а на деякому етапі навчання досягало його.*

Справді, чи завжди у повсякденному житті починають роботу з повторення, тренування тощо? Це доцільно, наприклад, спортсменам, співакам тощо, для яких так звана розминка – необхідність. Для більшості ж видів людської діяльності *основні прийоми та навички повинні бути доведені до автоматизму*. Щоб переконатись у цьому, візьмемо будь-яку масову професію, хоча б столяра. Хіба він, починаючи робочий день, повторює навички пиляння, стругання, забивання цвяхів і т.д.? Отже, в ідеалі процес навчання повинен забезпечувати на будь-якому етапі абсолютне засвоєння необхідних знань, умінь та навичок, тому, на нашу думку, вчителям слід організовувати навчальний процес якомога ближче до ідеального.

Поміркуємо, як цього досягти. Відповідь можна знайти в учителів-новаторів, наприклад у І.П. Волкова[5]. Під впливом одного із його виступів ще в

[5] І.П. Волков, учитель малювання та праці з м. Реутово Московської області, кандидат педагогічних наук, заслужений учитель школи РСФСР. Його основна новаторска ідея - створення механізму реалізації творчих здібностей дітей і здійснення педагогічного керівництва процесом розвитку індивідуальності кожного школяра.

першій половині восьмидесятих років XX століття закарбувалась чітка схема розвитку творчих здібностей та виявлення професійних нахилів учнів:

1) вивчаючи будь-який курс, предмет чи окрему тему слід виявити *основні поняття* та *їх властивості* (у І.П. Волкова, наприклад, інструменти у майстерні по дереву та основні навички роботи з ними);
2) досконало вивчити виявлені основні (робочі) поняття та до автоматизму відпрацювати стандартні прийоми їх використання (у майстерні – це правила техніки безпеки при використанні ножа, пилки, стамески тощо та стандартні прийоми їх застосування при роботі з деревом);
3) відпрацювати комплексне застосування робочих понять (наприклад, одночасне використання стамески і молотка);
4) створити умови для *самостійної творчої роботи* у рамках теми з вибраними поняттями та їх властивостями (створення виробу із дерева).

По суті, ми підійшли до другого робочого принципу (**Б**), необхідного для створення ЕКАПу. Коротко його можна сформулювати так: ***забезпечувати навчання не на рівні запам'ятання а на рівні розуміння, щоб після закріплення зникала потреба в повторенні раніше вивченого***.

На користь цього принципу наведемо досить тривіальний приклад. Вчителі інформатики знають, як важко призвичаїти учнів до строгості мови програмування, застерегти від багаторазового повторення банальних помилок порушення її синтаксису, дотримання структури програми тощо. Візьмемо фрагмент довільної програми, що містить блок вводу величини *a* з виводом підказки:

```
{...}  var a:real; {...}
begin
 {...} Write('a?');Read(a); {...}
end.
```

Можна спостерігати, як у подібних ситуаціях окремі учні на протязі багатьох уроків набирають "різні варіанти":

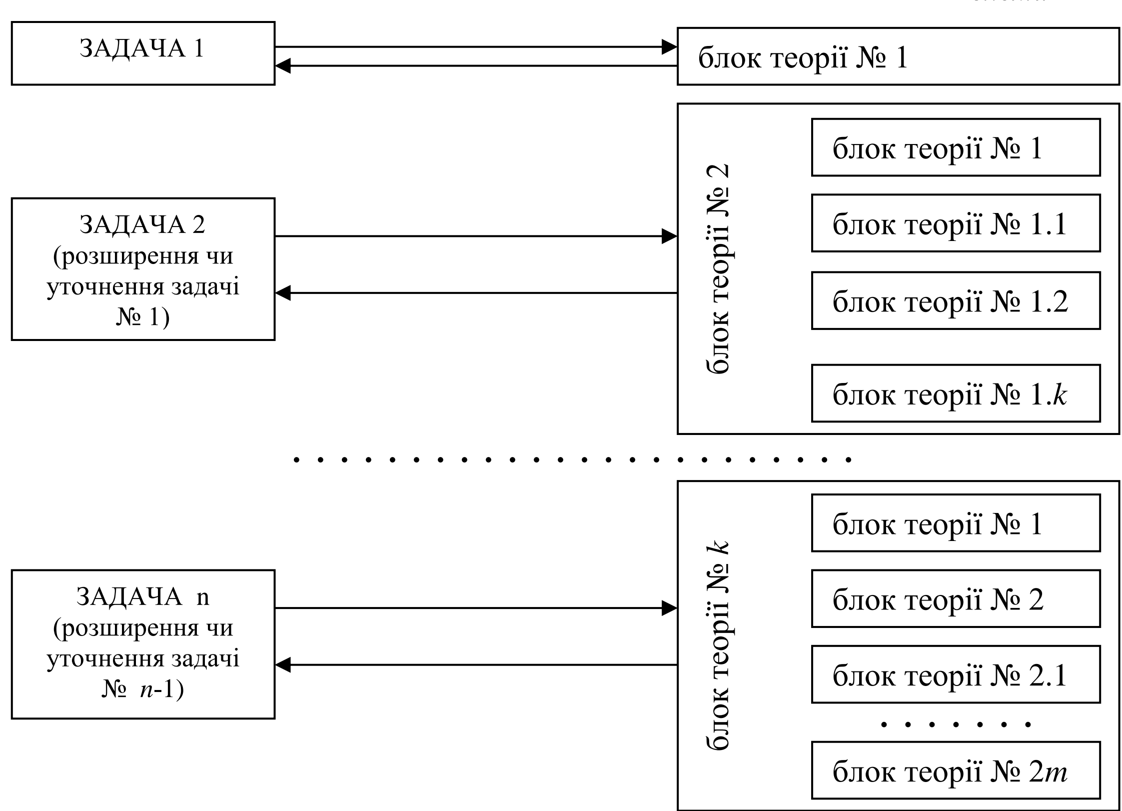
1) *Write(a?);Read(a);*
2) *Write('a?');Read('a');*
3) *Write('a?') Read(a)* тощо.

Як подолати раз і назавжди допущення таких помилок, щоб не тратити час на кожному уроці на повторення формату та різних варіантів команд вводу і виводу? Дуже просто. Дати учням можливість "набити" усі "шишки" відразу, виконавши наведені варіанти в середовищі системи програмування. Учень повинен *усвідомити*, а *не запам'ятати* правильний запис. Для цього у кожного є свій ліміт повторень ситуацій. Очевидно, що краще цей ліміт вичерпати відразу. Читачеві нічого не залишається, як погодитись, що після використання описаного дидактичного прийому важко буде знайти учня, якому для уникнення помилок необхідне повторення.

Усе це дасть значну економію часу і дозволить навчати без надмірних інтелектуальних напружень. А наведена щойно схема розвитку творчих здібностей учнів зовсім не є інноваційною, вона лише вимагає будувати ЕКАП по строгості й логічності подібно аксіоматичному методу побудови курсу геометрії.

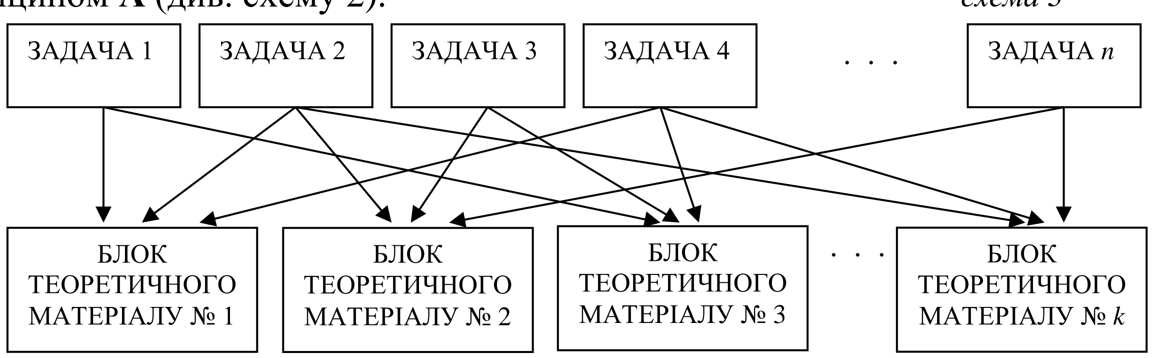
Безперечно, на практиці впровадження принципів **А** та **Б** не таке прозоре і зрозуміле, тому додатково скористаємось схемами 2 та 3.

схема 2



Якщо при вивченні алгоритмізації і програмування дотримуватись наведеної вище схеми 2, то можна зняти психологічні складності, отже підвищити якість засвоєння навчального матеріалу. Звичайно, при підборі задач необхідно уявляти весь курс і починати із останньої задачі. Нехай це *задача n* про пошук у відсортованому лінійному числовому масиві. Логічно найближча до неї задача про сортування лінійного числового масиву, тому назвемо її *задачею n-1* і будемо вважати попередньою. Для задачі *n-1* попередньою може бути задача *n-2* про відшукання найменшого елементу в лінійному числовому масиві і т.д. Міркуючи таким чином, поступово можна дійти до задачі *1*. Але задачу *n* можна сформулювати настільки детально, що вона міститиме всі попередні - від задачі *n-1* до першої включно. На кожному етапі деталізації чи ускладнення поточної задачі можна підключати необхідний блок теоретичного матеріалу. В кінці-кінців весь курс буде присвячений розгляду однієї глобальної задачі з поступовим нарощуванням блоку теоретичного матеріалу, як і передбачено принципом **А** (див. схему 2).

схема 3



Застосовувати **А** можна лише умовно, адже постійно доповнюючи умову задачі, ми одержуватимемо інші задачі (наприклад, із списку 1). Крім того, можна укрупнити ЕКАП, використавши не одну, а кілька базових задач, які не зводяться одна до одної. Уявимо, що в курсі їх буде *n*. Звернемо увагу на те, що і блоки теоретичного матеріалу тоді в задачах будуть застосовуватись не так, як показано у списку 1, а більш хаотично (див. схему 2).

Як уже зазначалось, для забезпечення вивчення алгоритмізації та програмування в прискореному і ефективному режимі, тобто режимі експрес-курсу, необхідні додаткові резерви часу і обов'язкова організація самостійного виконання учнями необхідних теоретичних і практичних завдань. Ми домовились застосовувати робочі принципи, два з яких (**А** та **Б**) уже сформульовані. Але по відношенню до учнів вони мають інтерактивний, тобто спонукальний із зовні, характер. У той же час будь-який курс повинен *не стільки викладатись*, як *вивчатись*. Отже необхідна ще хоча б одна робоча ідея, покликана бути *внутрішнім імперативом* або *стимулятором* для учнів. Назвемо її принципом **В** – ***орієнтувати учнів на самонавчання***.

Один мудрець вказав три шляхи удосконалення: *легкий – наслідування*, *важкий – досвід* і *шляхетний – міркування*. Проаналізуємо процес навчання. Кажуть, що воно буває пасивне, активне та інтерактивне. З цим погодитись ніяк не можна. Хіба не безглуздо стверджувати, що існує пасивне навчання? *Навчання – це процес, в якому одночасно і активно приймають участь або дві особи – учитель і учень, або одна особа - учень*. Якщо учень пасивний (читай - відсутній), то це зовсім не навчання, причому, можна сказати "по визначенню". Отже, ***будь-яке навчання активне***. Активність, тобто мотивація і стимуляція можуть бути внутрішніми чи зовнішніми. Інтерактивний - стимульований із зовні. Тому інтерактивні елементи в навчально-виховному процесі корисно застосовувати не постійно, а епізодично, наприклад, для актуалізації, концентрації, закріплення чи повторення, які в задуманому нами експрес-курсі повинні відігравати допоміжну роль. Слід завжди враховувати "протипоказання" інтерактивних методів, адже, як окремий вид маніпуляцій свідомістю, вони при надмірному "вживанні" можуть привести до формування в учнів ефекту (а то й дефекту) *залежності*, тобто *нездатності самостійно мислити, виконувати роботу, приймати рішення* тощо.

Дуже корисно на першому, ознайомлювальному уроці запитати учнів: "Чому ви сьогодні прийшли до кабінету інформатики?". Як правило, останні від несподіванки не знають, що відповісти. Ось зразки відповідей: "вивчати комп'ютери", "згідно розкладу", "грати в комп'ютерні ігри". Але, бачачи, що вчитель чекає іншої відповіді, учні можуть "гадати" далі: "слухати", "думати", "щоб нас навчили" тощо. Нарешті почується відповідь: "учитись". Це буде тим, що потрібно. Далі слід запитати: "Що означає *учитись*?" Скоріше всього, учні не зможуть правильно відповісти, бо вони ще не розуміють, що учитись – це самостійно, без учителя. Тоді їм слід пояснити те, що сказане в двох попередніх абзацах, переконати, що без орієнтації на самонавчання, вони ніколи не знатимуть більше за вчителя, будуть обмежені його знаннями та здібностями. І

тільки після цього учні можуть, якщо не сформулювати, то хоча б зрозуміти формулу: ***Учитись – це самостійно шукати запитання та відповіді на них***.

Щойно ми продемонстрували ефект несподіванки, висловлений ще одним древнім філософом. Взагалі, в навчально-пізнавальному процесі повинно бути якомога більше несподіваних ситуацій, це приводять до пізнання через здивування. Назвемо його принципом **Г** - ***пізнавати через здивування***, який разом із принципом **В** повинен стати *мотиваційною основою* ЕКАПу. Для забезпечення повноти системи взятих принципів, до них слід долучити ще один – ***принцип творчого співробітництва*** учнів з учителем (**Д**).

У наступному розділі пропонується орієнтовний ЕКАП з програмування мовою Паскаль, адже вище ми домовились розглядати методику вивчення теми "Алгоритмізація і програмування" у такій формі, а створення експрес-курсу – це, відповідно до умов і бажань, справа учителя. Кожна с тем, що називаються ***Уроками***, розрахована для вивчення протягом одного уроку, але при обов'язковій умові самостійних відповідей на запитання та виконання завдань, які пропонуються в кінці кожної теми. Курс не слід розглядати, як збірник поурочних планів учителя. Взявши пропонований ЕКАП за зразок, учителі повинні розуміти, що введення в дію принципів **А**-**Д** для учнів повинно бути поступовим, обережним, послідовним і не помітним. Наприклад, повністю відмовитись від систематичних повторень вивченого раніше (принцип **Б**) можна лише при забезпеченні свідомого дотримання принципу **А** у поєднанні з мотиваційтими принципами (**В-Д**).

I.3 Висновки[6]

Як зазначалось в другому пункті, пропонований ЕКАП повинен відповідати чинній програмі з основ інформатики для універсального профілю [1], тому будуватимемо його синхронно затвердженому МОН України підручнику [2]. Але, дотримуючись вимоги програми щодо кількості відведених навчальних годин і форми експрес-курсу, ми змушені шукати шляхи ущільнення.

У темі "Комп'ютерне моделювання. Основи алгоритмізації та програмування" фактично не відображається *власне програмування*, якщо не брати до уваги кількох окремих згадок терміну "програма", зокрема: "поняття програми", "структура опису алгоритму навчальною алгоритмічною мовою та мовою програмування" і "описання найпростіших алгоритмів навчальною алгоритмічною мовою та мовою програмування".

Підручник [2] містить три розділи: 3. "Етапи розв'язування задач за допомогою комп'ютера", 4. "Вступ до алгоритмізації" та 5. "Початки програмування". Як і частина програми про інформаційну модель, розділ 5 підручника дуже розпливчатий і не конкретний. Там багаторазово у різних контекстах згадується термін "модель", але, на наш погляд, так і нема однозначних відповідей про відмінність понять інформаційної і математичної

---

[6] Далі будемо використовувати особливі значки для виділення різних за призначенням частин тексту, наприклад, методичний аналіз і поради учителям позначатимемо - ✍, текст експрес-курсу - 🕮, означення і описи - ⓘ, приклади - ▤, завдання - ✎, запитання - ❓, корисні поради - 👂, обережно, будь уважним - ❗, детальніше - ⚲, додаткова інформація - § тощо.

моделей, комп'ютерного моделювання і побудови моделі, у підручнику немає чіткого і однозначного трактування етапів розв'язування задачі з допомогою комп'ютера. Ми намагатимемось лаконічно і прозоро сформулювати логічний ланцюжок *інформаційна модель – математичне моделювання – етапи комп'ютерного моделювання*

На наш погляд, фактично відсутній чіткий перехід *алгоритм – програма*. Окремий розгляд, як це зроблено в підручнику, вступу до алгоритмізації та початків програмування в умовах обов'язковості практичної форми вивчення теми також не доцільний. Не відходячи від істини, можна вважати, що *програма – це запис алгоритму розв'язання задачі мовою програмування*. Отже є сенс у ЕКАП об'єднати розділи " Вступ до алгоритмізації " та " Початки програмування".

Але для окремого розгляду алгоритмізації та програмування, що традиційно і цілком природньо робиться, є важливі причини. При розгляді питань алгоритмізації *основна увага концентрується на реалізації математичної моделі задачі в алгоритмі*, тобто домінує логічна лінія. Завдяки цьому при вивченні програмування *стає можливим зосередитись виключно на вивченні мови програмування* і нюансах запису алгоритму у вигляді програми. Підручники з мов програмування, як правило дуже об'ємні й детальні, очевидно, що шкільною програмою не передбачено їх вивчення. Отже виникає ще одна проблема: *як в ознайомлювальному курсі засвоїти основи мови програмування*? Ми також не бачимо можливості в нашому експрес-курсі для більш чи менш детального викладання основ мови *Pascal*. Тому нижче будемо розглядати лише вузлові питання запису програм, причому, як уже акцентувалось, по можливості об'єднуючи опис алгоритму і програми. Щоб заповнити прогалини в описі конструкцій *Pascal*, які не важко помітити при дуже стислому викладі матеріалу, ми будемо формулювати питання і ставити завдання, наведені в кінці кожного уроку, так, щоб учень, працюючи з ними, міг самостійно "відкривати" елементарні відомості, опис яких відсутній. Виконуючи завдання, учень буде змушений користуватись електронною довідкою, і таким чином отримувати достатньо повне уявлення про мову програмування. В зв'язку з цим ми будемо посилатись на версію *PascalABC*[7].

Піднявши питання перегляду ролі і місця теми "Комп'ютерне моделювання. Основи алгоритмізації та програмування" у курсі сучасної шкільної інформатики, ми переслідували ще одну важливу мету. Як уже зазначалось, згідно діючих програм ця тема вивчається фактично оглядово, що затрудняє для учнів її логічне усвідомлення в цілому. Але при цьому, в рекомендованих МОН України підручниках та навчальних посібниках продовжує відображатись класична, обтяжена деталями, схема подання і закріплення навчального матеріалу з алгоритмізації та програмування, яка багато в чому є калькою технологічного, призначеного для майбутніх студентів

[7] На рівні програми ЗОШ вона цілком відповідає найчастіше вживаним версіям *Pascal*. Крім того ця версія надає багато переваг, зокрема, допускає "довгі імена", використовує зручнішу систему числових типів, має стандартні підпрограми, наприклад функцію *Power*(*x*,*y*), має русифіковані меню, довідку та ін. Щодо потужності, то практика підтверджує, що ця версія дозволяє реалізувати навіть непересічні "олімпіадні" задачі.

технічних вузів, підходу до програмування. Але з позицій шкільного вчителя основ інформатики видно проблеми:

а) чіткості методологічних та філософських засад вивчення курсу;

б) підпорядкованості вивчення алгоритмізації та програмування простій і логічній загальній схемі;

в) слабкості органічного зв'язку із основною частиною курсу основ інформатики.

Вирішення проблеми *а)* бачиться в усвідомленні та реалізації тези "інформатика іде від математики" та вивченні теми раніше на один рік, тобто в першому семестрі 10-го класу, відразу після ознайомлення з основним системним програмним забезпеченням (операційні системи та сервісне програмне забезпечення) і вивчення простого текстового редактора.

Не виходячи із ліміту часу, передбаченого діючими програмами, подолати проблему *б)* можна, змінивши хронологію та акценти всередині цієї теми за такою схемою:

1. ***Поняття про алгоритми і найпростіші програми.***

1.1 Поняття інформаційної та математичної моделі. Етапи розв'язування задач з допомогою комп'ютера.

1.2 Алгоритми, їх властивості та виконавці. Система команд виконавця. Форми подання алгоритмів. Поняття програми.

1.3 Структура програми. Опис величин мовою програмування. Лінійні програми опрацювання величин. Команди вводу, виводу та присвоєння.

1.4 Виконання програми та аналіз її правильності. Покрокове виконання і відлагодження програм.

2. ***Структуровані алгоритми та їх програмна реалізація.***

2.1 Діалогові програми. Структури повторення та їх графічні схеми.

2.2 Структури розгалуження та їх графічні схеми.

2.3 Конструювання алгоритмів методами покрокової деталізації. Підпрограми і вказівки їх виклику. Формальні та фактичні параметри.

2.4 Структуровані алгоритми.

3. ***Структуровані величини та їх опрацювання.***

3.1 Структуровані типи величин. Табличні величини (масиви). Лінійні масиви та алгоритми їх опрацювання.

3.2 Алгоритми пошуку елементів у таблиці.

3.3 Алгоритми упорядкування лінійних таблиць.

3.4 Рядкові величини та алгоритми їх опрацювання.

Згідно цієї схеми тема розбита на три частини по чотири уроки. Перша з них присвячена формуванню понять алгоритма і програми. Як буде видно з наступного розділу "Експрес-курс алгоритмізації та програмування", вже в першій частині наявні дещо відмінні від канонічних трактування алгоритму, команди, програми, процедури, функції, багато уваги приділяється вводу/виводу (з цим пов'язані міркування про лінійні та діалогові алгоритми і програми, в ознайомлювальному плані розглядається поняття текстового файлу та питання використання зовнішніх файлів для вводу/виводу). Це зроблено з тим, щоб учні мали змогу краще уявити структуру і логіку теми та її місце в загальній системі знань.

Акцентується увага на терміні "*структура*",зокрема, дається чітке поняття *структури алгоритму та програми*. Далі структурованість також є

вузловим поняттям, у другій частині ідеться про *структуровані алгоритми*, а в третій про *структуровані типи*.

Ми вважаємо, що навіть при оглядовому вивченні теми повинні детально і чітко розглядатись не тільки питання математичної постановки задачі, а й питання відлагодження та виконання програми, цьому присвячений урок 4.

У другій частині вузловим є питання конструювання програми відповідно принципу покрокової деталізації. Умисно поставлені акценти на конструюванні методом "*згори донизу*", пов'язуючи його з поняттям про стандартні підпрограми та "*знизу догори*", розвиваючи ідею створення процедур і функцій користувача.

Логічним підсумком другої частини є останній урок, де вивченням циклу з параметром закінчується формування уявлень про структуру повторення, що дає змогу розглядати програми, які містять усі базові структури алгоритмів (БСА). Ми вважаємо за доцільне на відміну того, як це робиться в підручниках, БЗА розглядати окремо, вводячи їх поступово, відповідно до ускладнення задач. Це допоможе учням усвідомити суть структурного програмування взагалі.

В третій частині розглядаються найважливіші структуровані типи, зокрема масиви і рядки. У двох перших частинах вони лише згадуються та епізодично використовуються на прикладі типу *string*. Але, доводиться визнати, що, виходячи з ліміту навчального часу, у ЕКАП немає змоги для ознайомлення із записами та множинним типом, а також з рештою порядкових типів, вказівниками та процедурними типами.

Щодо проблеми *в)* можна сказати наступне. Необхідного органічного зв'язку із основною частиною курсу основ інформатики (інформаційно-комунікаційні технології) можна досягти, по-перше, вирішивши проблеми *а)-б)*, по-друге, забезпечивши наступне після теми "Комп'ютерне моделювання. Основи алгоритмізації та програмування" знайомство із об'єктно орієнтованим програмуванням (ООП) та візуальним проектуванням (ВП) і узгодивши, а в окремих випадках навіть підпорядкувавши, вивчення прикладного програмного забезпечення загального і навчального призначення (ППЗП, ППЗНП) із темою "Комп'ютерне моделювання. Основи алгоритмізації та програмування".

Кожен урок ЕКАПу є окремою темою, що, здебільшого, не можна вивчити за одну календарну годину, тому, як уже наголошувалось, слід шукати додаткові резерви часу. Але уроки написані так, що на розсуд учителя, їх можна вивчати фрагментарно. Для кращого закріплення у вигляді не обов'язкового додатку в кінці ЕКАПу є практикум із шести практичних робіт, які охоплюють прикладні і практичні задачі.

Поряд із цим відмітимо, що ЕКАП, або його ідея, може успішно застосовуватись і при значно ширшому та глибшому вивченні теми. Його можна розглядати, як *вступ до основ програмування*. Тому цілком природньо, що після практикуму розміщено збірник складніших задач із вказівками щодо розв'язування. Він включає деякі поради щодо письмового опису та дослідження алгоритмів, детальний розбір типових задач підвищеної складності та набір олімпіадних задач, які пропонувались протягом останніх років на ІІ-у і ІІІ-у турах Всеукраїнської олімпіади з основ інформатики, значна частина з яких – ексклюзивні.

II. ЕКСПРЕС-КУРС АЛГОРИТМІЗАЦІЇ ТА ПРОГРАМУВАННЯ

II.1 Поняття про алгоритми і найпростіші програми

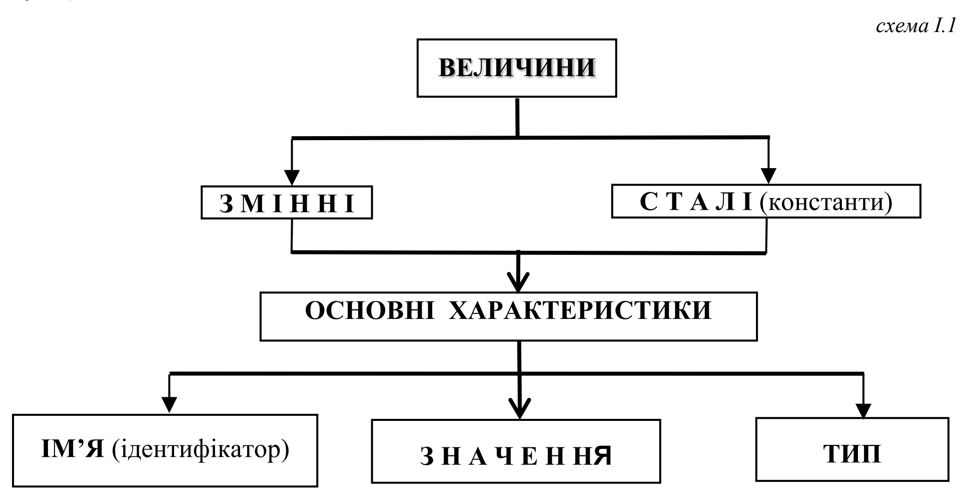
Урок I. Поняття інформаційної та математичної моделі. Етапи розв'язування задач з допомогою комп'ютера

Під *моделлю* прийнято розуміти *точну або подібну копію деякого об'єкта*, яку можна взяти за зразок, наприклад, модель (марку) мобільного телефона, зменшену, але діючу копію літака тощо.

ⓘ *Інформаційною моделлю* будемо вважати *точний опис* з допомогою тексту або аудіо-візуальних засобів *деякого матеріального чи абстрактного об'єкта або явища*, тобто *повну інформацію* про нього.

Кожна модель має *параметри*, тобто *величини*, що виражають її форму, розміри і т.д. Величини діляться на постійні (константи) та змінні і мають три характеристики: *ім'я* (інакше - позначення або ідентифікатор), *тип* та *значення* (див схему 1).

схема I.1



ⓘ *Математичною моделлю* називається *опис деякого об'єкта чи явища засобами математики, тобто у вигляді формул*.

Але для запису формул необхідно встановити, тобто описати величини, які будуть даними (*аргументами*), допоміжними та шуканими (*результатами*). Опис величин та побудова математичної моделі називається *математичною постановкою задачі*. Математична постановка – це перший етап розв'язування задачі[8], що передує визначенню її результатів. На уроках математики, фізики та інших предметів визначення результатів виконують особисто, тобто вручну, інколи для проміжних обчислень використовуючи калькулятор чи якусь спеціальну комп'ютерну програму, наприклад *Microsoft Excel*.

Існують задачі, що передбачають автоматизацію вводу аргументів, обчислень та виводу результатів через їх дуже велику кількість, тому розв'язуються з

[8] Під задачами далі будемо розуміти завдання, виконання яких потребують застосування математики, а тому й побудови математичної моделі.

допомогою комп'ютера в автоматичному режимі. Процес розв'язування таких задач складається з кількох етапів, зокрема:

a) Виконання математичної постановки задачі;

b) Опису, дослідження та доведення алгоритму[9];

c) Складання та тестування (перевірка правильності) програми;

d) Виконання програми і одержання результатів (відпові задачі).

Після математичної постановки задачі у "ручному" режимі виконуються необхідні обчислення і одержуються результати. При розв'язуванні задачі з допомогою комп'ютера в автоматичному режимі потреби обчислювати особисто немає, замість цього слід виконати п. *b) - d)*.

Приклад І.1 (**Прямокутник**) За даними довжиною і шириною прямокутника визначити його периметр і площу. Виконати математичну постановку та для даних значень аргументів обчислити результати.

Математична постановка:

- *Опис величин*. Нехай a та b – дані довжина і ширина прямокутника, а p та s його периметр і площа. Всі величини числові[10] і повинні мати дійсні значення.
- *Математична модель*. За відомими формулами $p = 2(a+b)$, $s = ab$.

Обчислення. При $a = 3{,}5$ см та $b = 2{,}2$ см $p = 2(3{,}5+2{,}2) = 11{,}4$ (см); $s = 3{,}5 \cdot 2{,}2 = 7{,}7$ (см).

Приклад І.2 (**Сума**) Обчислити суму перших 2500 натуральних чисел. Виконати математичну постановку розв'язання цієї задачі для n перших натуральних чисел.

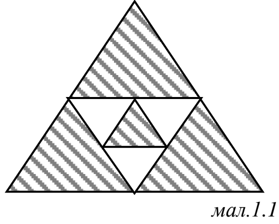
Уявивши вираз для обчислення суми перших 2500 натуральних чисел: 1+2+3+4+5+…+99+100+101+…+1000+1001+…+2499+2500, та час, потрібний для обчислення вручну, доведеться погодитись, що розв'язування цієї задачі слід почати з другої частини, тобто з математичної постановки.

Математична постановка:

- *Опис величин*. Нехай n – кількість доданків, а S_n – їх сума. Обидві величини повинні бути натуральними числами.
- *Математична модель*. Записавши шукану суму та підмітивши, що суми доданків, однаково віддалених від кінців, рівні між собою, а їх кількість у двічі менша кількості доданків, одержимо: $S_{2500} = (1+2500) + (2+2499) + \ldots + (1250 + 1251)$, після узагальнення, одержимо математичну модель: $S_n = (1+n)\cdot n/2$.

Обчислення. При $n = 2500$ $S_n = (1 + 2500)\cdot 2500/2 = 3126250$.

Приклад І.3 (**Фігура**) Виконати математичну постановку задачі обчислення площі заштрихованої фігури, що складається з n вписаних правильних трикутників, якщо сторона зовнішнього трикутника a і заштриховано трикутники з непарними номерами (на малюнку 1 $n = 3$)



мал.1.1

Математична постановка:

- *Опис величин*. Нехай n – кількість доданків, a – сторона зовнішнього правильного трикутника, а s_n – площа заштрихованої фігури. Додатково позначимо площу зовнішнього трикутника s_o. На відміну від попередніх прикладів, якщо n – натуральне, то a, s_n та s_o – повинні бути дійсними числами.
- *Математична модель*. Легко помітити, що площа вписаного трикутника завжди в четверо менша за площу описаного трикутника, тому: $s_n = s_o - s_o/4 + s_o/16 - s_o/64 + s_o/256 - \ldots$ Але ця формула з двома суттєвими недоліками: у виразі відсутній параметр n, вона має два варіанти – для парних і непарних n, крім того, s_o слід

[9] Окремо і детально поняття алгоритму, програми та етапи *b)-d)* будуть розглядатись нижче.

[10] В інформатиці, на відміну від математики, величини можуть мати не тільки числові значення.

виразити через a, тобто її слід узагальнити. Для цього достатньо помітити, що доданки становлять геометричну прогресію з першим членом s_0 та знаменником, рівним -1/4. Отже:

$$s_n = \frac{s_0((-\frac{1}{4})^n - 1)}{-\frac{1}{4} - 1}, \text{ при } s_0 = \frac{a^2\sqrt{3}}{4}. \qquad (I.3.1)$$

Тому остаточний вигляд формули такий: $s_n = \dfrac{a^2\sqrt{3}((-\frac{1}{4})^n - 1)}{5}$. (I.3.2)

Формули (II.3.1) та (II.3.2) – це два варіанти математичної моделі даної задачі. Але завжди слід визначати остаточний, найбільш загальний варіант[11], тому математичною моделлю задачі будемо вважати (II.3.2).

? *Питання до теми уроку*:

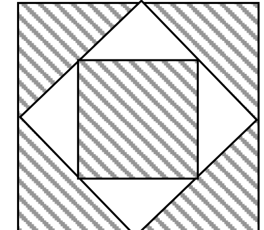
I.1 Чи можна вважати інформаційну модель математичною моделлю?
I.2 Чи можна вважати математичну модель інформаційною моделлю?
I.3 Яка різниця між інформаційною та математичною моделями?
I.4 Чи є в прикладах II.1-II.3 допоміжні величини, тобто ті, які не можна назвати ні аргументами, ні результатами?
I.5 Чи можна на свій розсуд вказувати типи величин, виконуючи математичну постановку задачі?
I.6 Чи можуть аргументи бути константами? змінними?
I.7 Чи можуть результати бути константами? змінними?
I.8 Чи можуть допоміжні величини бути константами? змінними?

Перед виконанням завдань ще раз перегляньте текст уроку, звернувши увагу на виділені курсивом слова та словосполучення, усвідоміть їх зміст та значення[12].

Завдання I.1 За даним радіусом кола визначити його довжину та площу відповідного круга. Виконати математичну постановку та для даного значення аргумента обчислити результати.

Завдання I.2 Обчислити кількість усіх діагоналей для опуклого n-кутника при даному n. Виконати математичну постановку розв'язання задачі.

Завдання I.3 Виконати математичну постановку задачі обчислення площі заштрихованої фігури, що складається з n вписаних квадратів, якщо сторона зовнішнього квадрата a і заштриховано квадрати з непарними номерами (на малюнку 2 $n = 3$)



мал.I.2

[11] Зведення формули до остаточного, найбільш загального і зручного вигляду умовно можна назвати дотриманням "правил інтелектуального етикету" (частіше вживають: "правил доброго смаку", тому умовно позначатимемо – ПДС) при математичних перетвореннях. Як побачимо далі, ПДС надзвичайно важливі у програмуванні.
[12] Цієї поради необхідно дотримуватись при вивченні всіх наступних уроків.

Урок II. Алгоритми, їх властивості та виконавці. Система команд виконавця. Форми подання алгоритмів. Поняття програми.

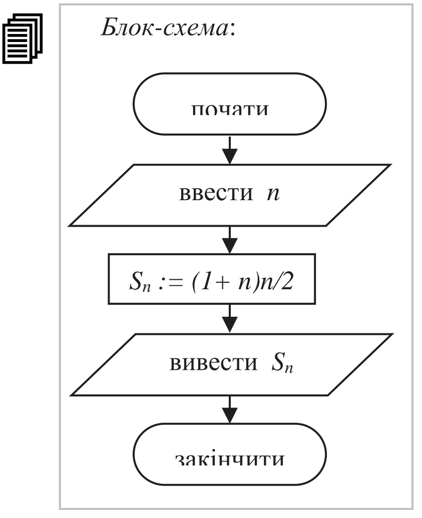
ⓘ Алгоритмом будемо вважати *скінченну* і *однозначну послідовність* команд, *повне* і *точне* виконання яких забезпечує *досягнення* поставленої мети або розв'язання даної задачі.

Алгоритм належить до основних, *неозначуваних* понять точних наук, зокрема математики та інформатики. Він складається із скінченної (не пустої!) кількості *команд*, тобто вказівок і адресується *виконавцеві*. Команди вказують на *дію* і *перехід*[13]. Виконавець уміє виконувати деякий набір команд, що називаються *системою його команд* (СКВ). До СКВ обов'язково повинні входити дві "пусті" команди, які не містять дії: *почати* – без дії перейти до наступної команди, та *закінчити* – не переходити до жодної команди.

Послідовність команд буде алгоритмом лише тоді, коли задовольняє *властивостям*:

1) Дискретність[14] – полягає в тому, що алгоритм складається із скінченної кількості *окремих* команд;
2) Зрозумілість – полягає в тому, що кожна команда алгоритма повинна входити у СК вибраного виконавця;
3) Точність – полягає в тому, що серед команд алгоритму повинна бути встановлена черговість, тобто переходи;
4) Масовість – полягає в тому, що виконання алгоритма забезпечує розв'язання будь-якої задачі даного типу;
5) Результативність – полягає в тому, що через *скінченне число кроків* (виконаних команд) та за *обмежений час* обов'язково отримується результат.

Особливістю алгоритму є *формальність виконання*, яка полягає в тому, що виконавець не розглядає суть поставленої задачі, він лише знає і точно та повністю виконує всі команди алгоритму, а це обов'язково приводить до очікуваного результату. Отже, виконавцем може бути як жива, розумна істота,



Словесна форма:

1. почати. п.2
2. ввести аргумент *n*. п.3
3. обчислити S_n за формулою $S_n = (1+n)n/2$ п. 4
4. вивести результат S_n. п.5
5. закінчити

Запис на УНАМ:

```
алг Сума
  дано n цілі
  знайти sn дійсні
поч
  Ввести n
  Sn := (1+ n)*n/2
  Вивести (sn)
кін
```

Запис мовою програмування Pascal:

```
program Summa;     {заголовок}
  var n : integer;   {блок описів}
      Sn: real;
begin   {початок тіла програми}
  Read (n);
  Sn := (1+ n)*n/2;
  Write (Sn);
end.    {кінець тіла програми}
```

[13] Тут, відповідно до властивості алгоритму 3), щоб не вводити окремого поняття зв'язку між командами, дію переходу свідомо включено в конструкцію команди. У словесному адреси переходів записані в кінці кожного пункту, у мові програмування Паскаль їх роль відіграють знак ";" та слова "*end*" і "*else*".

[14] Дискретний – не суцільний, той, що складається з окремих частин.

так і автомат. Завдяки цій особливості виконання алгоритмів покладено на комп'ютери, перевагою останніх є можливість виконувати необмежену кількість алгоритмів при обмежених пам'яті та СКВ. Кожен алгоритм повинен бути описаним відповідно до СКВ.

Є кілька основних форм опису алгоритмів: *словесний*, *графічний* (у вигляді *блок-схеми*), на спеціальній *умовній навчальній алгоритмічній мові* (УНАМ) та *мовою програмування*. Наведемо запис алгоритма розв'язування вибраної задачі (наприклад, І.2 з уроку І) у кожній з цих форм.

Описані у словесному вигляді алгоритми важко читати й розуміти, тому така форма застосовується мало, переважно у випадках, коли вимагається найбільш загальний опис. Блок-схеми складаються із блоків, з'єднаних стрілками, що вказують на перехід. Їх перевагою є наочність, а недоліком - громіздкість. Крім того, перевірка правильності описаних словесно та за допомогою блок-схем алгоритмів досить трудоємка.

УНАМ наближена до людської мови, вона дуже лаконічна і формалізована, тому зручна для опису складніших і більш громіздких алгоритмів, призначених для читання і схематичного виконання людиною, але через мовний бар'єр все ж обмежена в користуванні. Існують навіть програмні середовища, що дозволяють записані умовною навчальною алгоритмічною мовою алгоритми виконувати безпосередньо на комп'ютері. Проте, поряд із обмеженими можливостями, ці середовища мають багато недоліків, їх застосування практично обґрунтоване лише на початкових, навчальних етапах і передбачає обов'язковий перехід до використання більш потужних засобів опису алгоритмів. Тому, відповідно ПДС, алгоритми прийнято записувати з допомогою *спеціальних різновидів алгоритмічних мов*, "зрозумілих" комп'ютерам – *мов програмування*, у такому випадку самі алгоритми називаються *програмами*, або *програмними кодами*. Ми будемо користуватись мовою *Pascal*, яка універсальна і зручна для навчання та практичного програмування.

До особливостей алгоритмічних мов, зокрема мов програмування, належить наявність структури, тобто *заголовка*, *описової частини* , *тіла алгоритму* і т.д., можливість вставки коментарів (у мові *Pascal* беруться у фігурні дужки). Дуже важливою особливістю є *обов'язкова лінійна форма* запису математичних виразів, що вимагає:

- запису будь-якого математичного виразу в один рядок (порядок операцій змінюється з допомогою дужок);
- обов'язковість та особливість знаків математичних операцій;
- використання для окремих математичних операцій особливої форми запису – у вигляді функцій.

Таблиця основних знаків математичних операцій та деяких стандартних числових функцій, що застосовуються в алгоритмічних мовах:

таблиця ІІ.1

назва операцій чи функції	*форма запису*
Додавання	*x+y*
Віднімання	*x-y*
Множення	*x*y*
Ділення	*x/y*

квадрат числа	*Sqr(x)*
арифметичний квадратний корінь	*Sqrt(x)*
абсолютна величина	*Abs(x)*
Синус	*Sin(x)*
Косинус	*Cos(x)*
натуральний алгоритм	*Ln(x)*
остача від ділення	*x mod y*
неповна частка	*x div y*
ціла частина числа	*Int(x)*

Приклади записів математичних виразів у лінійній формі:

таблицяII. 2

	вираз	*лінійний запис*
Приклад II.1	$\frac{-1}{x^2}$	*-1/sqr(x)*
Приклад II.2	$\frac{a}{bc}$	*a/(b*c)*
Приклад II.3	$\frac{a}{b}c$	*a/b*c* або *(a/b)*c*
Приклад II.4	$\frac{-b+\sqrt{b^2-4ac}}{2a}$	*(-b+sqrt(sqr(b)-4*a*c))/(2*a)*
Приклад II.5	$\frac{ad+bc}{bd}$	*(a*d+b*c)/(b*d)*
Приклад II.6	$\frac{tg\alpha}{\sqrt{1+tg^2\alpha}}$	*Sin(a)/cos(a)/sqrt(1+sqr(sin(a)/cos(a)))*

Мови програмування бувають проблемно-орієнтованими і машинно-орієнтованими, низького і високого рівня. Комп’ютер виконує програми, записані лише відповідною йому машинно-орієнтованою мовою – мовою машинних кодів. Але записувати алгоритми такою мовою дуже важко, тому існують спеціальні програмі середовища, що дозволяють виконувати програми на комп’ютері, записані проблемно-орієнтованими мовами, аналогічними алгоритмічній мові. Їх називають трансляторами (перекладачами).

Усі мови, у тому числі й людські – це засоби спілкування, а тому мають як багато схожого, так і свою власну специфіку. Специфіка алгоритмічних мов полягає в їх призначені - виключно для опису алгоритмів і програм. Але, подібно людським мовам, вони мають алфавіт, словниковий запас та правила запису речень. До алфавіту алгоритмічних мов належать усі літери латиниці та кирилиці, арабські цифри, знаки відношень (у тому числі і такі: “<=“ - не більше, “>=“ не менше, “<>“ - не дорівнює), деякі спеціальні символи, наприклад, “@“, “&“ та деякі елементи псевдографіки, наприклад, “║“, “╣“, “╨“, “▒“ тощо.

Словниковий запас – це зарезервований набір слів, які не можуть використовуватись інакше, ніж дозволено у даній алгоритмічній мові. В записах алгоритма ***Сума*** УНАМ використовує зарезервовані слова ***алг***, ***дано***, ***знайти***, ***поч***, ***кін***, а *Pascal* у відповідній програмі слова – *program*, *var*, *begin* та *end* відповідно, що означають початок заголовка алгоритму, початок і кінець його основної частини – тіла та початок описової частини.

На відміну від людських, в алгоритмічних мовах можна записувати лише розповідні (твердження або умови) та спонукальні (команди) речення. Кожна алгоритмічна мова має свій дуже строгий правопис (синтаксис).

Питання до теми уроку:

II.1 Чи може бути алгоритм без команд?

II.2 Чи може не містити аргументів або результатів?

II.3 Що означає запис $S_n := (1+n)*n/2$ *у алгоритмі* ***Сума***?

II.4 Які команди повинні обов’язково належати до СКВ виконавця алгоритму ***Сума***?

II.5 Чи можна виконати команду, що не входить до СКВ (проаналізуйте таблиці 1 та 2)?

II.6 Які синтаксичні правила УНАМ можна встановити з алгоритму ***Сума***?

II.7 Які синтаксичні правила *Pascal* можна встановити з програми ***Summa***?

II.8 Які синтаксичні відмінності УНАМ та *Pascal* можна вказати за алгоритмом ***Сума*** та програмою ***Summa***?

Завдання II.1 Записати словесно, у вигляді блок-схеми та на УНАМ алгоритми завдань І.1-І.3.

Завдання II.2 Записати словесно та у вигляді блок-схеми алгоритм розв'язування квадратного рівняння.

Завдання II.3 Записати в лінійній формі математичні вирази:

$\frac{mv^2}{2} + mgh;\quad |x| + |x+1|;\quad \frac{\sqrt{x+1} - \sqrt{x-1}}{2\sqrt{x}};\quad \sqrt{ax^2 + bx + c}.$

Завдання II.4 Записати в лінійній формі математичні вирази у формулах (І.3.1) та (І.3.2).

Завдання II.5 Записати в математичній формі лінійні вирази:

a) sqrt(sin(A+B))/sqr(cos(A+B));

*b) 1/(1+x*x);*

*c) 2*sin((a+b)/2)* cos((a+b)/2).*

Урок III. Структура програми. Опис величин мовою програмування. Лінійні програми опрацювання величин. Команди вводу, виводу та присвоєння.

Блок-схема алгоритму складаються з окремих елементів - блоків, тобто має певну *структуру*. Програму ***Summa*** можна також уявити у вигляді блоків (схема III.1).

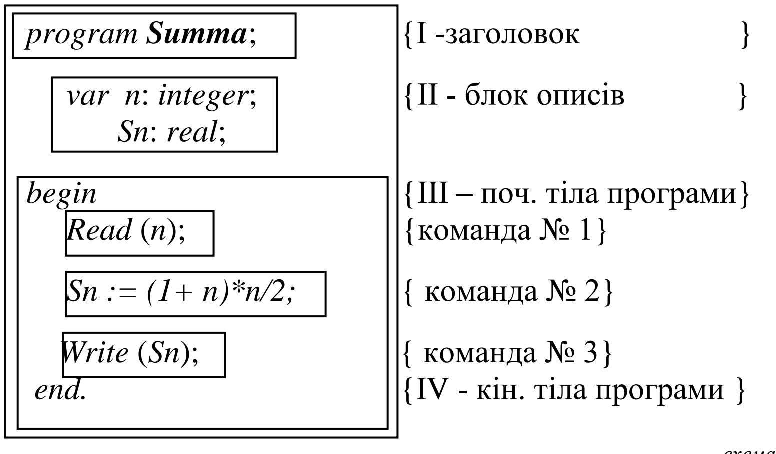


схема III.1

І не принципово, щоб вони мали різну форму. Легко помітити додаткові блоки – *заголовка* та *описів*, а також наявність в окремих блоках, наприклад у тілі програми, *вкладених блоків* окремих команд.

<u>*Структурою*</u> будемо називати *перелік елементів* чогось *цілого*, якщо вказано *форму елементів* та *взаємозв'язок* між ними.

Наведемо загальну структуру програм, написаних мовою *Pascal*. Але спочатку домовимось дотримуватись деяких правил та умовностей. У тексті програми *дозволяється*:

– використовувати тільки латиницю (до нього не входять коментарі та значення у *команді виводу*);
– імена складати тільки із допустимих символів;

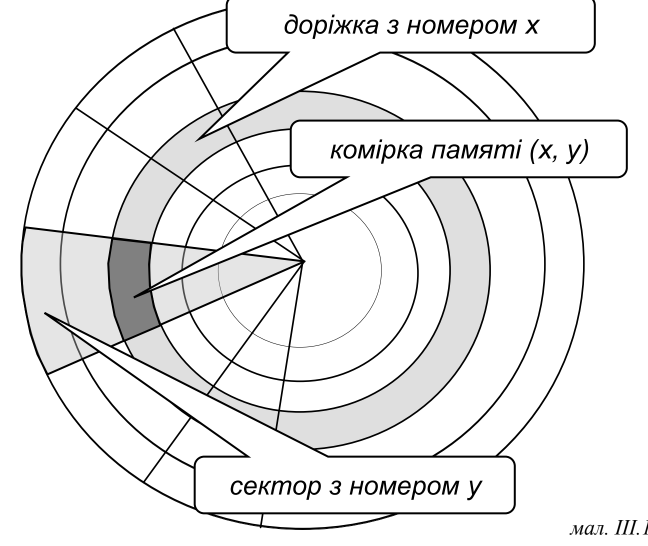
– не розрізняти великі й малі літери;
– пропуски між зарезервованими словами;
– ігнорувати іменами та розділовими знаками;
– у одному рядку записувати довільну кількість команд.

В узагальнених записах програм та їх фрагментів слід дотримуватись таких особливостей: записи, взяті в квадратні дужки – не обов'язкові, тобто, повинні бути відсутніми, або записаними без квадратних дужок; узагальнені назви для кращого розуміння можна записувати українською мовою. Для наочності вкладені блоки будемо виділяти окремим рядком і відступом на кілька позицій вправо, наприклад, до програми входить блок оголошень та описів змінних, тому його запишемо з нового рядка не три позиції правіше. Види розділових знаків: між елементами блоків – пропуск, між командами та блоками – ";", при переліку імен у описах чи операндів у командах – ",", при відкритті та закритті вкладених блоків – зарезервовані слова *begin* і *end*. У зв'язку з цим конструкція *begin…end* називається операторними дужками.

таблиця III.1

[*program* Назва програми;]		*{I - заголовок програми}*
[*uses* Ім'я модуля;]		*{II - розділ підключень модулів}*
[*const* ім'я величини = значення;]	*{розділ оголошень та описів величин}*	*{III - оголошення та опис констант}*
[*type* ім'я величини = ім'я типу;]		*{IV - оголошення та опис типів}*
[*var* ім'я величини : ім'я типу;]		*{V - оголошення та опис змінних}*
[*{розділ оголошень та описів підпрограм}*]		
Begin		*{VI - початок тіла програми}*
[*{ тіло програми}*]		
End.		*{VII - кінець програми}*

Як видно з таблиці III.1, заголовок програми не обов'язковий, але складається із зарезервованого слова *program*, пропуску, назви програми та розділового знаку ";". Уважно розглянувши кожен наступний блок, можна самостійно зробити висновки щодо форми його запису, маючи на увазі, що вона єдина.



мал. III.1

Перед використанням величин у командах програми їх необхідно *оголосити* і *описати*. Оголосити величину – це вказати її ім'я, описати – вказати її тип. Описи величин, залежно від їх типу, відрізняються, але завжди починаються зарезервованим словом: сталі – *const*, змінні – *var*. Якщо описується кілька однотипних змінних величин, їх можна перерахувати, розділяючи комою, і через двокрапку один раз указати тип, наприклад: *n, Sn*: *integer*;

Крім стандартних типів у програмі можуть бути описані типи користувача, блок їх опису починається зарезервованим словом *type*. Наприклад, у програмі ***Summa*** блок II можна записати так:

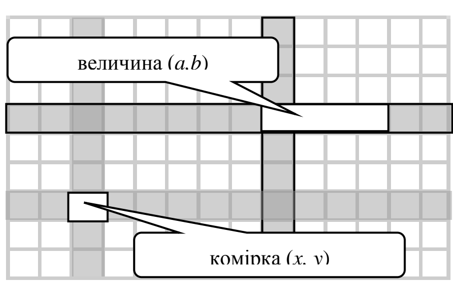
type userType = integer;

var n, *Sn*: *userType*;

Застосування типів користувача часто буває дуже зручним, тому вважається дотриманням ПДС при програмуванні.

Слід зауважити, що загальнонаукове поняття величини в інформатиці має дещо специфічний зміст. Це пов'язано з розміщенням величин у пам'яті комп'ютера. Тому, з точки зору програмування, *величина – це область пам'яті комп'ютера*, що має *адресу* і *розмір*. Величина, залежно від її типу, може займати одну, кілька або багато сусідніх *комірок* пам'яті. Комірка має дві координати (адресу): *x* – номер доріжки та *y* – номер сектора на диску (мал. III.1). Величина також має адресу – це координати її першої комірки. Тому величину в пам'яті комп'ютера можна уявити, як зображено на малюнку III.2.

Величини різних типів займають різну кількість комірок пам'яті (байтів). Це пов'язано з тим, що в програмуванні, на відміну від математики, число, наприклад ірраціональне, не може містити безліч знаків. Тому у мовах програмування, зокрема в *Pascal*, для представлення числових величин існує по кілька цілочисельних і дробових типів (див. табл. III.2)[15]. Не важко помітити, що кожен із дробових типів, на відміну від математики - дискретний, тому не охоплює жодну із множин чисел (раціональних, ірраціональних, дійсних тощо).



мал. III.2

таблиця III.2

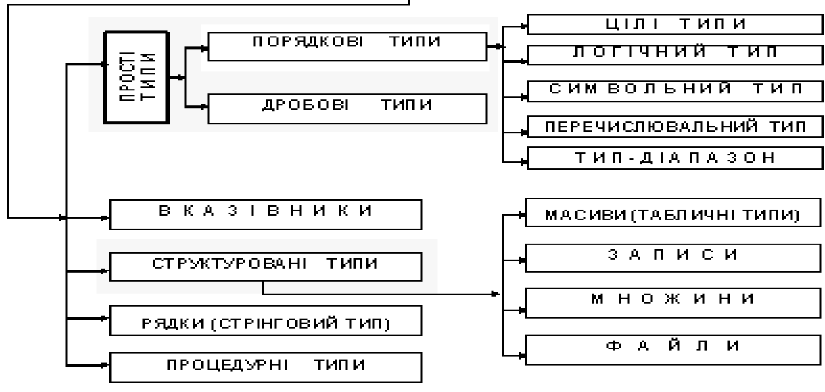
Ö²Ë² ÒÈÏ È

ò èï	äîïóñòèì ³ çíà÷åííÿ	ôîðìàò
SHORTINT	**-128 .. 127**	**1 áàéòç³ çí àêîì**
INTEGER	**-32768.. 32767**	**2 áàéòè ç³ çí àêîì**
LONGINT	**-2147483648 .. 2147483647**	**4 áàéòè ç³ çí àêîì**
BYTE	**0 .. 255**	**1 áàéò áåç çí àêà**
WORD	**0 .. 65535**	**2 áàéòè áåç çí àêà**

ÄÐÎ ÁÎ Â² ÒÈÏ È

ò èï	äîïóñòèì ³ çíà÷åííÿ	òî÷í ³ñòü	ôîðìàò
REAL	**$2.9*10^{-39}$ .. $1.7*10^{+38}$**	**11 — 12 çí àê³â**	**6 áàéò**
SINGL	**$1.5*10^{-45}$ .. $3.4*10^{+38}$**	**7 — 8 çí àê³â**	**4 áàéòè**
DOUBLÅ	**$5.0*10^{-324}$ .. $1.7*10^{-308}$**	**15 — 16 çí àê³â**	**8 áàéò**
EXTENDED	**$3.4*10^{-4932}$ .. $1.1*10^{4932}$**	**19 — 20 çí àê³â**	**10 áàéò**
COMP	**$-9.2*10^{+18}$ .. $9.2*10^{+18}$**	**19 — 20 çí àê³â**	**8 áàéò**

таблиця III.3



[15] Таблиця III.2 наводить типи для версії *Turbo Pascal*.

Це ж стосується й цілочисельних типів, крім того, цілі числа в програмуванні не входять, на відміну від математики, до дробових типів, вони разом із *логічними*, *символьними*, *перечислювальними* типами та *типом-діапазоном* утворюють групу *порядкових* типів величин (див. табл. III.3). Порядковими називаються типи величин (зверніть увагу – не тільки числових!), які передбачають встановлення порядку, тобто черговості значень величин такого типу. Крім числових, до них відносяться, як видно зі схеми, логічний тип, що має два значення : *true* та *false*, та символьний тип, який можна інтерпретувати, як текст довжиною в один знак. Перечислювальний тип та тип-діапазон можуть містити, як числові, так і символьні значення.

ⓘ *Лінійними* називаються програми, команди яких виконуються у порядку їх запису.

Обчислювальні лінійні програми можуть містити лише блоки *вводу даних*, *опрацювання інформації* та *виводу результатів*, які у найпростіших прикладах записуються з допомогою команд *вводу з клавіатури* (*Read*), присвоєння та виводу результатів на екран (*Write*). Ці команди об'єднані під загальною назвою – *структури слідування*. Подивившись на програму ***Summa***, легко помітити, що вона містить три команди, які виконуються в порядку запису. Перша команда забезпечує ввод аргументу *n*, вона називається *командою вводу*. Різні варіанти цієї команди такі: *Read(a,b)*, *ReadLn(a)* тощо. Формат (загальний вигляд) команди вводу:

Read[*Ln*](*параметр1*,[*параметр2,…*])

Друга команда забезпечує обчислення виразу *(1+ n)*n/2* та надання його значення величині *Sn*. Формат цієї команди такий: *ім'я величини := вираз*. Знак ":=" читається, як "*надати*", або "*присвоїти*".

Третя команда виводить результат *Sn* на екран. Подібно до команди вводу, команда виводу також має кілька модифікацій. Наприклад, для значення *Sn*= 314159265.36:

	команда:	*результат на екрані:*	*форма виводу:*
1	*Write(Sn)*	*314159265.36*	з фіксованою десятковою точкою
2	*Write('Sn=',Sn)*	*Sn=314159265.36*	з фіксованою десятковою точкою
3	*WriteLn('Sn=',Sn)*	*Sn=314159265.36*	з фіксованою десятковою точкою
4	*Write('Sn=',Sn:12:1)*	*Sn= 314159265.4*	з фіксованою десятковою точкою
5	*Write('Sn=',Sn:12:3)*	*Sn=314159265.360*	з фіксованою десятковою точкою

Для кращого розуміння фіксованого виводу чисел неведемо ще одну таблицю, у якій у вигляді "___" представлено пропуски зліва від числа, що виводиться.

WriteLn(14.859:10:3);	*___14.859*
WriteLn(14.859:10:5);	*_14.85900*
WriteLn(14.859:10:2);	*____14.86*
WriteLn(14.859:10:0);	*________15*
WriteLn(14.859:10:7);	*14.8590000*

Формат команди виводу:

Write[*Ln*](*операнд1*,[*операнд2,…*])

Операндами можуть бути числа, значення виразів та тексти (набори символів). Тексти обов'язково слід обмежувати з обох боків знаком апострофа.

Для виводу числових величин існує дві форми: *форматована* (з фіксованою десятковою точкою) та *експоненціальна* (з плаваючою десятковою точкою). Для демонстрації експоненціальної форми виводу можна для *Sn* взяти

значення 3141592222111265.36[16], у результаті команди *Write(Sn)* на екрані з'явиться 3.14159222211127E15. Ця форма виводу схожа на стандартний запис чисел у математиці, тобто відповідає запису: $3,14159222211127 \cdot 10^{15}$ (Е - експонента, назва функції $y = 10^x$ у алгебрі).

? *Питання до теми уроку*:

III.1 Чому в записах алгоритмів словесно та у вигляді блок-схеми відсутні блоки заголовка та описів?

III.2 Чи може програма мати вигляд "*begin end.*"? Якщо так, то скільки вона містить команд?

III.3 Чи може бути у програмі команда "*begin end.*"? "*begin end*;"?

III.4 Чи може команда бути вкладеною в іншу команду? Відповідь обгрунтуйте.

III.5 Чи випадково у програмі *Summa* окремі слова записано з великої літери? Чи обов'язково це?

III.6 Порівняйте записи:*(1+ n)*n/2:=Sn* та *Sn:=(1+ n)*n/2*. Який з них не є командою? Чому?

III.7 Чому в програмі *Summa* величина *Sn* не може бути описана, як ціла, хоча її значення, очевидно, є натуральним числом?

III.8 Чи можуть існувати команди *Read*? *Read*()? *Write*? *Write*()?

III.9 Назвати усі розділові знаки у програмі *Summa*. Чому після слова begin відсутній розділовий знак ";"?

III.10 Які блоки у програмі *Summa* - вкладені?

III.11 Чи завжди програма Summa виводитиме на екран правильну відповідь? Чому?

III.12 Яка різниця в роботі програм *Summa*, *Summa1* та *Summa2*?

```
program Summa1;
  var n:integer;Sn:real;
begin
  Write('n?');Read(n); Sn:=(1+ n)*n/2; Write('Sn=',Sn);
end.
  var n:integer;Sn:real; {програма Summa2}
begin
  Write('n?');Read(n); Sn:=(1+ n)*n/2; Write('Sn=',Sn);
end.
```

✍ <u>*Завдання III.1*</u> Написати програми обчислення значень виразів, наведених у завданні II.2.

<u>*Завдання III.2*</u> Написати програми завдання III.1 у найкоротшій формі.

<u>*Завдання III.3*</u> Написати програми завдання III.1 з коментарями.

<u>*Завдання III.4*</u> Написати програму *Circle*, яка виконує завдання I.1.

<u>*Завдання III.5**</u>[17] Написати програму *Figura_Triangle*, яка розв'язує задачу прикладу I.3.

<u>*Завдання III.6**</u> Написати програму *Figura_Square*, яка розв'язує задачу завдання I.3.

Урок IV. Виконання програми та аналіз її правильності. Покрокове виконання і відлагодження програми.

Програми або програмні коди, написані відповідно до попередньо *проаналізованих*, тобто доведених і оцінених, алгоритмів, призначені для *виконання*, що по суті є автономним процесом, тому результати можуть бути не зрозумілими або не очікуваними.

[16] Реалізовано у версії *PascalABC*.

[17] Завдання, позначені значком "*" – підвищеної складності.

ⓘ *Доведенням* алгоритма називають *обґрунтування його відповідності* властивостям 1)-5) (див урок II).

ⓘ *Оцінкою складності алгоритма* називають виявлення *залежності* орієнтовної кількості *елементарних арифметичних та логічних операцій*, що необхідно виконати процесору комп'ютера для повного виконання відповідної алгоритму програми від вхідних параметрів та застосованих методів.

Команду вводу з клавіатури або виводу на екран при оцінці складності можна вважати, як одну елементарну операцію, або ігнорувати.

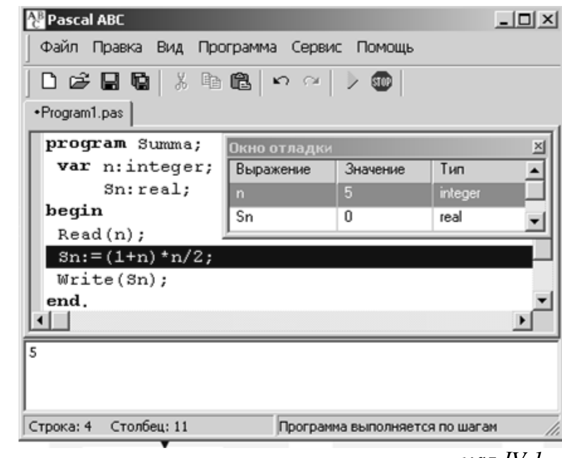
Алгоритми, що наводились вище є *очевидними*, тобто такими, що не вимагають доведення та оцінки складності. Як уже відмічалось, прості (очевидні) алгоритми слід записувати відразу мовою програмування. У такому випадку слід говорити про доведення та оцінку складності програми. Для прикладу візьмемо програму *Summa*. Очевидно, що вона повністю відповідає властивостям алгоритмів 1)-5), а кількість елементарних операцій вичерпується командою *Sn:=(1+ n)*n/2*, тобто дорівнює 3, якщо ж враховувати ввод/вивод, то 6.

Програма вважається *правильною*, або *коректною*, *якщо вона при виконанні завжди видає* (повертає) *коректні результати*. Але часто навіть досвідчені програмісти допускають помилки, через які програми або зовсім не працюють, або дають помилкові результати. Ця обставина вимагає процес написання програми поєднувати з *відлагодженням*, тобто можливістю покроково виконувати та тестувати її ще в процесі написання, а тому відокремлювати ввод і вивод, застосовуючи їх різні форми.

Помилки у програмах бувають *логічні*, *синтаксичні* та *помилки виконання*. Останні два типи помилок приводять до аварійних зупинок. Всі можливі типи таких помилок класифіковані, їх можна прочитати у довідці транслятора. На кожну синтаксичну помилку на екран видається повідомлення про її характер.

Помилки виконання зустрічаються у синтаксично правильних програмах, якщо компілятору пропонується виконати некоректну операцію, наприклад ділення на нуль, визначення арифметичного квадратного кореня з від'ємного числа тощо.

Логічні помилки приводять до одержання некоректних результатів. Значної частини логічних помилок вдається уникнути ще на етапі аналізу алгоритму. Для усунення решти логічних помилок застосовують тестування програми. *Тестами* називають варіанти вхідних даних та відповідних їм результатів. Тести повинні передбачати як тривіальні, так і екстремальні випадки вхідних даних та результатів.



мал.IV.1

Як уже згадувалось, процес виправлення помилок у програмі називається її відлагодженням. Найзручніше відлагоджувати програму в режимі її *покрокового виконання* (трасування) з використанням *вікна відлагодження Watch*, у яке під час покрокового виконання виводяться поточні значення величин (мал. IV.1)[18].

Для трасування програм застосовуються комбінації клавіш:

F2 – вихід з режиму трасування;

F4 – виконання програми до курсора;

F7 – трасування із покроковим виконанням підпрограми;

F8 – трасування без покрокового виконання підпрограми;

Ctrl+F5 – внесення змінної у вікно *Watch*.

Під час відлагодження програм доцільно уникнути введення з клавіатури аргументів, особливо, коли їх кількість значна. Цього можна досягти кількома способами:

1) Замінивши команду вводу командою присвоєння. Наприклад, у програмі *Summa* замінивши команду *Read*(*n*) командою *n* := 5. При цьому для виконання програми з іншим параметром його потрібно записати в команді присвоєння. Слід зауважити, що застосування цього способу не відповідає ПДС.

2) Застосувавши тип *const*. Кращим є спосіб з використанням типу констант. У таблиці III.1 оголошення та опис констант передбачено перед оголошенням типу змінних. У цьому випадку програма *Summa* матиме такий вигляд:

Приклад IV.1

```
program Summa;
  const n=5;                          {1}
  var {n:integer;} Sn:real;           {2}
begin
  {Read(n);}[19]                      {3}
  Sn:=(1+n)*n/2;  Write(Sn);
end.
```

Користуючись вводом аргументів у якості констант, треба мати на увазі, що у тілі програми не можна змінювати значення величини-константи з допомогою команди *Read* або команди присвоєння, тому слід звернути увагу, що запис {1} не сумісний із записами {2} та {3}. У зв'язку з цим такий спосіб також обмежений у використанні і не завжди відповідає ПДС. Але є випадки, коли спосіб 3) може бути дуже корисним. Це ми покажемо при вивченні табличних величин.

3) Застосувати для вводу даних та виводу результатів пов'язані з програмою зовнішні файли. Цей спосіб найбільш ефективний, тому обговоримо його детальніше на прикладі варіанту цієї ж програми *Summa1*:

Приклад IV.2

```
program Summa1;
  var n:integer; Sn:real;
      f: text;                                                          {1}
begin
  Assign(f, ' повне ім'я файлу, з якого читається аргумент');{2}
  Reset(f);                                                             {3}
  Read(f, n);                                                           {4}
  Close(f);                                                             {5}
  Sn:=(1+n)*n/2;  Write(Sn);
```

[18] На мал. IV.1, як і далі, показано вигляд процесу відлагодження та вікна *Watch* у *PascalABC*.

[19] Тут і далі будемо акцентувати на окремих фрагментах програм, що розглядаються, виділяючи їх напівжирним шрифтом.

end.

Перш за все файл, що містить значення аргумента (величини *n*) повинен бути створеним окремо. Для зв'язку з ним необхідна *файлова змінна* (у нашому випадку *f*), яка оголошується типом *text* {1}. Перед читанням необхідно з допомогою команди *Assign* файлову змінну *зв'язати* з файлом, що містить значення аргумента (див. формат *Assign* в {2}). Наступним кроком є відкриття файла для читання з допомогою процедури *Reset* (формат *Reset* у {3}). Саме читання здійснюється з допомогою команди *Read*, або *ReadLn*, але з указанням на першому місці файлової змінної (формат у {4}). Нарешті файл, відкритий для читання, слід закрити командою *Close* (її формат в {5}).

Тут показано найпростіший приклад читання аргументу з файлу. Безперечно для прикладу програми *Summa* його використання також не відповідає ПДС. Але зауважимо, що при читанні великої кількості даних, зокрема масивів, текстів тощо, про що йтиметься в наступних уроках, цей спосіб зчитування аргументів буває безальтернативним.

Крім того, подібним чином можна результати роботи програми виводити не на екран, а у зовнішній текстовий файл, що також додає великі перспективи у застосуванні програмування для розв'язування прикладних задач. Запис інформації у файл текстового формату забезпечується двома командами: *ReWrite(f)* та *Append(f)*, перша – створює файл з іменем, що записане у команді *Assign*, друга – відкриває файл для доповнення у його кінець. В обох випадках запис у файл здійснюється командою *Write* чи *WriteLn*, але з указанням у якості першого операнду імені файлової змінної.

? *Питання до теми уроку*:

IV.1 Чи можна вважати, що лінійна програма, яка містить *n* команд має складність *n*?
IV.2 Чи можна порівняти доведення алгоритму із доведенням математичної теореми?
IV.3 Для чого потрібно виконувати оцінку складності алгоритмів і програм?
IV.4 Чому не можна помилки виконання вважати логічними?
IV.5 Яка, на ваш погляд, різниця між програмою і програмним кодом?
IV.6 Чи можна вважати програмою програмний код із синтаксичною помилкою?
IV.7 У програмі *Summa* при описі величин замість "*var n:integer; Sn:real;*" записано "*var n, Sn:integer;*". При виконанні програми виникла аварійна зупинка із повідомленням "Помилка: Спроба надати змінній типу *integer* вираз типу *real*". Яка помилка має місце, логічна, синтаксична чи помилка виконання?
IV.8 Яка помилка допущена у програмі *Summa*, якщо замість "*var n:integer; Sn:real;*" написати "*var n:integer; S:real;*"?
IV.9 Яка помилка допущена в записі "*Write('розв'язків немає');*"?

Завдання IV.1 Оцінити складність алгоритмів завдань I.1-I.3.

Завдання IV.2 Довести та оцінити складність алгоритма розв'язування квадратного рівняння.

Завдання IV.3 Порівняти складність алгоритмів *Сума*[1] та *Сума*[2], (приклад I.2), складених відповідно моделей $S_n = 1+ 2+ 3 + \ldots + (n-1) + n$, та $S_n = (1+ n)\cdot n/2$.

Завдання IV.4 Виконати алгоритм *Сума*[2] для $n = 10$, заповнивши результати в табличку:

n	10	10	…	10
i	1	2	…	11
$i \leqslant n$	*true*		…	*false*
$S_i\,(S_0=0)$	0+1=1	1+2=3	…	?

II.2 Структуровані алгоритми та їх програмна реалізація

Урок V. Діалогові програми. Структури повторення та їх графічні схеми.

У наведених вище прикладах програм "втручатись" у процес їх роботи можна лише під час вводу даних. Але, таке "втручання" реалізує найпростіший діалог між користувачем і виконавцем програми[20].

ⓘ *Діалоговими* називаються програми, які дають користувачеві можливості впливати на хід їх виконання, тобто працювати у режимі "діалогу" з транслятором.

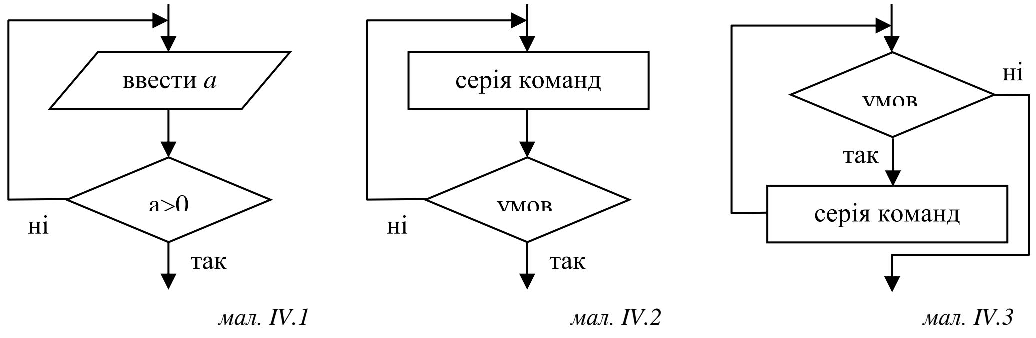
Приклад V.1 Розглянемо програму *Rectangular*, яка за даними довжиною *a* та шириною *b* визначає периметр *p* і площу *s* прямокутника (приклад I.1):

```
program Rectangular;
  uses Crt;
  var a,b,p,s:real;
begin
  WriteLn('a?');ReadLn(a);{1}
  WriteLn('b?');ReadLn(b);{2}
  p:=2*(a+b);s:=a*b;
  WriteLn('p=',p:4:2,'    s=',s:4:2)
end.
```

Через наявність команд {1}-{2} цю програму можна віднести до діалогових. Але це найпростіший, тобто лінійний, приклад діалогу. Розглянемо складніші варіанти діалогу в програмах.

При введенні не коректних значень *a* чи *b* (a, $b \leq 0$) вона виведе на екран не вірні значення результатів *p* або *s*, наприклад: при $a = -3.4$, $b = 7$ $p = 7.20$; $s =$ $= -23.80$.

Виникла проблема: *як забезпечити коректність вводу даних*? Щоб досягти цього, вивчимо *команду повторення з післяумовою*, яку інакше називають *циклом з післяумовою*, для наочності зобразивши її у вигляді блок-



мал. IV.1 *мал. IV.2* *мал. IV.3*

схеми.

Блок-схема (мал. IV.1) складається з двох блоків, один з яких, зображений у вигляді ромба тут зустрівся вперше. Так позначаються *умови* у *складених командах*.

[20] Виконавцем програми часто називають комп'ютер, але це не зовсім точно, адже програму виконує *система програмування*, тобто *транслятор*. Далі будемо мати на увазі компілятор *Turbo Pascal* або інтерпретатор *PascalABC*, чи скорочено - *Pascal*.

ⓘ *Умовою* називатимемо *твердження*, що входить до складеної команди і, залежно від його істинності здійснюється перехід до наступної команди[21].

Умови, або твердження називають ще *логічними виразами*, вважаючи, що вони мають можуть мати два значення *true* і *false*.

ⓘ *Складеними* називатимемо команди, що включають інші команди або *серії команд*. Серію команд, що входить у цикл називатимемо *тілом циклу*.

Істинність будь-якого твердження можна перевірити, одержавши при цьому відповідь про його істинність (*так* або *ні*).

Отже, після вводу *a* (мал.IV.1) слід перевірити умову *a*>0 і, залежно від її істинності, закінчити команду повторення, або знову ввести *a*. Така команда, як уже було сказано, називається командою повторення з післяумовою, або *циклом з післяумовою*.

Для забезпечення принципу **Б** корисно виробити в учнів навички *логічного розбору словосполучень та речень*, який складається з етапів:

1) розклад словосполучення або речення на *первинні* та *похідні* (вторинні, третинні і т.д.) терміни чи поняття;
2) послідовне визначення змісту термінів, первинних, вторинних і т.д., поки не буде визначено зміст вихідного словосполучення чи речення.

Для прикладу візьмемо словосполучення ***команда повторення з післяумовою***. Воно складається з первинних термінів:

- *команда;*
- *повторення;*
- *умова;*
- *післяумова*,

та вторинних (похідних) термінів:

- *команда повторення;*
- *команда повторення з післяумовою.*

Учні вже ознайомлені з поняттями команда, повторення та умова. Слід досягти розуміння, що післяумова – це умова, яка перевіряється після виконання деякої дії. Потім можна легко сформувати попереднє уявлення про *команду повторення*, як таку, що забезпечує *повторення деякого процесу певну кількість разів*. Нарешті можна сформувати уявлення про команду повторення з післяумовою, тобто таку, у якій після виконання деякого процесу (команди чи серії команд) слід перевірити на істинність поставлену умову, якщо умова істинна, то вважати команду повторення з післяумовою виконаною, інакше – знову, виконавши процес, перевірити умову на істинність.

Надалі будемо користуватись логічними розборами для формування понять, формулювання правил тощо.

ⓘ *Цикл з післяумовою виконується так*:

Виконується тіло цикла і перевіряється умова. Якщо вона істинна, то цикл з післяумовою вважається виконаним, інакше – знову виконується тіло цикла і перевіряється умова.

На мал. IV.2 зображено узагальнену команду повторення з післяумовою. Блок-схемою команди зручно користуватись для формулювання правила її

[21] Про простіші типи переходів між командами мовилось вище (див. зноску 12). Тут додамо, що умови вказують на перехід до наступної команди, або на перехід всередині команди, тобто до блоку, який може називатись “серія команд № *N*“ , “тіло цикла“ тощо.

виконання. Команди повторення дозволяють застосовувати в програмах крім структури слідування, що вивчались в уроці III, *структури повторення*. Крім циклу з післяумовою існують також *цикли з передумовою* (мал. IV.3).

ⓘ *Цикл з передумовою виконується так*:

Якщо істинна умова, виконується тіло цикла і знову перевіряється умова.

Якщо умова не істинна, то цикл з передумовою вважається виконаним.

! Не слід плутати, що *команди - виконуються*, а *умови - перевіряються*. Не можна формулювати правило виконання циклу з післяумовою так: *Виконується тіло цикла і перевіряється умова. Якщо вона не істинна, то знову виконується тіло цикла і перевіряється умова*, а правило виконання циклу з передумовою так: *Перевіряється умова. Якщо вона істинна, то виконується тіло цикла і знову перевіряється умова*. У наведених зразках не вказано обов'язковий у всіх командах перехід до наступної команди, тобто вихід з циклу.

Загальний запис (*формат*) розглянутих команд повторення:

Цикл з післяумовою:	*Цикл з передумовою:*
repeat *тіло цикла* ***until*** умова*;*	***while*** умова ***[begin]*** тіло цикла ***[end]****;* {якщо тіло цикла складається з однієї команди, то слова *begin* та *end* - не обов'язкові}

Для забезпечення коректності вводу даних програму ***Rectangular*** слід записати, представивши ввод довжини і ширини прямокутника, як тіло циклу з післяумовою:

Приклад V.2

```
program Rectangular1;
  uses Crt;
  var a,b,p,s:real;
begin
  repeat                                        {1}
    WriteLn('a?');ReadLn(a)                     {1}
  until a>0;                                    {1}
  repeat  WriteLn('b?');ReadLn(b) until b>0;    {2}
  p:=2*(a+b);s:=a*b;
  WriteLn('p=',p:4:2,'    s=',s:4:2)
end.
```

Звернемо увагу на запис аналогічних за змістом команд {1} та {2}. Якщо не виникає сумніву у результаті команди, її слід записувати якомога коротше, це відповідає (ПДС).

? *Питання до теми уроку*:

V.1 Які команди використовуються для лінійного та циклічного діалогу?
V.2 Який вид діалогу містить програма *Rectangular*?
V.3 Який вид діалогу містить програма *Rectangular1*?
V.4 Чи можна назвати програму *Summa* (урок IV) діалоговою? Чому?
V.5 Чи можна назвати програму *Summa1* (урок IV) діалоговою? Чому?
V.6 Як уникнути діалогу в програмі діалогу в програмі *Rectangular*?
V.7 Чи завжди можна замінити цикл *repeat…until* циклом *while…do*? Навести приклади.
V.8 Чи завжди можна замінити цикл *while…do* циклом *repeat…until*? Навести приклади.
V.9 Чи є помилка в наведеному записі команди повторення з післяумовою?

repeat
begin тіло цикла end
until умова*;*

✍ *Завдання V.1* Доповнити програму *Acquaintance циклічним діалогом.*

Завдання V.2 Замінити цикл *repeat WriteLn('a?');ReadLn(a) until a<>0;* циклом з передумовою.

*Завдання V.3** Замінити у програмі *Summa* команду *Sn:=(1+n)*n/2;* циклом *repeat…until.*

*Завдання V.3** Замінити у програмі *Summa* команду *Sn:=(1+n)*n/2;* циклом *while…do*?

Урок VI. Структури розгалуження та їх графічні схеми.

В уроках №3 і №5 розглядались *структури слідування* та *повторення*. Крім них у алгоритмах і програмах часто зустрічаються *структури розгалуження*. Разом їх називають *базовими структурами алгоритмів*. Діалогові програми здебільшого містять як лінійні, так і циклічні та розгалужені блоки.

Розглянемо програму розв'язування квадратного рівняння (алгоритм - завдання II.2). Пригадаємо, що квадратними називаються рівняння виду $ax^2+bx+c=0$, де a, b, c – числа, $a\neq 0$, x – невідоме.

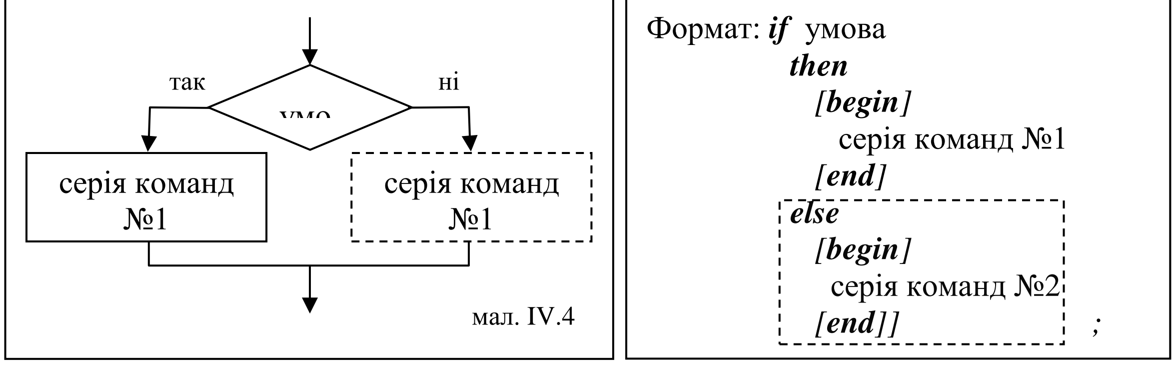
Приклад VI.1

```
program Sqr_Equat;
  var a,b,c,d,x1,x2:real;
begin
  repeat WriteLn('a?');ReadLn(a) until a<>0;          {1}

  WriteLn('b?');ReadLn(b);
  WriteLn('c?');ReadLn(c);
  d:=Sqr(b)-4*a*c;

  if d<0                                              {2}
   then WriteLn('розв"язків немає')                   {3}
   else begin                                         {4}
          x1:=(-b-Sqrt(d))/(2*a);                     {5}
          x2:=(-b+Sqrt(d))/(2*a);                     {6}
          WriteLn('x1=', x1:6:2, '; x2=', x2:6:2)     {7}
        end;                                          {8}
end.
```

Звернемо увагу на позначені рядки {1}-{8}. Перший свідчить, що з допомогою циклу *repeat…until* забезпечено коректність вводу *a*. Цикл, зважаючи на ПДС, записано в один рядок для зменшення кількості рядків програми.



мал. IV.4

Рядки {2}-{8} містять *команду розгалуження*. Щоб зрозуміти її призначення, розглянемо в програмі *Sqr_Equat* загальний вигляд команди розгалуження (див схему вище).

ⓘ *Команда розгалуження виконується так*:

> Якщо істинна умова, виконується серія команд №1 *і команда розгалуження вважається виконаною*. Якщо умова не істинна, то то виконується серія команд №2 *і команда розгалуження вважається виконаною*.

! Виділене курсивом – обов'язкове. Так вказується перехід до наступної команди. Важливо помітити, що після перевірки умови, залежно від її істинності, виконується лише одна з двох серій команд.

Якщо відсутня серія команд №2, команда розгалуження має скорочену форму. У блок-схемі і загальному записі на це вказано пунктирною рамкою.

Слід звернути увагу на операторні дужки *[begin]* та *[end]*. Вони обов'язкові, коли серія команд містить більше, ніж одну команду. У програмі *Sqr_Equat* серія команд №1 складається з однієї команди, тому операторні дужки відсутні, а серія команд №2 містить три команди, тому вони включені в операторні дужки.

Як згадувалось вище, команду розгалуження, можна використати для організації діалогу, наприклад:

Приклад VI.2

```
program Acquaintance;
  var name,answer:string;                          {1}
begin
  WriteLn('Я комп"ютер. А як звати тебе?');
  ReadLn(name);                                     {2}
  WriteLn(name,', чи хочеш зі мною дружити.');      {3}
  ReadLn(answer);
  if answer='ні'                                    {4}
    then WriteLn(name,', прощавай!')                {4}
   else WriteLn(name,', до побачення.');            {4}
end.
```

У програмі використано тип величин – *string*, або рядковий чи текстовий. Його слід розглядати окремо. Тут ми обмежимось тим, що величини *name* та *answer*, оголошені в рядку *{1}*, на відміну від числових величин зберігають набір символів, але не більше 255, тому займають у пам'яті по 256 байт, бо кожен символ займає 1 байт пам'яті. До "текстових" типів також можна віднести символьний тип *char*, кожен елемент такого типу містить один символ і займає 1 байт пам'яті. Якщо величину оголосити так: *answer:string*[10], вона займатиме 10 байт пам'яті. Але *a*:*char* ≠ *b*:*string*[1], бо *a* відноситься до складених, а *b* – до порядкових типів (див. таблицю III.3).

Команда *{2}* допомагає ввести з клавіатури у вигляді набору символів деяке ім'я, а команда *{3}* виводить його разом з текстом "пропоную дружити. ", створюючи ілюзію справжнього діалогу з комп'ютером. Команди блоку {4} ілюструють розгалужений діалог.

До структур розгалуження належить команда вибору (мал. IV.5). Її реалізація у мові *Pascal* має деякі особливості, які слід розглянути окремо. Формат:

```
case перемикач of               {перемикач – це вираз порядкового типу}
   значення1:серія команд 1;  {значення – це також вираз порядкового типу}
```

```
   значення2:серія команд 2;  {значення – це також вираз порядкового
типу}
        .  .  .
   значення N:серія команд N;
  [else серія команд № N+1]
end;
```

ⓘ *Команда вибору виконується так*:

якщо істинна *умова1*:***перемикач=значення*1**, виконується серія команд 1 і *команда вибору вважається виконаною*;

інакше, якщо істинна *умова2*:***перемикач=значення*2**, виконується серія команд 2 і *команда вибору вважається виконаною*;

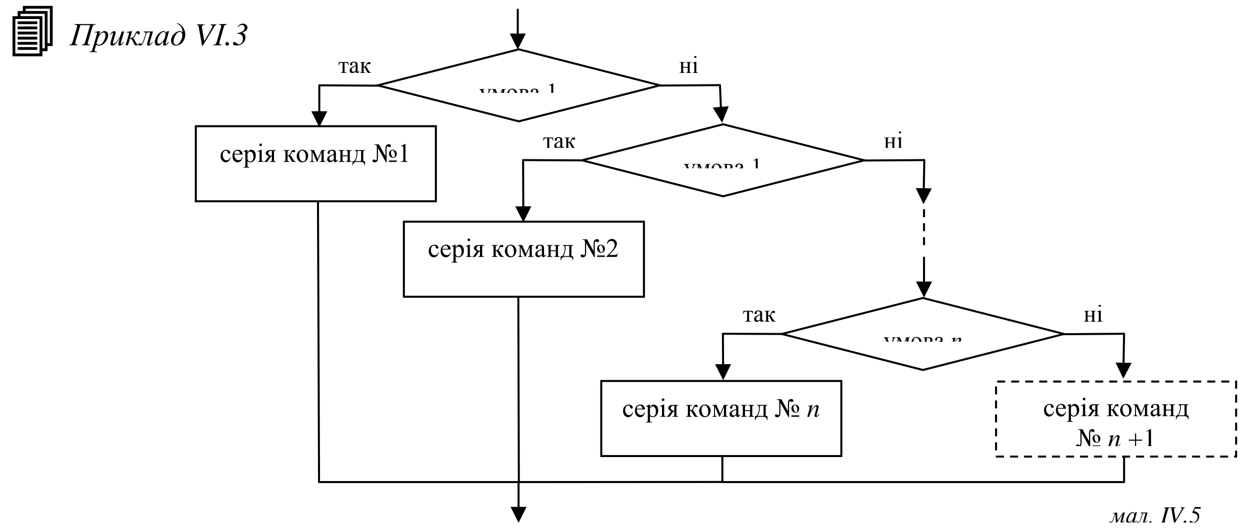
. . .

інакше, якщо істинна *умоваN*:***перемикач=значенняN***, виконується серія команд *N* і *команда вибору вважається виконаною*;

інакше, виконується серія команд *N+1* і *команда вибору вважається виконаною*.

Команда розгалуження, як і команди повторення, обов'язково містять

Приклад VI.3



мал. IV.5

вкладені команди (серії команд). Вкладеними можуть бути і складені команди. Розглянемо це на прикладі програми *Sqr_Equat_1*.

Блок {2} – це команда розгалуження, що утворює серію команд 2 команди розгалуження {1}. Вона в свою чергу містить серії команд {3} та {4}.

Нижче наведено програму *Quantity_Decision*, що визначає кількість коренів квадратного рівняння:

```
program Quantity_Decision;
  var k:integer;
begin
  repeat WriteLn('a?');ReadLn(a) until a<>0;
  WriteLn('b?');ReadLn(b); WriteLn('c?');ReadLn(c);
  d:=Sqr(b)–4*a*c;
  if d<0
    then k:=0                          {1}
    else if d=0  then k:=1             {2}
                 else k:=2             {3}
  WriteLn('равняння має ',k,' розв"язків')
end.
```

Тут коментарі {1}-{3} вказують місця, де визначається кількість розв'язків, причому {2}-{3} також містить вкладену команду розгалуження.

❓ *Питання до теми уроку*:

VI.1 Які команди використовуються організації діалогу?

VI.2 Які види діалогу містить програма *Acquaintance*?

VI.3 Який вид діалогу містить програма *Sqr_Equat*?

VI.4 Чи можна вважати рівноцінними наведені фрагменти програм?

фрагмент 1	*фрагмент 2*
if answer='ні'	*if answer='ні'*
then WriteLn(name,', прощавай!')	*then WriteLn(name,', прощавай!');*
else WriteLn(name,', до побачення.');	*if answer='так'*
	then WriteLn(name,', до побачення.');

VI.5 Чи можна записати повну форму команди розгалуження з допомогою команд(и) розгалуження у скороченій формі?

VI.6 Чи є помилка в наведеному записі команди розгалуження?

```
if answer='ні'
  then WriteLn(name,', прощаввай!')
 else WriteLn(name,', до побачення.');
```

VI.7 Чи є помилка в наведеному записі команди розгалуження?

```
if answer='ні'
  then WriteLn(name,', прощавай!');
 else WriteLn(name,', до побачення.');
```

VI.8 Чи можна записати в програмі *Quantity_Decision* команду розгалуження у вигляді команди вибору?

VI.9 Чи можна записати команду вибору з допомогою команд(и) розгалуження?

✍ *Завдання VI.1* Доповнити програму *Acquaintance розгалуженим діалогом.*

Завдання VI.2 Наведений фрагменті програми *Sqr_Equat* :

```
if d<0 then WriteLn('розв"язків немає')
else begin x1:=(-b-Sqrt(d))/(2*a);x2:=(-b+Sqrt(d))/(2*a);
           WriteLn('x1=', x1:6:2, '; x2=', x2:6:2) end;
```

не забезпечує виведення на екран одного розв'язку, коли *d=0*. Виправити цей недолік з допомогою додаткової команди розгалуження.

*Завдання VI.3** Змінити програму *Sqr_equat_1* з метою повного уникнення діалогу.

Урок VII. Конструювання алгоритмів методами покрокової деталізації. Підпрограми і вказівки їх виклику. Формальні та фактичні параметри.

У попередньому уроці розглядались основні логічні структурні елементи в алгоритмах та програмах – *слідування*, *розгалуження* і *цикли*. Здебільшого в задачах, призначених для розв'язування з допомогою програмування, через їх складність названі структурні елементи використовуються в комплексі, від чого самі програми стають великими, громіздкими, важкими для розуміння та відлагодження. Тому у програмуванні використовується *принцип покрокової деталізації*, що полягає у розбитті задачі на допоміжні задачі (*підзадачі*), складання програм розв'язування підзадач, які називаються *підпрограмами*, та їх використання в основній програмі.

Існують методи деталізації "*згори донизу*" і "*знизу догори*". Для ілюстрації методу деталізації "згори донизу" звернемось до програми *Sqr_Equat* (приклад VI.1 з уроку VI). У ній, використані підпрограми *Sqr*(*b*), яка визначає квадрат

величини *b*, та *Sqrt*(*d*), яка обчислює арифметичний квадратний корінь величини *d*. При цьому мається на увазі, що вони працюють правильно.

Поряд із процедурами вводу та виводу *Sqr* та *Sqrt* належать до *стандартних підпрограм*, які створені розробниками компілятора *Pascal*. Уважний читач міг помітити, що деякі з них називаються стандартними *процедурами* (*Write*, *Read*), інші – стандартними *функціями* (*Sqr*(*a*), *Sqrt*(*a*), *Abs*(*a*) тощо – див. таблицю II.1 уроку 2). Але слід усвідомлювати, що процедури та функції – це форми запису підпрограм.

ⓘ *Процедура* – це спеціальна форма запису програми, призначена для виклику іншою програмою.

ⓘ *Функція* – це *окремий вид процедури*, призначений для виклику у якості параметра.

Мова програмування надає користувачеві, відповідно до потреб, можливість створювати власні підпрограми – *функції та процедури користувача*, перевіряти їх правильність і використання в основній програмі. В цьому полягає метод покрокової деталізації "знизу догори".

Застосуємо цей метод для складання програми *Bi_Sqr_Equat* для розв'язування біквадратного рівняння. Нагадаємо, що біквадратним називається рівняння виду $ax^4+bx^2+c=0$, де $a\neq 0$. Пригадаємо з курсу алгебри, що для розв'язування біквадратних рівнянь застосовують метод заміни змінної, поклавши $x^2 = y$ та *застосувавши, як допоміжну, задачу розв'язування квадратного рівняння* $ay^2+by+c=0$.

Щоб скористатись раніше описаною програмою *Sqr_equat*, яка визначає розв'язки квадратного рівняння, залишається оформити її у вигляді підпрограми та забезпечити виклик в основній програмі. Програма розв'язування біквадратних рівнянь *Bi_Sqr_Equat* має такий вигляд:

Приклад VII.1

```
program Bi_Sqr_Equat;
 var a,b,c,d,y1,y2:real;f:boolean;
 procedure Sqr_Equat;           {1}
 begin
  d:=Sqr(b)-4*a*c;
  if d<0
   then f:=false               {2}
   else                      begin
    f:=true;                   {3}
    y1:=(-b–Sqrt(d))/(2*a);
    y2:=(-b+Sqrt(d))/(2*a) end[22];
 end;
begin
 repeat WriteLn('a?');ReadLn(a) until a<>0;
 WriteLn('b:');ReadLn(b);WriteLn('c:');ReadLn(c);
 Sqr_Equat;                                        {5}
 if f='y' then                               begin {6}
  if y1>=0 then
```

[22] Тут і нижче для підвищення читабельності програмного коду однаковим підкресленням будемо виділяти відповідні пари "*begin*" та "*end*".

```
    WriteLn('x1=',-Sqrt(y1):6:2,'; x2=',Sqrt(y1):6:2);
  if y2>=0 then
    WriteLn('x3=',-Sqrt(y2):6:2,'; x4=',Sqrt(y2):6:2)end;
 if ((y1<0)and(y2<0))or(f=false) then                    {7}
   WriteLn('розв“язків нема')
end.
```

Проаналізувавши рядки {1} та {5} можна зробити висновок, що *команда виклику* процедури[23] *Sqr_Equat* записується аналогічно її назві в заголовку.

! Тут ми вперше застосували величину *f* логічного типу, який згадувався вище (урок V). Її призначення, подібно "прапорцеві" – фіксувати існування розв'язків квадратного рівнянння, що необхідно для подальшого аналізу і визначення коренів біквадратного рівняння. У зв'язку з тим, що *f* набуває двох протилежних значень, її можна вважати своєрідною "радіокнопкою". Прийом "прапорця" чи "радіокнопки" часто використовується у програмах.

Вперше тут використано і складену умову "*((**y1<0**)**and**(**y2<0**))**or**(**f=false**)*".

ⓘ <u>*Умова називається складеною*</u>, якщо вона складається з однієї або кількох простих умов, об'єднаних з допомогою сполучників *not*, *and*, *or*.

Наприклад, позначивши ***A***="*y1*<0", ***B***="*y2*<0", ***C***="f=false" одержимо складену умову (*A**and**B*)***or**C*. Значення складених логічних виразів визначаються за схемами:

Приклад VII.2 — *таблиця VII.1*

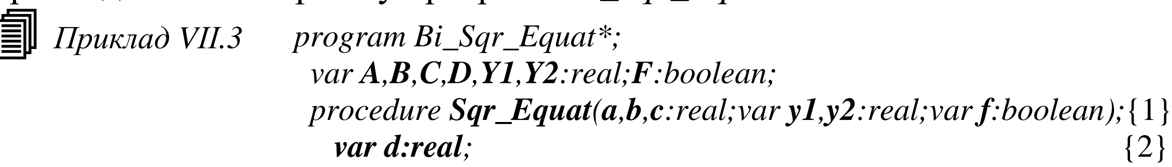
A	*B*	*A **and** B*	*A **or** B*	***not** A*
true	*true*	*true*	*true*	*false*
true	*false*	*false*	*true*	
false	*true*	*false*	*true*	*true*
false	*false*	*false*	*false*	

Згідно таблиці VII.1 логічний вираз має значення:

таблиця VII.2

(*A **and** B*)	*C*	*(A**and**B)**or**C*
true	*true*	*true*
false	*true*	*true*
true	*false*	*true*
false	*false*	*false*

Можна також помітити, що в *Sqr_Equat*: *а*) не оголошуються власні величини; *б*) відсутні блоки вводу/виводу. Пункт *а*) допустимий, бо розглядаючи підпрограми, як вкладені блоки основної програми, слід вважати усі оголошені й описані в основній програмі величини *глобальними*, тобто доступними для підпрограм основної програми. Зауважимо, що коли немає потреби вводити *локальні* величини (тобто оголошені, описані і доступні тільки всередині підпрограми), їх вводити, згідно ПДС, не слід. Але найчастіше задача вимагає *обміну* значеннями величин між основною програмою та її підпрограмами, тому в *Pascal* існують необхідні для цього засоби і в загальному випадку ПДС вимагають опису і виклику *підпрограм з параметрами*[24]. Розглянемо це на прикладі іншого варіанту програми *Bi_Sqr_Equat*:



Приклад VII.3

```
program Bi_Sqr_Equat*;
 var A,B,C,D,Y1,Y2:real;F:boolean;
 procedure Sqr_Equat(a,b,c:real;var y1,y2:real;var f:boolean);{1}
   var d:real;                                                   {2}
```

[23] Особливості та відмінності процедур і функції розглянемо нижче.

[24] Параметрами називаються *аргументи* та *результати* підпрограми.

```
    begin
      d:=Sqr(b)-4*a*c;
      if d<0 then f:=false
      else                        begin
       f:=true;
        y1:=(-b-Sqrt(d))/(2*a);
        y2:=(-b+Sqrt(d))/(2*a)end;
    end;
  begin
    repeat WriteLn('a:');ReadLn(A) until A<>0;
    WriteLn('b:');ReadLn(B);WriteLn('c:');ReadLn(C);
    Sqr_Equat(A,B,C,Y1,Y2,F);                              {3}
    if F=true then                              begin
      if Y1>=0 then
        WriteLn('x1=',-Sqrt(Y1):6:2,'; x2=',Sqrt(Y1):6:2);
      if Y2>=0 then
        WriteLn('x3=',-Sqrt(Y2):6:2,'; x4=',Sqrt(Y2):6:2)end;
    if ((Y1<0)and(Y2<0))or(F=false) then
      WriteLn('розв"язків нема')
  end.
```

Як і в попередньому прикладі, порівняємо заголовок процедури ***Sqr_Equat*** {1} із командою її виклику {3}. Відмітимо наявність *параметрів* у дужках.

Перш за все слід знати, що вони поділяються на *параметри-значення* і *параметри-змінні*. Значення перших процедура не може змінювати, тобто – це константи, значення інших (перед якими записується слово *var*) доступні для зміни. Іншими словами, для підпрограми *параметри-значення* – це аргументи, а *параметри-змінні* – це результати.

Необхідно також розуміти, що параматри, описані у заголовку підпрограми, умовні, або *формальні*, а відповідні їм параметри в команді виклику підпрограми – *фактичні*. Команду виклику підпрограми можна застосовувати багаторазово, при цьому щоразу замість формальних параметрів вставляючи *фак-тичні на даному етапі*. У програмі *Bi_Sqr_Equat** для простоти сприйняття формальні та відповідні їм фактичні параметри записано одними літерами, але у різних регістрах[25].

! Дотримуючись принципу покрокової деталізації, праклично кожну програму можна застосувати у якості підпрограми. Очевидно також, що кожна програма представляється у вигляді процедури.

Для ілюстрації скористаємось прикладом VI.1.

Приклад VII.4

```
procedure Quantity_Decision(a,b,c:real;var k:real);
   var d:integer;
 begin
   d:=Sqr(b)–4*a*c;
   if d<0  then k:=0
           else if d=0  then k:=1
```

[25] *Pascal* не розрізняє регістрів, але все ж, у подібних випадках, компілятор величинам зразка *A* та *a*, одна з яких є глобальною, а інша локальною, відводить у пам'яті різні адреси. Та, керуючись ПДС, слід уникати однакового позначення величин у різних блоках програми.

```
                  else k:=2
end.
```

З прикладу VII.4 видно, що при записі програми у вигляді процедури, слід виключити з неї команди вводу/виводу і після назви процедури в дужках оголосити та описати параметри. Вона стане коротшою і простішою для читання (порівняйте приклади прикладом VI.1 та прикладом VII.4).

Процедуру *Quantity_Decision* можна записати й у вигляді *функції*, що, як буде видно далі, дасть додаткові переваги.

У зв'язку з цим зауважимо, що термін ”функція” вибрано зовсім не випадково, адже тут встановлюється аналогія з поняттям математичної функції.

Приклад VII.5

```
function Quantity_Decision(a,b,c:real):real;
  var d:integer;
begin
  d:=Sqr(b)–4*a*c;
  if d<0  then Quantity_Decision:=0
          else if d=0  then Quantity_Decision:=1
                  else Quantity_Decision:=2
end.
```

Із запису функції вилучено параметр k, а назва результату тепер співпадає з назвою функції, тому запис “***Quantity_Decision****(a,b,c:real):real;*” одночасно є оголошенням і описом результату. Зрозуміло, що тоді в тілі підпрограми *замість величини k* використовується *величина Quantity_Decision*. Найпростіший виклик будь-якої підпрограми-функції[26] користувача аналогічний виклику стандартної функції (див. у прикладі VI.1, виклик $Sqr(b)$ та $Sqrt(d)$).

Існує особливий вид функцій – *рекурсивні*. Вони містять *команду виклику самих себе*. Наведемо приклади:

Приклад VII.6

```
function Fact(k:integer):integer;
begin
  if (k=0) or (k=1)
    then Fact:=1
    else Fact:=Fact(k-1)*k;       {1}
end;
```

Приклад VII.7

```
function Fib(k:integer):integer;
begin
  if (k=1) or (k=2)
    then Fib:=1
    else Fib:=Fib(k-1)+Fib(k-2);  {2}
end;
```

У прикладі VII.6 функція *Fact* визначає число $k!$ (добуток k перших натуральних чисел). У рядку {1} виділено *Fact(k-1)*, що є викликом цієї функції при значенні зменшеного на 1 значення параметра. У основі рекурсивної функції завжди лежить рекурентна формула, для цього випадку:

$$n! = \begin{cases} 1 \text{ при } n = 0 \text{ або } n = 1, \\ (n-1)! \cdot n \text{ при } n > 1. \end{cases}$$

У прикладі VII.7 функція *Fib* обчислює k-й елемент послідовності Фібоначчі, що обчислюється за рекурентною формулою:

$$Fib(k) = \begin{cases} 1 \text{ при } k = 1 \text{ або } k = 2, \\ Fib(k-1) + Fib(k-2) \text{ при } k > 1. \end{cases}$$

Питання до теми уроку:

---

[26] Нижче розглянемо складніші варіанти виклику підпрограм-функцій.

VII.1 Чим відрізняються методи покрокової деталізації алгоритмів “згори донизу“ і “знизу догори“?
VII.2 Чи може бути підпрограма також підпрограмою?
VII.3 Чи можна процедурою викликати функцію?
VII.4 Чи можна функцією викликати процедуру?
VII.5 Чи можна функцією викликати процедуру?
VII.6 Чи можна у команді виклику процедури записати команду виклику функції?
VII.7 Чи можна у команді виклику функції записати команду виклику функції?
VII.8 Чи можна у команді виклику функції записати команду виклику процедури?
VII.9 Чи можна у процедурі записати команду виклику її самої?
VII.10 Чи можна у функції записати команду виклику її самої?
VII.11 Чи можна у команді виклику процедури записати команду виклику її самої?
VII.12 Чи можна у команді виклику функції записати команду виклику її самої?
VII.13 Чи може бути команда виклику процедури параметром?
VII.14 Чи може бути команда виклику підпрограми формальним параметром?
VII.15 Чи може бути команда виклику функції фактичним параметром?

Завдання VII.1 Записати програму *Summa* (схема III.1) у вигляді процедури без параметрів.
Завдання VII.2 Записати програму *Summa* у вигляді процедури з параметрами.
Завдання VII.3 Записати програму *Summa* у вигляді функції без параметрів.
Завдання VII.4 Записати програму *Summa* у вигляді функції з параметріами.
Завдання VII.3 Записати програму *Summa* у вигляді функції без параметрів.
Завдання VII.5 Написати програму, яка викликає процедуру *Summa*.
Завдання VII.5 Написати програму, яка викликає функцію *Summa*.

Урок VIII. Структуровані алгоритми.

Ми вже розглянули три основні базові структури алгоритмів: *слідування, розгалуження* та *повторення*. Можна узагальнити: структурними одиницями програми є *дані* та *команди*. Отже, до структурних елементів алгоритмів належать також *команди виклику підпрограм*. Якщо кожен структурний елемент програми вважати окремим блоком, то по відношенню до блоків кожна програма буде лінійною.

ⓘ *Структурованими* називатимемо алгоритми, що містять кілька базових структур, поєднаних між собою.

Перед розглядом таких алгоритмів доповнимо структуру повторення *циклом з параметром*.

Приклад VIII.2 Написати програму, яка для введеного натурального числа *n*, замість формули $S_n=(1+n)\cdot n/2$ використавши цикл *for ... to* визначає значення S_n.

```
program Summa_For_To;
  var i,n,s:integer;
begin
  Read(n); s:=0;
  for i:=1 to n do s:=s+i;
  Write(s);
end.
```

Приклад VIII.1

Програму можна вважати *базовою* для багатьох задач, де потрібно визначати суму відомої кількості деяким чином заданих чисел. Її *суть зводиться до надання початкового значення суми та циклічного додавання до попереднього її значення наступного доданка*. Початкове значення суми найчастіше вважають рівним нулю. Але наведену програму природніше віднести не стільки до структурованих, як до *циклічних*, крім того, ній використано новий вид циклу, тому познайомимось із ним детальніше.

<u>Цикл *for … to*</u>

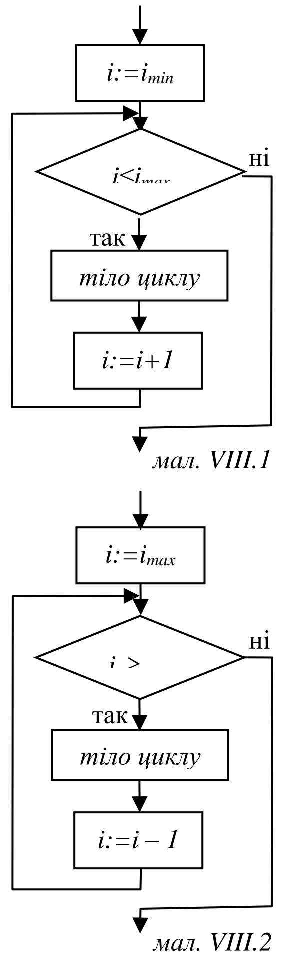
for i**:=i_{min} **to** i_{max} **do
*[**begin**] тіло цикла [**end**];*

i – параметр циклу
i надається значення i_{min} і перевіряється умова "i ⩽ i_{max}". Якщо вона істинна, то виконується тіло циклу, значення параметра збільшується на 1 і знову перевіряється умова. Якщо умова не істинна, цикл вважається виконаним.

<u>Цикл *for … downto*</u>

for i**:=i_{max} **downto** i_{min} **do
*[**begin**] тіло цикла [**end**];*

i – параметр циклу
i надається значення i_{max} і перевіряється умова "i ≥ i_{min}". Якщо вона істинна, то виконується тіло циклу, значення параметра зменшується на 1 і знову перевіряється умова. Якщо умова не істинна, цикл вважається виконаним.



мал. VIII.1
мал. VIII.2

Наступні приклади наочніше ілюструють власне структуровані програми.

Приклад VIII.2 Написати функцію, яка для двох чисел *a* і *b*, визначає їх найбільший спільний дільник.

Реалізуємо відомий алгоритм Евкліда, що полягає у зменшенні більшого з двох чисел на величину меншого до тих пір, поки числа не зрівняються (команди {3}-{4}). В результаті кожне з чисел дорівнюватиме найбільшому спільному дільникові. У зв'язку з тим, що *a* і *b* параметри-значення (подумайте – чому), слід ввести допоміжні величини *a1* та *b1* {1}-{2}.

В умові не випадково вказано, що треба написати функцію, а не процедуру чи програму. З огляду на те, що визначення найбільшого спільного дільника двох чисел може бути підзадачею для багатьох задач, і на те, що передбачається єдиний результат, саме форма функції буде найбільш прийнятною. Найкраще тут застосувати цикл *while…do* (подумайте – чому).

Приклад VIII.3 *function **Nsd**(a,b:integer):integer;*

```
  var a1,b1:integer;              {1}
begin
  a1:=a;b1:=b;                    {2}
  while a1<>b1 do
    if a1>b1 then a1:=a1–b1       {3}
             else b1:=b1–a1;      {4}
  Nsd:=a1
end;
```

Приклад VIII.3[1]

```
function Nsd(a,b:integer):integer;
     var c:integer;
begin
  c:=a mod b;                     {1}
  if c=0 then Nsd:=b
         else Nsd:=Nsd(b,c)       {2}
end;
```

Приклад VIII.3[1] ілюструє рекурсивну реалізацію функції *Nsd* (див. приклади VII.6– VII.7), яка записується коротше, але виконуватись може довше, ніж циклічна, наведена в попередньому прикладі. У прикладі з допомогою {2} помічено рекурсивний виклик функції *Nsd*. Тут є особливість, помічена {1}: команда *c:=a mod b* встановлює порядок параметрів функції, на першому місці повинее бути більше число.

Приклад VIII.4 Написати функцію, яка для визначає, чи буде дане натуральне число *k* простим.

За основу для функції можна взяти алгоритм перевірки числа на “простоту“ за означенням: *натуральне число, більше за 1, називається **простим**, якщо воно не ділиться на жодне натуральне число, крім 1 та самого себе*. Отже, досить перевірити число *k* на кратність числам з діапазону **2 .. (*k-1*)**. Але замість такого діапазону слід взяти діапазон **2 .. [*Sqrt*(*k*)]**, де квадратні дужки означають цілу частину числа ([*x*] – *ант'є від x*). Дійсно, якщо представити число *n* у вигляді двох натуральних множників, то найменший з них дорівнює 2, і тоді, коли *k* – складене, *k* = 2*[*Sqrt*(*k*)]. Вибір такого діапазону для більшості значень *k* суттєво зменшить кількість елементарних опереацій, тобто значно збільшить швидкість виконання програми, наприклад, якщо *k* близьке до 10^6 то цикл буде виконуватись не більше за 10^3 разів, тобто в тисячу разів менше (пригадайте *складність алгоритму*, урок IV)! Тоді функція матиме вигляд:

Приклад VIII.5

```
function PrimeNumder(k:integer):boolean;{1}
  var j,l:integer;b:boolean;           {2}
begin
  b:=true;                             {3}
  j:=2;l:=Trunc(Sqrt(k));              {4}
  while j<=l do begin
    if k mod j=0 then begin            {5}
      b:=false;j:=l    end;            {6}
    j:=j+1        end;
  PrimeNumber:=b
end;
```

Коментарі до програмного коду зайві, адже, враховуючи попередні зауваження, досить пояснити рядки {1}-{6}, звернувши увагу на виділене напівжирним шрифтом. Якщо не все буде зрозумілим, радимо, застосувавши вікно відлагодження *Watch*(див. урок IV), покроково виконати наступну програму, що є своєрідним доповненням до прикладу VIII.3:

Приклад VIII.6

```
program PrimeNumbers;
  var i,m,n:integer;
  {для уникнення дублювання
    опис функції PrimeNumber не наводимо}
begin
  repeat WriteLn('m?');ReadLn(m) until m>1;
  repeat WriteLn('n?');ReadLn(n) until n>=m;
  for i:=m to n do
    if PrimeNumber(i) then Write(i,'; ')
end.
```

Ця програма виводить на екран усі прості числа з діапазону $m..n$.

Питання до теми уроку:

VIII.1 Чи може у циклі *for...to* тіло виконуватись 1 раз? 0 разів?

VIII.2 Чи може у циклі *for…downto* тіло виконуватись 1 раз? 0 разів?

VIII.3 Чи можна цикл *for...to*(*downto*) замінити циклом *while...do*?

VIII.4 Чи можна цикл *while…do* замінити циклом *for...to*(*downto*)?

VIII.5 У яких випадках краще користуватись циклом *repeat…until*?

VIII.6 У яких випадках краще користуватись циклом *while…do*?

VIII.7 У яких випадках краще користуватись циклом *for…to*(*downto*)?

VIII.8 Скільки разів виконається тіло цикла у наведеному фрагменті програми?

```
s:=0; for i:=m to n do s:=s+1;
```

VIII.9 Скільки разів виконається тіло цикла у наведеному фрагменті програми?

```
s:=0;
for i:=m to n do
  for j:=k to p do
    s:=s+1;?
```

VIII.10 Скільки разів буде виконуватись тіло цикла у програмі *PrimeNumder* (приклад VIII.3) для *n*=557?

VIII.11 Скільки разів буде виконуватись тіло цикла у програмі *PrimeNumder* для *n*=558?

Завдання VIII.1 У програмі *Summa_For_To* (приклад VIII.1) цикл *for...to* замінити циклом *while…do*.

Завдання VIII.2 У програмі *Summa_For_To* цикл *for...to* замінити циклом *for…downto*.

Завдання VIII.3 У програмі *Summa_For_To* цикл *for...to* замінити циклом *repeat…until*.

Завдання VIII.4 Записати функцію *Nsd* (приклад VIII.2) у вигляді процедури.

Завдання VIII.5 Написати програму *Nsd_N*, яка з допомогою функції *Nsd* визначає найбільший спільний дільник трьох натуральних чисел.

*ЗавданняVIII.6** Написати програму *Prime_Numb_Dec*, яка з допомогою функції *Prime_Number* (приклад VIII.3) розкладає дане натуральне число на прості множники, записавши результат у вигляді: $n=a_1*a_2*\ldots*a_k$, де a_1–a_k–прості числа, записані у порядку зростання, наприклад: 2002=2*7*11*13.

*Завдання VIII.7** У програмі *Prime_Numb_Dec*, забезпечити читання аргументу із файлу *Prime_N.arg* і запис результату у файл *Prime_N.res*.

II.3 Структуровані величини та їх опрацювання

Урок IX. Структуровані типи величин. Табличні величини (масиви). Лінійні масиви та алгоритми їх опрацювання.

Програмістові важливо не тільки *формально* знати й уміти застосовувати ті чи інші команди, правила, алгоритми тощо, не менш важливим є вміння розрізняти зв'язки між командами, логічні переходи в алгоритмах, чітко виділяти структурні елементи.

У попередньому уроці названо структурні одиниці програми: *дані* та *команди*. Команди діляться на прості (які не містять у собі інших команд, наприклад, команди присвоєння, вводу/виводу) і складені, або структуровані команди (наприклад, цикли, розгалуження, команди виклику підпрограм). Дані також поділяються на *прості* (порядкові та дробові), *структуровані* (файли, масиви, множини, записи) та *інші* (див. табл. III.3 уроку III). Ми вже згадували файловий та рядковий типи. Нижче детальніше розглянемо масиви.

ⓘ <u>*Масивом*</u> (таблицею) називатимемо *скінченну* ***послідовність*** *однотипних елементів*.

Прикладами масивів можуть бути: пронумерований список прізвищ, упорядкована послідовність перших *n* простих чисел, алфавіт тощо.

Можна дати інші трактування поняття масиву, зокрема таке:

ⓘ <u>*Масивом*</u> називатимемо *скінченну* ***множину*** *однотипних пронумерованих елементів*.

Масив можна зобразити у вигляді таблиці:

таблиця IX.1

i	1	2	3	4	5	6	7	8	9	10
A[i]	7	-9	4	8	12	-56	-9	14	-23	11

! а) Не слід *плутати* масив із множиною, адже у множині елементи не нумеруються, крім того серед них не може бути однакових. У першому означенні термін "послідовність", означає, що елементи пронумеровані і серед них можуть бути однакові. У другому означенні термін "множина" передбачає у якості елементів об'єкти *A[i]*, тобто *A[1]* = 7, *A[5]* = 12. Тут і далі індексами називатимемо номери елементів, узагальнений індекс позначатимемо літерою, наприклад, "*i*", а елементи масиву позначатимемо "*A[i]*", де *A* – назва масиву. Отже, якщо значення *A[i]* та *A[j]* рівні між собою, наприклад, *A[2]* та *A[7]*, то елементи все одно різні, бо відрізняються індексами.

б) Якщо проводити паралелі між поняттями величин у математиці та в програмуванні (див. урок III), то масиви – це *аналог векторних величин*, але тут немає можливості це обґрунтувати.

Залежно від того, скількома числами (*індексами*) нумеруються елементи масиву, він може бути *лінійним*, *прямокутним* або іншим. Прізвища гравців футбольної команди – це лінійний масив, список глядачів у партері театру – це прямокутний масив (кожен має номер ряду та номер місця), список пасажирів поїзда – це тривимірний масив (кожен пасажир має номери вагону, купе та місця).

Масиви, подібно числовим чи іншим типам величин, поділяються на константи і змінні, вони обов'язково повинні бути *оголошені* та *описані*. Опрацювання масивів здійснюється звертанням до окремих елементів з допомогою циклів. На прикладі *базового алгоритму визначення суми*, розглянемо програму, що опрацьовує лінійний цілочисельний масив.

Приклад IX.1

Написати програму, що визначає та виводить на екран суму *n* даних цілих чисел.

Нехай ці числа записані в таблицю (див. табл. IX.1, де *n*=10):

i	1	2	3	4	5	6	7	8	9	10
A[i]	7	-9	4	8	12	-56	0	14	-23	11

Індекси елементів записані у першому рядку, а елементи масиву – у другому..

```
program Summa1;
  const n=10;                    {1}
  var i,s:integer;
      A:array[1..n]of integer;   {2}
begin
 for i:=1 to n do begin          {3}
    WriteLn('A[',i,']:');        {3}
    ReadLn(A[i]) end;            {3}
 s:=0;
 for i:=1 to n do s:=s+A[i];     {5}
 WriteLn('s=',s)
end.
```

У рядку {2} оголошено і описано лінійний масив *A*, *array* – означає табличний тип, у зв'язку з тим, що масив складається з елементів, на його розмірність вказує діапазон [1..*n*], де 1 та *n* – номери початкового і останнього елементів, "*of integer*" – так указано тип елементів масиву.

Рядки {3} призначені для вводу масиву з допомогою цикла. Рядок {5} визначає суму елементів масиву. Зверніть увагу, що у командах *ReadLn(A[i])* та *s:=s+A[i]* здійснюється звертання до елементу масиву з індексом *i*, що міняється циклом.

Постійно дотримуючись ПДС, слід записувати програму якомога коротше. Скоротити можна й програму *Summa2*:

Приклад IX.2

```
program Summa2;
  const n=10;
  var i,s:integer;
      A:array[1..n]of integer;
begin
 s:=0;
 for i:=1 to n do begin
    WriteLn('A[',i,']:');        {*}
    ReadLn(A[i]);                {*}
    s:=s+A[i]   end;             {*}
 WriteLn('s=',s)
end.
```

Зірочкою позначено об'єднані в один цикл команди вводу і додавання елементів масиву.

Часто, в процесі відлагоджування програм з даними, представленими масивом, його корисно представляти у вигляді константи або текстового файлу (див. приклади IX.3-IX.4).

Приклад IX.3

```
program Summa3;
  const n=10;
```

```
        A:array[1..n] of integer=(7,-9,4,8,12,-56,0,14,-23,11);
  var i,s:integer;
begin
 s:=0;
 for i:=1 to n do s:=s+A[i]
 WriteLn('s=',s)
end.
```

У прикладі IX.3 лінійний масив *A* оголошується не в розділі оголошення змінних, а в розділі оголошення констант, але після оголошення типу елементів ставиться знак “=“ і в дужках через кому вказуються усі елементи масиву. Тоді блок вводу у програмі зайвий.

Приклад IX.4

```
program Summa4;
  var i,s:integer; f:text;
      A:array[1..100] of integer;
begin
  Assign(f,'Summa.dat');Reset(f);
  s:=0;i:=1;
  repeat
    Read(f,A[i]);{1}
    s:=s+A[i];   {2}
    i:=i+1
  until EoF(f);
  Close(f);WriteLn('s=',s)
end.
```

У програмі *Summa4* з допомогою {1}-{2} виконується зчитування чисел, записаних у файлі *Summa.dat*, в масив *A* та їх додавання.

Скориставшись базовим алгоритмом визначення суми, наведемо також приклади опрацювання *прямокутних масивів*, застосувавши представлення масиву, як константи та зчитування елементів масиву (приклади IX.5-IX.6):

Приклад IX.5

```
program Summa5;
  const m=3;n=3;
      A:array[1..m,1..n]of integer=((7,-9,4),(8,12,-56),0,14,-23));{!}
  var i,j,s:integer;
begin
  s:=0;
  for i:=1 to m do
    for j:=1 to n do
      s:=s+A[i,j];
  WriteLn('s=',s)
end.
```

Програма аналогічна наведеній у прикладі VIII.3, відмінність лише в розмірності масиву. При його оголошенні {!} у дужках через кому записані елементи, що, на відміну від програми *Summa3*, є не числами, а рядками чисел. У зв’язку з цим опрацювання масиву здійснюється з допомогою циклу, вкладеного в інший цикл.

Приклад IX.6

```
program Summa6;
  var i,j,s:integer;f:text;
      A:array[1..1000,1..1000]of integer;{1}
begin
```

```
    Assign(f,'SummA.dat');Reset(f);
    s:=0;i:=1;
    repeat                                  {2}
      j:=1;                                 {2}
      repeat                                {2}
        Read(f,A[i,j]); s:=s+A[i,j]; j:=j+1 {2}
      until EoLn(f);                        {2}
      i:=i+1                                {2}
    until EoF(f);                           {2}
    Close(f);WriteLn('s=',s)
  end.
```

У програмі *Summa6* прямокутний масив навмисне оголошено з великою кількістю рядків та стовпчиків {1}, це дасть змогу для тестування програми використовувати вхідні файли різної розмірності, яка не перевищує 1000 рядків та стовпчиків. Зчитування прямокутного масиву з файлу найкраще виконувати з допомогою двох циклів *repeat…until* із умовами: в зовнішньому циклі *EoF(f)* (*End of File* - кінець файла), у внутрішньому циклі *EoLn(f)* (*End of Line* – кінець рядка).

Наведені програми цілком доступні для розуміння, але радимо уважно прослідкувати за результатами їх покрокового виконання.

Приклади IX.1– IX.6 приводять до висновку: поняття масиву розширює коло задач, доступних для розв'язування з допомогою програмування. Наприклад, при розгляді задачі визначення суми, на відміну від прикладу I.2 (урок I), де доданки можуть визначатись тільки аналітично (з допомогою формули), застосування масивів дає змогу зазделегідь, у довільному порядку встановити доданки.

? *Питання до теми уроку*:

IX.1 Скільки елементів містить масив *A*[13..27]?
IX.2 Скільки елементів містить масив *A*[13..27,12..100]?
IX.3 Яка узагальнена формула визначення кількості елементів лінійного масиву?
IX.4 Яка узагальнена формула визначення кількості елементів прямокутного масиву?
IX.5 Скільки байт займає масив *A*[1..100,1..100]*of byte* у пам'яті комп'ютера[27]?
IX.6 Скільки байт займає масив *A*[1..100,1..100]*of integer* у пам'яті комп'ютера?
IX.7 Чи можна вважати рівнозначними величини *A*:*array*[1..255]*of char* та *b*:*string* (див. приклад VI.2 уроку VI), тобто, чи буде масивом величина *b*?

Завдання IX.1 Написати програму, яка виводить на екран у вигляді прямокутної таблиці розмірами $m \times n$ (*m* рядків і *n* стовпчиків) *k* введених з клавіатури цілих чисел, заповнюючи вільні місця нулями, якщо $k < m \times n$.

Завдання IX.2 Написати програму, яка введений із файлу *matr.dat* прямокутний масив виводить у файл *matr.res* у вигляді одного рядка; одного ствпчика.

Завдання IX.3 Написати програму, яка виводить у файл *matr.res* *k*-й рядок (*k*-й стовпчик) введеного із файлу *matr.dat* прямокутного масиву.

*Завдання IX.4** Написати функцію, яка обчислює суму елементів *k*-го рядка (стовпчика) введеного із файлу *matr.dat* прямокутного масиву.

[27] У версії *PascalABC*, яку ми використовуємо, кожне значення типу *byte* займає 1 байт пам'яті, а кожне значення типу *integer* займає 4 байти пам'яті.

*Завдання IX.5** Написати функцію, яка визначає, чи буде "магічним квадратом"[28] введений із файла *matr.dat* прямокутний масив натуральних чисел *m×m*.

Урок X. Алгоритми пошуку елементів у таблиці.

Найпоширенішими *базовими задачами* при опрацюванні масивів є *пошук* і *сортування* елементів. Почнемо із задач пошуку, розглянувши ще один базовий алгоритм – *визначення найбільшого (найменшого) елемента лінійного числового масиву*. Візьмемо масив (табл. X.1).

таблиця X.1

i	1	2	3	4	5	6	7	8	9	10
A[i]	7	-9	4	8	12	-56	0	14	12	11

У ньому через малу кількість елементів можна моментально вказати найменше число -56. Але масиви можуть бути дуже великими за кількістю елементів, мати розмірність 2, 3, 4 і т.д., містити не цілі числа, а слова тощо. Тому програмне виявлення найменшого елемента полягає у тому, що масив розглядається, як закритий непрозорий ящик, з якого можна деяким чином брати по одному елементу.

Для пошуку найменшого елемента *min* лінійного числового масиву виділимо підзадачу визначення меншого з двох елементів. Відповідна функція матиме вигляд:

```
function Min2(x,y:integer):integer;
begin
  if y>x then Min2:=y
          else Min2:=x;
end;
```

Спочатку *min* покладемо рівним *A[1]*(див. {1} у програмі пошуку, яка наведена нижче).

```
program Min10;
 uses Crt;
   var i,max:integer;
        A:array[1..10]of integer;
        {опис функції Min2}
begin
   {ввод масиву}
   min:=A[1];                {1}
   for i:=2 to 10 do
     min:=Min2(min,A[i]); {2}
   {вивод результату}
end.
```

Для економії місця опис функції, введення даних і виведення результату не наведено. У рядку {2} величина *min* використовується, як проміжний результат і разом з тим, як один із аргументів у команді виклику функції *Min2*.

У наведеній програмі *характеристичною властивістю* елемента, що шукається, була його мінімальність. Задача пошуку універсальна, якщо змінювати характеристичну властивість, одержимо її різновиди. Наприклад, можна шукати не найменший, а *вказаний* елемент. Якщо організувати

---

[28] Магічним квадратом називають квадратний масив натуральних чисел, у якому суми елементів у кожному рядку, кожному стовпчику і кожній діагоналі однакові.

підрахунок кількості входжень вказаного елемента пошуку, виникне *задача виборки*, тобто *пошуку групи елементів* за вказаною характеристичною властивістю.

Але все це задачі на пошук в *неупорядкованому* лінійному масиві. Якщо брати *упорядкований* лінійний масив, то пошук можна організувати за *методом дихотомії*[29], значно швидшим і ефективнішим. Для його пояснення розглянемо приклад:

Приклад X.1 Задумане натуральне число в діапазоні 1..100. Назвати максимальну кількість спроб для його відгадування, якщо, назвавши число, можна одержувати відповідь "відгадано", "задумане число більше" або "задумане число менше".

Нехай задумане число 33. З малюнка X.1 видно, що при першій спробі відгадування слід назвати число 50=[(1+100)/2]. Одержавши у відповідь: " задумане число менше", слід назвати 25=[(1+50)/2], далі слід називати 37=[(25+50)/2], 31=[(25+37)/2]. Останньою буде відповідь: "задумане число більше". Залишиться назвати 34=[(31+37)/2], 32=[(31+34)/2] та 33=[(32+34)/2].

1 25 31 37 50 100

мал. X.1

Отже число 33 відгадано за 7 спроб. Очевидно – це найбільш оптимальний метод для такого класу задач, адже щоразу відкидалась половина діапазону, для 1..100 "довжина" діапазонів така: 100- 51-26-13-7-4-2-1, тобто остання спроба – це вибір на "нульовому" діапазоні, коли його кінці співпадають. Можна додати, що за 7 спроб завжди можна відгадати число й на проміжку 1..128, адже $128=2^7$. Метод дихотомії, як уже було сказано, дуже ефективний. Наприклад, для діапазону 1..30000000 буде досить 25 спроб, адже 2^{25}=33 554 432. Але, якщо задумане число дорівнює 16777216, то вже після першої спроби його буде відгадано.

! Приклад X.1, як і приклад VIII.5, дуже наочний при оцінюванні алгоритмів, адже вони мають однакову по ефективності властивість: легко встановити "плаваючу" швидкодію програми. Кількість елементарних арифметичних і логічних операцій у прикладі VIII.5 для числа $N \approx 1000000$ належить діапазону 1..1000, а у прикладі X.1 при цьому ж задуманому числі належить діапазону 1.. 19, бо $2^{20}=1048576$.

Приклад X.2 Перевірити, чи належить вказане число даному лінійному числовому масиву.

Візьмемо масив (табл. X.2) і будемо шукати в ньому число 8. Врахувавши упорядкованість даного масиву по зростанню, застосуємо метод дихотомії, але не для елементів, а для індексів, маючи на увазі запитання: "Чи дорівнює число 8 елементові масиву *A[i]*?". Пропонуємо самостійно знайти відповідь та визначити кількість кроків програми, потрібних для її одержання.

i	1	2	3	4	5	6	7	8	9	10
A[i]	7	9	17	19	21	23	38	38	38	39

таблиця X.2

[29] Дихотомія у перекладі означає поділ навпіл, тому цей метод інакше називають методом поділу відрізка пополам.

```
program Search_Dihotomia;
const n=10;
        A:array[1..n]of 1..100=(7,9,17,19,21,23,38,38,38,39);
 var i,a,b,c,x:integer;
      Arr:array[0..n+1]of integer;                              {1}
      res:boolean;
begin
   for i:=1 to n do Arr[i]:=A[i];                                {2}
   WriteLn('яке число шукати?');ReadLn(x);
   a:=0;                                                         {3}
   b:=n+1;                                                       {4}
   res:=false;                                                   {5}
   while b-a>1 do                   begin                        {6}
      c:=Trunc((a+b)/2);                                         {7}
        if Arr[c]=x  then     begin
                            res:=true;                           {8}
                            a:=b end                             {9}
                        else
                           if Arr[c]<x                           {10}
                              then a:=c                          {11}
                              else b:=c  end;                    {12}
   Writeln(res)
end.
```

У наведеній програмі слід звернути особливу увагу на логічний тип величини *res*, розмірність масиву *Arr* {1} та рядки, де опрацьовуються величини *a*, *b* і *c* ({3}-{4}, {6}-{12}). Величина *res* – це зразок "прапорця", що уже використовувався у попередніх уроках (див. приклад VII.1).

Розмірність *допоміжного масиву*, а такі масиви, як і прості величини (див. приклад VIII.3), часто використовуються в програмах, подібно допоміжним змінним у алгебрі при застосуванні методу заміни змінної (див. приклад VII.3), має два додаткові номери 0 та *n*+1. Без цього відповідь про належність крайніх елементів була б *false*. Причина виявляється при одержанні останнього діапазону, визначення ант'є від частки приводить до завершення на діапазоні, рівному 2, або *зациклювання*.

У програмах часто опрацьовуються діапазони, що звужуються. Для цього циклічно обчислюється проміжне значення, що потім воно стає одним із кінців діапазону. Таку роль у наведеному прикладі відіграє величина c, як середина відрізка [*a*, *b*]. Остаточно зрозуміти призначення величин *a*, *b* і *c* допоможе покрокове виконання програми з виводом проміжних результатів у вікно *Watch*.

У багатьох розглянутих вище прикладах застосовано позначення окремих рядків чи команд програми у вигляді коментаря-прапорця, наприклад {1}. Досвід свідчить, що застосування таких позначень з метою акцентування на окремих елементах програм та постановки запитань при їх розборі є досить ефективним дидактичним прийомом, особливо при використанні *принципу безпаперового навчання*, тобто тиражування дидактичних електронних матеріалів через локальну мережу та роботу у режимі інтерактивного мережевого класу з допомогою спеціального програмного забезпечення, зразка ***NetOp Scool***.

? *Питання до теми уроку*:

X.1 Що означає запис *Min2(min,A[i])* у програмі *Min10*?
X.2 Чи можна замінити запис *Min2(min,A[3])* записом *Min2(Min2(A[1],A[2]),A[3])*?
X.3 Чи можна обійтись без методу дихотомії при пошуку в упорядкованому цілочисельному масиві?
X.4 Яка максимальна кількість спроб необхідна при відгадуванні задуманого цілого числа в діапазоні 1000..5000?
X.5 Чи можна відгадати задумане в діапазоні 300..700 число за 8 спроб?
X.6 Чи можна застосувати метод дихотомії для пошуку в упорядкованому масиві, якщо його елементи мають тип *real*?
X.7 Яка дію записана у виразі *Trunc((a+b)/2)*?

Завдання X.1 Змінити програму *Min10*, щоб вона визначала найменше з *n* чисел.
Завдання X.2 Записати програму *Min10* у вигляді функції.
Завдання X.3 Написати функцію, яка визначає найбільший елемент в неупорядкованому числовому масиві.
Завдання X.4 Написати програму, яка визначає кількість елементів лінійного числового масиву, кратних даному числу.
*Завдання X.5** Написати програму, яка з допомогою функції *PrimeNumber* (приклад VIII.5) визначає всі елементи лінійного числового масиву, які є простими числами.
*Завдання X.6** Написати програму, що "ущільнює" лінійний числовий масив, залишивши в ньому лише по одному з однакових елементів.

Урок XI. Алгоритми упорядкування лінійних таблиць.

В попередньому уроці було розглянуто два різновиди пошуку – в *неупорядкованому* та в *упорядкованому* масиві, причому, другий значно ефективніший. Це одна з багатьох причин організації упорядкування, або *сортування* масивів. Сортування розділяють за порядком елементів (по зростанню та по спаданню) і за типом елементів. Якщо масив складається не із числових даних, а, наприклад, із текстових елементів, то тоді можна встановити прямий чи зворотний лексикографічний порядок елементів.

Приклад XI.1 Розглянемо один із найпростіших методів сортування. Візьмемо масив (табл. XI.1), елементи якого розміщені у довільному порядку, тобто *неупорядковані*. Поставимо задачу його *упорядкування по зростанню*, тобто одержання масиву:

таблиця XI.1

i	1	2	3	4	5	6	7	8	9	10
A[i]	7	-9	4	8	12	-56	0	14	-23	11

Тут ми вказали *ключ сортування*, тобто ознаку, за якою воно повинне виконуватись. Умовно поділимо масив на дві частини: зліва – упорядковану, а справа – неупорядковану (*хвіст*). Зрозуміло, що спочатку упорядкована частина не матиме елементів, а хвіст займатиме весь масив. Якщо на кожному кроці збільшувати на один елемент упорядковану частину і при цьому на один елемент зменшувати хвіст, то процес можна показати у вигляді таблиці X.2:

таблиця XI.2

	1	2	3	4	5	6	7	8	9	10
	7	-9	4	8	12	**-56**	0	14	-23	11
крок 1	-56	-9	4	8	12	7	0	14	**-23**	11
крок 2	-56	-23	4	8	12	7	0	14	**-9**	11
крок 3	-56	-23	-9	8	12	7	**0**	14	4	11

крок 4	-56	-23	-9	0	12	7	8	14	**4**	11
крок 5	-56	-23	-9	0	4	**7**	8	14	12	11
крок 6	-56	-23	-9	0	4	7	**8**	14	12	11
крок 7	-56	-23	-9	0	4	7	8	14	12	**11**
крок 8	-56	-23	-9	0	4	7	8	11	**12**	14
крок 9	-56	-23	-9	0	4	7	8	11	12	14

Упорядкована частина в таблиці виділена товстішими рамками. У задачі упорядкування таким чином[30] можна виділити дві підзадачі: *а*) пошук найменшого елемента, який розглядався у попередньому уроці; *б*) обмін місцями двох елементів масиву. Реалізація цих підзадач у наведеній нижче програмі виділена напівжирним шрифтом та рядком коментарів.

```
program Sorting_Arr;
  var i,j,n,min,k:integer;
      A:array[1..100]of integer;
begin
   WriteLn('n?');ReadLn(n);
  for i:=1 to n do                        begin
     WriteLn('A[',i,']?'); ReadLn(A[i])  end;
  for i:=1 to n-1 do        {1}        begin
     {визначення найменшого у хвості}
     min:=i;                {2}
     for j:=i+1 to n do     {3}
        if A[j]<A[min]      {4}
           then min:=j;     {5}
     {обмін місцями першого і найменшого елементів хвоста }
     k:=A[i];               {6}
     A[i]:=A[min];          {7}
     A[min]:=k              {8}       end;
  for i:=1 to n do
     Write(A[i],' ');
end.
```

Проаналізувавши {1}-{5}, бачимо, що тут реалізовано пошук найменшого елемента в неупорядкованому лінійному числовому масиві. Але зовнішній цикл управляє параметром *j* внутрішнього циклу. Із {3} видно, що початкове значення *j* завжди є номером першого елемента хвоста. Величина *min* зберігає номер найменшого елемента (див. {2}, {4}-{5}).

Команди {6}-{8} виконують *обмін* місцями двох вибраних елементів. У даному випадку *найменшого і першого* у хвості, але ідея обміну значеннями часто використовується в інших задачах, тому наведемо ще один її варіант. Нехай дано числові величини *a* та *b*. Напишемо серію команд, що проводить обмін їх значеннями *без використання допоміжної величини*: $a := a + b$; $b := a - b$; $a := a - b$ (пропонуємо перевірити правильність).

Програми сортування – це один із перших кроків до логічно більш насиченого програмування, тому при читанні, а особливо при відлагодженні таких програм часто виникають труднощі. Один із шляхів їх часткового подолання – це досягнення вищого рівня структурованості за рахунок

[30] Існує багато способів упорядкування, які поділяються на групи за методом та ефективністю, але в ЕКАП їх розглядати можливості немає.

Приклад XI.2

оформлення частин програм у вигляді підпрограм. У прикладі XI.1 можна виокремити функцію визначення номера найменшого елемента у хвості та процедуру обміну значеннями першого і найменшого елемента.

```
program Sorting_Arr*;
   var i,j,m,p,k:integer;
       A:array[1..100]of integer;
   function Min:integer;                {1}
   begin                                {1}
     m:=i;                              {1}
     for j:=i+1 to n do                 {1}
       if A[j]<A[m]  then m:=j;         {1}
     Min:=m                             {1}
   end;
   procedure Exchange;                  {2}
   begin                                {2}
      p:=Min;                           {2}
      k:=A[i]; A[i]:=A[p]; A[m]:=k      {2}
   end;                                 {2}
begin
   {ввод даних}                         {3}
    for i:=1 to n-1 do
      Exchange;                         {4}
   {вивод упорядкованого масиву}        {3}
end.
```

Це дало змогу скоротити тіло основної програми і зробити її максимально зручною в користуванні. Якщо програма працює із зовнішніми файлами, можна також описати і викликати процедури вводу та виводу {3}. Зверніть увагу на те, що функція *Min* {1}, описана вище процедури *Exchange* {2}. Це дозволило викликати функцію в процедурі, а не в програмі та перенести внутрішній цикл по *j* в неї, що спростило перевірку виконання зовнішнього циклу по *i*, який залишився у основній програмі і містить лише команду виклику процедури *Exchange* {4}. Щоб переконатись у перевагах програми *Sorting_Arr**, покроково виконайте їх обидві.

Приклад XI.3 Наведемо ще одну схему упорядкування лінійної числової таблиці:

таблиця XI.3

i	1	2	3	4	5	6	7	8	9	10
A[i]	***7***	***-9***	4	8	12	-56	0	14	-23	11
1-й крок	-9	***7***	***4***	8	12	-56	0	14	-23	11
2-й крок	-9	4	***7***	***8***	12	-56	0	14	-23	11
3-й крок	-9	4	7	***8***	***12***	-56	0	14	-23	11
4-й крок	-9	4	7	8	***12***	***-56***	0	14	-23	11
5-й крок	-9	4	7	8	-56	***12***	***0***	14	-23	11
6-й крок	-9	4	7	8	-56	0	***12***	***14***	-23	11
7-й крок	-9	4	7	8	-56	0	12	***14***	***-23***	11
8-й крок	-9	4	7	8	-56	0	12	-23	***14***	***11***
9-й крок	***-9***	***4***	7	8	-56	0	12	-23	11	***14***
10-й крок	-9	***4***	***7***	8	-56	0	12	-23	11	***14***
11-й крок	-9	4	***7***	***8***	-56	0	12	-23	11	***14***
12-й крок	-9	4	7	***8***	***-56***	0	12	-23	11	***14***

13-й крок	-9	4	7	-56	***8***	***0***	12	-23	11	***14***
14-й крок	-9	4	7	-56	0	***8***	***12***	-23	11	***14***
15-й крок	-9	4	7	-56	0	8	***12***	***-23***	11	***14***
16-й крок	-9	4	7	-56	0	8	-23	***12***	***11***	***14***
17-й крок	***-9***	***4***	7	-56	0	8	-23	11	***12***	***14***
18-й крок	-9	***4***	***7***	-56	0	8	-23	11	***12***	***14***
19-й крок	-9	4	***7***	***-56***	0	8	-23	11	***12***	***14***
20-й крок	-9	4	-56	***7***	***0***	8	-23	11	***12***	***14***
21-й крок	-9	4	-56	0	***7***	***8***	-23	11	***12***	***14***
22-й крок	-9	4	-56	0	7	***8***	***-23***	11	***12***	***14***
23-й крок	-9	4	-56	0	7	-23	***8***	***11***	***12***	***14***
24-й крок	-9	4	-56	0	7	8	-23	***11***	***12***	***14***

З таблиці XI.3 видно, що зліва направо порівнюються пари сусідніх елементів (виділені напівжирним курсивом), поки не будуть порівняні останній та передостанній елементи. При цьому, якщо елементи в парі *упорядковані по спаданню*, вони переставляються місцями. У наведеній таблиці процес закінчиться на 9-у кроці, при цьому найбільший елемент таблиці опиниться останнім, тобто упорядкованим (він виділений підкресленням). Назвемо упорядковану частину масиву хвостом. Після виконання 10-17 кроків, хвіст становитиме два елементи, а після 24-го кроку включатиме три елементи.

Пропонуємо продовжити таблицю до повного її упорядкування. Але наведених кроків цілком достатньо, щоб встановити ключ такого упорядкування і побудувати відповідний алгоритм.

Дотримання наведених щойно і в попередніх уроках порад відповідає ПДС, воно забезпечує вироблення *правильного стилю* програмування *ще на етапі ознайомлення*. Дійсно, ми з перших уроків акцентували увагу на структурності, способах вводу і виводу, прийомах трасування та відлагодження, методах покрокової деталізації, націлювали користувачів на пошуки найбільш ефективних форм програм, це дало змогу показати в дії ефективність комплексного застосування методів покрокової деталізації, адже запис і пояснення програм у вигляді процедур та функцій дозволив поступово формувати та використовувати *бібліотеку базових підпрограм*, що було проілюстровано у прикладі XI.2.

Слід також звернути увагу на те, що підпрограми у прикладі XI.2 не містять параметрів та взяти це на озброєння, бо на початкових етапах навчання програмуванню учням важко усвідомити у повному обсязі зміст формальних і фактичних параметрів, параметрів-значень та параметрів-змінних. Крім того, спрощена ієрархія величин, коли всі вони, або майже всі – глобальні, робить програму також більш прозорою. У той же час, наявність мінімального числа локальних змінних акцентує увагу на їх особливій ролі в програмі.

Не зважаючи на те, що упорядкування – одна із найширших тем навчального програмування, ми обмежимось лише цими прикладами. Потрібно було б дати класифікацію методів сортування та навести зразок упорядкування масивів з текстовими елементами. Але порівняно більша складність останніх, стислість експрес-курсу та те, що рядкові величини доцільно вивчати після масивів (у нас в уроці XII), не дає змоги тут навести такий приклад.

? *Питання до теми уроку*:

XI.1 Чим відрізняється сортування по зростанню від сортування по спаданню?

XI.2 Чи можна видозмінити програму *Sorting_Arr* для сортування за спаданням?

XI.3 Чи можна застосувати два ключі сортування одночасно, наприклад, сортування за спаданням і сортування за зростанням?

XI.4 Чи можна назвати сортуванням запис у лінійному числовому масиві зліва від'ємних елементів, потім нулів і справа додатних чисел?

XI.5 Яка складність програми *Sorting_Arr*?

XI.6 Чи зменшилась складність програми *Sorting_Arr** порівняно з програмою *Sorting_Arr*?

XI.7 Чи зменшиться складність програми *Sorting_Arr**, якщо у ній не використовувати величину *k* для обміну значеннями двох елементів масиву?

XI.8 Чи ефективнішою (за кількістю кроків до повного упорядкування) буде програма, що відповідає таблиці XI.3 порівняно з програмою *Sorting_Arr*?

✍ *Завдання XI.1* У програмі *Sorting_Arr** забезпечити зовнішній ввод і вивод, використавши текстові файли. Оформити ввод і вивод окремими процедурами *Inpt* та *Outp*.

Завдання XI.2 Проаналізувати залежність швидкодії програми *Sorting_Arr** від розміру масиву, що треба упорядкувати.

Завдання XI.3 Сформулювати ключ сортування методу, що описаний у прикладі XI.3.

*Завдання XI.4** Порівняти кількість кроків для повного упорядкування лінійного масиву з 20 чисел за ключами, описаними у прикладах XI.1 і XI.3.

*Завдання XI.5** Написати програму *Sorting_Arr***, що упорядковує масив за ключем прикладу XI.3.

Урок XII. Рядкові величини та алгоритми їх опрацювання.

У прикладі VI.2 (урок VI) було згадано тип *string* та наведено варіанти його оголошення й опису. Тут лише додамо, що для величини *a:string* елементи *a*[1] – *a*[255] є символами, тобто елементами типу *char*, а елемент *a*[0] є числом типу *byte*, він зберігає кількість символів, що містить *a*. Але звертання до *a*[0] вважається помилковим. З цієї причини, строго кажучи, величину *a:string* не можна вважати аналогом лінійного масиву, що складається із символів. Але в діапазоні 1-255 вона повністю є таким масивом, тому до неї застосовні всі алгоритми, що стосуються невеликих лінійних масивів, у тому числі й алгоритми пошуку та упорядкування.

У зв'язку з тим, що в програмуванні опрацювання текстів має важливе значення, на відміну від масивів, для величин типу *string* існують *стандартні процедури та функції*, що дозволяють звертатись *не до окремих символів*, як до елементів масиву, а до цілих *ланцюжків символів* і виражають *операції перетворення текстових величин*. Наведемо найважливіші з них:

Length(s*:string):byte* – функція, повертає довжину тексту *s*;

Copy(s*:string,* ***i****,* ***c****:integer):string* – функція, повертає частину *s*, довжиною *c*, починаючи з позиції *i*;

Delete*(var* ***s****: string;* ***i****,* ***c****: integer)* – процедура, видаляє в тексті *s c* символів, починаючи з позиції *i*.

Insert(s1*: string; var* ***s****: string,* ***i****: integer)* – процедура, вставляє текст *s1* в текст *s* з позиції *i*.

Str(x*:integer/real; var* ***s****:string)* – процедура, перетворює число *x* у текст *s*.

Val(s*:string;var* ***v****:integer/real;****code****:integer)* – процедура, перетворює текст *s* в число *v*, якщо перетворення можливе, то *code* = 0, інакше *code* ≠ 0.

👓 Тут опис наведених стандартних підпрограм має загальноприйнятий у літературі, так би мовити, *канонічний* вигляд. Він відповідає заголовку підпрограми. Радимо розвивати в учнів навички читання таких записів та використовувати їх, бо це не тільки відповідає ПДС, а й

спонукає до строгого стилю висловлення, як “про себе“, так і “вголос“ чи “письмового“, адже сучасній людині дуже не вистачає вербальних умінь і навичок.

Для текстових величин також застосовуються деякі операції, наприклад, операція конкатенації (злиття, або “склеювання“). Якщо *a, b:string*, *a*=’20’, *b*=’09’, то *a*+*b* =’2009’.

Окремим видом операцій є *операції порівняння*, які позначаються: “<“– менше, “>“ – більше, “<=“ – не менше, “>=“ – не більше, “=“ – дорівнює, “<>“ – не дорівнює. У зв’язку з тим, що кожен символ кодується натуральним числом, порівняння текстів виконується за правилом порівняння натуральних чисел, наприклад, ’Тарас’ >’ТАРАС’, бо серед букв *кирилиці* буква ’а’ має більший код, ніж буква ’А’.

Здебільшого операції порівняння застосовуються у поєднанні із процедурами та функціями опрацювання текстових величин. Проілюструємо це на прикладах:

Приклад XII.1

```
program Man_Wom;
  var name,s:string;
begin
  WriteLn('ввести повне ім"я:');ReadLn(name);
  s:=Copy(name,Length(name),1);{1}
  if s='a'                      {2}
    then WriteLn('жін.')
    else WriteLn('чол.')
end.
```

У рядку {1} одночасно застосовано функції *Copy* та *Length*. Друга визначає кількість символів у тексті *name*, а перша надає величині *s* значення останньої літери повного імені, тобто “ч“ (для чоловіків) або “а“ (для жінок). У рядку {2} порівнюється значення s із текстом ’а’.

Приклад XII.2

```
program Palindrom;
   var word:string;
   function Contr(st:string):string;
     var i,n:integer;
         st1:string;
   begin
     st1:='';                        {1}
     n:=Length(st);                  {2}
     for i:=1 to n do                {3}
       st1:=st1+Copy(st,n-i+1,1);    {4}
    Contr:=st1                       {5}
   end;
begin
   WriteLn('ввести слово:');ReadLn(word);{6}
   if word=Contr(word)               {7}
     then WriteLn('паліндром')       {8}
     else WriteLn('не паліндром')    {9}
end.
```

Під паліндромом розуміють текст, або число, що читаються однаково, як зліва-направо, так і справа-наліво, тобто симетричні відносно середини, наприклад, “потоп“. Функція *Contr* формує текст, зворотний до даного,

наприклад, при *st* = 'інформатика', *Contr(st)*='акитамрофні' (уважно розгляньте {1}–{5} та виконайте їх покроково). У ній, як і в попередньому прикладі, викликаються функції *Copy* та *Length*. Слід зауважити, що при перевірці текстів "Потоп" та "Ані Лорак – Кароліна" відповідь буде негативною (подумайте – чому?).

Приклад XII.3 На закінчення розглянемо складніший, але більш корисний приклад, у якому функція ***Long_Add*** виконує додавання двох багатоцифрових чисел, поданих у вигляді текстів *st1* та *st2*. Справа в тому, що навіть найпотужніші цілочисельні типи у *Pascal* не можуть зберегти багатоцифрове число, навіть таке досить невелике, як 100000000000, а тим більше виконувати арифметичні дії над подібними числами.

```
function Long_Add(st1,st2:string):string;                    {1}
   var i,l1,l2,pm,s1,s2,s,cod:word;st:string;                 {2}
begin
   if st1[0]<st2[0] then
     begin st:=st1;st1:=st2;st2:=st end;                      {3}
   st1:='0'+st1;                                              {4}
   l1:=Length(st1);l2:=Length(st2);pm:=0;                     {5}
   for i:=l1-l2 downto 1 do st2:='0'+st2;                     {6}
   for i:=l1 downto 1 do                          begin       {7}
     Val(st1[i],s1,cod);Val(st2[i],s2,cod);                   {8}
     s:=s1+s2+pm;                                             {9}
     st1[i]:=Char((s mod 10)+48);pm:=s div 10 end;            {10}
   if st1[1]='0' then Delete(st1,1,1);                        {11}
   Long_Add:=st1
end;
```

З {1} видно, що багатоцифрові числа вводяться з основної програми, як рядки *st*1 і *st*2, а функція повертає також у вигляді рядка результат додавання двох багатоцифрових чисел. Всі числові локальні змінні{2}, за винятком *cod*, яка згідно формату стандартної функції ***Val*** повинна мати тип *integer* або *word*, для економії пам'яті можуть бути оголошені, як *byte*. {3} переставляє на місце першого доданка число з більшою кількістю цифр, а {4} забезпечує ще один старший розряд, рівний 0 на випадок, коли сума буде довшою на одну цифру, ніж *st*1. {5} носить підготовчий характер для циклу {6}, де зліва від старшого розряду *st*2 добавляються нулі, щоб *st*1 та *st*2 мали однакову довжину. Основний смисл функції ***Long_Add*** реалізований в рядках{7}-{10}, спочатку з допомогою функції ***Val*** символи перетворюються в числа *s*1 та *s*2{8}, потім ці числа разом із числом *pm* утворюють число *s*.

Для роз'яснення значень величин *pm* та *s* пригадаємо алгоритм додавання натуральних чисел у стовпчик на прикладі:

```
  010  {pm}
  235  { s1}
+
   29  {s2}
 ____
  264  {  s}        (1)
```

З (1) видно, що *pm* – це цифри, які запам'ятовуються для додавання в старшому відносно поточного розряді, якщо сума цифр поточного розряду більша за 9, а для найменшого розряду pm надається значення 0. Число *s* – це результат додавання *s*1 та *s*2.

У {8} одержується нове значення *st*1[*i*] з допомогою функції ***Char***, як остача від ділення *s* на 10. Таким чином, *pm* = *s* div 10, а *st*1[i] дорівнює ***Char***((*s* *mod* 10)+48). Доданок 48 необхідний для того, щоб уникнути невідповідності коду цифри, як символу, та самої цифри.

Нарешті в {11} з допомогою процедури ***Delete*** видаляється лівий крайній символ *st*1, якщо він рівний 0.

Подібно функції ***Long_Add***, взявши за основу відомі арифметичні алгоритми, можна скласти ще деякі важливі підпрограми для опрацювання багатоцифрових чисел (віднімання, множення, цілочисельне ділення і т.д.), які можна об'єднати у *модуль*, тобто *бібліотеку підпрограм* користувача, який після *компіляції* буде зберігатись у файлі **.tpu* і може підключатись з допомогою зарезервованого слова *uses* (див. табл. III.1 уроку III), подібно стандартним модулям. Тоді не буде потреби оголошувати й описувати ці підпрограми в основній програмі.

? *Питання до теми уроку*:

XII.1 Чи можна для величин *s:string* та *c:char* виконати команду *c*:=*s*?
XII.2 Чим відрізняються стандартні підпрограми від підпрограм користувача?
XII.3 Для чого, на вашу думку, в процедурі *Val* параметр *c*?
XII.4 Чому, на вашу думку, *Copy* – не процедура, а функція?
XII.5 Чому, на вашу думку, *Insert* – не функція, а процедура?
XII.6 Чи може практично виникнути потреба у використанні операцій порівняння "<="та ">="?
XII.7 Чи можна для представлення багатоцифрових чисел застосувати не текстові величини, а масиви?

✍ *Завдання XII.1* Написати програму, що з допомогою функції *Long_Add* виконуватиме додавання багатоцифрових чисел.
Завдання XII.2 У програмі *Palindrom* (приклад XII.2) зробити зміни, що дозволять ігнорувати пропуски та не розрізнятимуть великі і малі літери.
Завдання XII.3 Написати функцію, що визначає модуль багаторозрядного числа.
*Завдання XII.4** Написати функцію, що виконує множення багаторозрядного числа на однорозрядне.
*Завдання XII.5** Написати програму, що виконує множення двох багаторозрядних чисел.
*Завдання XII.6** Написати програму, що обчислює *n!* при *n*>17.

III ПРАКТИКУМ З ПРОГРАМУВАННЯ

Вище було розглянуто поняття про алгоритми і програми, основні команди і типи величин у мові *Pascal*, основні принципи структурного програмування і деякі базові алгоритми. Уроки, що входять до експрес-курсу, містять необхідні для закріплення вивченого навчального матеріалу приклади, запитання та завдання, але будь-який курс програмування не може бути повноцінним без серії практичних чи лабораторних занять. Тому нижче наведено шість практичних робіт, які допоможуть більш якісному засвоєнню курсу. Вони мають аналогічну наведеним вище 12 урокам структуру, а також містять приклади і завдання. В той же час, ці практичні роботи побудовані так, що при дуже стислому ознайомленні з програмуванням їх можна розглядати вибірково, не розглядати зовсім, чи пропонувати для самостійного опрацювання.

Практична робота 1. Обчислення значення многочленна за схемою Горнера.

Спочатку ознайомимось із програмою швидкого обчислення значень многочлена за *схемою Горнера*, що часто буває дуже корисним. Нехай дано многочлен:

$$P(x)=a_0x^n+a_1x^{n-1}+a_2x^{n-2}+...+a_{n-2}x^2+a_{n-1}x+a_n, \quad (1)$$

де коефіцієнти позначені так, що сума індексу та відповідного степеня змінної рівна *n*. Для прикладу візьмемо: $P(x)=11x^5+7x^4-2x^2+x-8.$ (2)

Складемо таблицю коефіцієнтів для (2), включаючи нулі, у порядку зростання індексів:

i	0	1	2	3	4	5
a_i	11	7	0	-2	1	8

Тепер перетворимо многочлен таким чином:

$$\begin{aligned}P(x)&=11x^5+7x^4+0x^3-2x^2+x-8=(11x^4+7x^3+0x^2-2x^1+1)x-8=\\&=((11x^3+7x^2+0x-2)x+1)x-8=(((11x^2+7x+0)x-2)x+1)x-8=\\&=((((11x+7)x+0)x-2)x+1)x-8.\end{aligned} \quad (3)$$

Для остаточного варіанту (3) при даному значенні *x* алгоритм обчислення значення *P(x)* полягає у виконанні циклу для значень $0\le i\le n$, тіло якого містить множення попереднього значення (спочатку рівного a_0) на *x* та додавання a_i, що реалізовано функцією ***S_G*** {2}.

Крім того, представлення многочлена за схемою Горнера дозволяє уникнути операції піднесення до степеня і зменшити кількість арифметичних операцій при обчисленні його значення. Наприклад, (2) містить 16 операцій, а (3) тільки 10, різниця – 6 операцій. При великому *n* і незначній кількості коефіцієнтів, рівних нулю, ця різниця може бути дуже великою. Програма має вигляд:

```
program Gorners_Circuit;
  type mas=array[0..100] of real;              {1}
  var   i,n:byte; x,y:real; A:mas;
  function S_G(n:integer;x:real;B:mas):real;{2}
    var i:byte;y:real;
  begin
    y:=B[0];                                   {3}
    for i:=1 to n do y:=y*x+B[i];              {3}
    S_G:=y
```

```
    end;
begin
    repeat
        WriteLn('n?');ReadLn(n);
    until n<=100;
    WriteLn('x?');ReadLn(x);
    for i:=0 to n do    begin
        WriteLn('A[',i,']?');
         ReadLn(A[i])   end;
    WriteLn(S_G(n,x,A):12:2);
end.
```

! Слід звернути увагу на параметр *B* у заголовку функції *S_G*. З опису видно, що це масив (див {1}, {2}). Наголошуємо, що так описуються формальні параметри-масиви, які передаються у підпрограми. Раніше ми намагались уникати формальних параметрів у підпрограмах, але ПДС вимагають їх використовувати, бо це покращує читабельність програм і контроль за величинами під час їх трасування, крім того, допомагає уникати помилок виконання при одночасному використанні тих же глобальних величин програмою та її підпрограмами.

Корисно записати та зрозуміти загальну формулу (4) розкладу многочлена (1):

$$P(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-2}x^2 + a_{n-1}x + a_n{}^n = \\ = ((...(a_0x^{n-1} + a_1)x + a_2)x + ... + a_{n-1})x + a_n \qquad (4)$$

? *Питання до практичної роботи*:

1.1 Від чого залежить різниця у кількості операцій при записі многочлена у формах (1) та (4)?

1.2 Яка максимальна різниця в кількості операцій при обчисленні значення многочлена, представленого у формах (1) та (4)?

✍ *Завдання 1.1* Скласти таблицю коефіцієнтів для розкладу за схемою Горнера многочлена:

$$P(x) = -3x^9 + 17x^7 - 12x^6 + 5x^4 + 9x^3 - 11x^2 + 16x - 18..$$

Завдання 1.2 Написати многочлен, що відповідає таблиці коефіцієнтів:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a_i	1	2	3	0	4	0	5	6	7	0	0	8	0	9	10	11	12

Завдання 1.3 Змінити програму *Gorners_Circuit*, підключивши текстові файли для вводу/виводу.

Завдання 1.4 Вставити у програму *Gorners_Circuit* команди для підрахунку *k* – кількості арифметичних операцій при обчисленні значення многочлена і встановити залежність *k* від *n* та кількості коефіцієнтів, рівних 0.

Завдання 1.4 Записати програму *Gorners_Circuit* у вигляді функції *Gorners_Circuit_Func.*

Практична робота 2. Табулювання функції.

Табулювання функції – це визначення її значень для значень аргументу *на проміжку* [*a*, *b*] з *кроком табуляції h*, що можна використати для її дослідження та побудови графіка. Для цього необхідно ввести значення *a*, *b* та *n* – кількість значень *x* на [*a*, *b*]. Обчислення значення функції зручно записати у вигляді підпрограми-функції *F*(*x*). Нижче наведено програму ***Tab_Func*** для функції *y=x/(1+x)*. Щоб скоротити пояснення, у програмі застосовано поширені коментарі. Але окремі фрагменти програми потребують детальнішого

пояснення, вони виділені з допомогою додаткових коментарів у вигляді числа в фігурних дужках. Самі коментарі наведено нижче.

Приклад 2.1

```
program Tab_Func;                  {табулювання функції y = x/(1+x)}
  var a,b,x,h,y:real;
      n:integer;
  function F(x:real):real;
  begin
    if 1+x<>0               {1}  {перевірка належності x ОДЗ}
      then F:=x/(1+x)       {2}  {обчислення значень функції}
      else F:= –1000000     {3}  {для x∉ ОДЗ, y = –1000000}
  end;
begin
  WriteLn('a?');ReadLn(a);       {ліва межа відрізка табулювання}
  WriteLn('b?');ReadLn(b);       {права межа відрізка табулювання}
  WriteLn('n?');ReadLn(n);       {кількість кроків табулювання}
  h:=(b-a)/n;               {4}  {крок табуляції}
  x:=a;                          {початкове значення x}
  while x<=b do  begin
    WriteLn('y(',x:4:2,')=',F(x):8:4);{вивод значень функції}
    x:=x+h       end              {збільшення значення x на крок}
end.
```

При заданому значенні n може трапитись, що x не належить ОДЗ аргументу. Тоді відповідне значення функції не можна обчислити. Для уникнення цього у підпрограмі-функції використана команда розгалуження {1}. Але підпрограма-функція обов’язково повинна мати єдиний результат, тому у випадку виходу значення x з ОДЗ підпрограма-функція в якості результату видасть завідомо не правильне значення, наприклад -10000000 {3}.

Результати роботи програми для $a = –1$, $b = 2$, $n = 6$:

таблиця 2.1

x	-1	-0,5	0	0,5	1	1,5	2
y(x)	-1000000	-1,0000	0,0000	0,3333	0,5000	0,6000	0,6667

Змінною частиною програми є лише рядки {1} та {2}.

Приклад 2.2 Скласти програму розв’язування рівняння $y = kx + b$ для цілих значень аргументу з проміжку [-5; 5].

У наступній програмі, яка по суті аналогічна попередній (приклад 2.1), немає потреби визначати крок табуляції.

```
program Equation;   {розв'язування рівняння y=kx+b}
  var x, y:integer;
      k, b:real;
  function F(x:integer):real;
  begin
    F:=k*x+b                     {обчислення значень y}
  end;
begin
  WriteLn('k?');ReadLn(k);     {ввод коефіцієнта}
  WriteLn('b?');ReadLn(b);     {та вільного члена рівняння}
  x:= –5;                      {початкове значення x}
  while x<=5 do begin
    WriteLn('(',x,', ',F(x),');'); {вивод розв'язків}
```

```
            x:=x+1        end          {наступне значення x}
        end.
```

Результати роботи програми для $k = 2$ та $b = 3$:

x	-5	-4	-3	-2	-1	0	1	2	3	4	5
y	-7	-5	-3	-1	1	3	5	7	9	11	13

таблиця 2.2

? *Питання до практичної роботи*:

2.1 Чи можна у програмі *Tab_Func* замість аргумента *n* вказати аргумент *h* (крок табуляції)?

2.2 Чому у програмі *Tab_Func* у випадку, коли $x \notin$ ОДЗ, не можна вивести текстове повідомлення, наприклад "значення *y* обчислити не можна"?

2.3 Чи можна по таблиці 2.2 визначити кількість розв'язків рівняння $kx+b=0$?

2.4 Чи можна застосувати програму *Equation* застосувати для розв'язування нелінійних рівнянь з двома невідомими? Якщо можна, то у яких випадках?

Завдання 2.1 Написати програму, яка з допомогою функції *Gorners_Circuit_Func* виконує табулювання функції: $P(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_{n-2}x^2 + a_{n-1}x + a_n{}^n$.

*Завдання 2.2** Написати програму *P_Equation*, яка з допомогою програми *Equation* та функції *Gorners_Circuit_Func* визначає кількість розв'язків рівняння:

$$a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_{n-2}x^2 + a_{n-1}x + a_n{}^n = 0.$$

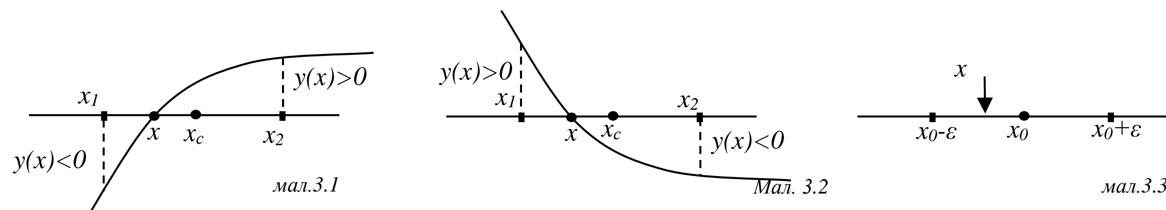
*Завдання 2.3** Написати програму, яка визначає усі відрізки [*k*; *k*+1], що містять корені рівняння $a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_{n-2}x^2 + a_{n-1}x + a_n{}^n = 0$.

Практична робота 3. Уточнення розв'язку рівняння.

Після розгляду практичних робіт 1–2 та виконання завдань 2.2*–2.3* логічно поставити питання про *уточнення розв'язку* рівняння, адже існує багато типів рівнянь, у тому числі й рівнянь $p(x)=0$, де $p(x)$ – многочлен *n*-го степеня, для яких не існує аналітичних методів розв'язування.

ⓘ *Уточненням розв'язку рівняння з точністю до $\varepsilon>0$* (епсілон) називають визначення його наближеного значення x_0, яке відрізняється від точного значення розв'язку *x* не більше, ніж на ε, тобто $|x-x_0|\le\varepsilon$.

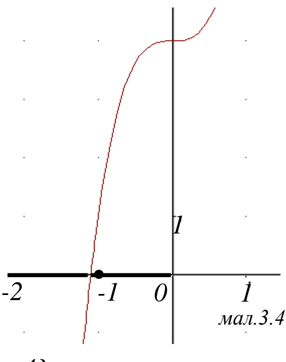
Нехай відомо, що на відрізку $[x_1, x_2]$ існує рівно один розв'язок *x* рівняння $y(x)=0$, де $y(x)$ – *неперервна* крива лінія. Тоді вигляд графіка функції $y(x)$ у *ε-околі* значення *x* (відрізок $[x-\varepsilon, x+\varepsilon]$, див. мал.3) матиме вигляд, показаний на мал.1, або на мал.2. *Характеристичною властивістю графіка у околі точного значення розв'язку* є те, що $y(x_1)$ та $y(x_2)$ мають різні знаки, тобто $y(x_1)*y(x_2)<0$.



Для визначення наближеного розв'язку рівняння можна застосувати відомий з уроку *X* метод половинного ділення, визначаючи значення $x_c=(x_2-x_1)/2$ і перевіряючи, чи істинна нерівність $|x_2-x_1|\le\varepsilon$. У випадку її істинності можна вважати x_c наближеним значенням розв'язку рівняння. У іншому випадку слід зменшити відрізок $[x_1, x_2]$, дотримуючись *характеристичної властивості*

графіка у околі точного значення розв'язку. Отже, програма уточнення розв'язку рівняння повинна містити щойно описані дії.

```
program SpecDec;                              {f(x)=3x^3+4}
   var a,b,c,x,e:real;
   function F(e:real;var a,b:real):real;   {1}
      var c:real;
   begin
     while abs(b-a)>2*e do   begin
        c:=(a+b)/2;                            {2}
        if (3*a*a*a+4)*(3*c*c*c+4)<=0{3}
          then b:=c else a:=c end;
     F:=(a+b)/2;
   end;
begin
   a:= -2; b:= -1;
   WriteLn('e?');ReadLn(e);
   WriteLn('x=', F(e,a,b):8:6);              {4}
end.
```



мал.3.4

Програма видає наближені значення розв'язку рівняння $3x^3+4=0$ при ε=0,1 x=-1,0625, при ε=0,0001 x=-1,100647. При зменшенні ε значення розв'язку стає більш точним. У першому випадку точність не перевищує 0,1. У другому випадку точність досягає 0,0001. Щоб побачити процес уточнення, розглянемо програму *SpecDec**.

```
program SpecDec*;                     {f(x)=3x^3+4}
  var i:integer;
      a,b,c,x,e:real;
  {опис функції F взяти з програми SpecDec};
begin
  a:=-2; b:=-1;
  e:=abs(b-a)/2;                     {1}
  for i:=1 to 12 do begin            {2}
     WriteLn('x=',F(e,a,b):12:10);
     e:=e/10          end;           {3}
end.
```

У ній ε змінне, воно поступово зменшується, починаючи з *abs(b-a)/2* {1}, тобто точність не перевищує 1. Кожне наступне значення ε менше попереднього в 10 разів {2}, тобто точність зростає на порядок, стаючи рівною 0,1; 0,01; 0,001 і т.д. Це видно із таблиці 3.1:

таблиця 3.1

ε	0,5 (точ. 1)	0,05 (точ. 0,1)	0,005 (точ. 0,01)	0,0005	0,00005	0,000005
x	-1,5	-1,09375	-1,09765625	-1,1010742188	-1,1006164551	-1,1006431580
ε	0,0000005	0,00000005	0,000000005	0,0000000005	0,00000000005	0,000000000005

Два останні значення x дорівнюють -1,1006424163, що означає досягнення точності, дозволеної параметром *l2* циклу {2} та форматованим виводом {3}. Для досягнення вищої точності слід збільшити ці величини. Але

уточнення розв’язку програмою *SpecDec** можливо лише в межах типу *real* (15-16 значущих цифр).

Недоліком наведених програм є те, що кількість цифр результату не залежить від встановленої точності. Програма *SpecDec*** самостійно встановлює кількість цифр виводу, для цього вона містить додаткову функцію *NumMark* (див {!}).

```
program SpecDec**;                    {f(x)=3x^3+4}
  var i,n:integer;
      a,b,c,x,e:real;
  procedure InpDat(var a,b,e:real);
  begin
    WriteLn('a,b?');ReadLn(a,b);
    WriteLn('e?');ReadLn(e);
  end;{end InpDat}
  function NumMark(e:real):integer;
     var i:byte;eps:real;
  begin
     i:=0;eps:=e;
     while eps<=1 do          begin
        eps:=eps*10;i:=i+1 end;
     NumMark:=i
  end;{end NumMark}
  function SpecRoot(e,a,b,c:real):real;
  begin
    while b-a>2*e do           begin
       c:=(a+b)/2;
       if (3*a*a*a+4)*(3*c*c*c+4)<=0
          then b:=c else a:=c end;
    SpecRoot:=(a+b)/2;
  end;{end SpecRoot}
begin
  InpDat(a,b,e);
  n:=NumMark(e);
  WriteLn('x=', SpecRoot(e,a,b,c):15:n) {!}
end.
```

Приклади роботи цієї програми для різних значень ε:

$\varepsilon=0{,}01,\ x=-1{,}102$; $\varepsilon=0{,}0001,\ x=-1{,}10065$; $\varepsilon=0{,}000001,\ x=-1{,}1006422$.

? *Питання до практичної роботи*:

3.1 Яка роль ε при уточненні розв’язку рівняння?

3.2 Що слід змінити у програмі *SpecDec***, щоб вона уточнювала розв’язок іншого рівняння?

3.3 Чи можна уточнювати розв’язки рівняння на проміжку [*a*,*b*], якщо на ньому графік відповідної функції має розриви?

Завдання 3.1 Написати програму, яка визначає усі проміжки, що містять розв’язки рівняння $p(x)=0$, де $p(x)$ – многочлен.

Завдання 3.2 Записати програму *SpecDec*** у вигляді функції.

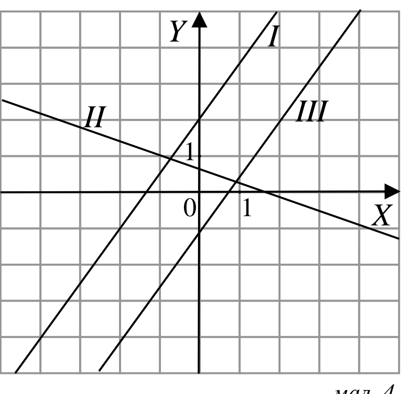
*Завдання 3.3** Написати програму, яка з допомогою функції *SpecDec*** визначає з указаною точністю усі розв’язки рівняння $p(x)=0$, де $p(x)$ – многочлен.

Практична робота 4. Розв'язування системи двох (трьох) лінійних рівнянь з двома (трьома) невідомими.

Нехай дано систему двох лінійних рівнянь з двома невідомими:

$$\begin{cases} a_1x + b_1y = c_1, \\ a_2x + b_2y = c_2 \end{cases} \qquad (1)$$

Як відомо, графіком кожного з них є пряма лінія. Дві прямі можуть мати лише три варіанти взаємного розміщення (див. мал.1): бути паралельними, співпадати, або перетинатись в одній точці. Отже система (1) може або не мати розв'язків (коли $a_1/a_2 = b_1/b_2 \neq c_1/c_2$), або мати безліч розв'язків (коли $a_1/a_2 = b_1/b_2 = c_1/c_2$), або мати єдиний розв'язок (коли $a_1/a_2 \neq b_1/b_2$).



мал. 4.1

Ці співвідношення можна покласти в основу алгоритму, але тоді необхідні додаткові формули для визначення розв'язків. Тому зручніше скористатись *правилом Крамера*. Але для цього спочатку введемо позначення визначника системи двох лінійних рівнянь з двома змінними Δ та визначників Δ_x і Δ_y:

$$\Delta = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1b_2 - a_2b_1; \; \Delta_X = \begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix} = c_1b_2 - c_2b_1; \; \Delta_Y = \begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix} = a_1c_2 - a_2c_1; \qquad (2)$$

Формули Крамера мають вигляд: $x = \dfrac{\Delta_x}{\Delta}, \; y = \dfrac{\Delta_y}{\Delta}$ (3)

Тоді програма для розв'язування системи двох лінійних рівнянь з двома невідомими методом Крамера матиме вигляд:

```
program SystLinEquat;
   var i,j:integer;d,dx,dy,x,y:real;
       a:array[1..2,1..3]of real;
begin
   for i:=1 to 2 do
     for j:=1 to 3 do  begin
        WriteLn('a[',i,',',j,']?'); ReadLn(a[i,j]) end;
   d:=a[1,1]*a[2,2]-a[2,1]*a[1,2];
   dx:=a[1,3]*a[2,2]-a[2,3]*a[1,2];
   dy:=a[1,1]*a[2,3]-a[2,1]*a[1,3];
   if d<>0                              {1}
     then              begin
        x:=dx/d; y:=dy/d;               {2}
        WriteLn('x=',x);                {2}
        WriteLn('y=',y) end             {2}
     else if dx=dy                      {3}
            then WriteLn('розв. безліч')
            else WriteLn('розв. нема');
end.
```

У програмі визначники позначено літерою "*d*". Команди {2} можна замінити на {4}: *WriteLn('x=',dx,'/',d,' y=',dy,'/',d)* {4}. Це не тільки скоротить кількість команд, а й дасть змогу завжди одержувати точні розв'язки. У *SystLinEquat* {1} виконує перехід до випадку, коли розв'язки існують ($d \neq 0$). Відповідно {3} передбачає, що $d=0$. Згідно правила Крамера, якщо $d=d_x=d_y=0$,

розв'язків безліч, інакше – жодного. Перевірку $d_x=d_y=0$ можна замінити на $d_x=d_y$ (перевірте самостійно).

Описаним методом можна розв'язувати й системи трьох лінійних рівнянь з трьома невідомими:

$$\begin{cases} a_1x+b_1y-c_1z=d_1, \\ a_2x+b_2y-c_2z=d_2, \\ a_3x+b_3y-c_3z=d_3 \end{cases} \tag{4}$$

Для цього замість формул (2) – (3) треба взяти формули (5) – (7):

$$\Delta = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = a_1b_2c_3 + b_1c_2a_3 + c_1a_2b_3 - c_1b_2a_3 - a_1c_2b_3 - b_1a_2c_3 \tag{5}$$

$$\Delta_X = \begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}; \ \Delta_Y = \begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}; \ \Delta_Y = \begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix} \tag{6}$$

$$x = \frac{\Delta_x}{\Delta}, \ y = \frac{\Delta_y}{\Delta}, \ \mathrm{z} = \frac{\Delta_z}{\Delta} \tag{7}$$

Пропонуємо формули для обчислення Δ_x, Δ_y і Δ_z написати та перевірити самостійно, по зразку (5). Можна замість формули (5) використовувати формулу (7):

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = a_1\begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} - b_1\begin{vmatrix} a_2 & c_2 \\ a_3 & c_3 \end{vmatrix} + \mathrm{c}_1\begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix} \tag{8}$$

? *Питання до практичної роботи*:

4.1 Скільки розв'язків може мати система трьох лінійних рівнянь з трьома невідомими?
4.2 Яка з формул (5) чи (7) краща для використання при обчисленні "вручну"?
4.3 Яка з формул (5) чи (7) краща для використання у програмі?

✍ *Завдання 4.1* Обчислити "вручну" визначник $\begin{vmatrix} 2 & -8 \\ 4 & 5 \end{vmatrix}$.

Завдання 4.2 Скласти систему рівнянь, відповідну визначнику завдання 4.1, взявши вільні члени довільно.

Завдання 4.3 Розв'язати методом Крамера систему завдання 4.2.

Завдання 4.4 Обчислити "вручну" визначник $\begin{vmatrix} -4 & 7 & 3 \\ 0 & 2 & 0 \\ -8 & -3 & 6 \end{vmatrix}$.

Завдання 4.5 Скласти систему рівнянь, відповідну визначнику завдання 4.4, взявши вільні члени довільно.

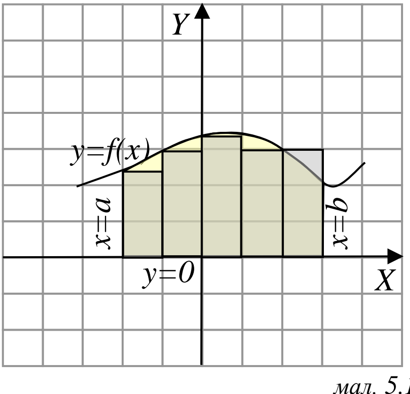
Завдання 4.6 Розв'язати методом Крамера систему завдання 4.5.

*Завдання 4.7** Змінити програму *SystLinEquat* так, щоб для вводу і виводу використовувались текстові файли.

*Завдання 4.8** Написати програму *SystLinEquat3*, яка визначає розв'язки системи трьох лінійних рівнянь з трьома невідомими методом Крамера.

Практична робота 5. Наближене обчислення площі криволінійної трапеції.

ⓘ *Криволінійною трапецією* називається фігура на координатній площині, обмежена віссю *OX*, прямими $x=a$, $x=b$ та графіком деякої функції $y=f(x)$.



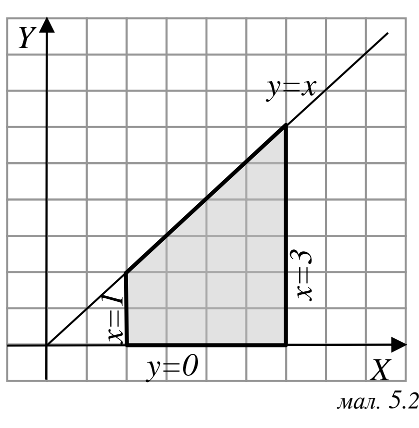
мал. 5.1

Існує багато способів наближеного обчислення площі криволінійної трапеції. Один із них (див. мал.) зводиться до обчислення площі ступінчатої фігури, що складається із *n* дотичних прямокутників однакової ширини $h=(b-a)/n$ і висотою $f(x_i)$, де x_i – абсциса лівого краю *i*-го прямокутника. Площа ступінчатої фігури буде наближатись до площі криволінійної трапеції із збільшенням *n* – кількості "сходинок", тобто буде наближеним значенням площі криволінійної трапеції з великою точністю, якою можна, як і в практичній роботі 3, управляти. Формула обчислення площі має вигляд:

$$S = \frac{b-a}{n} \cdot f(x_0) + \frac{b-a}{n} \cdot f(x_1) + ... + \frac{b-a}{n} \cdot f(x_{n-1}) =$$
$$\frac{b-a}{n} \cdot (f(x_0) + f(x_1) + ... + f(x_{n-1})) = h \cdot (f(x_0) + f(x_1) + ... + f(x_{n-1})) \qquad (1)$$

Програма обчислення площі може мати такий вигляд:

```
program Area;
  var i,n:integer;
      a,b,h,x,s:real;
begin
   WriteLn('a?');ReadLn(a);
   WriteLn('b?');ReadLn(b);
   WriteLn('n?');ReadLn(n);
   h:=(b-a)/n;              {1}
   s:=0;
   x:=a;
   for i:=1 to n do begin
     s:=s+h* sqrt(x);       {2}
     x:=x+h         end;    {3}
   WriteLn('s=',s:12:6);
end.
```



мал. 5.2

Величина *h* називається *кроком*, вона дорівнює *(b-a)/n* {1}. Початкове значення аргументу *x* дорівнює *a*, кожне наступне *x+h* {3}. У програмі взято функцію $y=\sqrt{x}$, тому змінною частиною програми є рядок {2}. Наведемо залежність значення площі від вибраного кроку. Для цього візьмемо функцію $y=x$, $a=1$, $b=3$. З малюнка 5.2 видно, що площа виділеної криволінійної трапеції дорівнює 4 кв.од.

таблиця 5.1

n	4	10	15	20	30	40	50	100	200	1000	2000
s	3,5	3,8	3,8(6)	3,9	3,9(3)	3,95	3,96	3,98	3,99	3,998	3,999

❓ *Питання до практичної роботи*:

5.1 Як позначиться на результаті заміна у формулі (1) значень $x_0, x_1, x_2, \ldots, x_{n-1}$ на $x_1, x_2, x_3, \ldots, x_{n-1}, x_n$?

5.2 Чи можна обчислити площу криволінійної трапеції на проміжку [a,b], якщо на ньому графік відповідної функції має розриви?

5.3 Який вигляд матиме криволінійна трапеція для функції $y=x$ при a=-1, b=3?

✍ *Завдання 5.1* Обчислити “вручну“ за формулою (1) площу фігури, утвореної функцією $y=x^2$ на відрізку [1;3] для n=5.

Завдання 5.2 Записати програму *Area* у вигляді функції.

*Завдання 5.3** Написати програму, яка з допомогою функції *Area* виводить на екран таблицю залежності значення площі від значення *n*.

*Завдання 5.4** Написати програму, яка з допомогою функції *Area* визначає наближене значення площі із указаною точністю ε (див. практичну роботу 3).

Практична робота 6. Розподіл температур в однорідному стержні.

Задача 1: На кінцях однорідного теплопровідного стержня підтримується температура t_1 і t_2, а по всій довжині стержень теплоізольований. Визначити, яка температура установиться у внутрішніх частинах стержня.

Розіб’ємо стержень на достатньо малі частини, температуру в яких можна вважати однаковою. Очевидно, що у кожній з таких частин температура у кожен момент часу визначатиметься за формулою: $t=(t_{left}+t_{right})/2$. Процес зміни температур матиме вигляд:

таблиця 6.1

t_{left}	t_1	t_2	t_3	t_4	t_5	t_{right}
10	0	0	0	0	0	50
10	5	0	0	0	25	50
10	5	2,5	0	12,5	25	50
10	6,25	2,5	7,5	12,5	31,25	50
10	6,25	6,875	7,5	19,375	31,25	50
10	8,4375	6,875	13,125	19,375	34,6875	50
10	8,4375	10,78125	13,125	23,90625	34,6875	50
10	10,390625	10,78125	17,34375	23,90625	36,953125	50
10	10,390625	13,8671875	17,34375	27,1484375	36,953125	50
10	11,9335938	13,8671875	20,5078125	27,1484375	38,5742188	50
10	11,9335938	16,2207031	20,5078125	29,5410156	38,5742188	50
10	13,1103516	16,2207031	22,8808594	29,5410156	39,7705078	50
10	13,1103516	17,9956055	22,8808594	31,3256836	39,7705078	50
10	13,9978027	17,9956055	24,6606445	31,3256836	40,6628418	50
10	13,9978027	19,3292236	24,6606445	32,6617432	40,6628418	50
10	14,6646118	19,3292236	25,9954834	32,6617432	41,3308716	50
10	14,6646118	20,3300476	25,9954834	33,6631775	41,3308716	50
10	**15**,1650238	20,3300476	26,9966125	33,6631775	41,8315887	50
10	**15**,1650238	**21**,0808182	26,9966125	**34**,4141006	41,8315887	50
10	**15,5404091**	**21**,0808182	**27,7474594**	**34**,4141006	**42,2070503**	50
10	**15,5404091**	**21**,6439342	**27,7474594**	**34**,9772549	**42,2070503**	50

У таблиці 6.1 показано 20 кроків. Останні рядки свідчать, що температура в частинках стержня встановлюється сталою з досить великою точністю (виділено шрифтом *Bold*).

Програма визначення температур у частинках стержня має вигляд:

```
program Core;
   const n=5;                                  {1}
   type A=array[0..n+1] of real;
   var i,k:integer;
        tL,tR,t0,eps,e:real;
        T,Th:A;f:char;
begin
   tL:=10;tR:=20;t0:=-1000;eps:=0.001;  {2}
   k:=0;e:=eps;
   repeat
     k:=k+1;e:=e*10 until not(e<1);
     Th[0]:=tL;Th[n+1]:=tR;
     for i:=1 to n do Th[i]:=t0;
     repeat
       for i:=1 to n do            begin
          T[i]:=(Th[i-1]+Th[i+1])/2;
          Write(T[i]:k+1:k,'   ') end;
     f:='y';WriteLn;
     for i:=1 to n do
        if abs(T[i]-Th[i])>=eps then f:='n';
     for i:=1 to n do Th[i]:=T[i];
   until f='y';
end.
```

Ця програма дозволяє встановити *n* – кількість частинок стержня {1} та *eps* – точність, з якою треба визначити остаточні температури в частинках {2}. Для *n* = 5 та *eps* = 0,001 два останні рядки таблиці мають вигляд:

таблиця 6.2

t_{left}	t_1	t_2	t_3	t_4	t_5	t_{right}
10	116665	13,330	14,997	16,663	18,332	20
10	116665	13,331	14,997	16,664	18,332	20

Це свідчить, що процес теплообміну зупинився з точністю до 0,001.

Задача 2: На кінцях однорідної теплопровідної квадратної пластини підтримується температура t_1, t_2, t_3 і t_4, а по всій довжині пластина теплоізольована. Визначити, яка температура установиться у внутрішніх частинах пластини.

? *Питання до практичної роботи*:

6.1 Яка формула відображає теплообмін у пластині (задача 2)?

6.2 Чи закінчиться теплообмін у стержні та пластині (задачі 1-2) взагалі?

✍ *Завдання 6.1* Змінити програму *Core*, щоб вона визначала час теплообміну до зупинки з указаною точністю, вважаючи, що кожен крок виконується за одну секунду.

*Завдання 6.2** Написати програму, що визначає теплообмін в частинах пластини (задача 2) з указаною точністю.

IV. РОЗВ'ЯЗУВАННЯ ЗАДАЧ З ПРОГРАМУВАННЯ ПІДВИЩЕНОЇ СКЛАДНОСТІ

IV.1 Письмовий опис розв'язку задачі.

Необхідною складовою розв'язування задачі з програмування є письмовий опис, до якого входять етапи:

Дослідження умови задачі та оголошення й опис величин;
Опис математичної моделі;
Загальний опис і доведення алгоритму;
Уточнення алгоритму та оцінка його складності;

Для ілюстрації наведемо приклад.

Задача **Num**. *(в більш ускладненому варіанті пропонувалась у 2000-2001 н.р. на обласній олімпіаді в Київській області)* Дано натуральне число *N*, записане *n* одиницями. Визначити число *k* - найменшу кількість чисел, в записі яких є лише цифри "3" і які в сумі дорівнюють числу *N*.

Технічні умови: вхідний файл *Num.dat* містить натуральне число *n* (1<*n*<*255*).
вихідний файл *Num.res* повинен містити число *k*.

а) Дослідження умови задачі та оголошення й опис величин.

1) Згідно умови задачі аргументом повинне бути число *n* цілого типу (1<*n*<*255*), результатом - число *k* (найменш можлива кількість доданків, також цілого типу).

2) Згідно ознаки подільності натурального числа на 3 робимо висновок: *дане число N представляється у вигляді суми, кожен доданок якої містить лише цифри "3" тоді і тільки тоді, коли кількість цифр числа N ділиться без остачі на число 3*. Отже, якщо *n* ділиться без остачі на 3, то представлення його згідно умови задачі можливе, інакше – ні.

3) Повний перебір всіх варіантів запису суми чисел, що мають у записі лише цифри "3" через обмеження 1<*n*<*255* неефективний.

b) Опис математичної моделі.

Для відшукання алгоритму спочатку покладемо *n* = 5 (тобто *N* = 11111) і помітимо, що воно представляється згідно умови задачі у вигляді:

$$N = \underbrace{\underbrace{(3333+3333+3333)}_{9999} + \underbrace{(333+333+333)}_{999} + \underbrace{(33+33+33)}_{99} + \underbrace{(3+3+3)}_{9}}_{\text{основна частина суми}} + \underbrace{3+3}_{\text{остача}} \qquad (1)$$

Як видно з (1), число *N* має 14 доданків, тобто *k* = 14. *Основною частиною суми* є число 12 = 3⋅4 (кількість груп доданків, що мають у записі цифру "3).

Наведена формула може бути узагальнена, і матиме вигляд:

$$N = \underbrace{(33...3+33...3+33...3)}_{\text{число з } n \text{ цифр"9"}} + ... + \underbrace{(33333+33333+33333)}_{99999} + \underbrace{(3333+3333+3333)}_{9999} + \underbrace{(333+333+333)}_{999} + \underbrace{(33+33+33)}_{99} + \underbrace{(3+3+3)}_{9} + \text{остача.} \qquad (2)$$

Остача – це число, менше за 9, воно може ділитись на 3 без остачі, або мати остачу 1 чи 2. В першому випадку це число додасть до числа *k* одиницю або двійку, у другому приведе до відповіді: $k = 0$. Очевидно, що остання формула дозволяє отримати найменш можливе значення числа *k*, тобто цілком відповідає умові задачі і може бути її математичною моделлю.

c) Загальний опис і доведення алгоритму.

Алгоритм розв'язування задачі у загальному вигляді буде такий:

```
алг NUM
   арг ціл n
   рез ціл k
поч ціл основна_частина_суми, остача
¦ основна_частина_суми ⎫
¦                 остача ⎭ згідно формул (1) та
¦ ввод n
¦ якщо mod(n, 3) <> 0
¦ ¦то k:=0
¦ ¦інакше  основна_частина_суми := 3(n - 1)
¦ ¦ визначити(остача)              ¦ допоміжний алгоритм
¦ ¦ k := основна_частина_суми + div(остача, 3)
¦ все
¦ вивод k
кін
```

Прокоментуємо форму запису алгоритму. Вона може бути довільною, в тому числі словесною чи у вигляді блок-схеми. Головне, щоб вибраний спосіб опису алгоритму був використаний доцільно й грамотно. Нам здається, що краще дотримуватись правил запису на алгоритмічній мові. При цьому слід привчати учнів дотримуватись позначень, введених у *a)* та *b)* що в коментарях (на наш погляд, обов'язкових) допоможе уникнути багатослів'я. Коментарі слід використовувати, як для опису величин, так і в тілі алгоритму.

При оголошенні величин бажано використовувати словесні імена, вживаючи знак "_" для з'єднання кількох слів в одне ім'я, наприклад: *основна_частина_суми*.

Під основним алгоритмом слід повністю за такими ж правилами навести допоміжні алгоритми, записані у вигляді *процедур* чи *функцій*.

Доведення існування та єдиності алгоритму – це теорема, тому воно записується згідно традиційних правил доведення теорем. Але спочатку слід чітко сформулювати умову й висновок, які важливо не сплутати з аргументами й результатами задачі. Для нашої задачі теорему існування її розв'язання можна сформулювати так:

***Теорема* NUM**. Якщо *дано натуральне число N, записане n одиницями (1<n<255)*, то *існує алгоритм, який визначає число k - найменшу кількість чисел, в записі яких є лише цифри "3" і які в сумі дорівнюють числу N*. (умова й висновок виділені курсивом)

Доведення випливає з 2), (1) та (2). Дійсно, ця формула представляє число *N*, як суму найменшої кількості доданків, записаних дев'ятками, а кожен доданок, у свою чергу, виділяє найменш можливу кількість чисел, записаних трійками. *Остача* - це число, менше 9 і з нього легко визначити кількість доданків, рівних 3 із допомогою функції *div*(*остача*, 3). Нарешті, процес віднімання від числа *N* доданків, записаних лише з допомогою цифр "3" скінченний. Ці міркування повністю доводять теорему NUM.

IV.2 Олімпіадні задачі з інформатики.

При підготовці учнів до олімпіад з інформатики важливо уточнити поняття "олімпіадна задача". Чим же особливі олімпіадні задачі з інформатики (програмування)? Перш за все вони значно "важчі" від тих, які передбачено розглядати згідно програми вивчення курсу інформатики. Тому задачі слід класифікувати перш за все за типами математичних моделей, тобто відповідністю певним розділам математики. Під основними типами олімпіадних задач за математичними моделями тут розуміються традиційні задачі на *повний і удосконалений перебір*, *рекурсії*, *методи оптимізації та динамічного програмування*, *координатний метод*, *графи*, *пошук виграшних стратегій* і т.д.

По рівню складності задачі можна поділити на три типи: прості ("втішальні"), звичайні та складні ("непідйомні"). Очевидно, що рівень складності – поняття відносне для різних рівнів олімпіади. Особливо слід виділити так звані задачі-пастки. Вони водночас можуть бути й простими й підвищеної складності. Справа в тому, що при знайденні вдалої математичної моделі така задача стає занадто легкою з точки зору програмування. Як приклад можна навести задачу (IV етап XIII Всеукраїнської олімпіади з інформатики):

> Дано послідовність, що складається з *2N* натуральних чисел, які можна розбити на пари таким чином, що сума чисел у всіх парах буде однаковою. Наприклад, числа послідовності *99, 23, 77, 1* можна поділити на пари *1+99=77+23*. Написати програму, яка для даної послідовності встановлює, чи можна її розбити на пари таким чином, щоб добуток чисел у всіх парах був також однаковим.

Після застосування в математичній моделі відомої всім учням теореми Вієта програма розв'язання цієї задачі записується буквально в кілька рядків.

На олімпіаді одночасно слід пропонувати задачі різного рівня складності, але ні в якому разі не можна давати задач, доведених за рахунок обмежень в умовах "до абсурду", що в попередні роки часто траплялось, особливо на IV етапі.

Можна порадити таку послідовність підготовки:

- Повторення й узагальнення традиційного матеріалу про алгоритми та програми, способи їх запису, основні структури алгоритмів, етапи розв'язування задач на комп'ютері;
- Вивчення основ алгоритмічної мови та мови програмування з розглядом і узагальненням понять числа, величини, процедури, функції, перетворення типів в інформатиці;

- Розв’язування так званих “технічних” задач, яке вимагає мінімум зусиль для побудови математичної моделі задачі, але дозволяє відпрацювати “до автоматизму” навички оперування циклами, розгалуженнями, масивами, процедурами, функціями, кодуванням, файлами вводу/виводу, ресурсами комп’ютера (пам’яттю й швидкодією), директивами компілятора, динамічною пам’яттю і т.д.;
- Розв’язування основних типів олімпіадних задач (за математичними моделями);
- Розв’язування задач на евристику.

Можна пропонувати в якості “технічних” задачі на пошук, сортування “довгу арифметику”.

При підготовці до олімпіади замість терміну “кінцевий етап” краще вживати “зрілий етап”. На ньому пропонуються комбіновані в розумінні типів задачі, а учні *обов’язково* повинні вміти працювати у автономному режимі.

IV.2 Олімпіадні задачі.

1 **Planet**. На Планеті Птахів жителі мають не по п’ять, як у людей, а по чотири пальці на кожній лапі, тому замість звичної десяткової у них поширена вісімкова система числення. Для міжпланетного спілкування математиків потрібно написати програму переведення десяткових чисел у вісімкові і навпаки.

Формат вхідних даних: У файлі *Planet.dat* записано через пропуск два числа, десяткове m_{10} та вісімкове n_8, (m_{10}, $n_8 \le 1000000000$).

Формат вихідних даних: У файлі *Planet.sol* записати через пропуск числа $k_8 = m_{10}$ та $p_{10} = n_8$.

Приклади вхідних та вихідних даних:

Planet.dat: 12345678 1234567 *Planet.sol*: 57060516 342391

Для переведення чисел з десяткової системи числення у вісімкову і навпаки найприродніше використати алгоритми, подібні алгоритмам переведення чисел з десяткової системи числення і навпаки.

2 **Calc**. Всі знають, що п’ятикласник Петя, на прізвисько Майбутній Король Математики, вміє дуже швидко рахувати. Таке відкриття зробив учитель математики, який одного разу, щоб Петя не заважав на уроці, запропонував йому обчислити суму n перших натуральних чисел, а для того, щоб не було використано калькулятор, на число наклав умову: $10^{50} \le n \le 10^{100}$. Петя написав у зошиті правильну відповідь, помітивши під нею для учителя: “Я склав справді чудову програму, але, на жаль, тут вона не помістилась“. Відтворіть програму Петі.

Формат вхідних даних: У файлі *Calc.dat* записано число n.

Формат вихідних даних: У файлі *Calc.sol* записати число s, що дорівнює $1+2+3+…+(n-1)+n$.

Приклади вхідних та вихідних даних:

Calc.dat: 100000000000000000000000000000000000000000000000000

Calc.sol:

500000000000000000000000000000000000000000000000005000000000000000000000000000000000000
0000000000000

Задача на застосування арифметичних дій над багатоцифровими числами. Варто скористатись формулою обчислення суми n перших членів арифметичної

прогресії при $a_1 = 1$ і $d = = 1$: $s = (1 + n) * n / 2$. Отже, достатньо скласти функції додавання, множення і цілочисельного ділення багатоцифрових чисел.

3 **Zero**. На числовій прямій здавна, як і мусить бути, жили НУЛЬ та упорядковані між собою цілі числа, по праву руку від нього - *n* ДОДАТНИХ, а по ліву руку - *n* ВІД'ЄМНИХ. Одного разу НУЛЬ, вважаючи себе найважливішим і прихильніше ставлячись до ВІД'ЄМНИХ чисел, бо вони ніколи не казали, що більші за НУЛЯ, вирішив поселити їх по праву руку від себе, а ДОДАТНІ числа перемістити по ліву руку. Щоб не було суперечок між ДОДАТНИМИ та ВІД'ЄМНИМИ числами, НУЛЬ став особисто робити перестановки, кожного разу міняючись місцями з одним із чисел – сусіднім, або через одне, проте, не допускаючи, щоб його запідозрили в нечесності, коли між числами однакового знаку зміниться порядок. Через деякий час, після *m*-го кроку, НУЛЬ сів перепочити і став думати, продовжувати йому реформу, чи повернутись до старого, звичного порядку, але не міг визначити, чого можна швидше досягти. Написати програму, яка допоможе НУЛЕВІ, визначивши число *k* – кількість кроків, що слід зробити для закінчення задуманої реформи.

Формат вхідних даних: У файлі *Zero.dat* записано через пропуск *n* – кількість чисел кожного знаку ($n \le 46300$) і *m* – число зроблених НУЛЕМ кроків ($m \le$ ≤ 2143782599).

Формат вихідних даних: У файлі *Zero.sol* записати шукане число *k,* або слово *error*, якщо *k* не існує.

Приклади вхідних та вихідних даних: *Zero.dat*: 3 6 *Zero.sol*: 9

Представивши у вигляді таблиці, у першому рядку якої записано послідовність кроків НУЛЯ згідно файлу *Zero.dat*, легко підрахувати шукане число *k* і з допомогою останнього рядка встановити закономірність.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-3	-3	-3	-3	-3	-3	0	1	1	1	1	1	1	1	1	1
-2	-2	-2	-2	-2	0	-3	-3	-3	-3	-3	-3	0	2	2	2
-1	0	1	1	1	1	1	0	2	2	2	2	2	0	3	3
0	-1	-1	-1	0	-2	-2	-2	-2	-2	-2	0	-3	-3	-3	0
1	1	0	2	2	2	2	2	0	3	3	3	3	3	0	-3
2	2	2	0	-1	-1	-1	-1	-1	-1	0	-2	-2	-2	-2	-2
3	3	3	3	3	3	3	3	3	0	-1	-1	-1	-1	-1	-1
	-	+	+	-	-	-	+	+	+	-	-	-	+	+	-

Очевидно, загальна кількість кроків $m + k = 2\sum_{i=1}^{n} i + n.$

4 **Hares**. На засніженій лісовій галявині, де під кущами є *n* ($1 \le n \le 100$) заячих нірок, мисливці побачили сліди їх господарів. Відчувши поряд мисливців, зайці деякий час заплутували сліди, перебігаючи від одного куща до іншого, врешті поховалися в нірках. Перед мисливцями стоїть завдання: визначити *k* - кількість зайців, що сховалось на галявині, та *x* – кількість зайців, схованки яких можна визначити, тобто уполювати. Взяти до уваги, що зайці не ховаються у чужих нірках.

Формат вхідних даних: У файлі *Hares.dat* записано у вигляді прямокутної таблиці $n \times n$, заповненої нулями та одиницями, інформацію про нірки та сліди зайців, якщо між нірками *i* та *j* ($1 \le i, j \le n$) є слід, на перетині *i*-го рядка та *j*-го стовпчика записано одиницю, інакше – нуль.

Формат вихідних даних: У файлі *Hares.sol* записати через пропуск числа *k* та *x*.

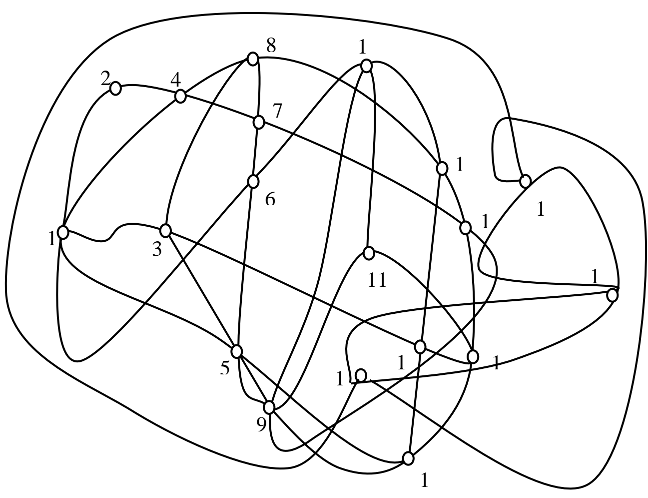
Приклади вхідних та вихідних даних: *Hares.dat*: 0 1 1 1 *Hares.sol*: 1 1

1 0 1 0

1 1 0 0

1 0 0 0

Задачу можна проілюструвати графом з вершинами у нірках (тут обмежимось неорієнтованим графом, адже в умові нічого не сказано про напрямки слідів), зобразивши сліди його ребрами, як показано на малюнку. Відомо, що у графі число непарних вершин – парне. Отже, в кожній нірці, зв'язаній непарним числом слідів, заєць або переховувався спочатку, або перебуває в даний момент. Як видно з малюнка, серед вершин під номерами 1-19 лише перша та одинадцята - непарні. Отже у одній із цих нірок переховується заєць. Зауважимо, що у випадку парності усіх вершин заєць може знаходитись у будь-якій нірці, що зробить його пошуки практично не можливими. При умові, що заєць не забігає в чужу нірку, кількість зайців, що переховуються на галявині рівна кількості зв'язних компонентів графа. Очевидно, що гарантована кількість зайців у нашому прикладі дорівнює 2, а кількість зайців, положення яких можна встановити, а отже і уполювати, дорівнює половині кількості непарних вершин, тобто 1. Задача зводиться до підрахунку кількості непарних вершин та зв'язних компонентів графа.



5. **POINTS** На колі взято *k* точок ($3 \le k \le 100$), частина з яких (можливо всі) сполучені відрізками (хордами). Написати програму, що визначає *min* - мінімальну кількість хорд, які треба додатково провести, щоб утворився вписаний у коло опуклий многокутник, для якого дані хорди є сторонами або діагоналями, та *n* – число ще не проведених діагоналей утвореного многокутника.

Формат вхідних даних:

У єдиному рядку через пропуск записано числа $x_1\ y_1\ x_2\ y_2\ \ldots x_i\ y_i\ \ldots$ – номери кінців даних хорд.

Формат вихідних даних:

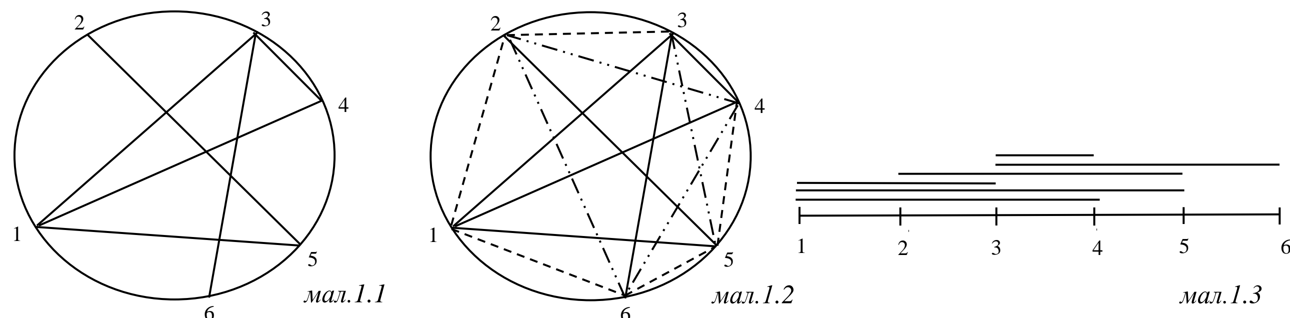
У єдиному рядку через пропуск записати числа *min* і *n*.

Приклад вхідних та вихідних даних:

Points.inp: 1 3 1 4 1 5 2 5 3 4 3 6 *Points.out*: 5 4

Вказівки до розв'язування.

Перш за все необхідно визначити і записати у порядку зростання вершини многокутника, їх кількість – *k* дорівнює числу точок, що є кінцями даних хорд (на мал. 1.1 $k = 6$). Зауважимо, що точки, не сполучені відрізками, до уваги брати зовсім не слід, адже вони не використовуються у задачі. Саме тому *k*, яке в умові означає загальну кількість точок, на першому ж етапі розв'язування становитиме кількість задіяних точок.



мал.1.1 *мал.1.2* *мал.1.3*

У даному прикладі k дорівнює c – числу проведених хорд. Сторонами даного многокутника є хорди, які не мають внутрішніх перетинів, а його діагоналями (мал. 1.2) є хорди, що перетинаються. Дані задачі можна *ізоморфно* відобразити на відрізок з координатами 1 - 6 (мал. 1.3), поєднавши допоміжними відрізками координати, відповідні номерам кінців даних хорд. Кількість даних діагоналей многокутника d дорівнює кількості пар допоміжних відрізків, які мають не пусті і не рівні одному з відрізків перерізи. Проте, можна без цього обійтись, якщо помітити, що ті з хорд, що є сторонами многокутника, на кінцях мають точки, різниця номерів яких дорівнює 1. Якщо позначити p – число проведених сторін многокутника, то $d = c - p$.

Число *min* – кількість не побудованих сторін многокутника (на мал. 1.2 зображені пунктиром) дорівнює $k\text{-}(l - d)$, тобто $min = k - p$. Кількість не проведених діагоналей обчислюється за формулою $n = k*(k - 3)/2 - d$.

Основною робочою підпрограмою може бути логічна функція перевірки на перетин двох хорд. Кожну хорду зручно записати у записі *lin = record*
begln, endln : integer end,
де *begln* та *endln* – координати початку та кінця хорд, а всі хорди – у масиві *lines*:*array*[1..100] *of lin*.

6. **TRANSP** У місті працює мережа мікроавтобусів, що має кілька маршрутів, кожен із яких не замкнений і без самоперетинів. На кожному з маршрутів є по кілька зупинок, причому, деякі стоять на перетині маршрутів. Всі зупинки пронумеровані натуральними числами от 1 до n. Написати програму, яка за даним описом транспортної мережі визначить найменшу кількість пересадок, щоб дістатись від зупинки А до зупинки В.

Формат вхідних даних:

У першому рядку через пропуск записані числа: m ($1 \le m \le 20$) - кількість маршрутів, n ($2 \le n \le 100$) - кількість зупинок, A та B – номери зупинок, для яких потрібно підрахувати кількість пересадок. Кожен із наступних m рядків складається з p_i чисел ($2 \le p_i \le 100$) – номерів зупинок на i-му маршруті.

Формат вихідних даних:

У єдиний рядок записати число k - найменшу кількість потрібних пересадок або число -1, якщо це не можливо.

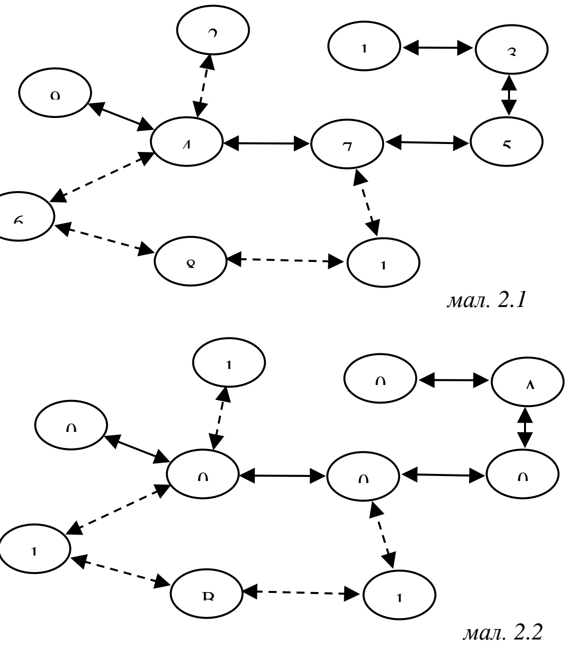
Приклади вхідних та вихідних даних:

Transp1.inp	*Transp1.out*	*Transp2.inp*	*Transp2.out*	*Transp3.inp*	*Transp3.out*
2 5 3 1	0	2 10 3 8	1	2 4 1 3	-1
1 2 3 4		1 3 5 7 4 9		1 2	
5 3		2 4 6 8 10 7		3 4	

Вказівки до розв'язування.

Зобразимо транспортну мережу у вигляді графа, де зупинками будуть пронумеровані вершини. На малюнку 2.1, який відображає другий приклад, є два маршрути із зупинками 1 – 10. Для наочності зупинки першого маршруту з'єднаємо суцільними стрілками, а зупинки другого маршруту - пунктирними стрілками. Всі зупинки першого маршруту, що містить зупинку *A*, тобто №3, помітимо числом 0. Зупинки другого маршруту, який має спільні з першим маршрутом зупинки 4 та 7, помітимо числом 1 (малюнок 2.2). Очевидно, до будь-якої зупинки, поміченої числом 1, в тому числі і до зупинки *B* (№8) від зупинки *A* можна дістатись з однією пересадкою.

Подібним чином, всі зупинки, суміжні з маршрутом, зупинки якого помічені числом 1, помітимо числом 2 і. т.д. Залишиться визначити число, яким помічені зупинки маршруту, що містить зупинку *B*. Воно і буде шуканим числом *k*.


мал. 2.1

мал. 2.2

Використаємо структуру даних *множина* (*set*). Нехай *L*[*i*] - множина зупинок, розміщених на *i*-му маршруті (тобто *L* - масив множин), *cur* - множина зупинок, до яких можна дістатись, здійснивши не більшее *step* пересадок. Побудуємо множину *next* зупинок, до яких можна дістатись, зробивши не більше (*step*+1) пересадок. По-перше, всі зупинки з множини *cur* входять до множини *next*. По-друге, якщо на якусь зупинку можна потрапити не більше ніж за *step* пересадок, і ця зупинка належить в тому числі й маршруту *i*, то тоді на будь-яку зупинку маршруту *i* можна потрапити за не більше ніж (*step*+1) пересадок. Тобто, якщо переріз множин *cur* та *L*[*i*] не пустий, всі елементи множини *L*[*i*] потрібно включити в множину *next*: *next* := *next* + *L*[*i*] (об'єднання множин).

Залишилось помітити, що коли транспортна мережа міста складається з *m* маршрутів, то або із зупинки *A* на зупинку *B* можна потрапити не більш ніж з (*m*-1)-ю пересадкою, або не можна потрапити взагалі (*A* і *B* лежать *в різних компонентах зв'язності графа*).

Таким чином, для розв'язання задачі *Transp* необхідно вміти оперувати типом "множина" (*set*), до речі, знати елементарні відомості з алгебри логіки та перевіряти граф на зв'язність немає, адже перевірка на зв'язність графа *G*, який

має множини вершин зв'язних підграфів U та V ($G=U\cup V$), здійснюється шляхом перевірки умови $U\cap V\neq\varnothing$.

7. **RELATIV** Якщо погодитись, що люди походять від Адама та Єви, то всі вони – родичі, правда, серед них небагато прямих родичів (батьки та діти, рідні брати та сестри і т.д.), більшість людей пов'язані між собою лише через спільних родичів. Написати програму, яка для n людей ($1\leq n\leq 100$), що мають родинні зв'язки, визначає, якщо це можливо, число *min* ($0\leq min\leq n/2$) - мінімальну кількість тих, що між собою не мають родинних зв'язків, і без кого решта з n людей втратять будь-які родинні зв'язки

Формат вхідних даних:

На перетині i-го рядка та j-го стопчика квадратної таблиці $n\times n$ записано число 1, якщо i-а і j-а людини є прямими родичами, і число 0, якщо для цих людей відсутній прямий родинний зв'язок.

Формат вихідних даних:

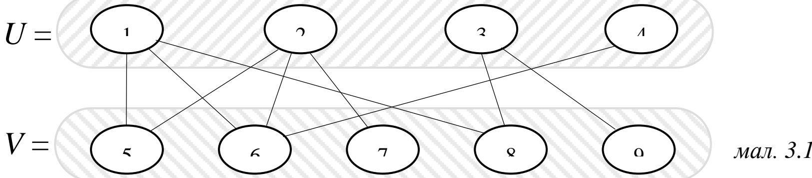
В єдиному рядку записати число *min*, або слово *"not"*, якщо це число знайти не можливо.

Приклад вхідних та вихідних даних:

Relativ.inp:
```
000011010
000011100
000000011
000001000
110000000
100100000
010000000
101000000
001000000
```
Relativ.out: 4

Вказівки до розв'язування.

Дані зручно проілюструвати на графі (див. мал. 3.1). У наведеному прикладі маємо справу із зв'язним графом. Але, як видно з малюнка, відповідно умові, він двохдольний, тобто множина всіх вершин $G=U\cup V$, де множини U=(1,2,3,4) та V=(5,6,7,8,9), причому $U\cap V=\varnothing$.

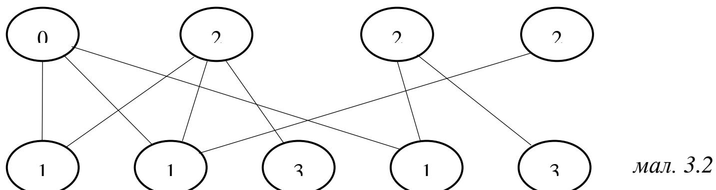


мал. 3.1

Отже, розв'язування задачі зводиться до:

1) перевірки, чи він двохдольний;
2) якщо так, то визначення серед множин U і V тієї, яка має меншу кількість елементів.

Найпростіше це можна зробити, здійснивши пошук в ширину, позначивши початкову вершину числом 0, безпосередньо пов'язані з нею вершини – числом 1, наступні вершини, безпосередньо пов'язані з вершинами, позначеними числом 1, позначити числом 2 і т.д. У наведеному графі одержимо (мал 3.2):


мал. 3.2

Всі вершини множини U будуть позначені парними числами, а вершини множини V – непарними.

8. **PARTS** У прямокутному лісі, розбитому на квадрати, заблукав мисливець. Після того, як силами місцевого населення було безрезультатно обстежено k довільно розміщених квадратів, вирішено викликати рятівників для одночасного пошуку у всіх ще не обстежених ізольованих ділянках. Написати програму, яка визначить кількість необхідних бригад рятівників.

Формат вхідних даних:

У першому рядку записано через пропуск числа m та n – довжина і ширина лісу (m, n ≤ 50), у наступних i ($i \leq m*n$) рядках по два числа x_i та y_i – координати обстежених квадратів.

Формат вихідних даних:

У єдиному рядку записати кількість необхідних бригад рятівників.

Приклад вхідних та вихідних даних:

Parts.inp: 3 3 *Parts.out*: 3
1 3
2 2
3 1
3 3

Вказівки до розв'язування.

З малюнку 4, де зображено ліс із розмірами та розміщенням обстежених квадратів (заштриховані) відповідно наведеному прикладу, видно, що ізольованих не обстежених ділянок три (вони пронумеровані). Для розв'язування задачі зручно використати структуру "черга".

мал 4

Спочатку в чергу слід занести будь-який ще не обстежений квадрат. Потім, доки черга не стане пустою, слід видаляти з неї перший квадрат і додавати в чергу всі суміжні з поточним необстежені квадрати. Черга обнулиться, коли закінчиться перша не обстежена ділянка. Далі, доки не закінчаться всі ще не обстежені квадрати, слід шукати новий не обстежений квадрат і повторювати вже щойно описане.

Програма може вийти досить громіздкою, тому бажано виділити в окремі підпрограми ВВОД даних, ВИЗНАЧЕННЯ НАЯВНОСТІ НЕ ОБСТЕЖЕНИХ КВАДРАТІВ, ДОДАВАННЯ В ЧЕРГУ та ВИЛУЧЕННЯ З ЧЕРГИ.

9. **Coll** Створено електронний збірник робіт учасників МАН Київської області у 2007-2008 н.р., де на n сторінках формату А4 посекційно розміщені рефетати, причому кожен реферат починається із нової сторінки. Першим у кожній секції міститься реферат абсолютного переможця даної секції. Але, за браком коштів, для видання вибрано лише реферати абсолютних преможців. Кожну із сторінок електронного збірника, відібрану для майбутнього друкованого збірника, вирішено зменшити до формату А5 і надрукувати їх по дві на кожній стороні аркуша формату А4. З перегорнених удвоє аркушів потрібно зшити брошури по $4*k$ ($k \leq 10$)

сторінок, а з них потім виготовити друкований збірник. Чисті сторінки потрібно помітити: “Для нотаток“. Написати програму, яка визначить l - кількість листків формату А4, потрібних для збірника, та номери сторінок електронного збірника, які будуть послідовно розміщені у друкованому збірнику.

Технічні вимоги:

Вхідний файл: ***Coll.dat***

Вихідний файл: ***Coll.sol***

Формат вхідних даних:

Перший рядок вхідного файлу містить число n ($n \leq 1000$) – кількість сторінок електронного збірника та через пропуск число k ($k \leq 10$) – кількість листків формату А4 у брошурі. Кожен наступний рядок містить через пропуск номери початкових сторінок рефератів наступної секції.

Формат вихідних даних:

Перший рядок вихідного файлу містить число l. Кожен наступний рядок містить через пропуск послідовний перелік сторінок чергової брошури, включаючи сторінки для нотаток, які помічаються літерою “н“, якщо вони є.

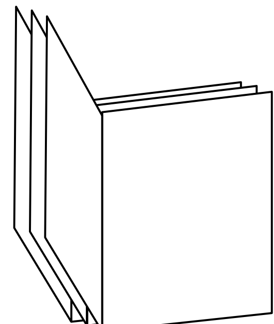
Приклад:

Coll.dat	*Coll.sol*
200 16 1 19 44 78 110 111 145 180	64 н 1 2 н н 3 4 н н 5 6 н н 7 8 н н 9 10 н н 11 12 н 144 13 14 143 142 15 16 141 140 17 18 139 138 111 112 137 136 113 114 135 134 115 116 133 32 117 118 131 130 119 120 129 128 121 122 127 126 123 124 125

Вказівки до розв’язування.

Уявимо одну брошуру, як зображено на малюнку. Для трьох листків формату А4 на обох сторонах буде 3*4 = 12 сторінок формату А5, а для n листків на обох сторонах буде, відповідно, 4*n сторінок. Легко переконатись, що нумерацію сторінок можна обчислювати за формулами, наведеними у таблиці.

Якщо не враховувати нюанси умови задачі, процедура нумерації сторінок



№ листка	перша сторона листка		друга сторона листка	
1	4*n*-2*0	1+2*0 = 1	2+2*0 = 2	4*n*-2*0-1
2	4*n*-2*1	1+2*1 = 3	2+2*1 = 4	4*n*-2*1-1
3	4*n*-2*2	1+2*2 = 5	2+2*2 = 6	4*n*-2*2-1
4	4*n*-2*3	1+2*3 = 7	2+2*3 = 8	4*n*-2*3-1
5	4*n*-2*4	1+2*4 = 9	2+2*4 = 10	4*n*-2*4-1
. . .	. . .	. . .	. . .	. . .
n	4*n*-2*(*n*-1)	1+2*(*n*-1)	2+2(*n*-1)	4*n*-2*(*n*-1)-1

однієї брошури очевидна. Не складно також створити процедуру, яка визначить k_1 – кількість заповнених сторінок майбутнього друкованого збірника. Залишиться тільки визначити кількість брошур: k_1 div 4*k, якщо k_1 mod 4*k = 0, інакше (k_1 div 4*k) +1, та k_2 - кількість сторінок формату А5, що слід додати для нотаток (якщо остання брошура містить $0 < k_2 < 4*k$ сторінок, то для нотаток слід відвести 4*k - k_2 сторінок). Залишається зауважити, що нумерація сторінок наступної брошури починається не з 1, а з наступного за номером останньої вже пронумерованої сторінки.

10. **Lotto** Для визначення грошового призу розіграшу лотареї створено прямокутний лототрон з $n \times m$ комірок, заповнений по спіралі за годинниковою стрілкою, починаючи з лівої верхньої комірки, числами від 1 до $n \times m$. Розіграш відбувається так: кинуті навмання на область лототрона дві фішки визначають діагональні комірки блоку, сума розміщених у ньому чисел становить виграш в USD. Написати програму, що встановлює суму виграшу s, якщо відомо числа m та n і координати діагональних комірок (x_1, y_1) та (x_2, y_2).

Технічні вимоги:

Вхідний файл ***Lotto.dat***

Вихідний файл ***Lotto.sol***

Формат вхідних даних:

Вхідний файл містить записані через пропуск числа m, n (<1m, n ≤ 100), x_1, y_1, x_2, y_2.

Формат вихідних даних:

Вихідний файл повинен містити число s.

Приклад:

Lotto.dat	*Lotto.sol*
10 10 3 5 8 3	1511

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	36	37	38	39	40	41	42	43	44	11
3	35	64	65	66	67	68	69	70	45	12
4	34	63	84	85	86	87	88	71	46	13
5	33	62	83	96	97	98	89	72	47	14
6	32	61	82	95	100	99	90	73	48	15
7	31	60	81	94	93	92	91	74	49	16
8	30	59	80	79	78	77	76	75	50	17
9	29	58	57	56	55	54	53	52	51	18
10	28	27	26	25	24	23	22	21	20	19

Вказівки до розв'язування.

Єдиною проблемою при розв'язуванні задачі є процедура створення прямокутного масиву по спіралі, як вказано в умові. При встановленні меж слід урахувати варіанти: 1) $x_1 < x_2$; 2) $x_1 > x_2$; 3) $x_1 = x_2$ та 4) $y_1 < y_2$; 5) $y_1 > y_2$; 6) $y_1 = y_2$, адже від цього залежать параметри вкладених циклів для обчислення числа s. Наведена таблиця ілюструє приклад.

11. **Sale** На базарі зібралось n видів товарів, що необхідно виставити на розпродаж, який вирішено зробити так: 1) усі товари, вартість яких відрізняється не більше, ніж на k копійок між собою, слід об'єднати в окрему групу і оцінити найменшим з чисел, що виражають вартості товарів цієї групи; 2) усі товари кожної групи, тобто однаково оцінені, виставити на розпродаж у окремій палатці. Написати програму, яка визначить m – найменшу кількість палаток, що слід встановити для розпродажу.

Технічні вимоги:

Вхідний файл ***Sale.dat***

Вихідний файл ***Sale.sol***

Формат вхідних даних:

Перший рядок вхідного файлу містить два записаних через пропуск числа: n – кількість видів товарів, що підлягають розпродажу (n≤100) і k (k≤10000). Другий рядок містить n чисел – початкову вартість кожного виду товару в копійках.

Формат вихідних даних:

Вихідний файл повинен містити число m.

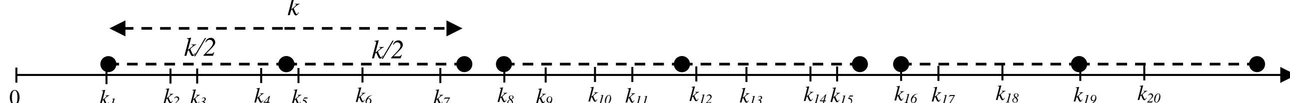
Приклад:

Sale.dat	*Sale.sol*

3 10 15 11 26	1

Вказівки до розв’язування.

Перш за все необхідно упорядкувати по зростанню вартості усіх *n* товарів. Щоб зробити корисні для пошуку алгоритму розв’язування задачі висновки, проілюструємо на числовій прямій вартості даних товарів.



Нехай найменша вартість серед товарів – k_1. Відкладемо від нього вправо відрізок довжиною k. На малюнку цей діапазон охопить групи товарів з вартістю k_1-k_7. Починаючи з k_8 вправо відкладемо наступний діапазон довжиною k. Він охопить k_8 - k_{15}. Цей процес продовжимо доти, поки не досягнемо найбільшої вартості товару.

12. **Gas.** Газотранспортна система країни, розрахована на постачання природного газу з кількох сусідніх країн, складається з *n* вузлів та *m* трубопроводів, що їх сполучають (приклад дивись на малюнку). У зв’язку із переорієнтацією імпортної політики країни окремі вузли, через які надходив природний газ, та з’єднані з ними трубопроводи необхідно демонтувати (на малюнку перекреслено). Написати програму, яка визначає число *k* - найменшу кількість додаткових трубопроводів, які треба добудувати (на малюнку показано пунктиром), щоб не порушити забезпечення газом споживачів.

Технічні вимоги: вхідний файл gas.dat, вихідний файл gas.sol.

Формат вхідних даних: у першому рядку через пропуск записано номери вузлів, через які надходить природний газ, ті з них, які слід демонтувати, взято в дужки, у кожному з *m* наступних ($1<m\leq100$) рядків через пропуск записано номери вузлів, з’єднаних *m*-им трубопроводом.

Формат вихідних даних: вихідний файл повинен містити число *k*.

Коректність вхідних даних не перевіряти.

Приклад (відповідно малюнку) :

Вхідний файл*:* gas.dat

(1) (10) 13
1 2
2 4
3 4
4 5
6 7
6 8
8 9
8 15
9 10
11 12
12 13
12 14
12 15

вихідний файл gas.sol

1

РОЗВ'ЯЗАННЯ.

Розв'язання зводиться до відшукання зв'язних компонентів графа (нехай G), побудованого згідно умови задачі. Скористаємось вхідними даними і малюнком прикладу, з якого видно, що число *k* залежить від: *а*) кількості зв'язних компонент графа G; *б*) розміщення вузлів типу 13 (таких, через які газ поступає в газотранспортну систему і які не підлягають демонтуванню). Очевидно, що вилучення вершин типу 1, 10 та відповідних ребер (на малюнку перекреслено) на розв'язок задачі не впливає, тому для скорочення програмного коду відповідний фрагмент не будемо писати. Підготуємо масив *M*1[1..*n*], у якому натуральними числами позначено вершини, що входять у зв'язні компоненти:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M[i]	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2

На малюнку один із двох для даного прикладу зв'язних компонентів виділено контуром, обмеженим точками, а у таблиці позначено цифрою "2". Після цього залишається запам'ятати в іншому масиві (наприклад *M*2) вершини, записані у першому рядку вхідного файлу без дужок (у прикладі вершина 13) і підрахувати кількість зв'язних компонент графа G, які не містять жодної вершини, записаної у *M*2. Це й буде шукане число *k*.

```
program GAS;
 const m=100;
 type arr1=array[1..m,1..m] of byte; arr2=array[1..2,1..m] of byte; lin = array[1..m] of byte;
 var  i,j,n:byte;
       fv:text;
       C:arr1;D:arr2; use : lin; t,res: integer;
procedure INPUT(var cont:arr1);
 var i,j,m : byte; st1,st2 : string;
begin
 Assign(fv, 'gas.dat'); Reset(fv);
 n:=0; m:=0; i:=1; readln(fv, st1);
 while pos('(', st1) <> 0 do
  if st1[pos('(', st1)-1]=' ' then
   delete(st1,pos('(', st1)-1,(pos(')', st1)-pos('(', st1)+2))
  else delete(st1,pos('(', st1),(pos(')', st1)-pos('(', st1)+1));
  if st1[1]=' ' then delete(st1,1,1);
 i:=1;
 if pos(' ', st1) <> 0 then
  while pos(' ', st1) <> 0 do                    begin
    st2:=copy(st1, 1, pos(' ', st1)-1);
    delete(st1, 1, pos(' ', st1)); val(st2, use[i], t); inc(i)  end
  else val(st1,use[1], t);
 if st1 <> '' then val(st1, use[i], t);
 while not EoF(fv) do   begin
   read(fv, i); readln(fv, j);
   if i > n then n:=i; if j > n then n:=j;
   cont[i,j]:=1; cont[j,i]:=1;
   inc(m)                    end;
 Close(fv)
```

```
end;
procedure CONNECT(A:arr1;n:byte;var B:arr2);
 var i,j,k,l,m,f,i1,j1:byte; check : lin;
begin
 for i:=1 to 2 do for j:=1 to n do B[i,j]:=0;
 f:=0;l:=1;
 while f=0 do                              begin
  i:=1;j:=1;
  while B[1,i]<>0 do i:=i+1;B[1,i]:=l;
  while j<=n do
   if (B[1,j]=l) and (B[2,j]=0) then begin
   for k:=1 to n do
     if A[j,k]=1 then B[1,k]:=l;
    B[2,j]:=1;j:=1                           end
   else j:=j+1;
 f:=1;for m:=1 to n do
       if B[1,m]=0 then f:=0;l:=l+1 end;
 for i:=1 to n do check[i]:=0;
  t:=1;
 for i:=1 to n do                                  begin
   for j:=1 to n do
    if D[1,j] = i then    begin
     check[t]:=j; inc(t) end;
   for i1:=1 to n do
    for j1:=1 to n do
     if (use[i1] = check[j1]) and (use[i1]<>0) then  begin
      i1:=n; j1:=n; inc(res)                          end;
    t:=1; for i1:=1 to n do check[i1]:=0        end;
  t:=0;
  for i:=1 to n do
   if d[1,i]>t then t:=d[1,i];
 Assign(fv,'gas.sol');ReWrite(fv);
 writeln(fv,t-res);
 Close(fv)
end;
begin
 Input(C); Connect(C,n,D)
end.
```

У наведеній нижче програмі **GAS** процедура **INPUT** відповідає за читання вхідних даних та створення матриці суміжності графа G і списку вузлів-постачальників газу після модернізації газотранспортної системи. Процедура **CONNECT** визначає зв’язні компоненти графа G, зберігає інформацію про них і обчислює число k.

13. **Fift** Написати програму, яка визначає, чи ділиться на 15 натуральне число n, подане у двійковій системі числення, якщо кількість двійкових розрядів не більше 10000.

Технічні вимоги

Вхідний файл: ***fift.dat***, вихідний файл: ***fift.res***

Формат вхідних даних:

Вхідний файл містить послідовно записані через пропуск цифри числа n.

Формат вихідних результатів:

У вихідному файлі повинно міститись слово “yes“ або слово “no“.

Приклад: *fift.dat* *fift.sol*

fift.dat	fift.sol
100011111100011000000011111000	yes
100011101100011000000011111011	no

Вказівки щодо розв’язування

Відомо, що коли сума цифр числа, записаного в 16-ковій системі числення, ділиться на 15, то це число кратне 15.

Доведення: Спочатку число слід представити в 16-ковій системі числення, взявши до уваги, що кожні чотири сусідні розряди будь-якого двійкового числа записуються однією цифрою 16-кової системи числення, наприклад: $100011111100011000000011111000_2 = 11F8C0F8_{16}$, $11011101001_2 = 6E9_{16}$ і т.д.

$$N = \sum_{k=0}^{n} a[k] \cdot 16^k = a[0] + \sum_{k=1}^{n} 16 \cdot a[k] \cdot 16^{k-1} = \sum_{k=0}^{n} a[k] + \sum_{k=1}^{n} 15 \cdot a[k] \cdot 16^{k-1}.$$

Другий доданок ділиться на 15, бо має множник 15. Отже *N* ділиться на 15, якщо й перший доданок, що розписується, як $\sum_{k=0}^{n} a[k] = a_0 + a_1 + a_2 + ... + a_{n-1} + a_n$ і становить суму цифр числа *N*, ділиться на це число.

Враховуючи, що коли число, записане в одній позиційній системі числення, кратне 15, то воно буде кратним 15 і в будь-якій іншій позиційній системі числення, зауважимо, що для розв’язування задачі Fift немає потреби у переводі числа *n* в 16-кову систему числення. У програмі можна обійтись 10-ковою системою числення, замінивши кожну двійкову тетраду аргументу її десятковим представленням.

```
program Fift;
 var i,n,m,k,y,s,code:integer;
     A,H:array[0..10000]of integer;
     st:string;{ch:string[1];}res:char;
     f:text;
 procedure Inpt;
 begin
  Assign(f,'Fift10.dat');Reset(f);
  Read(f,st);Close(f);
 end;
 procedure Transf;
 begin
  n:=Length(st);
  for i:=n downto 1 do
   Val(Copy(st,i,1),A[n-i],code);
  k:=(n div 4)+1;
  for i:=0 to k-1 do
   begin
    m:=4*i;
    H[i]:=A[m]+A[m+1]*2+A[m+2]*4+A[m+3]*8
   end;
 end;
 procedure Result;
 begin
```

```
 s:=0;
 for i:=0 to k-1 do
  s:=(s+H[i]);
 if s mod 15=0
  then res:='y'
  else res:='n'
end;
procedure Outp;
begin
 Assign(f,'Fift10.res');ReWrite(f);
 Write(f,res);Close(f);
end;
begin
 Inpt;
 Transf;
 Result;
 Outp
end.
```

14. **Least** На координатній площині знаходиться *n* точок, які є вершинами деякого опуклого многокутника або належать його внутрішній області. Написати програму, що визначає *k* – кількість вершин цього многокутника та їх координати.

Технічні вимоги

Вхідний файл: ***least.dat***, вихідний файл: ***least.res***

Формат вхідних даних:

Перший рядок вхідного файлу повинен містити число *n* – кількість заданих точок ($3 \le n \le 100$). У кожному з наступних *n* рядків треба записати два розділених одним пропуском числа *x* та *y* – координати точок на площині ($-100 \le x,\ \ y \le 100$).

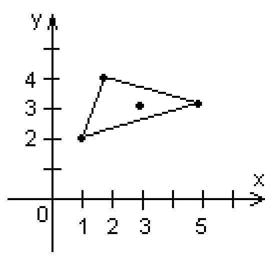
Формат вихідних результатів:

Перший рядок вихідного файлу містить єдине число *k* – кількість вершин отриманого опуклого многокутника. Наступні *k* рядків містять по два числа *x* та *y* – координати вершин опуклого многокутника, розділених одним пропуском, у порядку обходу проти годинникової стрілки, починаючи з вершини, розміщеної найнижче і найлівіше.

Приклад:

least.dat	*least.res*
4	3
3 3	1 2
1 2	5 3
5 3	2 4
2 4	

Вказівки щодо розв’язування



Розглянемо розв’язок, який випливає з фізичних міркувань. Нехай в підлогу забито *N* цвяхів; почнемо обв’язувати їх ниткою, будуючи опуклу оболонку. Спочатку прив’яжемо нитку до найлівішого і найнижчого цвяха, він обов’язково належить опуклій оболонці. Далі почнемо повертати нитку проти годинникової стрілки, поки не зустрінемо на ще один цвях. Такими поворотами обв’яжемо всі цвяхи - вершини

опуклої оболонки. Правильність даного підходу практично очевидна, але як реалізувати його програмно? Дуже просто, якщо здогадатися, що кожного разу нове положення нитки відхиляється на мінімальний кут. Припустимо, що спочатку нитка проходила горизонтально. Тоді нову точку опуклої оболонки потрібно обирати такою, щоб кут між новим і старим положенням нитки (сторони многокутника) був найменшим із можливих. Скористаємось відомостями з векторної алгебри. Вектором називається напрямлений відрізок із точки *(0; 0)* в точку *(x; y)*. Нормаллю векторної пари (x_1, y_1) і (x_2, y_2) називається вираз $x_1 y_2 - y_1 x_2$, який додатний при переході від вектора (x_1, y_1) до вектора (x_2, y_2) проти годинникової стрілки і від'ємний при переході від вектора (x_1, y_1) до вектора (x_2, y_2) за годинниковою стрілкою. Тоді кут між векторами (x_0, y_0) і (x_1, y_1) менший, ніж кут між (x_0, y_0) і (x_2, y_2), якщо $x_0 y_1 - y_0 x_1 > 0$ і $x_1 y_2 - y_1 x_2 > 0$ (помітимо, що вектор (x_1, y_1) лежить між векторами (x_0, y_0) і (x_2, y_2)). Виходячи з цього, легко скласти програму побудови опуклого многокутника найменшої площі, який покриває всі задані точки.

```
program Least;
  type Point = record
          x,y: integer;
        end;
  var  f: text;
       Points: array[1..100] of Point;
       n, H: integer;
       x0, y0: integer;
       i, j: integer;
      shell: array[1..100] of Point;
      used: array[1..100] of boolean;
begin
 Assign(f,'least.dat'); Reset(f);
 ReadLn(f,n);
 for i:=1 to n do
   with Points[i] do
     ReadLn(f,x,y);
 close(f);
 Assign(f,'least.res'); Rewrite(f);
 i:=1;
 for j:=2 to n do
   if (Points[j].y<Points[i].y) or
     ((Points[j].y=Points[i].y) and (Points[j].x<Points[i].x))
   then i:=j;
 x0:=1;
 y0:=0;
 H:=0;
 repeat
   used[i]:=true;
   H:=H+1;
   shell[H]:=Points[i];
   i:=1;
   while (i<=n) and not(x0*(Points[i].y-shell[H].y)-y0*(Points[i].x-shell[H].x)>0) do
     i:=i+1;
   for j:=i+1 to n do
```

```
    if (x0*(Points[j].y-shell[H].y)-y0*(Points[j].x-shell[H].x)>=0) and
    ((Points[j].x-shell[H].x)*(Points[i].y-shell[H].y)-(Points[j].y-shell[H].y)*(Points[i].x-
shell[H].x)>0)
    then i:=j;
   x0:=Points[i].x-shell[H].x;
   y0:=Points[i].y-shell[H].y;
 until (i>n) or used[i];
 WriteLn(f,H);
 for i:=1 to H do
   with shell[i] do
  WriteLn(f,x,' ',y);
  Close(f);
end.
```

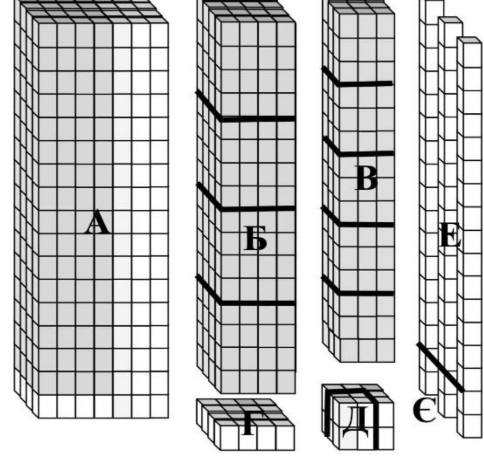
15. **CubLad** (35 *балів*) Куби, на які розрізають прямокутний паралелепіпед, укладають в один ряд так, щоб вони дотикались гранями. Написати програму, яка визначить найкоротший за довжиною ряд, що може при цьому утворитись.

Технічні вимоги

Вхідний файл: ***cublad.dat***, вихідний файл: ***cublad.res***

Формат вхідних даних:

Вхідний файл містить у одному рядку через пропуск три числа a, b і c ($a, b, c \in N,\ a, b, c \le 100$) – виміри даного паралелепіпеда.



Формат вихідних результатів:

У вихідному файлі записати через пропуск числа k – найменш можливу кількість кубів та l – довжину одержаного найкоротшого за довжиною ряду.

Приклад:

cublad.dat	*cublad.res*
17 7 4	80 104

Вказівки щодо розв'язування.

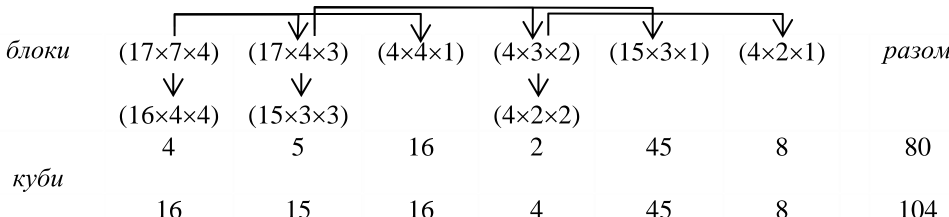
Як відомо, подібна за формулюванням задача про прямокутникик, який треба розрізати на найменшу кількість квадратів, розв'язується з допомогою алгоритма Евкліда (визначення найбільшого спільного дільника двох чисел). Але застосувати його для розв'язування задачі **CubLad** не можна.

Очевидно, що довжина ряду буде найменшою, коли паралелепіпед розрізати на найменш можливу кількість кубів (пропонуємо довести це самостійно). На кожному кроці слід виділяти куби з максимальною стороною. Для продовження аналізу алгоритма скористаємось зображеним на малюнку прикладом (блок **А**), який має виміри 17×7×4. Першим слід вирізати блок **Б**, що містить 4 куби із стороною 4. Залишок слід розрізати на два паралелепіпеди, один з яких має висоту 1 (блок **Г**). Його розміри можуть бути 1×4×4 або 1×4×7. Як видно з малюнка, оптимальніший перший варіант, бо він дає 16 кубів, а другий 28 кубів. Тоді об'єднання блоків **В**∪**Д**∪**Е**∪**Є** дозволяє вирізати ще 5 кубів із стороною 3, після чого слід згрупувати блоки **Д** і **Є,** які дадуть два куби 2×2×2 із

стороною 2 та 8 кубів 1×1×1, разом 10 кубів. Останній блок **E** дасть 45 кубів 1×1×1. Всього одержиться 80 кубів, що утворять ряд довжиною 4×4+5×3+2×2+(16+8+45)×1=104.

Для розв’язування задачі **CubLad** зручно скористатись чергою (структура “першим зайшов – першим вийшов“) з використанням динамічної пам’яті. Кожен елемент черги повинен містити три числа – виміри чергового елемента – прямокутного паралелепіпеда.

Опишемо це на розглянутому вище прикладі (див. схему), де стрілками на схемі показано елементи черги, які виникають у результаті розрізання її



блоки	(17×7×4)	(17×4×3)	(4×4×1)	(4×3×2)	(15×3×1)	(4×2×1)	*разом*
	(16×4×4)	(15×3×3)		(4×2×2)			
куби	4	5	16	2	45	8	80
довжина	16	15	16	4	45	8	104

поточного елемента, наприклад, у результаті розрізання прямокутного паралелепіпеда 17×7×4 виникають прямокутні паралелепіпеди 16×4×4, 17×4×3 та 4×4×1.

Одержана схема ідентична орієнтованому дереву (див. малюнок), але застосувати його обхід немає сенсу, адже сам граф в умові задачі невідомий, а в процесі його побудови автоматично визначаються й числа l та k.

```
program CubLad;
 var x,y,z,i,res,p,min,l:integer;
 f:text;
 procedure Inpt;
  var f:text;
 begin
  Assign(f,'cublad1.dat');
  Reset(f);
  Read(f,x,y,z);
  Close(f)
 end;
 procedure Outp;
  var f:text;
 begin
  Assign(f,'cublad.res');
  Rewrite(f);
  Write(f,res,' ',l);
  Close(f)
 end;
 procedure Decomp(a,b,c:integer);
 begin
  if (a=0)or(b=0)or(c=0)
   then exit;
  if (a=b)and(b=c)then begin
```

```
  res:=res+1;
  l:=l+a;exit     end;
 if (a=1)or(b=1)or(c=1)then begin
  if a=1 then begin
   res:=b*c+res;
   l:=l+b*c;
   exit     end;
  if b=1 then begin
   res:=a*c+res;
   l:=l+a*c;
   exit     end;
  if c=1 then begin
   res:=a*b+res;
   l:=l+a*b;
   exit     end;
  exit              end;
  if a>b then begin
   p:=a;
   a:=b;
   b:=p     end;
  if b>c then begin
   p:=b;
   b:=c;
   c:=p     end;
  if a>b then begin
   p:=a;
   a:=b;
   b:=p     end;
 if (b mod a)>(c mod a)
  then min:=c mod a
  else min:=b mod a;
 res:=res+((c div a)*(b div a));
 l:=l+((c div a)*(b div a)*a);
 if min=b mod a
  then          begin
    Decomp(a,b mod a,(c-(c mod a)));
    Decomp(a,b,c mod a)end
  else               begin
   Decomp(a,b mod a,c);
   Decomp(a,(b-(b mod a)),c mod a)end;
 end;
begin
 Inpt;
 res:=0;
 l:=0;
 Decomp(x,y,z);
 Outp
end.
```

16. **Animal** У квадратній матриці, заповненій різноманітними символами, приховано деякий малюнок, який може містити зображені однаковими символами окремі точки, горизонтальні чи вертикальні лінії та блоки,

обмежені контурами з вертикальних і горизонтальних ліній. Написати програму, що розпізнає малюнок за даним ключем.

Технічні вимоги

Вхідний файл: ***anim.dat***, вихідний файл: ***anim.res***

Формат вхідних даних:

Вхідний файл у першому рядку містить ключ – символ, що обмежує малюнок. У наступних рядках розміщено саму квадратну матрицю розмірами $n\times n$ ($3\leq n\leq 100$).

Формат вихідних результатів:

Вихідний файл повинен у вигляді символів "*" зображати розпізнаний малюнок.

Приклад: *anim.dat*

```
a
cvbaaabbbb
bbaabaabbb
bbabklanbm
sdaabaabbb
bbbaaabbyn
kibbabbpxb
bbbbabbbbb
baaaaaaabb
```

anim.res

```
  ***
 *****
 *****
 *****
  ***
   *
   *
*******
```

Вказівки щодо розв'язування.

Якщо уважно прочитати умову, то стане зрозумілим, що елементи малюнка: точки, лінії і контури зображені з допомогою символа, записаного у першому рядку вхідного файла. Залишається лише розпізнати зображені таким же символом блоки, тобто замкнені контури.

У авторських тестах малюнки містять не більше одного блоку, тому "логічна" складова задачі незначна і зводиться до вміння написати програму, яка заповнює один контур, що є класичною задачею, відомою юним програмістам навіть із невеликим практичним досвідом.

Але задача також включає достатньо потужну "технічну" складову, що полягає у дуже абстрактному формулюванні умови та "незручному" символьному представленні вхідних даних. Саме це, а також провокуючий на ускладнення математичної моделі натяк "обмежені контурами" дозволяють оцінити задачу досить високо.

```
program Animal;
 type Bin=array[1..255,1..255]of 0..1;
 var  n:integer;
      x,y:byte;
      Ar:array[1..255]of string;
      Arr:Bin;
      key:char;
      f:text;
 procedure Inpt;
 begin
  Assign(f,'Animal1.dat');Reset(f);n:=1;
  ReadLn(f,key);
  repeat
   ReadLn(f,Ar[n]);Inc(n);
  until EoF(f);
  Dec(n);Close(f);
```

```
end;
procedure Change;
 var i,j,k:byte;
     st:string;
begin
 for i:=1 to n do
  begin
   st:=Ar[i];
   for j:=1 to n do
    if Copy(st,j,1)=key
     then Arr[i,j]:=1
     else Arr[i,j]:=0
  end;
end;
procedure In_Cont(Arr:Bin;N:byte;var k,l:byte);
 var i,j:integer;
begin
 for i:=1 to N do
  for j:=1 to N do
   if Arr[i,j]=1 then
    begin
     k:=i+1;l:=j+1;
     i:=N; j:=N;
    end;
end;
procedure Fill_Cont(N,k,l:byte;var Arr:Bin);
 var i,j:integer;
begin
 i:=k;j:=l;
 if Arr[i,j]=0 then
  begin
   Arr[i,j]:=1;
   i:=i-1; Fill_Cont(N,i,j,Arr);i:=i+1;
   j:=j+1; Fill_Cont(N,i,j,Arr);j:=j-1;
   i:=i+1; Fill_Cont(N,i,j,Arr);i:=i-1;
   j:=j-1; Fill_Cont(N,i,j,Arr);j:=j+1;
  end;
end;
procedure Back;
 var i,j:integer;
     st:string;
begin
 for i:=1 to n do
  begin
   Ar[i]:='';
   for j:=1 to n do
    if Arr[i,j]=1
     then Ar[i]:=Ar[i]+'*'
     else Ar[i]:=Ar[i]+' '
  end
end;
procedure Outp;
```

```
 var i:integer;
begin
 Assign(f,'Animal.res');Rewrite(f);
 Write(f,'');
 Close(f);
 Assign(f,'Animal.res');Append(f);
 for i:=1 to n do
  WriteLn(f,Ar[i]);
 Close(f)
end;
begin
 Inpt;
 Change;
 In_Cont(Arr,N,x,y);
 Fill_Cont(N,x,y,Arr);
 Back;
 Outp
end.
```

17 Turn. Бажаючі отримати квитки на концерт вишикувалися у чергу із N - чоловік, кожен із яких має намір викупити лише 1 квиток. Оскільки працювала лише одна каса, продаж квитків відбувався повільно, а люди у черзі приходили у відчай. Однак, найкмітливіші дійшли до висновку, що декілька квитків в одні руки касирка продає набагато швидше, ніж коли ті ж квитки продаються поодинці. Тому вони запропонували тим, хто стоїть поруч, віддавати гроші першому з них, щоб той викупив квитки на всіх. Для боротьби зі спекулянтами касирка продавала не більше, ніж по 3 квитки в одні руки. Таким чином домовитися між собою мали можливість лише 2 або 3 людини. Скласти програму, яка визначатиме найменший час, за який касирка зможе продати квитки усім бажаючим. Відомо, що на продаж і-ій людині із черги одного квитка касирка витрачає A_i секунд, на продаж двох квитків - B_i секунд, а трьох квитків - C_i секунд. Зверніть увагу, що квитки на групу завжди купує перший із них, а зайвих квитків не купує ніхто.

Технічні вимогоги

Формат вхідних даних: Перший рядок містить єдине число N - кількість чоловік ($1 \le N \le 5000$), що утворюють чергу. Нумерація людей у черзі починається від каси. Наступні N- рядків містять трійки натуральних чисел A_i, B_i, C_i, значення кожного з яких не перевищує 3600.

Формат вихідних даних: Рядок містить одне число - найменший час, за який касирка зможе продати квитки усім бажаючим.

Важливо відчути ланцюжок: *математична модель – цільова функція – програма – тести*. Спробуємо прослідкувати його, скориставшись програмою і деякими з тестів.

Вказівки щодо розв'язування.

Цю задачу найефективніше розв'язувати методом динамічного програмування, у основі якого лежить *принцип оптимальності* та поняття *цільової функції*. Основні труднощі виникають на етапі формулювання рекурентних співвідношень, які виражають цільову функцію. Вони пов'язані з особливостями умови задачі та формою представлення вхідних даних і шуканого результату. Формула, що виражає цільову функцію – це свого роду функціональне рівняння, складаючи яке, ми, іншими словами, описуємо його розв'язок, що для учнів із пересічними математичними здібностями та підготовкою нелегко. Щоб частково спростити цю проблему, ми, по-перше, однаково позначали таблиці: для зберігання вхідних даних – *G* або *g*, для формування оптимального рішення *F* або *f*. Застосування елементів таблиці *f*, наприклад**,** ***f[i]***:=*Min(**f[i-1]**+g[1,i], Min(**f[i-2]**+g[2,i-1], **f[i-3]**+g[3,i- -2]))* у програмах має ще й інше значення, це – функціональне рівняння, залежне від констант *g*(*i*), рекурентних звернень *f*(*i*-1), *f*(*i*-2) і т.д. у поєднанні з функціями *Max* або *Min*. де права частина рівняння виражає розв'язок рівняння. По-друге, для глибшого усвідомлення методу динамічного програмування, ми радимо користуватись готовою програмою до задачі динамічного програмування і ручним покроковим виконанням окремих тестів до цієї програми. При уважному спостереженні відповідних таблиць та процедури *TimePurchase* стане очевиднішим і зрозумілішим вигляд рекурентних співвідношень, а набутий досвід допоможе у розв'язуванні наступних задач на метод динамічного програмування та складанні тестів до них.

```
program Turn;
 var i,n:integer;g:array[1..3,1..5000] of integer;
      f:array[1..5000] of longint;
procedure Inp;
begin
  Assign(input,'t_8.tst');Reset(input);
  ReadLn(input,n); i:=1;
  while i<=n do  begin
    Read(input,g[1,i]);
    Read(input,g[2,i]);Read(input,g[3,i]);
    i:=i+1          end;
   Close(input)
end;
procedure Outp;
begin
   Assign(output,'t_.ans');ReWrite(output);
   Write(output, f[n]);
   Close(output)
end;
function Min(x,y:longint):longint;
begin  if x>y
           then Min:=y
           else Min:=x; end;
procedure TimePurchase;
begin
  f[1]:=g[1,1]; f[2]:=Min(f[1]+g[1,2], g[2,1]);
  f[3]:=Min(f[2]+g[1,3], Min(f[1]+g[2,2], g[3,1]));
```

```
  for i:=4 to n do
    f[i]:=Min(f[i-1]+g[1,i], Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))
end;
begin
  Inp;
  TimePurchase;
  Outp;
end.
```

Нижче наведено кілька тестів до задачі та аналіз, що дозволяє зрозуміти процес формування значень цільової функції.

j/i	1	2	3
1	**5**	**10**	**15**
2	**2**	**10**	**15**
3	**5**	**5**	**5**
4	**20**	**20**	**1**
5	**20**	**1**	**1**

таблиця g (для тесту e t_1.tst)

таблиця f (для тесту et_1.tst)

i	f[i]			
1	g[1,1]	5	5	5
2	Min(f[1]+g[1,2], g[2,1])	Min(5+2, 10)	7	7
3	Min(f[2]+g[1,3], Min(f[1]+g[2,2], g[3,1]))	Min(7+5, Min(5+10, 15))	Min(12,Min(15,15))	12
4	Min(f[i-1]+g[1,i],Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))	Min(12+20,Min(7+5,7+15))	Min(32,Min(12,22))	12
5	Min(f[i-1]+g[1,i],Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))	Min(12+20,Min(12+20,7+5))	Min(32,Min(3,12))	**12**

j/i	1	2	3
1	**1**	**10**	**10**
2	**2**	**3**	**4**
3	**2**	**3**	**4**
4	**2**	**3**	**4**

таблиця g (для тесту t_2.tst)

таблиця f (для тесту t_2.tst)

i	f[i]			
1	g[1,1]	1	1	1
2	Min(f[1]+g[1,2], g[2,1])	Min(1+3, 10)	4	4
3	Min(f[2]+g[1,3], Min(f[1]+g[2,2], g[3,1]))	Min(4+2, Min(1+3, 10))	Min(6,Min(4,10))	4
4	Min(f[i-1]+g[1,i],Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))	Min(4+2,Min(4+3,1+4))	Min(6,Min(7,5))	**5**

j/i	1	2	3
1	**1**	**2**	**3**
2	**1**	**1**	**3**
3	**1**	**2**	**3**
4	**1**	**2**	**1**
5	**1**	**2**	**3**
6	**1**	**2**	**3**

таблиця g (для тесту t_3.tst)

таблиця f (для тесту et_3.tst)

i	f[i]			
1	g[1,1]	1	1	1
2	Min(f[1]+g[1,2], g[2,1])	Min(1+1,2)	2	2
3	Min(f[2]+g[1,3], Min(f[1]+g[2,2], g[3,1]))	Min(2+1, Min(1+1, 3))	Min(3,Min(2,3))	2
4	Min(f[i-1]+g[1,i],Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))	Min(2+1,Min(2+2,1+3))	Min(3,Min(4,4))	3
5	Min(f[i-1]+g[1,i],Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))	Min(3+1,Min(2+2,2+3))	Min(4,Min(4,5))	4
6	Min(f[i-1]+g[1,i],Min(f[i-2]+g[2,i-1], f[i-3]+g[3,i-2]))	Min(4+1,Min(3+2,2+1))	Min(5,Min(5,3))	**3**

IV.3 ВІД ЗАДАЧІ ДО ЗАДАЧІ

Перед тренерами учасників учнівських олімпіад з інформатики, а точніше, з програмування, постійно виникають питання, *якими повинні бути задачі* та *як навчати їх розв'язуванню*. Питання зрозумілі і незрозумілі водночас, якщо враховувати, що програмування – це нетрадиційний для школи предмет, який вимагає вільного володіння математичним апаратом, а задачі важко розділяти за віковими категоріями. Перш за все треба розібратись, які розділи математики та в якому порядку і поєднанні доцільно розглядати у процесі підготовки юного програміста, на яких типових задачах слід починати шліфовку знань, умінь і практичних навичок? На наш погляд, відповідь на це питання дуже вдало дано у [3], де після розгляду арифметики багаторозрядних чисел поставлено комбінаторні алгоритми, а вже потім розглянуто теми перебору і методів його скорочення, алгоритми на графах, обчислювальну геометрію і т.д. Тому і ми тут, зовсім не випадково, зосередимось на розгляді комбінаторних алгоритмів, взявши у якості робочого матеріалу дуже цікаву задачу з [3]. Додатково до цього спонукала ще й необхідність своєрідної післямови до ІІІ етапу Всеукраїнської олімпіади-2005 з інформатики у Київській області, де розглянута нижче задача була представлена.

Задача 18. ([3], стор. 75) На смужці клітинкового паперу висотою в одну і довжиною *N* клітинок деякі клітинки зафарбовані в чорний і білий кольори.

Кодом клітинки є послідовність чисел – кількості ідучих підряд чорних клітинок зліва направо. Наприклад, для наведеної смужки кодом буде послідовність 2, 3, 2, 8, 1. При цьому кількість білих клітинок, які розділяють групи чорних ніде не враховуються (головне, щоб дві сусідні групи розділені принаймні однією білою клітинкою). Одному й тому ж коду можуть відповідати кілька смужок, наприклад наведеному коду відповідає й така смужка:

Задача полягає в тому, щоб знайти кількість смужок довжини *N*, які відповідають заданому коду.

Вхідні дані. У єдиному рядку файлу файла *input.txt* записано число *N* – довжина смужки ($1 \le N \le 200$), потім число *K* ($0 \le K \le (N+1)/2$) – кількість чисел у коді і далі *K* чисел, що визначають код.

Вихідні дані. У вихідний файл *output.txt* записується кількість смужок довжини *N*, які відповідають заданому коду.

а) ***Дослідження умови та виділення параметрів***. З умови видно, що задача має комбінаторний характер, причому на підрахунок смужок, а не на їх генерацію, отже слід знайти потрібні формули. На користь цього також свідчить можливість надто великого числа смужок, генерація яких може виходити за межі стандартних розрядів та допустимих проміжків часу.

Зробимо позначення: *s* - сума чисел коду та числа *K* - 1, $Ns = N - s$ - кількість незафарбованих клітинок, які слід розмістити серед $K1 = K + 1$ місць. Кількість варіантів такого розподілу, яку позначимо *KS* і буде шуканою відповіддю. З огляду на це, запитання задачі 3 можна сформулювати інакше:

*знайти число KS - кількість способів розміщення Ns пропусків на K*1 *місцях, щоб їх сума дорівнювала Ns*. (*)

Зауваження. Слід звернути увагу на те, що у [1] за s помилково взято суму чисел коду та числа *K*. Там же, у вказівці до розв'язування, помилково стверджується, що залишається підрахувати *число розміщень N – s по відомій кількості місць – K* + 1. Але жодна із формул - A_N^k = = N^k (розміщень з повтореннями) та $A_N^M = N! / (N - M)!$ (розміщень без повторень) у нашому випадку не підходить. Очевидно, задача відноситься до категорії складних (не випадково вона наведена під №15 із 17, включених у якості додаткових до розділу "Комбінаторні алгоритми"), тому, на нашу думку, короткої вказівки (причому з указаними неточностями та обмеженнями, що потребують застосування багаторозрядної арифметики) зовсім не достатньо. Поставимо перед собою завдання: *знайти ефективний алгоритм пошуку числа KS, яке за рахунок умов* $1 \le N \le 200$ *та* $0 \le K \le (N+1)/2$ *може бути багаторозрядним числом*. Забігаючи наперед, зауважимо, що в процесі розв'язання цього завдання доведеться шукати відповіді й на інші, які виникнуть по ходу і з яких випливатимуть важливі наслідки.

b) ***Пошуки математичної моделі***. Як уже згадувалось вище, формули числа розміщень застосувати до задачі не можна. Але при визначенні кількості варіантів завжди виникає спокуса використати одну з комбінаторних формул. Нескладний аналіз допоможе представити шукане число у вигляді числа комбінацій без повторень, але з огляду на наступні міркування, це буде зроблено значно нижче.

Справа в тому, що математичну модель задачі вибирають здебільшого залежно від кількох обставин: обмежень на вхідні та вихідні дані, віку виконавців або стану їх математичної підготовки. Ми поки що не будемо звертати увагу на формат даних, обмежившись стандартними типами цілих чисел, зокрема типом *longint*[31]. Дві інші обставини проігнорувати ніяк не можна. За браком достатньої математичної підготовки, учні змушені користуватись спрощеною моделлю задачі, що приводить до неповного чи неефективного розв'язку. Виходячи з цього, можна стверджувати, що учні, розв'язуючи подібну задачу, майже обов'язково, підуть не шляхом застосування комбінаторики, яка вивчається, на жаль тільки в 11 класі, причому без достатнього закріплення та застосування.

Отже, прослідкуємо за емпіричним ходом пошуку робочих формул. Зупинимось на прикладі смужки при *N* = 8, *K* = 3, код 1, 2, 1. Як початкову, візьмемо смужку (малюнок 1):

мал. 1

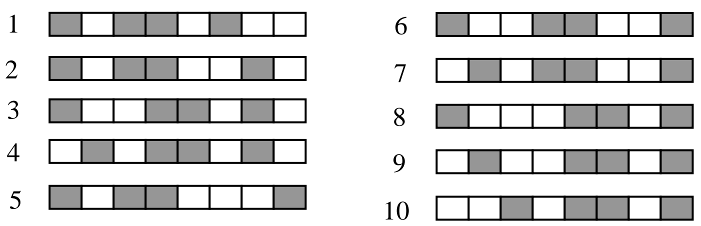
Зробимо підрахунок, попередньо згенерованих варіантів смужок.Як видно з малюнка 2, таких смужок 10. Як уникнути генерування смужок? З (*) випливає, що задача для смужки, зображеної на малюнку 1, зводиться до варіантів, які видно з наступної таблиці:

№	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

[31] Маємо на увазі тип *longint* для мови Turbo Pascal.

1	0	0	0	0	0	0	1	1	1	2
2	0	0	0	1	1	2	0	0	1	0
3	0	1	2	0	1	0	0	1	0	0
4	2	1	0	1	0	0	1	0	0	0

таблиця 1



мал. 2

Очевидно, що *KS* буде функцією від двох величин – *Ns* та *K*+1, тобто *KS*(*Ns*,*K*+1), у нашому прикладі *KS*(2, 4). Спробуємо визначити формулу для знаходження *KS*(2, 4), а на її основі формулу для визначення *KS*(*i*, *j*). Очевидно, що $KS(1,j) = j$, $KS(i,1)=1$, $KS(0,j)=1$. (1)

Отже: $KS(2, 1) = 1$;

$KS(2, 2) = KS(2, 1) + KS(1, 1) + KS(0, 1) = 1 +1 + 1 = 3$;
$KS(2, 3) = KS(2, 2) + KS(1, 2) + KS(0, 2) = 3 + 2 + 1 = 6$; (2)
$KS(2, 4) = KS(2, 3) + KS(1, 3) + KS(0, 3) = 6 + 3 + 1 = 10$

Міркуючи подібним чином, визначимо *KS*(3, 4) і одержимо (3), а потім, узагальнивши (1), (2) та (3), запишемо рекурентні співвідношення для визначення *KS*(*i*, *j*), які пропонуємо довести самостійно. Зауважимо, що кількість доданків у сумі дорівнює *i*+1.

$KS(3, 1) = 1$;
$KS(3, 2) = KS(3, 1) + KS(2, 1) + KS(1, 1) + KS(0, 1) = 1 +1 + 1 + 1 = 4$;
$KS(3, 3) = KS(3, 2) + KS(2, 2) + KS(1, 2) + KS(0, 2) = 4 + 3 + 2 + 1 = 10$; (3)
$KS(3, 4) = KS(3, 3) + KS(2, 3) + KS(1, 3) + KS(0, 3) = 10 + 6 + 3 + 1 = 20$

Отже, остаточно маємо:

$KS(1, j) = j$, $KS(i, 1) = 1$, $KS(0, j) = 1$,
$KS(i,j) = KS(i,j-1)+KS(i-1,j-1)+KS(i-2,j-1)+\ldots+KS(1,j-1)+KS(0,j-1)$ (4)

Останні рекурентні співвідношення дають змогу скласти алгоритм розв'язування задачі, але для його спрощення проведемо додаткове дослідження з допомогою Microsoft Excel (див. таблицю 2). Проаналізувавши знайдені в ній результати (2) та (3), помічаємо, наприклад, що *H*9 = *H*8 + *G*8 + *F*8 + *E*8 + *D*8 + *C*8 = 252, тобто значення вибраної комірки *X*(*i*, *j*) дорівнює сумі значень комірок діапазону *X*(*i* - 1, 0):X(*i* - 1, *j*). Але тоді *H*9 = *H*8 + *G*9, тобто *X*(*i*, *j*) = *X*(*i*, *j*-1) : X(*i* - 1, *j*).

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
кількість пропусків, які треба розмістити												
	j \ i	0	1	2	3	4	5	6	7	8	9	10
для розміщення	1	1	1	1	1	1	1	1	1	1	1	1
	2	1	2	3	4	5	6	7	8	9	10	11
	3	1	3	6	10	15	21	28	36	45	55	66
	4	1	4	10	20	35	56	84	120	165	220	286
	5	1	5	15	35	70	126	210	330	495	715	1001
	6	1	6	21	56	126	252	462	792	1287	2002	3003

	7	1	7	28	84	210	462	924	1716	3003	5005	8008
	8	1	8	36	120	330	792	1716	3432	6435	11440	19448
	9	1	9	45	165	495	1287	3003	6435	12870	24310	43758
	10	1	10	55	220	715	2002	5005	11440	24310	48620	92378
	11	1	11	66	286	1001	3003	8008	19448	43758	92378	184756
	12	1	12	78	364	1365	4368	12376	31824	75582	167960	352716

	48	1	48	1176	19600	249900	2598960	22957480	177100560	1217566350	7575968400	43183019880
місця	49	1	49	1225	20825	270725	2869685	25827165	202927725	1420494075	8996462475	52179482355
	50	1	50	1257	22100	292825	3162510	28989675	231917400	1652411475	10648873950	62828356305
	51	1	51	1326	23426	316251	3478761	32468436	264385836	1916797311	12565671261	75394027566
	52	1	52	1378	24804	341055	3819816	36288252	300674088	2217471399	14783142660	90177170226
	53	1	53	1431	26235	367290	4187106	40475358	341149446	2558620845	17341763505	1,07519E+11

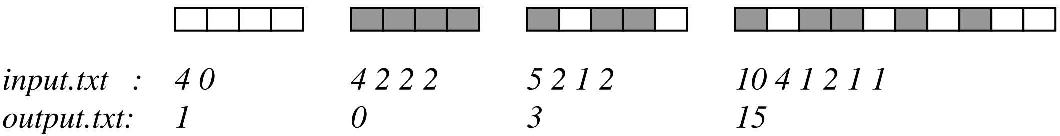
таблиця 2

Виходячи з цього, з (4) одержуємо остаточні робочі формули:

$$KS(1, j) = j,\ KS(i, 1) = 1,\ KS(0, j) = 1,\ KS(i,j) = KS(i,j\text{-}1)+KS(i\text{-}1,j) \qquad (5)$$

Для перевірки та закріплення формул (5) пропонуємо:

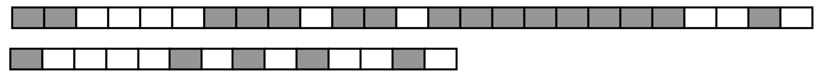
1. Обчислити з її допомогою та перевірити генеруванням кількість



input.txt :	*4 0*	*4 2 2 2*	*5 2 1 2*	*10 4 1 2 1 1*
output.txt:	*1*	*0*	*3*	*15*

мал. 3

смужок, взявши за початкові:



мал. 4

2. Обома способами визначити *KS* для смужок, наведених на малюнку 4.

Наголосимо на деяких цікавих особливостях задачі. Врахувавши (*), не вплинемо на результат, якщо перетворимо початкову смужку, наведену в умові задачі і зображену на малюнку 4 ($N = 25$, $K = 5$, код 2, 3, 2, 8, 1). Визначивши для обох смужок $s_1 = 20$, $Ns_1 = 5$, $K_1 + 1 = 6$ та $s_2 = 14$, $Ns_2 = 5$, $K_2 + 1 = 6$, бачимо, що $Ns_1 = Ns_2$ і $K_1 + 1 = K_2 + 1$, отже для цих смужок *KS* буде однакове. Назвемо смужки з однаковими *Ns* та $K + 1$ *еквівалентними* і будемо першим кроком алгоритму виконувати заміну початкової смужки на їй *мінімальну еквівалентну*, назвавши цю дію *мінімізацією* смужки. Це принесе значний ефект при визначенні *KS* генеруванням нових смужок. Наведемо алгоритм мінімізації:

1) Початок. п. 2
2) Визначити суму коду (s'). п. 3
3) Одержати новий код (s''), замінити кожне його число на 1. п. 3
4) N := s' - s''. п. 4

5) Кінець.

c) ***Пошуки ефективного алгоритму розв'язування задачі***.[32] Рекурсивний алгоритм, який не потребує доведення і оцінки на складність, легко одержати, використавши (5). Але необхідно дослідити, в яких межах він ефективно працюватиме без використання багаторозрядної арифметики, а також, чи можна його застосувати для багаторозрядних чисел. Серед проблем можуть бути переповнення через надто велике заглиблення в рекурсію та не допустимий час виконання.

```
program Strip _REC; {рекурсивна програма}
   var n,k,s,i,j,b:integer;f:text;
       a:array[1..50]33of longint; p:array[1..50,1..50]of longint;
   function KS(i,j:longint):longint;
   begin
      if i=1 then KS:=j
      else if j=1 then KS:=1 else KS:=KS(i,j-1)+KS(i-1,j)
    end;
begin
   assign(f,'n3.dat');reset(f); read(f,n,k);
   for i:=1 to k do begin read(f,a[i]);s:=s+a[i] end;
   close(f);
   i:=n-(s+(k-1));j:=k+1;
   assign(f,'n3.res');rewrite(f); writeln(f,KS(i,j));close(f);
end.
```

Значно ефективнішим є циклічний алгоритм, який передбачає з допомогою вкладених циклів для обчислення *KS*(5, 6) послідовно, рядок за рядком, визначити таблицю 6×6:

```
program Strip_For; {циклічна програма}
    var n,k,s,i,j,b:longint;f:text;
    a:array[1..256]of longint; p:array[1..100,1..100]of longint;
begin
    assign(f,'n3.dat');reset(f);read(f,n,k);
   for i:=1 to k do begin read(f,a[i]);s:=s+a[i] end;
   close(f);
   b:=n-(s+(k-1));
   for i:=1 to b+1 do p[1,i]:=1;  for j:=1 to k+1 do p[j,1]:=1;
   for j:=2 to k+1 do
     for i:=1 to b+1 do  p[i,j]:=p[i-1,j]+p[i,j-1];
   assign(f,'n3.res');rewrite(f); writeln(f,p[b+1,k+1]);close(f);
end.
```

Крім того, в циклічному алгоритмі простіше використати "довгу арифметику". Але слід врахувати, що, наприклад, при застосуванні текстового

[32] Алгоритми будемо наводити у вигляді програм на мові Turbo Pascal.
[33] Тут і далі обмеження для масивів взято довільно.

представлення багатоцифрових чисел, коли кожен елемент таблиці матиме тип *string* (256 байт), виникнуть проблеми переповнення пам'яті, тому краще замість одночасного обчислення прямокутної таблиці багаторазово обчислювати лінійну таблицю. Можна знайти інші виходи, наприклад, застосування динамічної пам'яті. Цим багато учнів і закінчать роботу над подібною задачею, хоча одержану програму ще не можна назвати ефективною.

Для пошуку справді ефекивного алгоритму ще раз скористаємось таблицею 2. З неї перш за все можна побачити межі дії алгоритму без використання багаторозрядних чисел. Права та нижня межі таблиці відсутні, що вказує на можливе продовження її в цих напрямках. Пунктирні межі між рядками №13-47 вказують на розрив таблиці. Нижній фрагмент (рядки 48-53) наведено лише для того, щоб указати одне із значень, що виходить за межі типу *longint* - $KS(10, 53) = 1{,}07519E{+}11$. У лівому верхньому куті таблиці виділено прямокутну область, яка ілюструє обчислення $KS(5, 6) = 252$, що дорівнює кількості смужок наведеного в умові задачі зразка (мал. 4). Як виявляється, таблиця відображає відомий числовий трикутник біноміальних коефіцієнтів Паскаля, якщо за його вершину взяти її лівий верхній кут. У цій частині таблиці заштрихованими та білими клітинками показано ”поверхи” трикутника Паскаля. На перший погляд, обчислювати значення таблиці з допомогою формули числа комбінацій $C_n^m = n!/(m!(n-m)!)$ не зрозуміло як, адже, наприклад, $KS(5,\ 6) = C_{10}^5$. Але, представивши частину таблиці 2, в дещо іншому вигляді, де для вирівнювання ширини стовпчиків всі числа замінено відповідними записами зразка C_j^i і сірими лініями виділено рядки трикутника Паскаля (табл. 3), легко встановити залежність між *i*, *j* з одного боку та *n*, *m* з іншого.

C	***D***	***E***	***F***	***G***	***H***	***I***	***J***	***K***	***L***	***M***
0	1	2	3	4	5	6	7	8	9	10
C_0^0	C_1^1	C_2^2	C_3^3	C_4^4	C_5^5	C_6^6	C_7^7	C_8^8	C_9^9	C_{10}^{10}
C_1^0	C_2^1	C_3^2	C_4^3	C_5^4	C_6^5	C_7^6	C_8^7	C_9^8	C_{10}^9	C_{10}^{10}
C_2^0	C_3^1	C_4^2	C_5^3	C_6^4	C_7^5	C_8	C_9^7	C_{10}^8	C_{11}^9	C_{10}^{10}
C_3^0	C_4^1	C_5^2	C_6^3	C_7^4	C_8^5	C_9^6	C_{10}^7	C_{11}^8	C_{12}^9	C_{10}^{10}
C_4^0	C_5^1	C_6^2	C_7^3	C_8^4	C_9^5	C_{10}^6	C_{11}^7	C_{12}^8	C_{13}^9	C_{10}^{10}
C_5^0	C_6^1	C_7^2	C_8^3	C_9^4	C_{10}^5	C_{11}^6	C_{12}^7	C_{13}^8	C_{14}^9	C_{10}^{10}
C_6^0	C_7^1	C_8^2	C_9^3	C_{10}^4	C_{11}^5	C_{12}^6	C_{13}^7	C_{14}^8	C_{15}^9	C_{10}^{10}
C_7^0	C_8^1	C_9^2	C_{10}^3	C_{11}^4	C_{12}^5	C_{13}^6	C_{14}^7	C_{15}^8	C_{16}^9	C_{10}^{10}
C_8^0	C_9^1	C_{10}^2	C_{11}^3	C_{12}^4	C_{13}^5	C_{14}^6	C_{15}^7	C_{16}^8	C_{17}^9	C_{10}^{10}
C_9^0	C_{10}^1	C_{11}^2	C_{12}^3	C_{13}^4	C_{14}^5	C_{15}^6	C_{16}^7	C_{17}^8	C_{18}^9	C_{10}^{10}
C_{10}^0	C_{11}^1	C_{12}^2	C_{13}^3	C_{14}^4	C_{15}^5	C_{16}^6	C_{17}^7	C_{18}^8	C_{19}^9	C_{10}^{10}
C_{11}^0	C_{12}^1	C_{13}^2	C_{14}^3	C_{15}^4	C_{16}^5	C_{17}^6	C_{18}^7	C_{19}^8	C_{20}^9	C_{10}^{10}

таблиця 3

Очевидно, що $m = i$, $n = i + j - 1$, тобто $KS(i, j) = C_{i+j-1}^i$, а, врахувавши позначення (*), остаточно одержимо: $KS = C_{i+j-1}^i$ (6)

Без значної і систематичної практики застосування комбінаторики, що характерно для більшості учнів, (6) встановити аналітично не легко, а з допомогою таблиці 3 вона стає очевидною.

Покажемо, як можна одержати цю формулу без наведених вище міркувань. Зрозуміло, що загальна кількість білих клітинок у смужці після усіх вставок дорівнюватиме $j + i$. Якщо прийняти, що вставка кожної з i додаткових білих клітинок можлива тільки між двома іншими білими клітинками, то першу з i білих клітинок можна розмістити j - 1 способами, наступну - j способами і т.д., а останню можна розмістити $i + j$ - 1 способами. Отже кількість таких розміщень дорівнюватиме $A_{i+j-1}^{\ i} = (i + j - 1)! /(i + j - 1 - i)! = (i + j - 1)! /(j - 1)!$. Але тут враховано всі розміщення білих клітинок, а в нашій задачі білі клітинки, які вставляються - рівноправні, тому одержане число слід ще поділити на кількість перестановок з i елементів, тобто на $i!$, у результаті чого отримаємо рівність (6).

Цю формулу важко побачити учневі, який з відомих причин навіть не обов'язково знайомий з комбінаторикою, тому, для нього більш звична математична модель, яка використовує (5). Поза сумнівом, вищеописана схема розв'язування даної задачі, тобто: рекурсивна програма ***Strip_REC*** ⇒ циклічна програма ***Strip_FOR*** ⇒ програма на базі (6) - це найбільш природний шлях пошуку повного і ефективного розв'язання, тому можна порадити учителям саме такої схеми дотримуватись при роботі з учнями над подібною задачею.

Необхідно також зауважити, що розглянуту задачу слід давати на олімпіаді диференційовано, відповідно до віку учасників. Це можна зробити за рахунок зміни формату вхідних і вихідних даних, а також тестів, для 8-9 класів достатньо обмежитись використанням стандартних числових типів, що дозволить обмежитись рекурсивним чи циклічним алгоритмами, для 10-11 класів можна дати наведені технічні умови, що вимагатиме опрацювання багаторозрядних чисел і може привести до застосування комбінаторики.

Отже, доведення формули (6), хоч і доступне далеко не всім учасникам олімпіади, але радикально спрощує алгоритм та забезпечує одержання повного розв'язку даної задачі, в тому числі на множині багаторозрядних чисел. Наведемо відповідну програму:

```
program Strip;[34]
   uses Long_Ar;
  var n,k,i,s,j,m:integer;a:array[1..101] of integer;f,f1:text;b, ks:string;
begin{Strip}
  Assign(f,'input.txt');  Reset(f); Assign(f1,'output.txt'); ReWrite(f1);
  Read(f,n); Read(f,k);
  for m:=1 to k do  Read(f,a[m]);
  if (n<1) or (n>200) or (k<0) or (k>(n+1)/2) then
     begin WriteLn(f1,'ERROR'); Close(f); Close(f1) end
  else  for m:=0 to k do s:=s+a[m+1];
  j:=k+1; i:=n-s-k+1;
```

[34] Програма враховує застосування "довгої арифметики" з представленням чисел у вигляді рядкових величин, необхідні процедури і функції для реалізації базових алгоритмів опрацювання багаторозрядних чисел, зокрема порівняння чисел, додавання, віднімання, множення багацифрового числа на одноцифрове, множення багатоцифрових чисел, визначення факторіалу та цілочисельне ділення наведено нижче, передбачається, що вони включені у модуль Long_Ar, який оголошений у програмі.

```
  MoDiv(Fact(i+j-1), Mult(Fact(i), Fact(j-1)),b, ks);
  WriteLn(f1,ks); Close(f); Close(f1);
end.{end  Strip}
```

Далі наведено підпрограми модуля Long_Ar.

```
function Comp(s1,s2:string):boolean;{порівняння st1 та st2}
   var j,l1,l2:byte;st:string;
begin l1:=Length(s1);l2:=Length(s2);
   if l1<l2 then for j:=l1-l2 downto 1 do s1:='0'+s1
   else            for j:=l2-l1 downto 1 do s2:='0'+s2;
   if s1>=s2 then Comp:=true else Comp:=false
end; {end порівняння st1 та st2}
function Add(st1,st2:string):string;{Add=st1+st2}
   var i,l1,l2,pm,s1,s2,s:byte;cod:integer;st:string;
begin
  if st1[0]<st2[0] then begin st:=st1;st1:=st2;st2:=st end;
  st1:='0'+st1;l1:=Length(st1);l2:=Length(st2);pm:=0;
  for i:=l1-l2 downto 1 do st2:='0'+st2;
  for i:=l1 downto 1 do                                   begin
    Val(st1[i],s1,cod);Val(st2[i],s2,cod);s:=s1+s2+pm;
    st1[i]:=Chr((s mod 10)+48);pm:=s div 10            end;
  if st1[1]='0' then Delete (st1,1,1);Add:=st1
end;{end Add}
function Sub(st1,st2:string):string;{Sub=st1-st2}
   var i,l1,l2,pm,s1,s2,s:byte;cod:integer;st:string;
begin
   if st1[0]<st2[0] then begin st:=st1;st1:=st2;st2:=st end;
   l1:=Length(st1);l2:=Length(st2);pm:=0;
   if st1=st2 then Sub:='0'
   else                                                begin
     for i:=l1-l2 downto 1 do st2:='0'+st2;
     for i:=l1 downto 1 do                        begin
       Val(st1[i],s1,cod);Val(st2[i],s2,cod);
       if s1>=s2+pm then begin s:=s1-(s2+pm);pm:=0 end
    else        begin s:=(s1+10)-(s2+pm);pm:=1 end;
    st1[i]:=Chr(s+48)                             end;
   while st1[1]='0' do Delete (st1,1,1);  Sub:=st1   end
end;{end Sub}
function Mult1(st1,st2:string):string;{Mult1=st1*st2, st2 - одноцифрове}
   var i,l1,l2,pm,s1,s2,cod,s:word;st:string;
begin
  st1:='0'+st1;l1:=Length(st1);l2:=Length(st2);pm:=0;
  for i:=l1 downto 1 do                                   begin
    Val(st1[i],s1,cod);Val(st2,s2,cod);s:=s1*s2+pm;
    st1[i]:=Chr((s mod 10)+48);pm:=s div 10           end;
  if st1[1]='0' then Delete(st1,1,1);Mult1:=st1
```

```
end;{end Mult1}
function Mult(st1,st2:string):string;  {Mult = st1*st2}
  var i,j,l:word;st0,st:string;
begin
  st0:='';l:=Length(st2);
  for i:=l downto 1 do          begin
    st:=Mult1(st1,Copy(st2,i,1));
    for j:=1 to l-i do st:=st+'0';
    st0:=Add(st,st0)             end;
  Mult:=st0;
end; {end Mult}
function Fact(n:integer):string; { n!}
   var i:integer; f, f1:string;
begin
function Fact(n:integer):string;
  var i:integer;f,f1:string;
begin
  f:='1';
  if n>1 then for i:=2 to n do            begin
                    str(i,f1);f:=Mult(f,f1) end;  Fact:=f;
end; { end Fact}
procedure MoDiv(st1, st2:string; var stmod, stdiv:string);{mod, div}
  var i,l,k, j:integer;st_1:string; bZero: boolean;
begin
  st_1:='';stdiv:='';i:=1;l:=Length(st1);
  repeat
    k:=0;  if st2 = '' then break;
    if st2 = '0' then begin st2 := 'ERROR';Break end;
    while not Comp(st_1,st2) do                begin
      st_1:=st_1+st1[i]; Inc(i); stdiv:=stdiv+'0';
      if i>length(st1) then break              end;
    Delete (stdiv,Length(stdiv),1);
    while st_1[1]='0' do                       begin
      if st_1 = '' then break; Delete (st_1,1,1) end;
    while Comp(st_1,st2) do           begin
      st_1:=Sub(st_1,st2);  k:=k+1  end;
    stmod:=st_1; if st_1 = '' then stmod := '0'; stdiv:=stdiv+Chr(k+48);
    while Length(stdiv) > Length(st1) do Delete(stdiv, Length(stdiv), 1);
  until i>l;
  while stdiv[1]='0' do                        begin
    if stdiv= '0' then break; Delete (stdiv,1,1) end;
end;  {end MoDiv}
```

Зауважимо, що програма ***Strip*** не у всіх випадках повертає правильне значення *ks*. Проаналізуємо кілька вхідних файлів і результати занесемо до таблиці 4 (неуточнені вихідні файли наведено у додатку).

input1.txt: 8 3 1 2 1
input2.txt: 25 5 2 3 2 8 1
input3.txt:
111 52 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1
input4.txt:
200 50 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
input5.txt:
200 60 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
input6.txt:
200 62 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
input7.txt:
200 65 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
input8.txt:
200 75 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
input9.txt:
200 90 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
input10.txt:
200 20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11

тест	*N*	*K*	*i*	*j*	*i+j-1*
1	8	3	6	4	9
2	25	5	5	6	10
3	111	52	8	53	60
4	200	50	101	51	151
5	200	60	81	61	141
6	200	62	77	63	139
7	200	65	71	66	136
8	200	75	51	76	126
9	200	90	21	91	111
10	200	20	151	21	171

таблиця 4

Згідно (6), наприклад, у тесті №10 необхідно обчислити 20!, 151! та 171!, але представлення багаторозрядних чисел у рядковому вигляді ставить під сумнів правильність визначення факторіалів таких великих чисел та значення неповної частки при їх цілочисельному діленні. Для уникнення цієї проблеми слід модернізувати програму ***Strip***, використавши перетворення:

$$C_{171}^{151} = \frac{171!}{151! \cdot 20!} = \frac{152 \cdot 153 \cdot ... 170 \cdot 171}{20!}$$

З огляду на елементарність при наявності модуля ***Long_Ar*** пропонуємо виконати це перетворення самостійно та одержати результати до наведених тестів.

На цьому дослідження задачі можна вважати цілком вичерпаним. Проте деякі виявлені факти дозволяють зробити додаткові висновки. Наприклад, ми поки-що не побачили практичної потреби у згаданій вище мінімізації смужки. Але можна запропонувати нову задачу:

Задача 18′. На смужці клітинкового паперу висотою в одну і довжиною N клітинок деякі клітинки зафарбовані в чорний і білий кольори. Кодом клітинки є непуста множина чисел – кількості ідучих підряд чорних клітинок зліва направо. Наприклад, для наведеної смужки кодом буде множина {2, 3, 2, 8, 1}. При цьому кількість білих клітинок, які розділяють групи чорних ніде не враховуються (головне, щоб дві сусідні групи розділені принаймні однією білою клітинкою). Задача полягає в тому, щоб знайти кількість смужок довжини N, які відповідають заданому коду.

Вхідні дані. У єдиному рядку файлу файла *input.txt* записано число N – довжина смужки ($1 \le N \le 25$), потім число K ($0 \le K \le (N+1)/2$) – кількість чисел у коді і далі K чисел, що визначають код.

Вихідні дані. У вихідний файл *output.txt* записується кількість смужок довжини N, які відповідають заданому коду.

Задача 18′ лише зовнішньо дуже схожа на задачу 18. Насправді ж з умови випливає, що слід враховувати всі смужки з послідовностями чорних клітинок, що є перестановками з повторенням у коді. Звідси легко вивести формулу для визначення Ks у задачі 15′. У зв’язку з цим у файлі *input.txt* взято обмеження $1 \le N \le 25$. Очевидно, що перша задача легша від другої, більше того, вона є її частинним випадком, а побачити це допоможе введене нами поняття мінімізації смужки.

Не будемо вказувати на інші виявлені факти. По-перше, це дасть можливість уважним читачам зробити власні маленькі відкриття, по-друге, нема потреби розкривати таємниці ще кількох нових задач, які можуть бути представлені у якості олімпіадних. Зауважимо лише, що ми намагались прослідкувати, як, шукаючи розв’язок однієї задачі можна прийти до інших, можливо не менш цікавих, а то й зовсім несподіваних, тобто продемонструвати шлях *від задачі до задачі*.

Надзвичайно цінною якістю особистості є вміння швидко, за екстремальних обставин, як буває на олімпіадах, шукати розв’язки задач, та не менш важливе вміння без поспіху, але грунтовно, проводити повне дослідження поставленої проблеми, що часто приводить до нових проблем.

Література:

1) “Програми для загальноосвітніх навчальних закладів. Навчальні програми для профільного навчання. Програми факультативів, спецкурсів, пропедевтичних курсів, гуртків. Інформатика”, видавництва “Прем’єр”, Запоріжжя, 2003р.
2) І.Т.Зарецька, А.М.Гуржій, О.Ю.Соколов, Інформатика, частина 2, підручник для 10-11 класів загальноосвітніх навчальних закладів, Київ, “Форум“, 2004 р.
3) С. Окулов. Программирование в алгоритмах. Москва. Бином. Лаборатория Знаний. 2002