# **04.** ОЦЕНКА РЕСУРСОВ. НАБОРЫ ДАННЫХ

*курс лекций по информатике и программированию*
*для студентов первого курса ИТИС КФУ (java-поток)*
*2023/2024*

**М.М. Абрамский**

кандидат технических наук, доцент кафедры программной инженерии

# Эффективный алгоритм

Оптимально использует **ресурсы:**

- Время работы алгоритма
    - «Количество» «шагов» («длина» трассы)
- Память, требуемая для работы
    - «Количество» «ячеек» памяти, необходимых для работы

*Вопрос: Как измерить объем (количество, размер) этих ресурсов?*

# Измеряем точно

- Замеряем время работы алгоритма в микро- (нано-) секундах
  - System.currentTimeMillis(), System.nanoTime();
- Замеряем точный размер использованной памяти (все переменные)

*Почему не годится?*

# Если считать время

>java Task2 100
time spent: 57300

>java Task2 100
time spent: 89300

>java Task2 100
time spent: 54900

>java Task2 100
time spent: 68600

>java Task2 100
time spent: 57800

>java Task2 100
time spent: 55100

# Если память

```
int x = 2;
int y = 5;
int z = x + y;
```

12 байтов

```
...
byte x = 2;
byte y = 5;
byte z = (byte) (x + y);
```

... байтов

```
...
byte x = 2;
byte y = 5;
System.out.println(x + y);
```

... байтов

# От чего зависят используемые ресурсы?

От самой задачи

**От входных данных задачи**

**ОТ РАЗМЕРА ВХОДНЫХ ДАННЫХ!**

*Что есть размер входных данных?*

# «Размер входных данных» - «Количество» «ячеек»

Вообще говоря, понятие относительное

- Если алгоритм обрабатывает массив, то **размер входа = размер массива** = количество чисел в нем (одно число – одна ячейка)
- Если алгоритме обрабатывает целое число, то **размер входа – количество цифр числа** (одна цифра – одна ячейка)

*Почему такой неоднородный подход нас не беспокоит?*

# Почему такой подход нас не беспокоит?

- *А зачем нам сравнивать между собой алгоритм обработки массива и алгоритм обработки числа?*
  - Нужно сравнивать между собой алгоритмы, решающие одну задачу.

- Итак, ресурсы зависят от размера входа.
  - Раз зависят, то тогда это функция, которая имеет в компьютерных науках специальное название.

На экзамене:

**- ОЦЕНИТЕ СЛОЖНОСТЬ АЛГОРИТМА.**
**- ОГО! ДЕЙСТВИТЕЛЬНО, ОЧЕНЬ СЛОЖНО!**

# Сложность вычислений

## Computational Complexity

- **T(n) – временнАя сложность** (сложность по времени)
- **S(n) – пространственная сложность** (сложность по памяти)

В рамках этого предмета мы пользуемся сложностью *«в худшем случае»*:

- Для оценки сложности берется самый худший случай, на котором сложность имеет наибольшее значение.
- Пример: в отсортированном по возрастанию массиве минимальный элемент a[0], который мы найдем за один шаг, но это не значит, что сложность по времени алгоритма поиска минимума массива – 1 шаг.

# Измерение сложности

- Сложность как-то явно зависит от размера входа n.
  - Значит, наверное можно выразить формулой типа ‘T(n) = n’
  - Но помните, что точный подсчет количества шагов/ячеек затруднен?

**Внимание: нам не нужно задавать точную формулу зависимости**

- Нам важно знать *порядок* зависимости от n – насколько сильно растет сложность при увеличении размера входа.
  - *Почему? И как нам это облегчает жизнь?*

# O - символика

«O-большое» (не путать с «o-маленьким»)

*f = O(g), если есть константа C, что*

$$f(x) <= C \cdot g(x)$$

*«Оценка сверху»*
*«Ну точно будет не больше, чем g(x)»*
***«Асимптотическая оценка»***

# Свойства O(f)

Это не равенства. Выражения верны только слева направо:

$$f \cdot O(g) = O(f \cdot g)$$
$$C \cdot O(f) = O(f)$$
$$O(f) + O(g) = O(max(f, g))$$

*Пример:*

$$O(n^2) + O(n) = O(n^2)$$
$$n \cdot O(n) = O(n^2)$$

# Какие бывают сложности

- **Полиномиальная – O($n^k$)**
  - Частные случаи – линейная и константная
- **Экспоненциальная – O($k^n$)**
- **Логарифмическая – O($\log n$)**
  - Не важно основание логарифма
    - *! Почему, кстати?*

(n – размер входа, k - константа)

# Максимум массива

Ввод массива

```
int max = a[0];
for (int x : a) {
      if (x > max)
            max = x;
}
```

Какая сложность?

# Сортировка массива выбором

```
for (int i = 0; i < n - 1; i++) {
    m = i;
    for (int j = i + 1; j < n; j++) {
        if (a[j] > a[m]) {
            m = j;
        }
    }
    h = a[i];
    a[i] = a[m];
    a[m] = h;
}
```

Сложность?

# Логарифмическая сложность

- Бинарный поиск в отсортированном массиве;
- Игра «Угадай число от 1 до N».

# Циклы по всем элементам массива – источник сложности

Два соседних цикла – это какая сложность?

```
for (int i = 1; i <= n; i++) {
    //...
}
for (int i = 1; i <= n; i++) {
    //...
}
```

Как на сложность влияет вложенность? (см. сортировка)

# Еще про циклы и сложность

- Источники ПОЛИНОМИАЛЬНОЙ СЛОЖНОСТИ - циклы (в т.ч. и вложенные) по каждому элементу входа. Тем самым, получается степень n.
- Источники ЭКСПОНЕНЦИАЛЬНОЙ - тоже циклы, которые работают за экспоненциальное (к размеру входа) количество итераций.
- Источники ЛОГАРИФМИЧЕСКОЙ - тоже циклы, которые идут не по каждому элементу массива, а как-то по-другому, например, с помощью бинарного поиска.

Кстати, сложности могут быть и "сложные" - в смысле, сложных функций. Например, O(n log n) - сложность хороших алгоритмов сортировок.

# Cruel World

- Было бы классно, если бы все алгоритмы имели полиномиальную сложность.

- Но:
  - *Перебрать все числа из N цифр*
    - *Сколько сочетаний?*
    - *Вот такая и сложность – экспоненциальная!*

# P и NP

- класс P – задачи, решаемые детерминированной машиной Тьюринга за полиномиальное время

- класс NP – задачи, решаемые недетерминированной машиной Тьюринга за полиномиальное время.
- класс NP – задачи, решаемые детерминированной машиной Тьюринга за экспоненциальное время.

- Понятно, что задача из P может быть и задачей из NP
  - Но можно ли задачу из NP решить за полиномиальное время?

# Who knows?

Вроде очевидно.

- Но никто не доказал…
- Лучший результат: Александр Разборов (1990-е)
  - *Ввел natural proofs, показал, что в современной математике пока нет инструментов, способных доказывать такие теоремы.*

Экспоненциальная сложность – вроде плохо (есть медленные алгоритмы)

- Но и хорошо – *см. пример далее*

# Правда о паролях

- Их знаете только вы!
- Сайты не должны знать пароли своих пользователей:
  - Злоумышленники, украв базу, узнают пароли
  - Админы могут получать доступ туда, куда не надо.

# Что происходит на самом деле?

**РЕГИСТРАЦИЯ:**

- hash(ваш пароль) -> хэш
    - строка вида **d6aabbdd62a11ef721d15**
    - легко подсчитать, сложно узнать исходный пароль

**ВХОД НА САЙТ:**

- hash(введенный пароль) сравнивается с хэшем пароля, который вы ввели при регистрации:
    - Если хэши равны – значит и пароль совпадает с тем, что лежит в базе – вас пускают.

# Пример хэша (sha256)

password
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

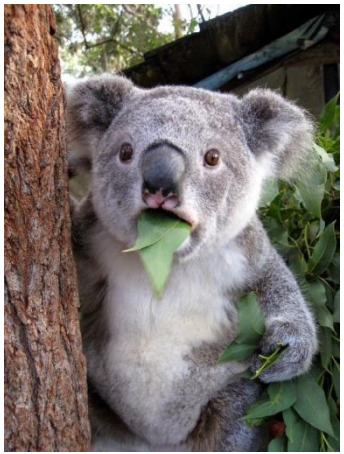
Password
e7cf3ef4f17c3999a94f2c6f612e8a888e5b1026878e4e19398b23bd38ec221a

Password1
19513fdc9da4fb72a4a05eb66917548d3c90ff94d5419e1f2363eea89dfee1dd


# *Сасса бен Дахир*

# Сколько зерен?

$\mathbf{N} = 1 + 2 + 4 + \ldots + 2^{63} = 18\,446\,744\,073\,709\,551\,615$



*Если 1 зерно = 1 секунда:*

307 445 734 561 825 860 **мин** = 5 124 095 576 030 431 **часов** = 213 503 982 334 601 **дней** = 584 942 417 355 **лет**

**Возраст земли:** 14 000 000 000 **ЛЕТ**

**Разных значений хэша: примерно N · N · N · N**

**(77 цифр).**

**Подберем?)**

# Если P = NP

**Можно выкидывать** все системы авторизации, аутентификации, банковские системы, электронные средства оплаты, криптографические системы, военные тайны, блокчейн и т.д.

Скорее всего не равно. Но… не доказано.

# Сложность факторизации

- Факторизация (разложение на множители) числа размером 428 бит заняло 8 месяцев, 600 человек и 1600 компьютеров.

- Алгоритмы шифрования, хэширования, цифровой подписиидр. основаны на том, что факторизацию быстро не сделать
    - ! Алгоритм RSA
    - Есть теоретически квантовый алгоритм, делающий факторизацию быстро.
        - При его реализации на квантовом устройстве произойдет коллапс всей системы электронной безопасности – пароли, банкинг, шифрование.

# Нужно строить алгоритмы с оптимальной сложностью.

- Типичный пример: 1! + 2! + 3! + … + n!
- Типичная ошибка: fact(1) + fact(2) + fact(3) + …

```
for (int i = 1; i <= n ; i++) {
    p = 1;
    for (int j = 1; j <= i; j++) {
        p *= j;
    }
}
```

# Как нужно

- 1! + 2! + 3! + … + n!

```
int p = 1;
int s = 0;
for (int i = 1; i <= n; i++) {
    p *= i; // мемоизация
    s += p;
}
```

# Другой пример

Катастрофическое рекурсивное вычисление n-го числа Фибоначчи:

```
int fib(int n) {
    если n == 1, то вернуть 1
    иначе если n == 2, то вернуть 1
    иначе вернуть fib(n-1) + fib(n-2)
}
```

*– что плохого в этом примере?*

# Потоковая обработка (это про сложность по памяти)

- Не нужно хранить все входные данные в процессе работы:
- Пример: сумма n чисел (и типичная ошибка в решении)

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
int [] a = new int[n];
int s = 0;
for (int i = 0; i < n; i++) {
    a[i] = sc.nextInt();
    s = s + a[i];
}
System.out.println(s);
```

# Потоковая обработка

```
Scanner sc = new Scanner(…);
int n = sc.nextInt();
int s = 0;
for (int i = 0; i < n; i++) {
    s = s + sc.nextInt();
}
System.out.println(s);
```

# Итак

Сложность по памяти алгоритмов, использующих только скалярные (примитивные) типы – O(1)

– *понятно, почему?*

*Все ли алгоритмы так работают?*

```
if (n == 2) {
    a11 = sc.nextInt();
    a12 = sc.nextInt();
    a21 = sc.nextInt();
    a22 = sc.nextInt();
} else if (n == 3) {
    a11 = sc.nextInt();
    a12 = sc.nextInt();
    a13 = sc.nextInt();
    a21 = sc.nextInt();
    …
}
```

# Необходимость

1. Структура данных, хранящая входные данные для их последующей [неоднократной] обработки.

2. Размер структуры должен задаваться динамически во время работы программы.

    - А не как в Pascal – сначала объявляем 10 000-й массив, а затем вводим размер массива 5.

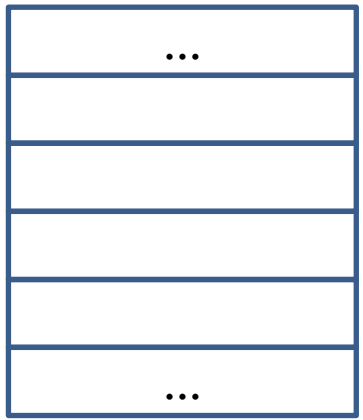
# Массив

- *Набор* данных **одного** типа;
- Объявление: тип [ ] имя;
- Выделение памяти: arr = **new** тип[размер]
  - размер – целочисленная переменная, может быть вычислена заранее
  - каждый элемент массива получает значение типа по умолчанию
    - » нулевое значения – для boolean, char, ссылочного типа?
- a[i] – обращение к элементу под номером i;
- Если массив размера n, то индексы его элементов от **0** до **n-1.**

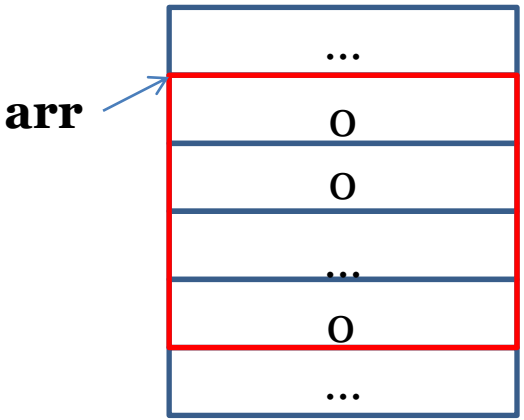
# Как массив хранится в памяти

**int [] arr**

**arr**




**arr = new int[5]**




Да, массив инициализирует свои элементы значением по умолчанию для типа массива (false, 0, 0.0, null)

# Обратите внимание

1. Адрес массива совпадает с адресом его первого элемента
2. Ячейки массива – одного размера (т.к. одного типа)
3. Адреса вообще говоря – числа (0xfab3123)

*Как получить адрес i-го элемента?*

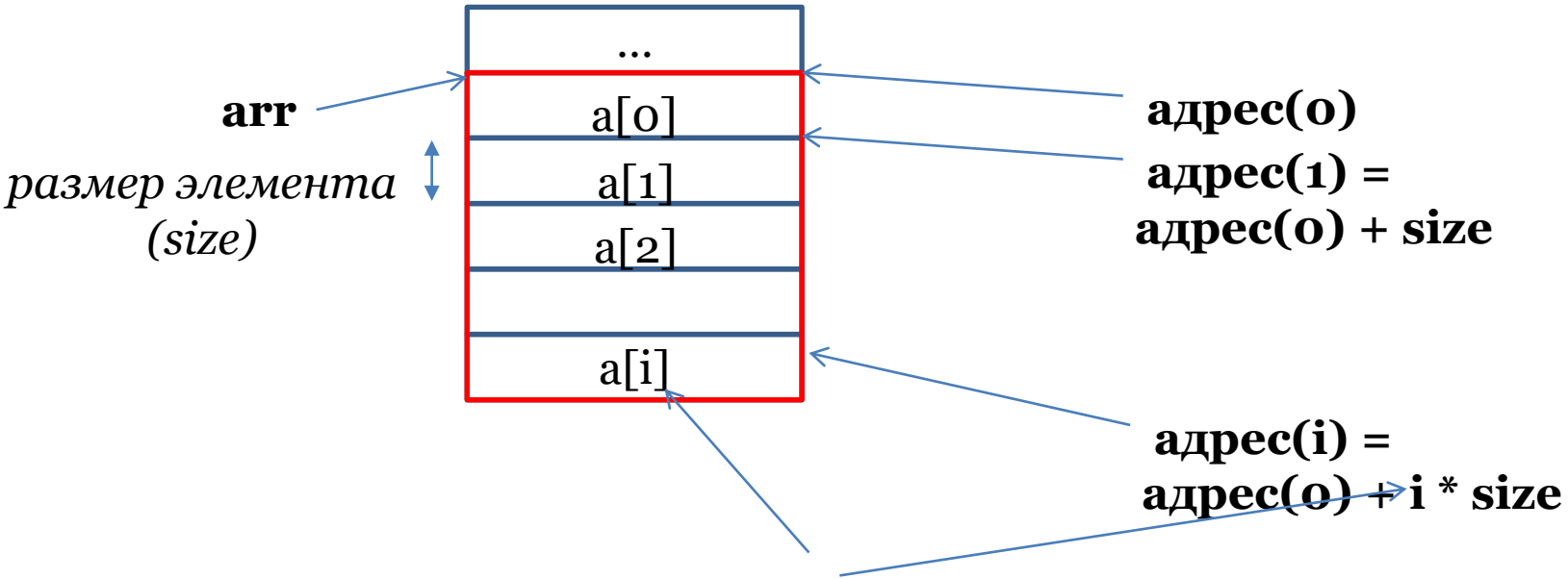
# Что стоит за a[i]

a[i] – обращение к содержимому элемента массива a под номером i

обратить надо по адресу, а адрес легко считается:

- **`адрес(i) = адрес(0) + i * size`**
  - » адрес(0) = адрес массива = a
  - » size – размер типа данных массива

# Что стоит за оператором a[i]




Сложность доступа к любому элементу массива?

**Прямая адресация**

# Цикл прохода по массиву

- arr.length – длина массива
  - Не всегда вы располагаете переменной ее длины
- Учимся считать с нуля
  - Длина: n, первый элемент 0, второй – 1, …, последний – n – 1.

```
for (int i = 0; i < n; i++) {
    обработка a[i]
}
```

# for each

- ТОЛЬКО ЧТЕНИЕ

```
for (int x : a) {
    обработка с x
    x = … - бесполезный оператор
}
```

# Ошибки работы с массивом. Присваивание

```
int[] b = a;
```

// «ждем тут, что все скопировалось в b»

Но вспомните, что такое a и b?

# Типовые ошибки работы с массивом

Выход за пределы массива (ошибки с его индексами):

```
for (int i = 1; i <= n; i++)
```

*или*:

```
for (int i = 0; i < n; i++) {
        a[i + 1] = …;
```

# Хардкод

```
int n = 10;
int[] arr = new int[n];
for (int i = 0; i <= 9; i++) {
    //ввод arr
}
arr[9] = arr[1];
for (int i = 0; i < 5; i++) {
    // обнуление первой половины
    arr[i] = 0;
}
```

# Сложности операций с массивами

- Вставка на произвольную (random) позицию
- Вставка на ою позицию (на i-ю)? На последнюю?
- Удаление ого элемента (i-го)?
- Поиск в массиве?
- Поиск в упорядоченном массиве?

# CAPACITY & size

```
final int CAPACITY = 1000;
int [] storage = new int[CAPACITY];
int storageSize = 0;

//Сколько в массиве элементов?

storage[storageSize] = 100;
storageSize++;
```

# Очистить массив (если есть и size, и capacity)

```
storageSize = 0;
```


To Be Continued