Лекція 7. Метод грубої сили.
План.

1. Основні поняття рекурсії.
2. Визначення методу розробки алгоритмів.
3. Квадратичні алгоритми сортування.

1.

1. Об'єкт називається **рекурсивним**, якщо він містить сам себе або визначений за допомогою самого себе. Рекурсія зустрічається не тільки в математиці, але й у повсякденному житті.

Рекурсія є особливо потужним засобом у математичних визначеннях. Відомі приклади рекурсивних визначень натуральних чисел, деревоподібних структур і деяких функцій:

1. Натуральні числа:

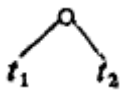
(a) 1 є натуральне число;

(b) ціле число, що випливає за натуральним, є натуральне число.

2. Деревоподібні структури:

(a) O є дерево (називане порожнім деревом);

(b) якщо t1 і t2 - дерева, то



є дерево (намальоване зверху вниз).

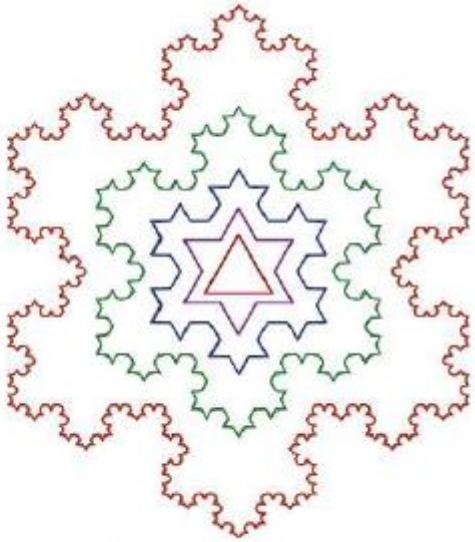
3. Функція факторіал *n!* для ненегативних цілих чисел:

(a) 0! = 1,

(b) якщо n> 0, то n! = n*(n-1)!

4. Фрактали (самоподібні множини нецілої розмірності)

Треугольники Серпинского



Кривые Коха

5. Числа Фібоначчі.

$$f(n) = \begin{cases} 1, & n \le 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

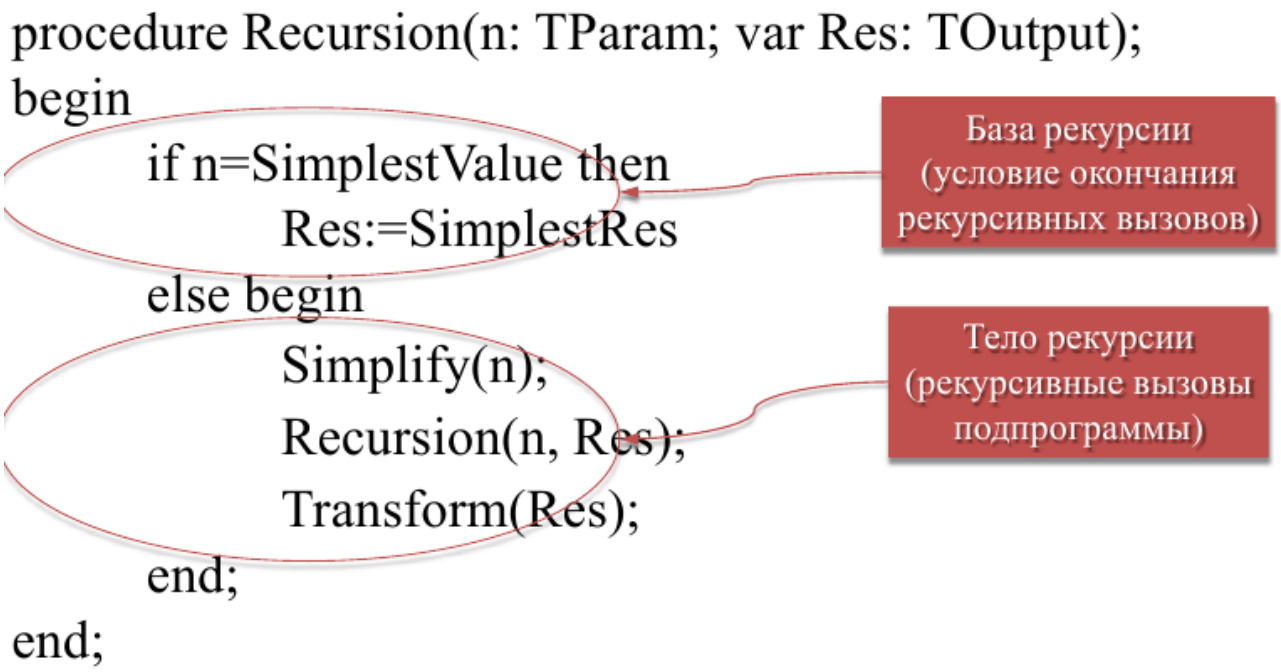
```
function Fib(n: Integer): Integer;
begin
        if (n=1) or (n=2) then
                Result:=1
        else
                Result:=Fib(n-
1)+Fib(n-2);
end;
```

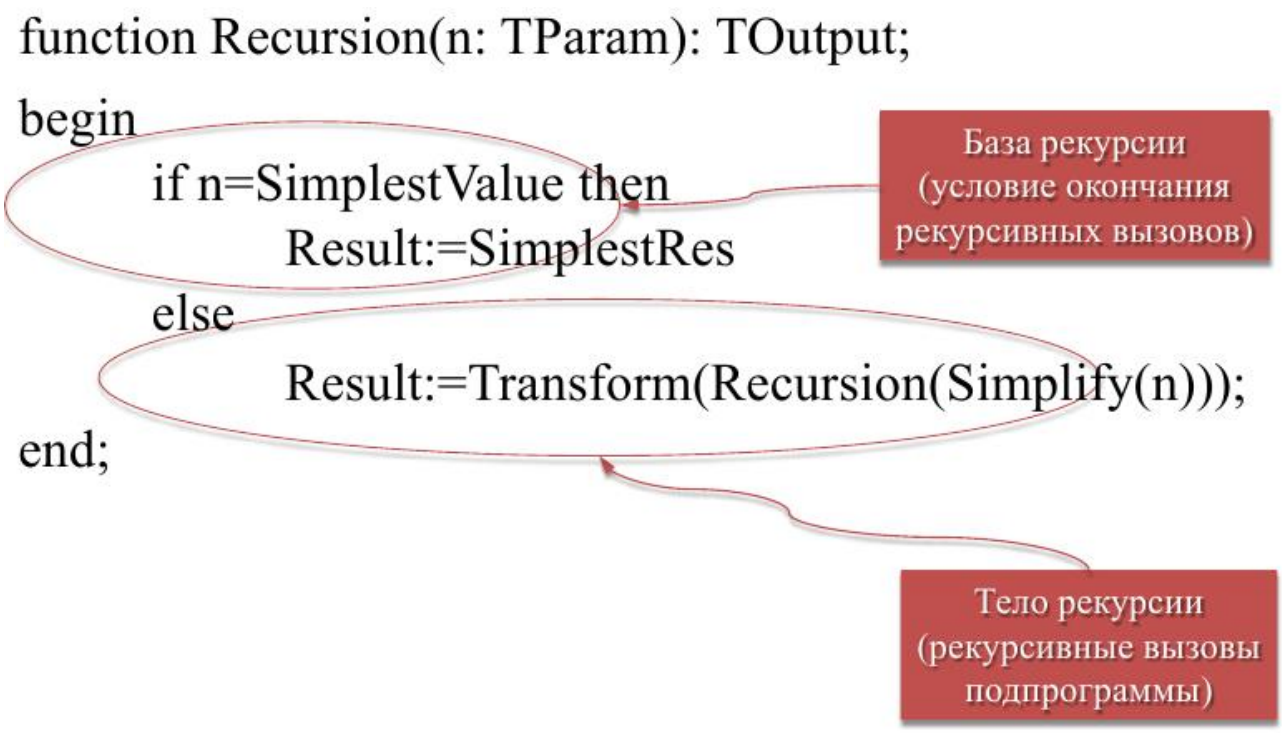
Очевидно, що потужність рекурсії пов'язана з тим, що вона дозволяє визначити нескінченну множину об'єктів за допомогою кінцевого висловлення. Точно так само нескінченні обчислення можна описати за допомогою кінцевої рекурсивної програми, навіть якщо ця програма не містить явних циклів. Однак **найкраще** використовувати рекурсивні алгоритми в тих випадках, коли розв'язувана задача, або обчислювана функція або оброблювана структура даних визначені за допомогою рекурсії.

Необхідний й достатній засіб для рекурсивного подання програм - це опис процедур, або підпрограм, тому що він дозволяє привласнювати якому-небудь операторові ім'я, за допомогою якого можна викликати цей оператор.

Програми, у яких використаються рекурсивні процедури відрізняються простотою, наочністю й компактністю тексту. Такі якості рекурсивних алгоритмів випливають із того, що рекурсивна процедура вказує що потрібно робити, а нерекурсивна більше акцентує увагу на тім, як потрібно робити.

Якщо процедура P містить явне звертання до самої себе, то вона називається **прямо рекурсивною**; якщо P містить звертання до процедури Q, що містить (прямо або побічно) звертання до P, те P називається **побічно рекурсивною**. Тому використання рекурсії не завжди відразу видно з тексту програми.





Із процедурою прийнято зв'язувати деяку множина локальних об'єктів, тобто змінних, констант, типів і процедур, які визначені локально в цій процедурі, а поза її не існують або не мають змісту. Щораз, коли така процедура рекурсивно викликається, для неї створюється нова множина локальних змінних. Хоча вони мають ті ж імена, що й відповідні елементи множини локальних змінних, створеного при попереднім звертанні до цієї ж процедури, їхні значення різні. Наступні правила області дії ідентифікаторів дозволяють виключити який-небудь конфлікт при використанні імен: ідентифікатори завжди посилаються на множину змінних, створене останнім. Те ж правило ставиться до параметрів процедури.

Подібно операторам циклу, рекурсивні процедури можуть привести до нескінченних обчислень. Тому необхідно розглянути проблему закінчення роботи процедур. Очевидно, що для того, щоб робота коли-небудь, завершилася, необхідно, щоб рекурсивне звертання до процедури P підкорялося **умові B**, що у якийсь момент перестає виконуватися.

Основний спосіб довести, що виконання операторів циклу коли-небудь, закінчується, - визначити функцію f(x) (x - множина змінних програми), таку, що f(x)<=0 задовольняє умові закінчення циклу (із предумовою або з

постумовою), і довести, що при кожному повторенні f(x) зменшується. Точно так само можна довести, що виконання рекурсивної процедури P коли-небудь, завершиться, показавши, що кожне виконання P зменшує f(x). Найбільш надійний спосіб забезпечити закінчення процедури - зв'язати з P параметр (значення), скажемо n, і рекурсивно викликати P зі значенням цього параметра n-1.

На практиці потрібно обов'язково переконатися, що найбільша глибина рекурсії не тільки кінцева, але й досить мала. Справа в тому, що при кожному рекурсивному виклику процедури P виділяється деяка пам'ять для розміщення її змінних. Крім цих локальних змінних потрібно ще зберігати поточний стан обчислень, щоб повернутися до нього, коли закінчиться виконання нової активації P и потрібно буде повернутися до старого.

Реалізація механізму рекурсії.

При виклику підпрограми її локальні змінні й фактичні параметри містяться в сегмент стека.

Прямий хід рекурсії. Рекурсивний виклик підпрограми залишає в сегменті стека дані, поміщені туди на попередньому виклику підпрограми, і записує на вершину стека дані поточного виклику.

Зворотний хід рекурсії. При досягненні бази рекурсії рекурсивні виклики припиняються й починається серія завершень викликів з добуванням зі стека збережених значень і використанням їх для продовження обчислень. Останнім завершується перший виклик.

```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then Result:=1
        else
                Result:=n*Factorial(n-1);
end;
```

Прямой ход

Factorial(5)
5*Factorial(4)
5*4*Factorial(3)
5*4*3*Factorial(2)
5*4*3*2*Factorial(1)

```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then Result:=1
        else
                Result:=n*Factorial(n-1);
end;
```

Обратный ход

Factorial(5) **5*24=120**
5*Factorial(4) **4*6=24**
5*4*Factorial(3) **3*2=6**
5*4*3*Factorial(2) **2*1=2**
5*4*3*2*Factorial(1) **1**

Рекурсія й ітерації

Рекурсія

- вирішує задачу методом "від складного до простого"; алгоритм - опис складного об'єкта через більше простий (прості) об'єкт того ж типу

- запис алгоритму, як правило, компактна, інтуїтивно зрозуміла, гарна
- виконання алгоритму пов'язане з більшими накладними витратами й, як правило, більше повільне, чим ітеративне

Ітерація

- вирішує задачу методом "від простого до складного"; алгоритм - опис процесу побудови складного об'єкта з більше простих об'єктів
- запис алгоритму не завжди компактна, інтуїтивно зрозуміла
- виконання ітеративного алгоритму, не пов'язане з більшими накладними витратами й, як правило, більше швидке, чим рекурсивне

Приклад: обчислення факторіала

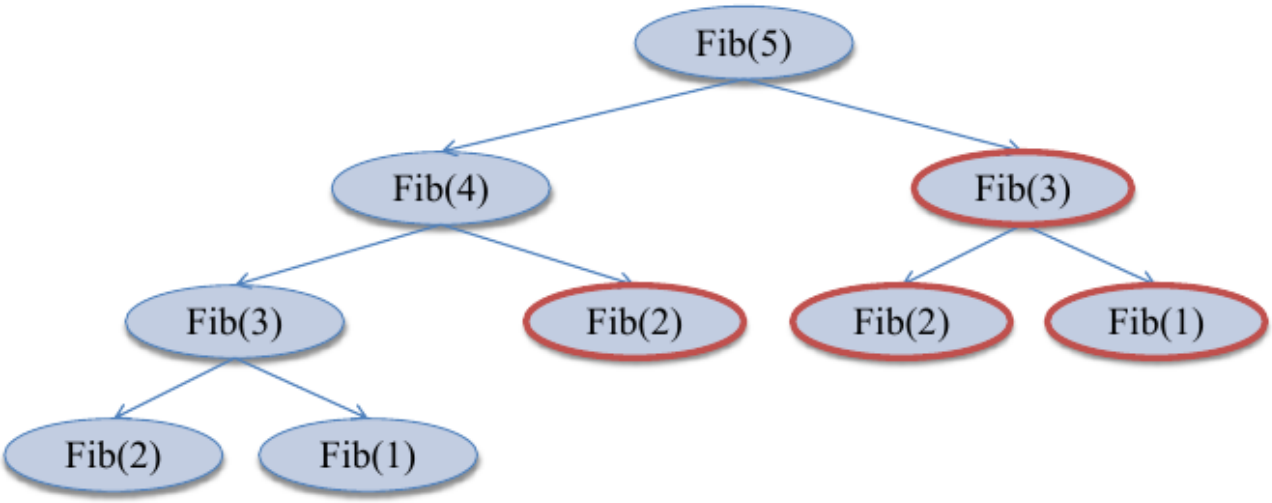
```
function Factorial(n: Integer): Integer;
begin
        if (n=0) or (n=1) then
                Result:=1
        else

        Result:=n*Factorial(n-1);
end;
```

```
function Factorial(n: Integer): Integer;
var i, Res: Integer;
begin
        Res:=1;
        for i:=2 to n do
                Res:=i*Res;
        Result:=Res;
end;
```

Приклад: обчислення ряду Фібоначчі.

Оператор Result:=Fib(n-1)+Fib(n-2) породжує "зайві" рекурсивні виклики




```
function Fib(n: Integer): Integer;
begin
   if (n=1) or (n=2) then
      Result:=1
   else
      Result:=Fib(n-1)+Fib(n-2);
end;
```

$$Fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

```
function Fib(n: Integer): Integer;
var i, n1, n2, Res: Integer;
begin
    N1:=1; N2:=1;
    Result:=1;
    for i:=3 to n do begin
        Res:=n1+n2;
        n2:=n1;
        n1:=Res;
    end;
    Result:=Res;
end;
```

Задача про Ханойські вежі.

В одному з буддійських монастирів ченці вже тисячу років займаються перекладанням кілець. Вони розташовують трьома пірамідами, на яких нанизані кільця різних розмірів.

У початковому стані 64 кільця були нанизані на першу піраміду й упорядковані по розмірі. Ченці повинні перекласти всі кільця з першої піраміди на другу, виконуючи єдину умову - кільце не можна покласти на кільце меншого розміру. При перекладанні можна використати всі три піраміди.

Ченці перекладають одне кільце за одну секунду. Як тільки вони закінчать свою роботу, наступить кінець світу.

Рекурсивний алгоритм рішення задачі "Ханойські вежі" пересунути потрібна кількість дисків з піраміди джерело на піраміду завдання, використовуючи запасну піраміду

- Якщо кількість дисків дорівнює одному, тоді пересунути диск із джерела в завдання
- У противному випадку:

рекурсивно пересунути всі диски, крім останнього, із джерела в запас, використовуючи завдання як запас

пересунути диск, що залишився, із джерела в завдання

пересунути всі диски із запасу в завдання, використовуючи джерело як запасло

2. КОЛИ НЕ ПОТРІБНО ВИКОРИСТОВУВАТИ РЕКУРСІЮ

Рекурсивні алгоритми найбільш придатні у випадках, коли поставлена задача або використовувані дані визначені рекурсивно. Але це не виходить, що при наявності таких рекурсивних визначень кращим способом рішення задачі неодмінно є рекурсивний алгоритм.

Варто уникати рекурсії, коли є очевидне ітеративне рішення поставленої задачі.

2.

Метод проектування алгоритму (algorithm design technique) (або "стратегія", або "принцип") — це універсальний підхід, застосовуваний для алгоритмічного рішення широкого кола задач, що ставляться до різних областей обчислювальної техніки.

Метод грубої сили являє собою прямий підхід до розв’язання задачі, звичайно заснований безпосередньо на формулюванні задачі й визначеннях використовуваних нею концепцій.

Перефразувати визначення даної стратегії можна простіше: "Нема чого думати, треба діяти!". Найчастіше стратегія грубої сили виявляється найбільш простою у застосуванні.

Як приклад розглянемо задачу піднесення в ступінь: обчислення a^n для деякого числа a й ненегативного цілого n. Хоча ця задача може здатися тривіальної, вона дозволяє проілюструвати кілька методів розробки алгоритмів, у тому числі й підхід грубої сили. По визначенню піднесення в ступінь

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ раз}}.$$

Звідси відразу треба найпростіший алгоритм обчислення a^n — шляхом множення на a початкового значення, рівного 1, n раз.

Ми вже зустрічалися на попередніх лекціях як мінімум із двома алгоритмами з використанням грубої сили: послідовна перевірка всіх цілих чисел при пошуку *gcd* (*m*,*n*) і алгоритм множення матриць, заснований на визначенні даної операції. Багато інших прикладів ви знайдете на цій лекції. (чи можете ви вказати трохи відомих вам алгоритмів, заснованих на використанні грубої сили?)

Хоча метод грубої сили рідко дає красиві або ефективні алгоритми, його розгляд не можна опустити, оскільки даний метод являє собою важливу стратегію розробки алгоритмів. По-перше, на відміну від інших стратегій, метод грубої сили застосуємо до дуже широкого діапазону задач. (Схоже, це єдиний підхід, для якого істотно складніше вказати задачу, для рішення якої він незастосовний.) Зокрема, метод грубої сили використається для багатьох елементарних, але важливих алгоритмічних задач, таких як обчислення суми n чисел, пошук найбільшого елемента в списку й тому подібних. По-друге, для деяких важливих задач (наприклад, сортування, пошуку, множення матриць, пошуку підстрок) метод грубої сили дає цілком раціональні алгоритми. По-третє, вартість розробки більше ефективного алгоритму може виявитися неприйнятної, якщо потрібно вирішити тільки кілька екземплярів задачі, а алгоритм, заснований на грубій силі, дозволяє вирішити їх за прийнятний час. По-четверте, навіть будучи неефективним у загальному випадку, метод грубої сили може виявитися корисний для рішення невеликих по розмірі екземплярів задачі.

Нарешті, алгоритм, заснований на грубій силі, може служити для важливих теоретичних або дидактичних цілей, наприклад мірилом для визначення ефективності інших алгоритмів для рішення даної задачі.

3.

Розглянемо застосування методу грубої сили до задачі сортування, що полягає в наступному: даний список з n елементів, що впорядковуються, (наприклад, чисел, символів деякого алфавіту, символьних рядків) , які треба розмістити в неубутному порядку. Є десятки алгоритмів, розроблених для рішення цієї дуже важливої задачі. Деякі з них вам уже, можливо, зустрічалися. У такому випадку спробуйте на час забути про їх.

Бульбашкове сортування

Застосування методу грубої сили до задачі сортування складається в порівнянні сусідніх елементів і їхньому обміні, якщо вони перебувають не в належному порядку. Неодноразово виконуючи цю дію, ми змушуємо найбільший елемент "спливати" до кінця списку. Наступний прохід приведе до спливання другого найбільшого елемента, і так доти, поки після *n-1* ітерації список не буде повністю відсортований, *i-ий* прохід *0<=i<=n-2*) бульбашкового сортування можна представити наступною діаграмою:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leq \cdots \leq A_{n-1}$$

Элементы в окончательных позициях

Нижче наведений псевдокод даного алгоритму.

АЛГОРИТМ $BubbleSort\,(A\,[0..n-1])$

// Сортировка массива пузырьковой сортировкой
// **Входные данные:** Массив $A[0..n-1]$ упорядочиваемых
// элементов
// **Выходные данные:** Массив $A[0..n-1]$, отсортированный
// в неубывающем порядке
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$
      Обмен $A[j]$ и $A[j+1]$

Кількість порівнянь ключів у даній версії бульбашкового сортування однаково для всіх масивів розміром *n*. Воно представляється сумою, практично ідентичній сумі для сортування вибором:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] =$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)\,n}{2} \in \Theta\left(n^2\right).$$

Кількість же обмінів елементів залежить від вхідних даних. У найгіршому випадку зменшуваного масиву воно дорівнює кількості порівнянь:

$$S_{worst}(n) = C(n) = \frac{(n-1)\,n}{2} \in \Theta\left(n^2\right).$$

Найчастіше при використанні методу грубої сили перша версія алгоритму може бути вдосконалена ціною досить скромних зусиль. Зокрема, наведену сиру версію бульбашкового сортування можна поліпшити, скориставшись наступним спостереженням: якщо при проході за списком не зроблено жодного

обміну, виходить, список відсортований, і виконання алгоритму можна припинити.

Сортування вибором

Ми починаємо сортування вибором з пошуку найменшого елемента в списку й обміну його з першим елементом (таким чином, найменший елемент міститься в остаточну позицію у відсортованому списку). Потім ми скануємо список, починаючи із другого елемента, у пошуках найменшого серед що залишилися *n - 1* елементів і обмінюємо знайдений найменший елемент із другим, тобто поміщаємо другий найменший елемент в остаточну позицію у відсортованому списку. У загальному випадку, при *i-ом* проході за списком (0 =< *i* =< *n-2*) алгоритм шукає найменший елемент серед останніх *n-i* елементів і обмінює його з A_i.

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{Элементы в окончательных позициях}} \mid \overbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}^{\text{Последние } n-i \text{ элементов}}$$

Після виконання *n - 1* проходів список виявляється відсортований.

От псевдокод даного алгоритму, у якому для простоти передбачається, що список реалізований у вигляді масиву.

```
АЛГОРИТМ SelectionSort (A[0..n − 1])
  // Сортировка массива методом выбора
  // Входные данные:  Массив A[0..n − 1] упорядочиваемых
  //                  элементов
  // Выходные данные: Масєив A[0..n − 1], отсортированный
  //                  в неубывающем порядке
  for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
      if A[j] < A[min]
        min ← j
    Обмен A[i] и A[min]
```

Аналіз сортування вибором виконується досить просто. Розмір вхідних даних визначається кількістю *n* елементів у списку. Базовою операцією алгоритму є порівняння ключів *A [j] < A [min]*. Загальна кількість порівнянь залежить тільки від розміру масиву й визначається наступною сумою:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

Обчислите зазначену суму.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{(n-1)\,n}{2}.$$

Таким чином, для будь-яких вхідних даних алгоритм сортування вибором належить $\Theta(n^2)$. Помітимо, однак, що кількість обмінів елементів масиву дорівнює $\Theta(n)$, точніше, рівно n-1 — по одному для кожної ітерації циклу *i*. Ця властивість відрізняє сортування вибором від багатьох інших алгоритмів сортування.

Сортування вставкою.

Самостійне вивчення.