academy

Алгоритми

Лекція 1

План лекції

- Складність алгоритмів
- Алгоритм бінарного пошуку
- Рекурсія
- Задача на закріплення знань

Складність алгоритмів

Де це мені пригодиться?

- На парах матану
- На співбесідах
- Коли необхідна оптимізація алгоритму

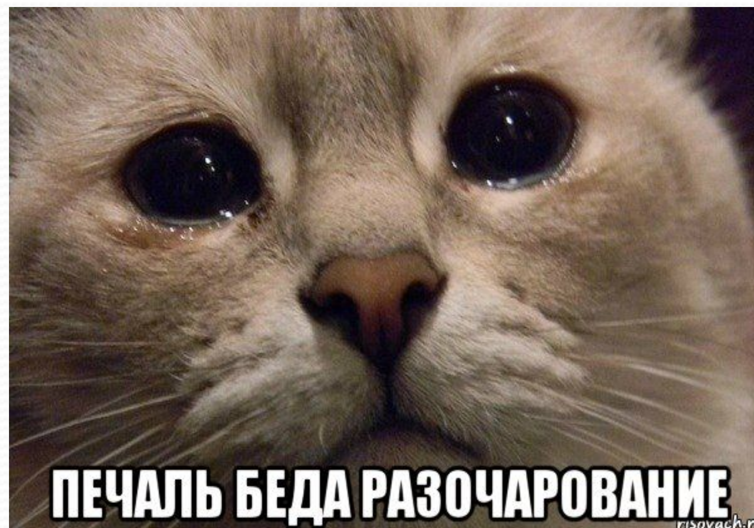
Профайлери

Профайлери – інструменти для вимірювання швидкості виконання програм.

```
clock_t t;
cout << "Start calculation..." << endl;
t = clock();              /* get current time */
some_stupid_calc(14/88);
t = clock() - t;          /* substract saved time (t)
from current time */
cout << "Execution time: " << (float)t << "
sec"<<endl;
```

Це дуже зручний інструмент, але він не має ніякого зв’язку зі складністю алгоритмів.

Алгоритм	Супер повільний	Оптимізований
Мова реалізації	Assembler	Java
Швидкість виконання	~ 0.001 сек.	~ 0.1 сек.



Суть

- *Складність алгоритму* - це спосіб його оцінки без прив’язки до низькорівневих деталей таких як реалізація мови програмування чи апаратне забезпечення.
- *Ціль* – розглянути алгоритм з точки зору ідеї, на якій базуються обчислення.

Приклад: знайти максимальний елемент масиву.

```
int max = a[0];
for(i=0; i<n; i++)
    if(a[i] > max)
        max = a[i];
```

Спробуємо порахувати кількість операцій…

Для аналізу цього коду треба поділити його на атомарні операції (прості інструкції, які будуть виконуватися за однаковий проміжок часу, на приклад за 1 такт процесора).

До атомарних операцій віднесемо:

- Присвоєння значення
- Доступ до елемента масиву по індексу
- Порівняння двох значень
- Основні арифметичні операції (*, +)

- Перша строчка містить 2 операції: доступ до елемента масиву по індексу a[0] та присвоєння значення max = a[0];
- Ініціалізація циклу потребує 2 операції: присвоєння i = 0 та порівняння $i < n$;
- Кожен крок циклу (за умови порожнього тіла) проводить порівняння $i < n$ та присвоєння i++ (ще 2n операцій);

В загальному прохід порожнього циклу займе 4 + 2*n операцій.

Тепер можемо визначити функцію f(n), таким чином, що знаючи n отримаємо кількість інструкцій необхідну для роботи алгоритму.

Для циклу for з порожнім тілом

$$f(n) = 4 + 2n.$$

Аналіз найгіршого випадку

- Тіло циклу містить 2 операції `if(a[i] > max)` (доступ до елемента масиву та порівняння);
- Але тіло умови `max = a[i]`це 2 операції) буде виконуватись не завжди, що ускладнює точну оцінку кількості операцій;
- В такому разі говорять про найгірший випадок – коли вважаємо що алгоритм виконує максимально можливу кількість інструкцій;
- Отже тіло циклу в найгіршому випадку виконує 4n операцій, а $f(n) = 4 + 2n + 4n = 6n + 4$.

В теорії складність алгоритму характеризують трьома варіантами вхідних даних:

- найкращий випадок
- середній випадок
- найгірший випадок

Для обраного прикладу найкращий випадок буде коли максимальний елемент стоїть на першому місці, а найгірший – масив відсортований по зростанню (доведеться робити присвоєння на кожній ітерації).

Асимптотична складність

- Під час аналізу алгоритму найбільшу цікавість викликає поведінка f(n) при великих значеннях n.
- Якщо алгоритм працює швидко з великим набором даних то є велика ймовірність що він буде так же добре працювати з малими наборами.

$$f(n) = 6n + 4 \quad \Rightarrow f(n) = n$$

- Іншими словами асимптотична складність являє собою ліміт f(n) при n прямує до нескінченності.

ДАВАЙТЕ ПОМОЖЕМ ДАШЕ
НАЙТИ АСИМПТОТИЧЕСКУЮ
СЛОЖНОСТЬ
risovach.ru

Знайти асимптотичну складність:

$f(n) = 5n + 12$

$f(n) = 109$

$f(n) = n^2 + 3n + 112$

$f(n) = n + \text{sqrt}(n)$

$f(n) = n^3 + 1999n + 1337$

$f(n) = n$

$f(n) = 1$

$f(n) = n^2$

$f(n) = n$

$f(n) = n^3$

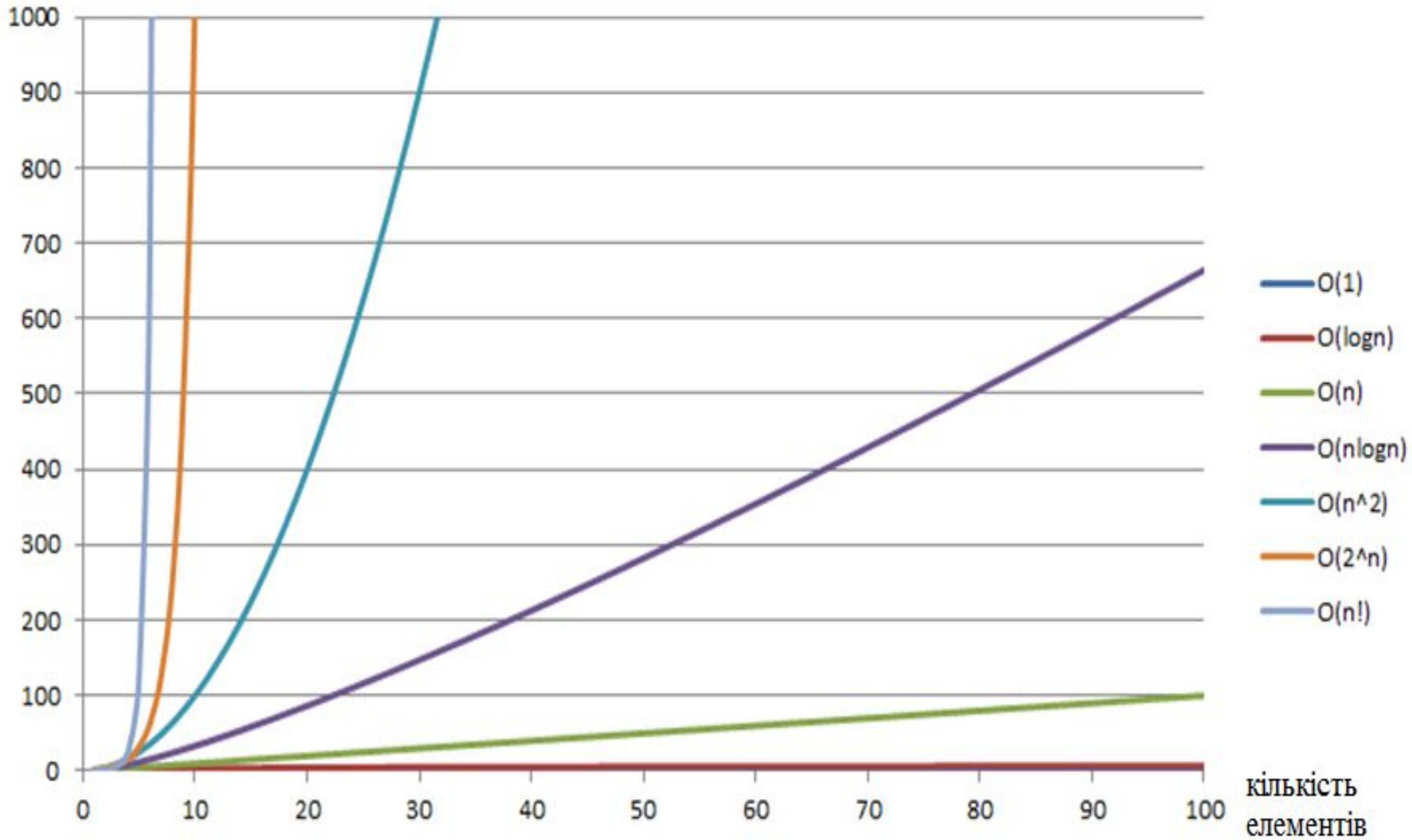
Нотації асимптотичної складності

Позначення	Межі	Характер
Θ (Тета)	Нижня та верхня границі, точна оцінка	Дорівнює
O (O-велике)	Верхня границя, точна оцінка невідома	Менше або дорівнює
o (o-мале)	Верхня границя, не точна оцінка	Менше
Ω (Омега-велике)	Нижня границя, точна оцінка невідома	Більше або дорівнює
ω (Омега-мале)	Нижня границя, не точна оцінка	Більше

Коротше кажучи, якщо алгоритм має складність ___, тоді його ефективність ___

Складність	Ефективність
$\Theta(N)$	= N
$O(N)$	<= N
$o(N)$	< N
ОМЕГА(N)	>= N
$\omega(N)$	> N

Графік росту О-велике



Рекомендації по вибору складності

В залежності від об'єму вхідних даних N важливо правильно оцінити складність алгоритму, який би забезпечив оптимальний час роботи програми.

Порядок N	10^6	10^5	10^4	10^3	10^2
Складність	O(N)	O(N*log N)	$O(N^{3/2})$	$O(N^2)$	$O(N^3)$

Алгоритм бінарного пошуку

Бінарний пошук є найефективнішим пошуком даних у відсортованому масиві.

Він досить простий у реалізації, суттєво швидший за лінійний, що сприяло його широкому застосуванню у програмуванні та багатьох інших сферах діяльності.

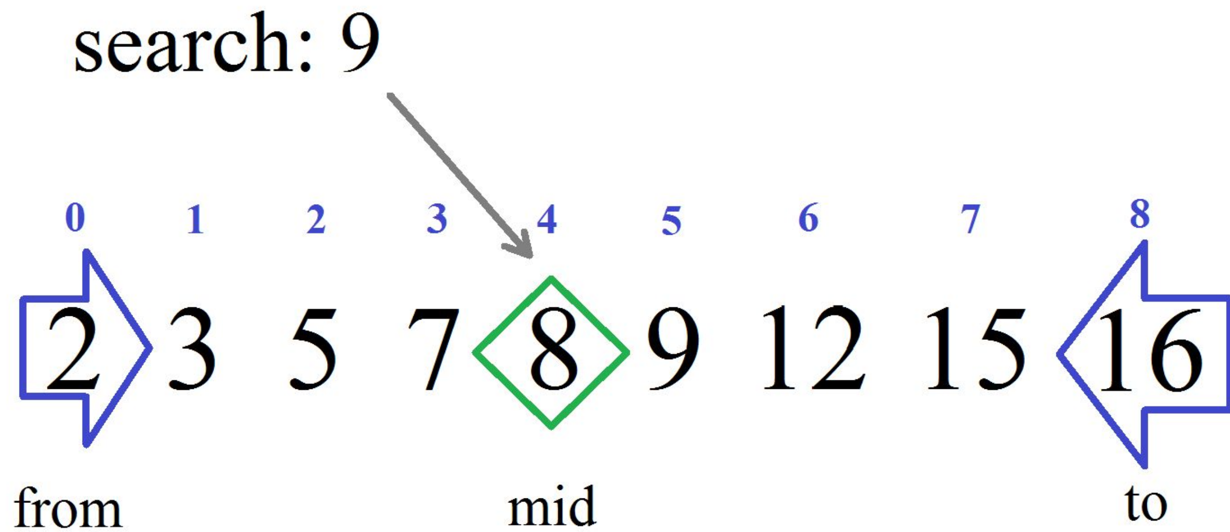
Алгоритм бінарного пошуку

1) Знаходимо індекс середнього елементу
2) Порівнюємо значення, яке шукаємо з середнім елементом масиву. Тут розглядаємо 3 можливі випадки:

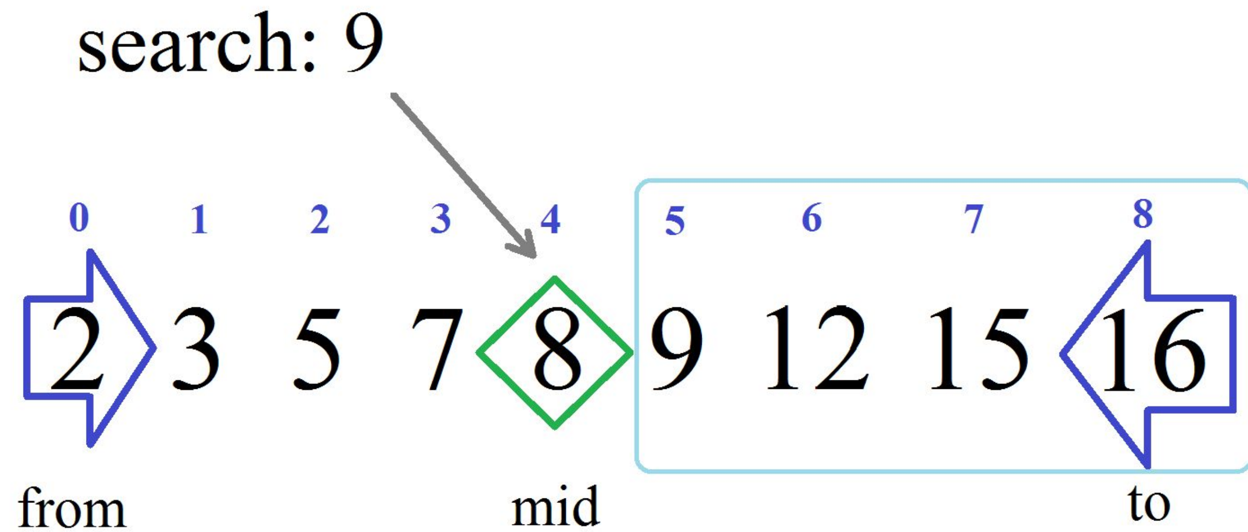
- шукане значення = середньому елементу, тоді пошук завершено;
- шукане значення < середнього елементу, тоді здійснюємо пошук у першій половині масиву;
- шукане значення > середнього елементу, тоді здійснюємо пошук у першій половині масиву.

Продовжуємо поки інтервал пошуку не перетвориться в одне число або поки не буде знайдений елемент.

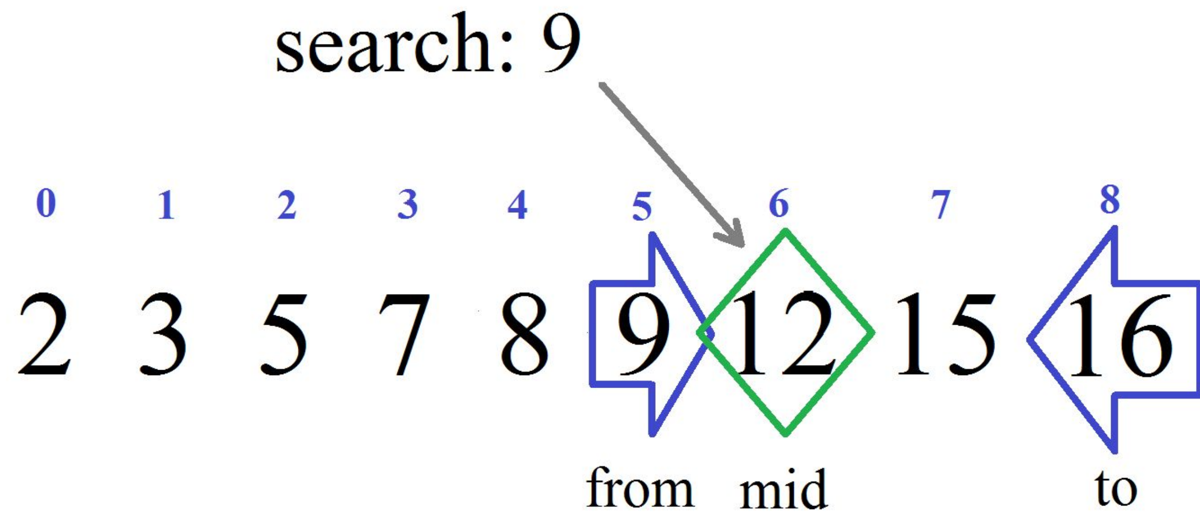
Алгоритм бінарного пошуку



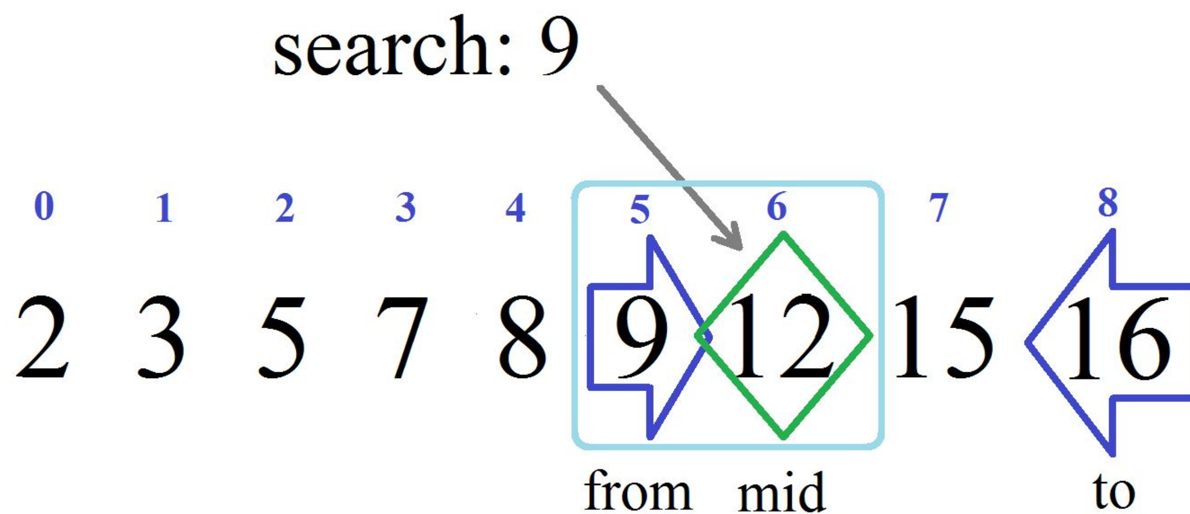
Алгоритм бінарного пошуку



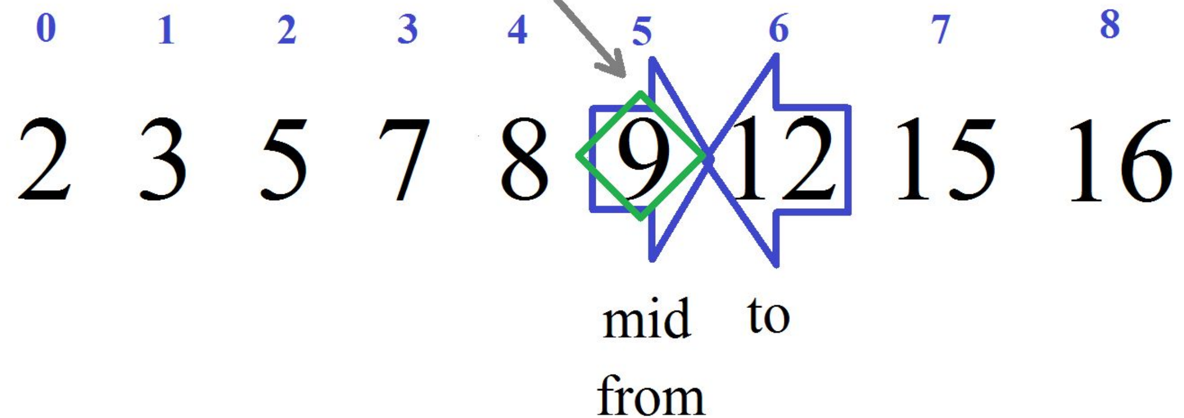
Алгоритм бінарного пошуку



Алгоритм бінарного пошуку



Алгоритм бінарного пошуку



Реалізація

- Функція повертає індекс елементу пошуку, тому на початку алгоритму присвоїмо йому значення -1, яке і буде повертати функція якщо елемент не знайдено.
- Обов'язковою умовою використання алгоритму бінарного пошуку є відсортований масив пошуку.
- Формула mid = (from + to)/2 може призвести до переповнення пам'яті, якщо from і to будуть занадто великі, їх сума вийде за межі типу int, тому рекомендується mid = from - (from - to)/2 щоб уникнути цього.

Реалізація

```
int binarySearch(int arr[], int search)
{
    int searchIndex = -1;
    int from = 0;
    int to = SIZE;
    int mid;
    std::sort(arr, arr + SIZE);
    if (search < arr[0] || search > arr[SIZE-1]) {
        searchIndex = -1;
    } else {
        while ((to - from) >= 0) {
            mid = from - (from - to)/2; /* the same that (from + to)/2 but avoid */
            if (arr[mid] < search)      /* overflow: sizeof(from+to) > sizeof(int) */
                from = mid + 1;
            else if (search < arr[mid])
                to = mid - 1;
            else if (search == arr[mid]) {
                searchIndex = mid;
                break;
            }
        }
    }
    return searchIndex;
}
```

Складність алгоритму

- Бінарний пошук вимагає єдиного проходу по масиву - O(n) але таким чином що він бере до уваги тільки певні елементи (з індексом mid). Таким чином кількість елементів які будуть розглянуті вираховується простою формулою:

$$\log_2 n.$$

- Отже складність алгоритму O(log n)

Рекурсія

Рекурсія – це виклик функції всередині цієї ж функції.

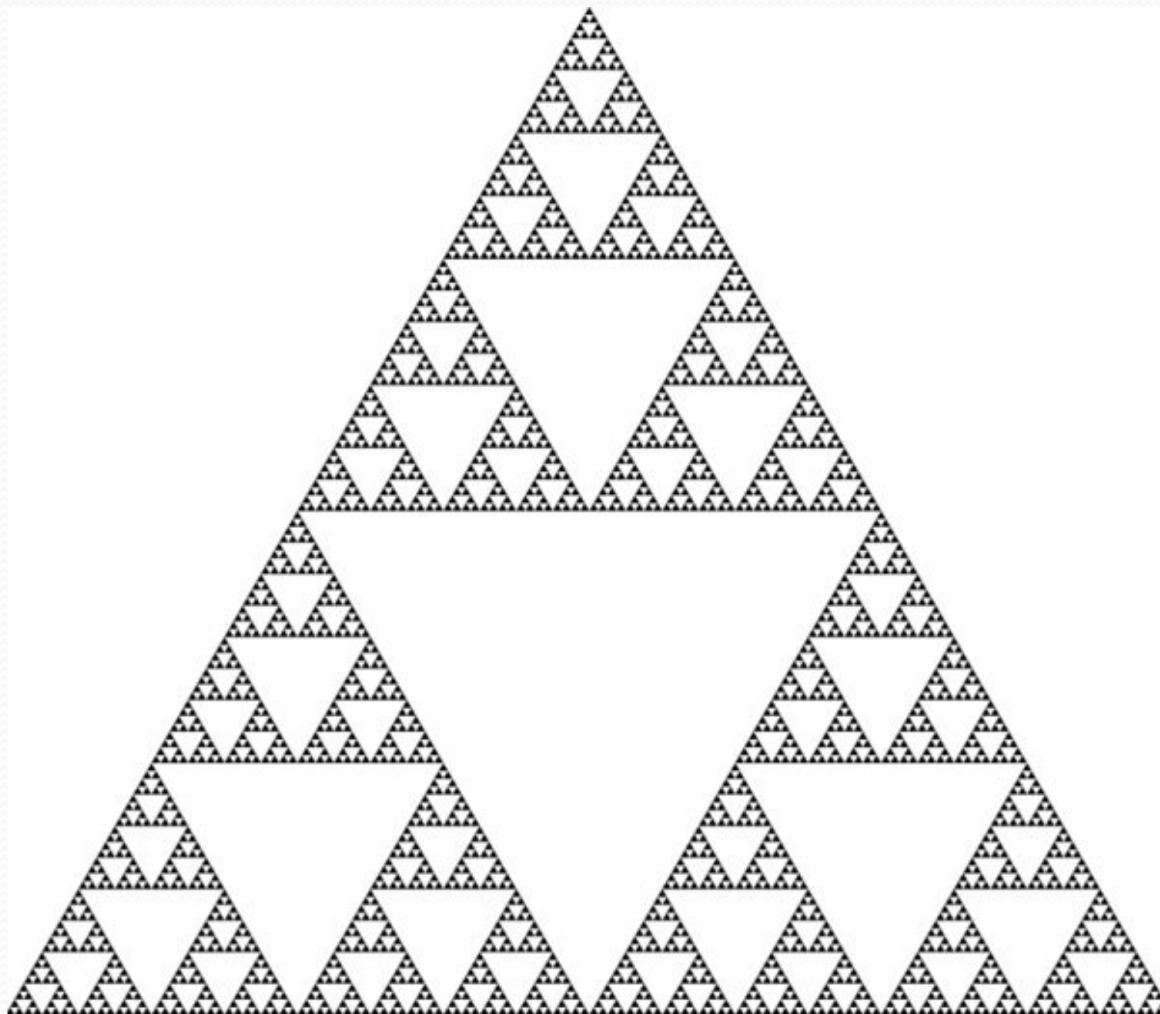
```
int recursion(int value) {
    doSomeStaffWith(value);
    if (value > 100500) {
        return value;
    }
    recursion(value);
}
```

Приклад рекурсії

Якщо абстрагуватись від програмування то рекурсія це повторення об’єкта всередині самого себе.

WINE is not emulator

- Трикутник Серпінського



Використання рекурсії

Знайти суму чисел від 1 до n

```
int sumOfFirstNNumbers(int n) {
    if (n == 0) {
        return 0;
    }
    return n + sumOfFirstNNumbers(n-1);
}
```

Використання рекурсії

Знайти факторіал числа n.

1) Реалізація через тернарний оператор

```
int factorial (int n)
{
   return (n < 2) ? 1 : n * factorial (n - 1);
}
```

2) Реалізація через умовний оператор

```
int factorial (int n)
{
    if (n < 2) {
        return 1;
    }
    return n * factorial (n - 1);
}
```

Завдання

- Реалізувати бінарний пошук використовуючи рекурсію.

Рекурсивний бінарний пошук

```
int recursiveBinarySearchRange(int from, int to, int arr[], int search)
{
    if ((to - from) >= 0) {
        int mid = from - (from - to)/2;
        if (arr[mid] < search)
            recursiveBinarySearchRange(mid + 1, to, arr, search);
        else if (search < arr[mid])
            recursiveBinarySearchRange(from, mid - 1, arr, search);
        else if (search == arr[mid])
            return mid;
    }
    else {
        return -1;
    }
}
```

Задача на закріплення знань

Є масив цілих чисел. Числа йдуть підряд від 1 до k, але в масиві пропущені два числа. Знайти ці числа.

1 _ 3 4 5 _ 7 8 9

Ваші ідеї?

Підхід до вирішення

- Використати (рекурсивний) бінарний пошук для пошуку відрізку між пропущеними числами
- Знайти бінарним пошуком по одному пропущеному елементу в лівій та правій частині.

- Для перевірки використати різницю між значенням та індексом елемента в масиві.

- Реалізуємо самостійно

Дякую за увагу!