

## Занятие 11

# Оценка сложности алгоритмов. Поиск и сортировка данных

# Цель занятия

Сформировать знания об оценке сложности алгоритмов, поиске и сортировке данных. Изучить алгоритмы линейного поиска и сортировки подсчетом.

## Программа занятия

1. Логарифм
2. Оценка сложности алгоритма. Нотация  $O$ -большое
3. Константные алгоритмы.  $O(1)$
4. Линейные алгоритмы.  $O(N)$
5. Логарифмические алгоритмы.  $O(k \log N)$
6. Полиномиальные алгоритмы.  $O(N^p)$
7. Экспоненциальные алгоритмы.  $O(k^N)$
8. Факториальные алгоритмы.  $O(N!)$
9. Поиск и сортировка данных
10. Линейный поиск
11. Сортировка подсчетом
12. Случайные числа. Функции `rand()` и `srand()`

# Логарифм

**ЛОГАРИФМ** положительного числа  $B$  по основанию  $A$  (обозначается, как  $\log_A B$ ) — это степень, в которую нужно возвести  $A$ , чтобы получить  $B$ .

Допустим, что  $\log_A B = C$ , тогда  $A^C = B$

$$B > 0$$

$$A > 0$$

$$A \neq 1$$

# Оценка сложности алгоритма. Нотация $O$ -большое

$O$

Опуская сложную математику, мы будем смотреть на алгоритм, как на некоторую функцию  $g(N)$  и будем сравнивать ее с  $O(C_0 \times f(N) + C_1)$

$N$  — количество входных данных нашего алгоритма

$C_0$  и  $C_1$  — некоторые постоянные значение, которые опускаются для простоты понимания полученной оценки.

# Оценка сложности алгоритма. Нотация $O$ -большое

## Константные алгоритмы. $O(1)$

```
int N = 1000;  
cout << "N = " << N << '\n';
```

Неважно какое значение содержит  $N$  — фрагмент кода выше всегда будет выполняться за константное время.

```
int N = 1000;  
cout << "N = " << N << '\n';  
cout << "N = " << N << '\n';  
cout << "N = " << N << '\n';
```

# Оценка сложности алгоритма. Нотация $O$ -большое

## Линейные алгоритмы. $O(N)$

$O(N)$  описывает алгоритм, чья скорость работы линейно и прямопропорционально зависит от размера входных данных.

```
bool isContainingValue(string text, char letter) {  
    for (size_t i = 0; i < text.length(); ++i) {  
        if (text.at(i) == letter) {  
            return true;  
        }  
    }  
  
    return false;  
}
```

Когда мы говорим, что что-то растет с линейной скоростью, мы имеем в виду, что оно растет прямопропорционально количеству его элементов (например, линейная функция  $f(x) = kx + b$ ).

# Оценка сложности алгоритма. Нотация $O$ -большое

## Линейные алгоритмы. $O(N)$

```
for (int i = 0; i < N; ++i) {  
    cout << i << '\n';  
}
```

Этот фрагмент выполнится  $N$  раз.

```
for (int i = 0; i < N; ++i) {  
    for (int j = 1; j <= 5; ++j) {  
        cout << i << ' ';  
    }  
    cout << '\n';  
}
```

# Оценка сложности алгоритма. Нотация $O$ -большое

## Логарифмические алгоритмы. $O(k \log N)$

$O(\log N)$  описывает алгоритмы, в которых происходит циклическое деление количества входных элементов.

```
for (int i = 1; i < N; i *= 2) {  
    cout << i << '\n';  
}
```

Если  $N = 8$ , то вывод будет следующим:

```
1  
2  
4
```

Таким образом, наш алгоритм выполнялся за  $\log_2 8 = 3$  единицы времени.



# Оценка сложности алгоритма. Нотация $O$ -большое

Логарифмические алгоритмы.  $O(k \log N)$

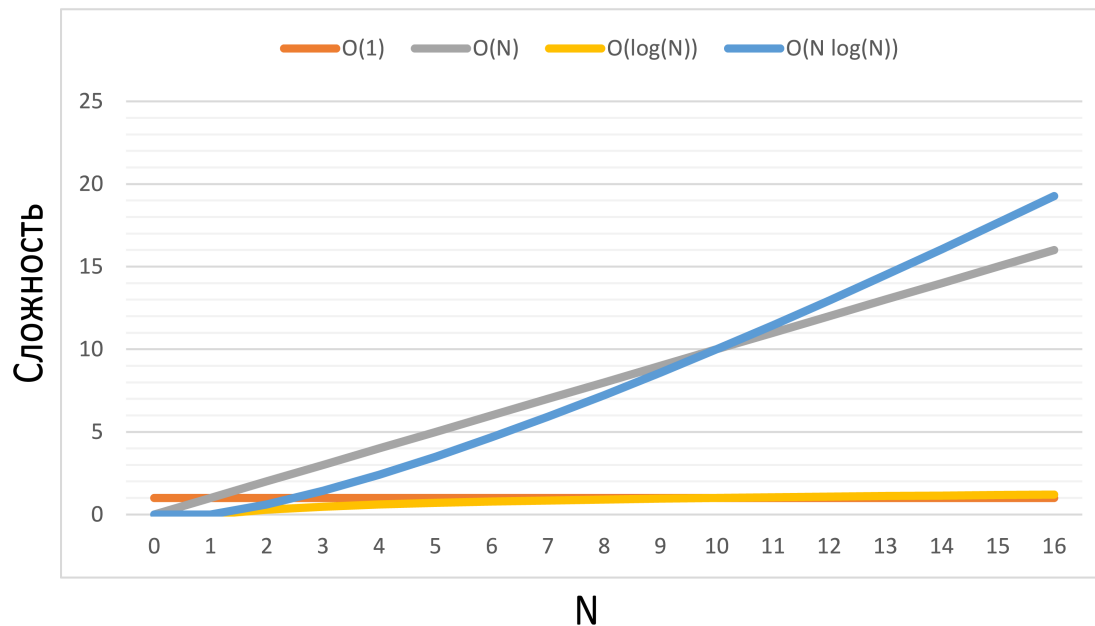
$O(N \log N)$

```
for (int i = 1; i <= N; ++i) {  
    for (int j = 0; j < N; j *= 2) {  
        cout << i << ' ' << j << '\n';  
    }  
}
```

Пусть  $N = 8$ , тогда алгоритм будет выполняться за  $8 \log_2 8 = 8 \times 3 = 24$  единицы времени.

# Оценка сложности алгоритма. Нотация $O$ -большое

Сравнение  $O(1)$ ,  $O(N)$ ,  $O(\log N)$  и  $O(N \log N)$



# Оценка сложности алгоритма. Нотация $O$ -большое

## Полиномиальные алгоритмы. $O(N^p)$

Термин **полиномиальный** — это обобщающий термин, включающий в себя квадратные ( $N^2$ ) функции, кубические ( $N^3$ ), квадратические ( $N^4$ ) и так далее. Наиболее важно то, что  $O(N^2)$  быстрее, чем  $O(N^3)$ , который, в свою очередь, быстрее, чем  $O(N^4)$  и так далее.

# Оценка сложности алгоритма. Нотация $O$ -большое

## Полиномиальные алгоритмы. $O(N^p)$

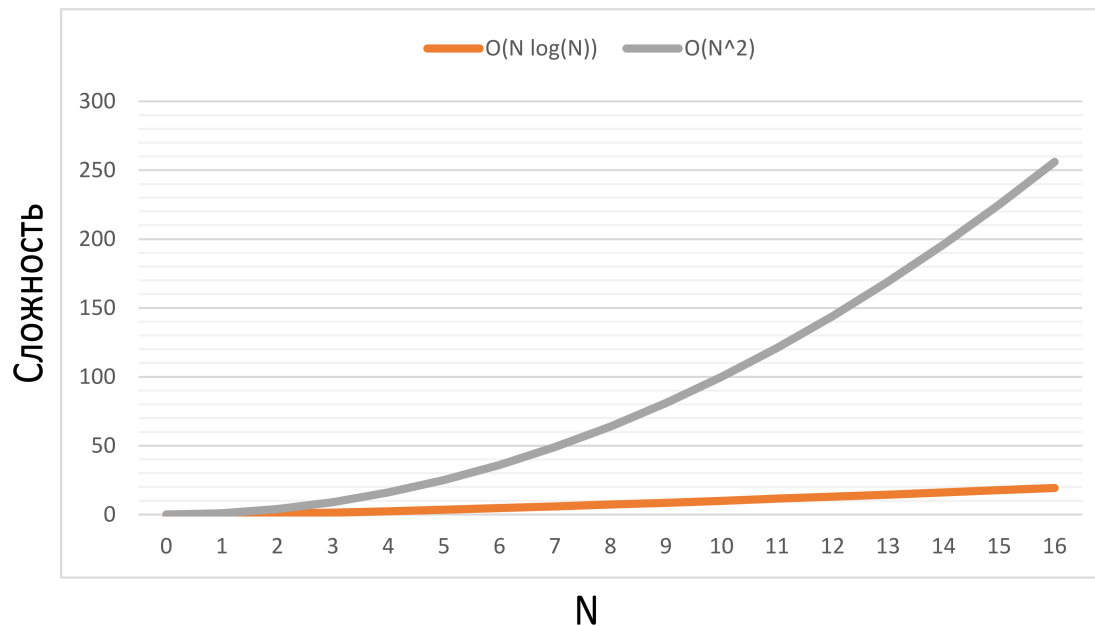
$O(N^2)$  описывает алгоритм, чья производительность прямопропорциональна квадрату размера входных данных.

```
bool containsDuplicates(string text) {  
    for (size_t outer = 0; outer < text.length(); ++outer) {  
        for (size_t inner = 0; inner < text.length(); ++inner) {  
            if (outer == inner) { continue; }  
            if (text.at(outer) == text.at(inner)) { return true; }  
        }  
    }  
    return false;  
}
```

Допустим, что строка `text` содержит 8 символов. Тогда алгоритм будет выполнен за  $8^2 = 64$  единицы времени.

# Оценка сложности алгоритма. Нотация $O$ -большое

Сравнение  $O(N \times \log N)$  и  $O(N^2)$



# Оценка сложности алгоритма. Нотация $O$ -большое

## Экспоненциальные алгоритмы. $O(k^N)$

Время исполнения этих алгоритмов возрастает в  $k$  раз при каждом дополнительном входном значении.

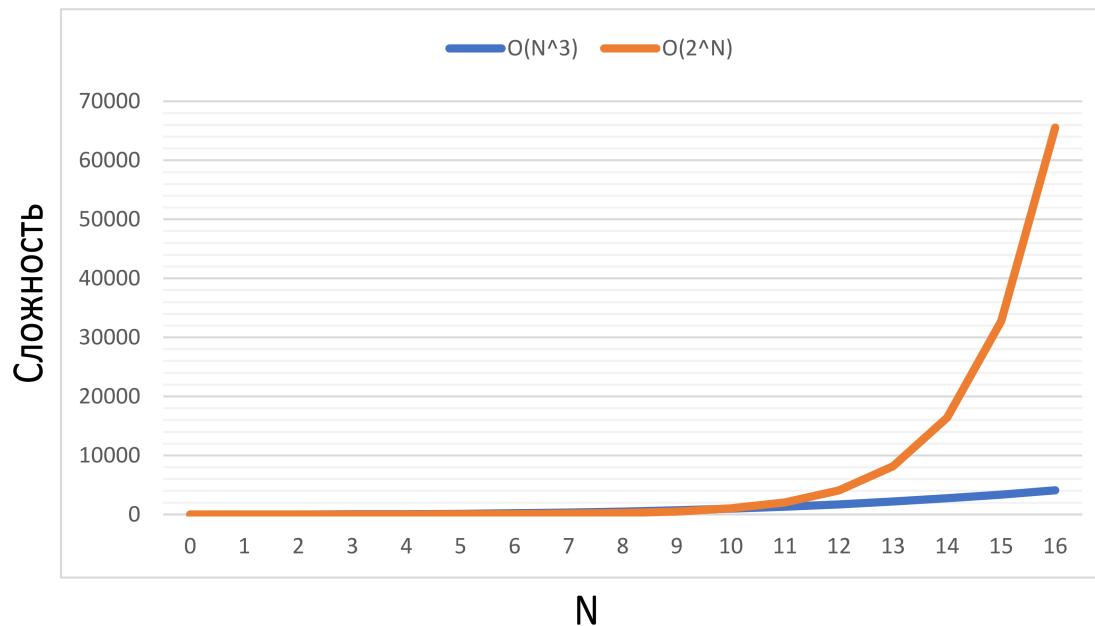
$O(2^N)$  описывает алгоритм, чей рост удваивается с каждым добавлением в множество входных данных. Примером такого алгоритма является рекурсивное вычисление  $n$ -ного числа Фибоначчи:

```
int fibonacci(int number) {  
    if (number <= 1) { return number; }  
    return fibonacci(number - 2) + fibonacci(number - 1);  
}
```

Если  $number = 8$ , то алгоритм выполнится за  $2^8 = 256$  единиц времени.

# Оценка сложности алгоритма. Нотация $O$ -большое

Сравнение  $O(N^3)$  и  $O(2^N)$



# Оценка сложности алгоритма. Нотация $O$ -большое

## Факториальные алгоритмы. $O(N!)$

Этот класс алгоритмов выполняется за время, прямопропорциональное факториалу количества входных данных.

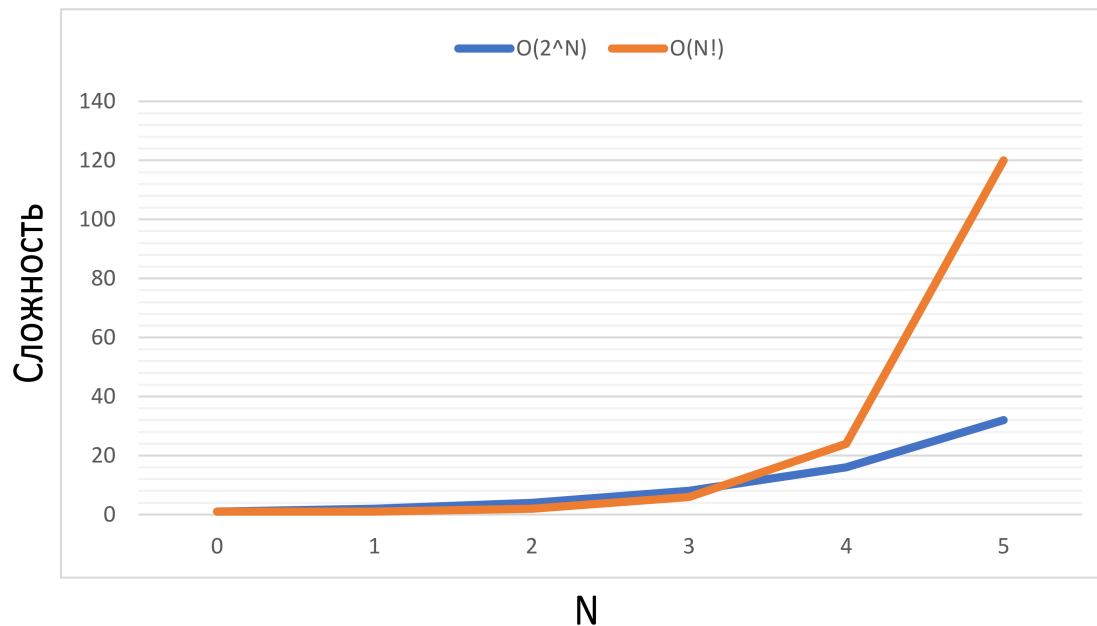
```
for (int i = 0; i < factorial(N); ++i) {  
    cout << i << '\n';  
}
```

Где `factorial(N)` — функция, вычисляющая факториал числа  $N$ . Допустим,  $N = 8$ , тогда алгоритм выполнится за  $8! = 40320$  единиц времени.



# Оценка сложности алгоритма. Нотация $O$ -большое

Сравнение  $O(2^N)$  и  $O(N!)$



# Поиск и сортировка данных

**Коллекция данных** — специальным образом организованное множество элементов.

**Поиск данных** — процесс поиска элемента коллекции по заданному параметру поиска.

**Сортировка данных** — процесс упорядочивания коллекции по заданному критерию сравнения (по какому принципу следует располагать элементы коллекции).

# Поиск и сортировка данных

## Линейный поиск

Пусть имеется целочисленный массив размера  $N$ . Необходимо найти все индексы, по которым в массиве расположено число 4.

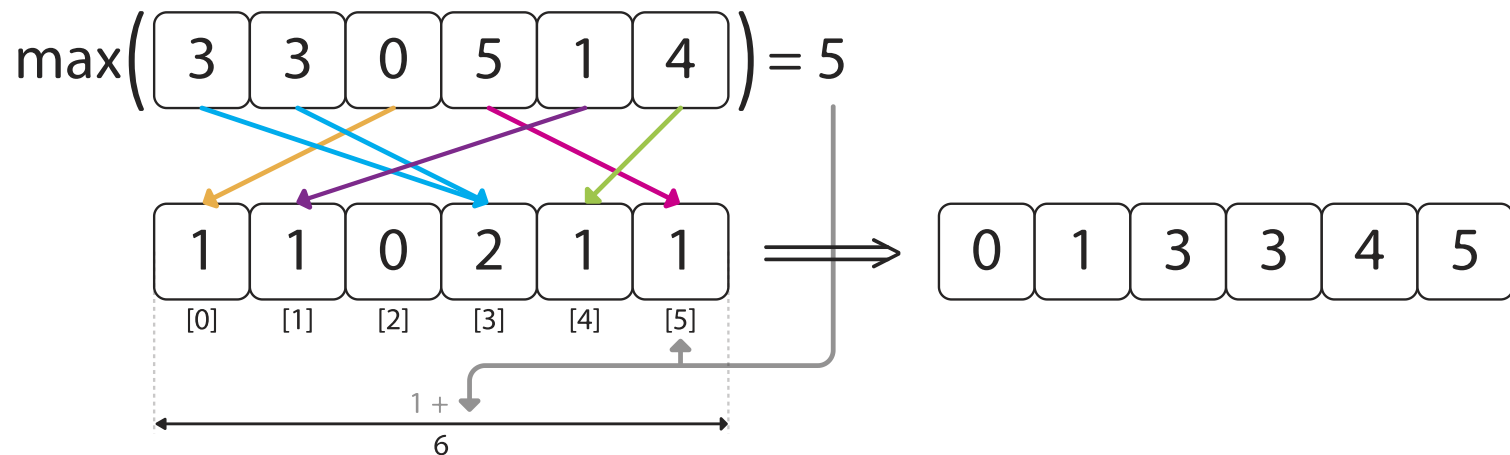
Используем линейный поиск, сравнивая каждый элемент массива с числом 4 и выводя все индексы, в которых было встречено совпадение:

```
for (int i = 0; i < N; ++i) {  
    if (arr[i] == 4) { cout << i << '\n'; }  
}
```

Время выполнения линейного поиска оценивается, как  $O(N)$ , потому что каждый элемент массива проверяется единожды.

# Поиск и сортировка данных

## Сортировка подсчетом



### Сложность алгоритма

- Время выполнения —  $O(N)$
- Используемая память —  $O(N)$

# Поиск и сортировка данных

## Сортировка подсчетом

### Шаги сортировки подсчетом

1. Найдем  $K$  — максимальное число в коллекции  $C$
2. Создадим дополнительную коллекцию  $T$  размера  $K + 1$
3. Пройдемся по коллекции  $C$ . Будем прибавлять единицу к элементам коллекции  $T$  с индексами, равными значениям элементов коллекции  $C$ .

```
for(size_t i = 0; i < N; ++i) { ++T[C[i]]; }
```

4. Пройдемся по коллекции  $T$ . Теперь вернем в коллекцию  $C$  каждый элемент из  $T$  столько раз, сколько он встречался в  $C$ .

```
for(size_t i = 0; i <= K; ++i) {  
    int currentIndex = 0;  
    for(size_t j = 0; j < T[i]; ++j) { C[currentIndex++] = i; }  
}
```

# Случайные числа. Функции `rand()` и `srand()`

- Псевдослучайная последовательность чисел
- Генератор псевдослучайных чисел (ГПСЧ)
- Зерно ГПСЧ

# Случайные числа

Функция `rand()`

В C++ имеется встроенная функция `rand()`, возвращающая целое число из отрезка `[0, INT_MAX]`.

Для того, чтобы получить "случайное" число из определенного отрезка, используется следующая конструкция:

```
a + rand() % (b - a + 1);
```

Где `a` — левая граница отрезка, а `b` — правая граница отрезка.

# Случайные числа

## Функция `srand()`

В C++ имеется встроенная функция `srand()`, задающая зерно ГПСЧ.

Для того, чтобы получать различные последовательности случайных чисел при независимых запусках программы, в качестве зерна используется текущее время, которое легко получить, используя функцию `time()` из библиотеки `<time.h>` с аргументом `NULL`: `time(NULL)`.

```
#include <time.h>

...

srand(time(NULL));
int r = a + rand() % (b - a + 1);
```



# Практические задания

## 1. Лине́йный поиск

Массив заполняется  $N$  случайными числами из отрезка  $[a, b]$ . Необходимо найти все индексы массива, в которых находится значение  $k$ .

Пользователю предлагается ввести  $N$ ,  $k$ ,  $a$  и  $b$ .

Вывести на экран сгенерированный массив и все индексы элементов массива, в которых находится число  $k$ . Вывести  $-1$ , если число не было найдено в массиве.

# Практические задания

## 1. Лине́йный поиск

### Пример 1

```
Enter N, k, a, b:  
5 8 -10 10  
  
-1 -8 -3 8 -5  
Indices of 8: 3
```

### Пример 2

```
Enter N, k, a, b:  
5 8 -10 10  
  
5 -7 -6 -3 9  
Indices of 8: -1
```

# Практические задания

## 2. Сортировка подсчетом

Массив заполняется  $N$  случайными числами из отрезка  $[a, b]$ . Необходимо отсортировать массив по возрастанию и по убыванию значений.

Пользователю предлагается ввести  $N$ ,  $a$  и  $b$ .

Вывести на экран сгенерированный массив, отсортированный по возрастанию массив и отсортированный по убыванию массив.

# Практические задания

## 2. Сортировка подсчетом

### Пример 1

```
Enter N, a, b:  
5 50 100
```

```
59 98 93 68 74  
59 68 74 93 98  
98 93 74 68 59
```

### Пример 2

```
Enter N, a, b:  
5 -50 50  
'a' must be bigger than 0!
```