**ёёЛабораторна робота № 2.**

***Тема:* “Рекурсія.Рекурсивні процедури і функції”.**

***Мета:*** Розлянути поняття рекурсії, дослідити доцільність застосування рекурсивних алгоритмів та заміну рекурсивних алгоритмів ітеративними.

***Теоретичні відомості:***

Визначення називається *рекурсивним*, якщо воно задає елементи множини за допомогою інших елементів цієї ж множини. Об'єкти, задані рекурсивним визначенням, також називаються рекурсивними. І нарешті, рекурсія – це використання рекурсивних визначень.

*Приклад 1*

Значення функції факторіал задаються виразом: 0!=1, n=n\*(n-1)!. Вони утворюють множину {1,2,6.}: 0!=1, 1=1\*0!, 2=2\*1!, 6=3\*2! і т.д. Всі його елементи, окрім першого, визначаються рекурсивно.

*Приклад 2*

Арифметичні вирази з константами і знаком операції “+” в інфіксній дужковій формі задаються таким визначенням:константа є виразом;якщо Е і F є виразами , то (Е)+(F) також є виразом.Такими виразами є, наприклад, 1, 2 (1)+(2), ((1)+(2))+(1). Всі вони, окрім констант 1 і 2, визначаються рекурсивно.

Об'єкти, визначені в прикладах є рекурсивними.

У рекурсивному визначенні не повинно бути “зачарованого кола”, коли об'єкт визначається за допомогою себе самого або за допомогою інших, але заданих через нього ж.

*Приклад 3*

Змінимо визначення функції факторіал на наступне: n!=n\*(n-1)! При n>0, 0!=1!. Спочатку значення функції від 1 виражається через її ж значення від 0, яке, у свою чергу, – через значення від 1. За таким визначенням так і не дізнатися, чому ж рівне 1!.

Щоб подібна нескінченність не виникала в рекурсивному визначенні, повинні виконуватися наступні умови:

-безліч визначуваних об'єктів є частково впорядкованою;

-кожна убуваюча по цьому впорядкуванню послідовність елементів закінчується деяким мінімальним елементом;

-мінімальні елементи визначаються нерекурсивно;

-немінімальні елементи визначаються за допомогою елементів, які менше за них по цьому впорядкуванню.

Неважко дійти переконання, що визначення з прикладів 1 і 2 задовольняють цим умовам, а з прикладу 3 – ні.

Відношення називається відношенням часткового порядку, якщо воно має такі властивості:

1) Для кожного елементу а множини у відношенні є пара (а, а);

2) Якщо у відношенні є пара (а, b) з різними елементами а і b, то пари (b, а) там немає. При цьому ми говоримо, що а менше за b.

3) Якщо а менше за b, а b менше з, то а менше с.

Втім, елементів а, b, с таких, що а менше за b, а b менше с, в множині може і не бути – при виконанні властивостей (1) і (2) відношення буде відношенням часткового порядку.

Множина із заданим на ньому відношенням часткового порядку називається частково впорядкованою. Елемент частково впорядкованої множини називається мінімальним, якщо в множині немає елементів, менших його.

Рекурсивні алгоритми в програмуванні реалізовані в механізмі так званих рекурсивних підпрограм. **Рекурсивною** називається підпрограма, виконання якої приводить до її ж **повторного** виклику з зміненими параметрами.

Якщо підпрограма просто викликає сама себе, то така рекурсія називається прямою. Якщо ж декілька підпрограм викликають одна одну, але ці виклики "замкнуті в кільце", то така рекурсія називається непрямою.

З рекурсивними підпрограмами пов'язані два важливі поняття – *глибина рекурсії* і *загальна кількість викликів*, *породжених викликом рекурсивної підпрограми*.

Розглянемо перше з них. При обчисленні факторіалу виклик функції з аргументом, наприклад 4, закінчується лише після закінчення виклику з аргументом 3, а той, у свою чергу, після виклику з аргументом 2 і т.д. Такі виклики називаються *вкладеними*. Таким чином, виклик з аргументом 4 породжує ще три вкладені виклики. Взагалі, при виклику цієї функції з аргументом *n* породжується ще *n*-1 виклик, і загальна кількість незавершених викликів досягає *n*. Таким чином, *глибиною рекурсії виклику підпрограми* називається максимальна кількість незавершених рекурсивних викликів при виконанні її виклику.

При виконанні виклику з глибиною рекурсії *т* одночасно існує *т* екземплярів локальної пам'яті. Кожен екземпляр має певний розмір, і якщо глибина буде занадто великою, то пам'яті, наданої процесу виконання програми, може не вистачити.

Друге поняття можна назвати *загальною кількістю вкладених викликів*, породжених викликом рекурсивної підпрограми. Ця кількість впливає на час виконання виклику.

Таким чином, вживання рекурсивних підпрограм вимагає обережності і уміння оцінити можливу глибину рекурсії і загальну кількість викликів.

Рекурсивний виклик може бути непрямим. В цьому випадку підпрограма звертається сама до себе опосередковано, шляхом виклику іншої підпрограми, в якій звернення до першої.

**Обмеження глибини рекурсії**

Теоретично, рекурсія може бути нескінченною. Проте такий варіант навряд чи кого-небудь влаштує: рекурсивний алгоритм, як і будь-який нерекурсивний його побратим, зобов'язаний видавати результат своєї роботи за якийсь осяжний час. Крім того, пам'ять у комп'ютера не гумова, в ній може поміститися лише кінцеве число контекстів одночасно відкритих екземплярів рекурсивної підпрограми.

Отже, кожна рекурсивна підпрограма повинна містити в собі **ознаку закінчення** - своєрідну "огорожу", що визначає максимальну глибину вкладеності для цієї рекурсії. Ознака кінця рекурсії може бути як явною, так і неявною.

Рекурсія і ітерація в деякому розумінні протилежні. Рекурсія вирішує задачу від складного до простого, ітерація - від простого до складного. Текст рекурсивного алгоритму виражає складний об'єкт через більш простий (простіші) такого ж типа. Текст ітеративного алгоритму описує процес будівництва, починаючи з дрібних деталей. Рекурсивний алгоритм виражає невідоме через невідоме, але зв'язок між цими двома невідомими досить простий, математично прозорий, тому довести правильність алгоритму легко. Стани об'єкту на двох різних кроках ітеративного алгоритму - як два стани недобудованого будинку: поки не покладена остання цеглина, не очевидно, що вийде в кінці. Тому довести правильність такого алгоритму звичайно досить складно.

Якщо обчислення починається з відшукання значення функції при мінімальному значенні аргументу, використовує його для визначення наступного і так далі, таке обчислення, записане у вигляді:

у := а;

for i:= 1 to n do

у := f(у, i);

називається ітерацією.

Важлива відмінність між рекурсією і ітерацією полягає в механізмі реалізації.

Виконавець рекурсивного алгоритму зводить невідоме до іншого невідомого, накопичуючи інформацію і відкладаючи реальні обчислення до моменту зведення аргументу функції до такого значення, при якому значення функції відоме.

Виконавець ітеративного алгоритму починає з відомого значення і крок за кроком перетворить відомі вже значення функції в значення для великих аргументів. В цьому відношенні ітерація подібна зворотному ходу в реалізації рекурсії. З цих слів (ітераційний процес є частина рекурсивного) можна зробити висновок про те, що ітерація простіше (як частина цілого) і необхідно вирішувати задачі ітераційними методами.

Насправді ситуація складніша. У декількох словах її можна охарактеризувати так: рекурсивне рішення задачі складається з двох частин - прямого ходу (зведення завдання до ітераційної форми) і зворотного (виконання ітерацій). Звільнивши машину від виконання прямого ходу, полегшуємо її роботу, але перекладаємо цю частину роботи на програміста, якому ставиться в обов'язок перетворити програму. Іноді це не складає труднощів для програміста, і він відразу може написати ітераційний варіант, а іноді проблема буде важко вирішуваною.

Розглянемо докладніше зв'язок між рекурсією і ітерацією на прикладі обчислення чисел Фібоначчі.

*Рекурсивний варіант*.

function Fr (n: integer): integer;

begin

if n = 1 or n = 2 then

Fr:= 1

else Fr:=Fr(n-l) + Fr(n-2);

end;

*Ітеративний варіант*

function Fi (n: integer): integer;

var i N1, N2, Result: integer ;

begin

N1:= 1; N2:= 1; { Установка початкових значень}

Result:= 1; { на той випадок, якщо n = 1 або 2.}

for i:= 3 to n do { Не виконується, якщо n < 3.}

begin

Result:= N1 + N2; { Відповідає основній формулі чисел Фібоначчі.}

N2:= N1; { Зрушення значень на один індекс.}

N1:= Result { Зрушення значень на один індекс.}

end;

Fi:= Result { Установка значення функції.}

end;

До порівняльної оцінки цих двох алгоритмів можна підходити з двох позицій: використання ресурсів ЕОМ і складності написання і відладки програми.

A. Ресурси. В даному випадку під ресурсами розуміється час процесора і об'єм зайнятої програмою (і даними) пам'яті.

Рекурсивна програма включає наступні операції, що займають час процесора: виклик функції, порівняння n з одиницею і двійкою, складання; при кожному виклику відбуваються порівняння і в частині викликів (якщо n>2) відбуваються складання. Тому вирішальним параметром в оцінці часу є кількість викликів функції. Всі виклики можна розбити на рівні таким чином. На першому рівні знаходиться єдиний перший виклик з параметром n. На другому рівні знаходяться два виклики (з параметрами n-1 і n-2), проведені при першому виклику. На третьому рівні знаходяться чотири виклики з параметрами n-2, n-3, n-3, n-4 і т.д. із збільшенням кількості на кожному рівні в два рази по відношенню до попереднього рівня до тих пір, поки величини n-i не почнуть досягати значення 2. Ці виклики можна зобразити у вигляді дерева (малюнок 1).

Тут для визначеності узято n=7. Непросто оцінити, скільки вершин (викликів) в цьому дереві, але можна швидко одержати нерівності, що обмежують цю кількість зверху і знизу. Ліва гілка помічена числами n n-1, n-2, n-3, ..., 2. Отже, кількість рівнів не перевершує величини n-1. Права гілка помічена числами n n-2, n-4, n-6 = 1. Можна зробити висновок, що кількість рівнів не менша n/2. Якщо дерево має к рівнів, то в ньому 1+2+4+8+...+2k-1=2k вершин або, враховуючи нерівність n-1 > до > n/2, одержуємо 2n-1-1 > число викликів > 2n/2.

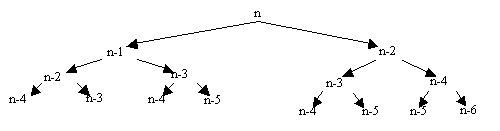


Рис.1. Дерево викликів рекурсивної функції обчислення чисел Фібоначчі

Як верхня, так і нижня межі швидко ростуть із збільшенням n. Вже при n=20 потрібно не менше 1000 викликів, хоча, як це ясно видно з дерева, багато викликів дублюють один одного - текст функції не передбачає контролю дублювання. З кожним незавершеним викликом функції пов'язано зберігання в стеку певних даних (це обговорювалося, коли розглядався механізм реалізації рекурсії). З огляду на те, що прохід по дереву проводиться в глибину, у будь-який момент часу потрібно зберігати від n-1 до n/2 груп даних.

Ітеративна програма включає складання, пересилки (привласнення) і перевірки (неявні) циклу на закінчення. Зайнятість процесора в цьому випадку зручно оцінити кількістю виконань тіла циклу. Заголовок циклу ясно указує на те, що тіло циклу (одне складання, три пересилки і команди управління циклом) виконується n - 2 рази; окрім цього три пересилки (привласнення) на початку і одне в кінці. Таким чином, кількість операцій може бути оцінена формулою an+b. Кількість використаної пам'яті - пам'ять для чотирьох змінних.

Ці розрахунки показують явну перевагу ітеративного методу порівняно з рекурсивним по використанню ресурсів ЕОМ для вирішення даного завдання.

B. Складність написання і відладки програми. По суті це питання про витрати праці програміста. Формальні числові оцінки тут важкі. Але порівняння двох текстів показує, що рекурсивна програма пишеться автоматично на основі постановки завдання, тоді як ітеративна вимагає перетворення завдання і "винаходження" алгоритму, введення додаткових змінних, відсутніх в постановці завдання, і неочевидних операцій. Все це приводить до необхідності доказу того, що одержаний алгоритм дійсно вирішує поставлену задачу, а не якусь іншу, що він правильно працює для всіх допустимих початкових даних. Положення ускладнюється тим, що замість статичного аналізу тексту (у разі рекурсії) необхідно представляти, як алгоритм розгортається в часі за допомогою якогось виконавця, тобто застосовувати модель комп'ютера.

З цієї точки зору рекурсивний алгоритм кращий ітеративного.

Аналіз пунктів (А) і (Б) дає нам протилежні висновки. У разі таких простих завдань як обчислення чисел Фібоначчі питання звичайно розв'язується на користь ресурсів комп'ютера.

**Заміна рекурсивних алгоритмів ітеративними**

Оскільки новий контекст створюється кожного разу, коли черговий екземпляр рекурсивної підпрограми (сам ще залишаючись незавершеним) наново викликає себе ж, то пам'ять комп'ютера витрачається дуже швидко. Тому рекурсію при всієї її наочності не можна віднести до економічних способів програмування. Існує навіть спеціальна наука - **динамічне програмування**, що вивчає способи заміни рекурсивних алгоритмів адекватними ітеративними алгоритмами.

Якщо виконання підпрограми приводить тільки до одного виклику цієї ж самої підпрограми, то така рекурсія називається лінійною. Лінійну рекурсію досить легко замінити ітеративним алгоритмом. Наприклад, можна реалізувати функцію факторіалу двояко.

|  |  |
| --- | --- |
| **Рекурсивна реалізація** | **Ітеративна реалізація** |
| function fact(k:byte):longint;  var x: longint;  begin  if к = 0  then fact:= 1  else begin x:= fact(k-1)\*k;  fact:=x;  end;  end; | fact:= 1  for i:= 2 to до do  fact:= fact \* i; | |

Якщо ж кожен екземпляр підпрограми може викликати себе кілька разів, то рекурсія називається нелінійною або такою, що розгалуджується. Для того, щоб перетворити таку рекурсію на ітеративний алгоритм, доведеться докласти багато зусиль і, можливо, навіть внести деякі зміни в оброблювану структуру даних.

***Реалізація механізму рекурсії***

Виконання рекурсивної процедури (функції) виявляється для комп'ютера не зовсім простим завданням. Для того, щоб підтримати всі можливості, закладені в схемі рекурсії, комп'ютер повинен виконати ряд дій, прихованих від програміста.  Щоб ясніше уявити собі те, що відбувається на машинному рівні під час виконання рекурсивних процедур (а, означає, вирішити, наскільки можливо і ефективно їх використання в тому або іншому випадку), розглянемо декілька прикладів.

***Виконання оператора привласнення ...* Z:= 2\*а + b; *... для скалярних змінних в головній програмі***. Компілятор генерує очевидні команди: множення, складання операндів і пересилка результату в область пам'яті, відведену для змінної **Z**. Тут немає прихованих дій за виключенням, мабуть, введення робочих змінних, якщо вираз в правій частині оператора привласнення достатньо складний.

***Обчислення з викликом функції***. Дано визначення функції:

function Sum(X, Y: integer): integer;

begin

Sum:= X + Y;

end;

У основній програмі є звернення:

begin

...

Z:= Sum(2\*a, b);

...

end.

Компілятор згенерує команди, які

в основній програмі:

-обчислять значення фактичних параметрів;

-у спеціально відведеній області пам'яті основної програми сформують список параметрів (як правило, безперервна ділянка пам'яті, в яку занесені адреси параметрів);

-запишуть адресу списку параметрів в один з регістрів процесора;

виконають перехід до початкової команди функції **Sum** і передачу цій функції адреси повернення;

-після повернення з функції перешлють отриманий результат в область пам'яті, відведену для змінної **Z**.

У програмі, що викликається (функції **Sum**):

-забезпечується збереження вмісту регістрів процесора в оперативному пристрої, що запам'ятовує;

-значення фактичних (вхідних) параметрів, узяті із списку, переписуються в пам'ять формальних параметрів **X** і **Y**;

-проводяться обчислення відповідно до операторів тіла функції;

-відновлюється вміст регістрів процесора;

-виконується команда повернення в зухвалу програму.

З приведеного переліку видно, що використання процедур і функцій вимагає додаткових дій. Кожна процедура має контекст, в якому вона виконується. Спрощено кажучи, це набір даних, пов'язаних з процедурою, і їх значень; зокрема - локальні змінні і формальні параметри. При вході в процедуру, що викликається, контекст створюється, а при поверненні в основну програму її контекст повинен відновлюватися.

При компіляції процедури необхідно вирішити одне важливе питання: як бути з її локальними змінними і параметрами, передаваними по значенню (роль яких усередині процедури така ж, як і роль локальних змінних)?

З погляду мови вони не існують, поки процедура не активізована (викликана); і перестають існувати після завершення процедури. Це означає, що можна виділяти для них пам'ять у момент активізації і віддавати її назад системі після закінчення виконання процедури. Але якщо виклики процедури знаходяться в циклі, дії по роботі з пам'яттю можуть значно збільшити сумарний час роботи програми, чи проводяться запити відносно виділення пам'яті до операційної системи або пам'ять береться із спеціальної області - стека.

Підвищення швидкості обчислень можна добитися, якщо призначити локальним змінним адреси (відносні) при компіляції. Тоді область пам'яті буде закріплена за ними протягом всього часу виконання програми і команди в процедурі можна згенерувати з урахуванням відомих адрес. При цьому, правда, дещо зменшується ефективність використання пам'яті, але якщо в процедурі немає великих локальних масивів, то на це можна піти.

Небезпека такого підходу не в неефективному використанні пам'яті. Вона набагато серйозніша! Уявимо собі, що робота відбувається в середовищі багатопрограмної операційної системи и/или на багатопроцесорній обчислювальній системі і функція **Sum** викликана майже одночасно двома програмами (з різними, зрозуміло, фактичними параметрами). Точніше кажучи, друга програма викликала функцію **Sum** після того, як це зробила перша, але до того як завершився перший виклик. Програмний код - ресурс не витрачається (якщо немає вірусів), тому два процесори цілком можуть поділити команди коду, прочитуючи з пам'яті одну і ту ж ділянку коду, кожен в свій кеш команд або іншим способом. Річ у тому, що команди не залежать від фактичних параметрів функції. Але значення локальних змінних залежать! Якщо локальна пам'ять функції вже використовується першим викликом (тобто першою програмою), то початок роботи другого виклику зруйнує значення в полі локальних даних. В результаті обидва виклики заважатимуть один одному, перекреслюючи те, що було тільки що записане викликом, що змагається, і ні перша, ні друга програми не отримають очікуваних результатів.

Процедури і функції, які забезпечують правильну роботу в такій ситуації, тобто можливість знов викликати процедуру ("увійти" до процедури), хоча ще не завершилися попередні виклики, називаються реєнтерабельними (повторновходимыми). Для забезпечення реєнтерабельності потрібно пам'ять локальних змінних пов'язувати не з процедурою, а з кожним викликом процедури.

Аналогічна проблема виникає при реалізації ***рекурсивних процедур, навіть якщо вони виконуються на однопроцесорній обчислювальній системі в середовищі однопрограмної ОС***. Як випливає з тексту рекурсивної процедури, вона звертається до самої себе, тобто генерує новий виклик до завершення попереднього. Але ця проблема одночасно і складніше і простіше за попередню.

Складніше тому, що глибина вкладення (глибина рекурсії) може бути дуже великою, як показує хоч би випадок обчислення факторіалу:

function FACTORIAL( N: integer ): integer;

begin

if N = 1 then FACTORIAL:= 1

else FACTORIAL:= N \* FACTORIAL (N -1)

end;

:

begin

... { Фрагмент основної програми.}

K:= FACTORIAL(M); { Цей виклик породжує ще М-1 викликів. }

...

end.

Старти і завершення різних викликів процедури в багатопрограмній або багатопроцесорній системі ніяк не синхронізовані: раніший виклик може закінчитися і раніше і пізніше за пізніше. Це залежить як від відмінностей у фактичних параметрах викликів (виконання піде по різних гілках в процедурі), так і від відмінностей в швидкості процесорів або із-за особливостей в порядку надання квантів часу процесора в однопроцесорній багатопрограмній системі. Асинхронность декілька ускладнює управління пам'яттю. В той же час при викликах рекурсивних процедур дотримується чіткий порядок.

Раніший рекурсивний виклик завжди закінчиться пізніше. Цей факт дозволяє застосувати для управління локальною пам'яттю рекурсивних процедур таку структуру даних як ***стек***. Стек характеризується наступними властивостями: він має невизначений розмір (реально розмір обмежений технічними параметрами комп'ютера); елементи стека не нумеруються, доступний (для читання) тільки один елемент, верхній; після видалення із стека верхнього елементу доступним стає наступний і т.д.; елемент, що знов поміщається в стек, стає верхнім. Таким чином, ***стек - лінійна структура даних, доступна для читання і запису тільки з одного кінця***, тобто робота із стеком підкоряється принципу LIFO (last **in - first out,** останнім ***включений - першим видалений)***. Стек можна моделювати лінійним списком, але це неефективно за часом, тому звичайно стек моделюється одновимірним масивом і двома індексними змінними - покажчиками на дно і на вершину стека. Сучасні комп'ютери підтримують стек апаратний - спеціальними регістрами-покажчиками на сегменти пам'яті, призначені для розміщення стеків, на вершини стеків і командами для виконання операцій із стеками.

В загальному випадку при виклику процедурою A процедури B відбувається наступне:

1. У вершину стека поміщається фрагмент потрібного розміру. У нього входять слідуючі дані: (а) покажчики фактичних параметрів виклику процедури В; (б) порожні осередки для локальних змінних, визначених в процедурі В; (в) адреса повернення, тобто адреса команди в процедурі A, яку слід виконати після того, як процедура B закінчить свою роботу. Якщо B - функція, то у фрагмент стека для B поміщається покажчик осередку у фрагменті стека для A, в яку належить помістити значення цієї функції (адреса значення).

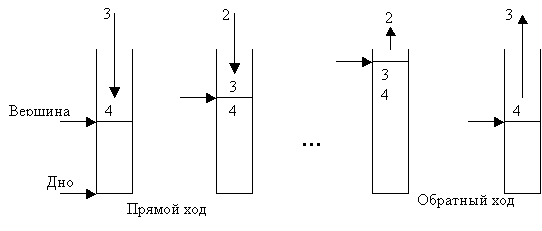
2. Управління передається першому оператору процедури B.

3. При завершенні роботи процедури B управління передається процедурі A за допомогою наступної послідовності кроків: (а) адреса повернення витягується з вершини стека; (б) якщо B - функція, то її значення запам'ятовується в осередку, вказаному покажчиком на адресу значення; (в) фрагмент стека процедури B витягується із стека, у вершину ставиться фрагмент процедури A; (г) виконання процедури A поновлюється з команди, вказаної в адресі повернення.

При виклику підпрограмою самої себе, тобто в рекурсивному випадку, виконується та ж сама послідовність дій.

Робота із стеком проводиться таким чином. При першому виклику рекурсивної процедури (функції) в стеку резервується місце для всіх значень, пов'язаних з цим викликом (це можуть бути не тільки локальні змінні і параметри, але і деяка службова інформація, прихована від програміста). Робиться це простою зміною значення покажчика на вершину стека: воно збільшується на необхідне число байт, якщо стек росте у бік старших адрес пам'яті, або (як в багатьох комп'ютерах) зменшується, якщо стек росте у бік молодших адрес пам'яті.

Перед другим викликом у області пам'яті першого виклику знаходитимуться деякі значення, відповідні початим, але не завершеним обчисленням. Другий виклик резервує в стеку область над областю першого виклику тим самим, перекриваючи до неї доступ.

  
Рис.2. Зміна вмісту стека при обробці рекурсивних викликів

Таким чином, аж до останнього виклику стек заповнюватиметься (так званий прямий хід рекурсії). На малюнку 2 показані стани стека при виконанні виклику **Factorial (4)**; нижня область між дном і вершиною вже використана раніше викликаними іншими процедурами (що викликають по відношенню до функції **Factorial**). Перший виклик поміщає в стек значення 4, другий 3 і т.д. до тих пір, поки фактичним параметром виклику не стане значення 1. Потім починається серія завершень викликів з витяганням із стека збережених значень і використанням їх для продовження обчислень (у даному прикладі для виконання операції множення). Останнім завершується перший виклик. На малюнку 2 зображено зсув покажчика на вершину області.

**Завдання:**

**Варіант 1**

1) Напишіть рекурсивну підпрограму піднесення числа до степеня.

2) Напишіть рекурсивну підпрограму переведення натурального числа з десяткової системи в двійкову.

**Варіант 2**

1) Напишіть рекурсивну підпрограму визначення, чи є дане натуральне число простим.

2) Написати рекурсивну підпрограму друку десяткового запису цілого додатнього числа n.

**Варіант 3**

1) Напишіть рекурсивну підпрограму визначення кількості цифр натурального числа.

2) Запрограмуйте рекурсивний алгоритм розв'язання задачі про Ханойські вежі. Ця задача полягає в наступному:

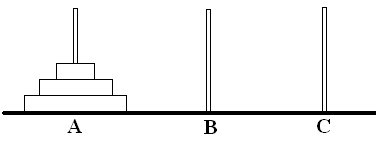
Є три стержня і *n* дисків різного розміру. Диски можна надівати на стержні, утворюючи з них вежі. Спочатку всі диски в спадаючому порядку розташовані на стержні A. Задача полягає в тому, щоб перенести *n* дисків зі стержня А на стержень С, зберігши їх початковий порядок. При перенесенні потрібно керуватись наступними правилами:

а) На кожному кроці зі стержня на стержень переноситься рівно один диск;

б) Диск не можна класти на диск меншого розміру;

в) Для тимчасового зберігання можна використовувати стержень B.

Вежу зручно розглядати як таку, що складається з одного верхнього диску і вежі, утвореної іншими дисками.



**Варіант 4**

1) Напишіть рекурсивну підпрограму знаходження цифрового кореня натурального числа. (Додати всі цифри числа, потім всі цифри знайденої суми і т.д. доки не отримається одна цифра).

2) Написати рекурсивну підпрограму підрахунку числа вершин в дереві.

**Варіант 5**

1) Напишіть рекурсивну підпрограму знаходження суми перших n членів арифметичної прогресії.

2) Написати рекурсивну програму підрахунку числа листків в дереві.

**Варіант 6**

1) Напишіть рекурсивну підпрограму знаходження індекса мінімального елемента в масиві з n елементів.

2) Написати рекурсивну програму підрахунку висоти дерева (корінь має висоту 0, його сини - висоту 1 і т.п.; висота дерева - це максимум висот його вершин).

**Варіант 7**

1) Написати рекурсивну підпрограму для обчислення функції Аккермана для невід'ємних чисел n m, яка визначається таким чином:

A(n,m)=m+1 якщо n=0;

A(n,m)=A(n-1,1) якщо n<>0, m=0;

A(n,m)=A(n-1,A(n,m-1)) якщо n>0, m>0.

2) Напишіть рекурсивну підпрограму знаходження n-го члена геометричної прогресії.

**Варіант 8**

1) Написати рекурсивну підпрограму яка зчитує з клавіатури послідовність чисел і виводить її на екран в зворотньому порядку (закінчення послідовності при введенні 0).

2) Написати рекурсивну програму, яка знаходить всі перестановки чисел 1..n по одному разу.

**Варіант 9**

1) Написати рекурсивну підпрограму визначення найбільшого спільного дільника двох чисел.

2) Написати рекурсивну програму, яка дозволяє перерахувати всі представлення позитивного цілого числа n у вигляді суми послідовності незростаючих цілих позитивних доданків.

**Варіант 10**

1) Написати рекурсивну підпрограму що виводить цифри натурального числа в зворотньому порядку.

2) Написати рекурсивну програму, яка дозволяє розкласти натуральне число на співмножники всіма можливими способами (без повторень)

**Варіант 11**

1) Написати рекурсивну програму, яка по заданому n рахує число всіх вершин висоти n у заданому дереві.

2) Написати рекурсивну програму, яка дозволяєреалізувати індійський алгоритм піднесення до степеня. Цей алгоритм обчислення натурального *n-го (n*>1) степеня цілого числа *а* виглядає таким чином:

an=a при n=1,

an=an mod 2 \* (an div 2)2 при n>1.

Основна мета цього алгоритму – скоротити кількість множень при піднесенні до степеня.

Обчислення *аn* зводиться до обчислення *аn* **div** 2, запам'ятовуванню результату, зведенню в квадрат і множенню його на *х* при непарному *n*.

**Варіант 12**

1) Напишіть рекурсивну підпрограму знаходження суми перших n членів геометричної прогресії.

2) Запрограмувати рекурсивний алгоритм сортування злиттям. Цей алгоритм полягає в наступному:

а) нехай К - індекс середнього елемента масиву А;

б) відсортуйте елементи до А(к) включно;

в) відсортуйте елементи після А(к);

г) з'єднайте ці два пiдмасиви в один відсортований масив

**Варіант 13**

1) Використовуючи лише команди write(x) при x=0..9, написати рекурсивну підпрограму друку десяткового запису цілого додатнього числа n.

2) Є деяка сума грошей S і набір монет з номіналами a1, ., an. Монета кожного номінала є в єдиному екземплярі. Написати рекурсивну програму, яка дозволяє знайти всі можливі способи розміняти суму S за допомогою цих монет.

**Варіант 14**

1) Написати рекурсивну підпрограму для визначення, чи є симетричною частина рядка s починаючи з i-го елемента і закінчуючи j-м.

2) Написати рекурсивну програму відшукання максимального з N елементів, збережених в масиві а[1],. . ., а[N].

**Варіант 15**

1) Напишіть рекурсивну підпрограму обчислення суми цифр натурального числа.

2) Написати рекурсивну програмущо здійснює пошук елементу у впорядкованому масиві (двійковий пошук). Є впорядкований масив і еталонний елемент. Потрібно визначити, чи міститься еталон в масиві. Якщо так, то повернути відповідний номер позиції. Якщо ні - вивести повідомлення. Для вирішення завдання використовується метод ділення початкового масиву навпіл. З еталоном порівнюється «середній» (розташований посередині) елемент масиву. Якщо він менше еталону – пошук продовжується в правій половині масиву. Якщо більше – в лівій. Пошук ведеться до тих пір, поки не буде виявлена відповідність, або поки довжина ділянок масиву, в яких ведеться пошук, не стане менше 1.

***Контрольні питання:***

1. Що таке рекурсія;
2. Що таке глибина рекурсії;
3. Які види рекурсії ви знаєте.