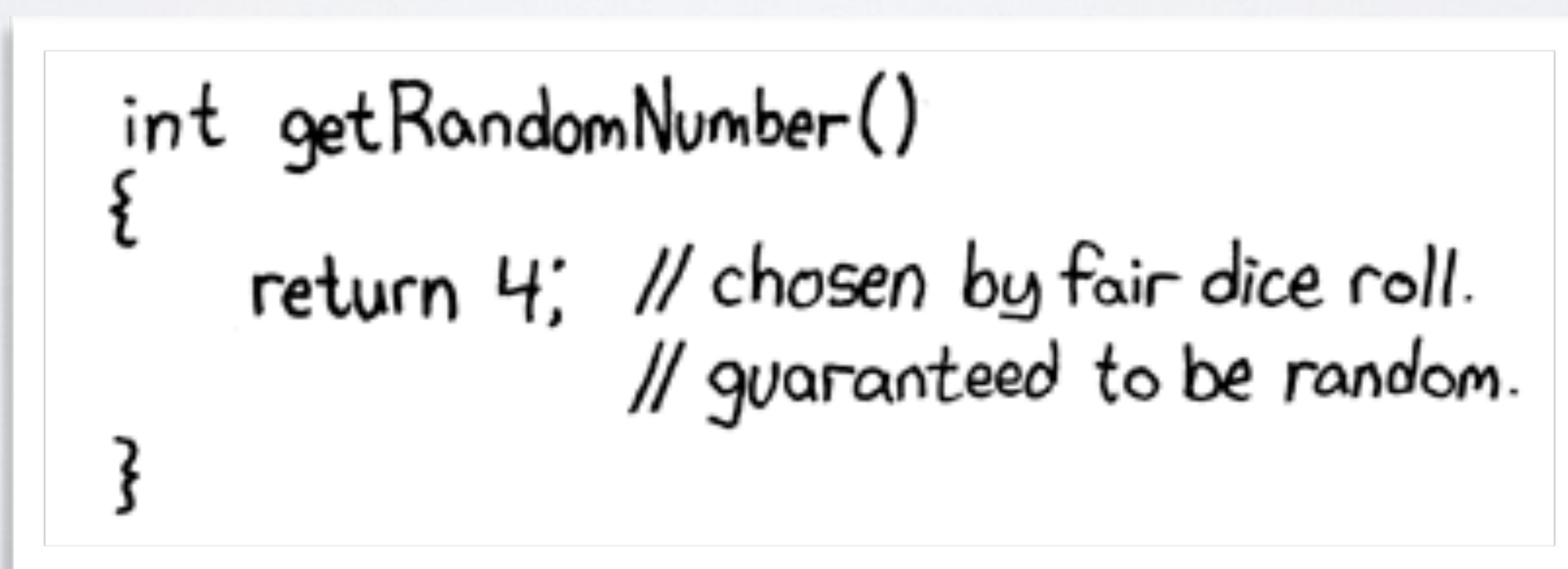
ОСНОВЫ ПРОГРАММНОГО КОНСТРУИРОВАНИЯ

Лекция № 3

19 сентября 2016 г.



ВРЕМЕННАЯ СЛОЖНОСТЬ

- Количество требуемых операций (время): **Q(n)=O(f(n))**, где **n** – размер структуры данных.
- **f(n)=O(g(n))** $\leftrightarrow$ $\exists\ n_0, M: \forall\ n \geq n_0$ $|f(n)| \leq M \cdot |g(n)|$.

- **O(1)** – константная.
- **O(log n)** – логарифмическая.
- **O(n)** – линейная.
- **O(n log n)** – квази-линейная.
- **O(n^2)** – квадратичная.
- **O(2^n)** – экспоненциальная.

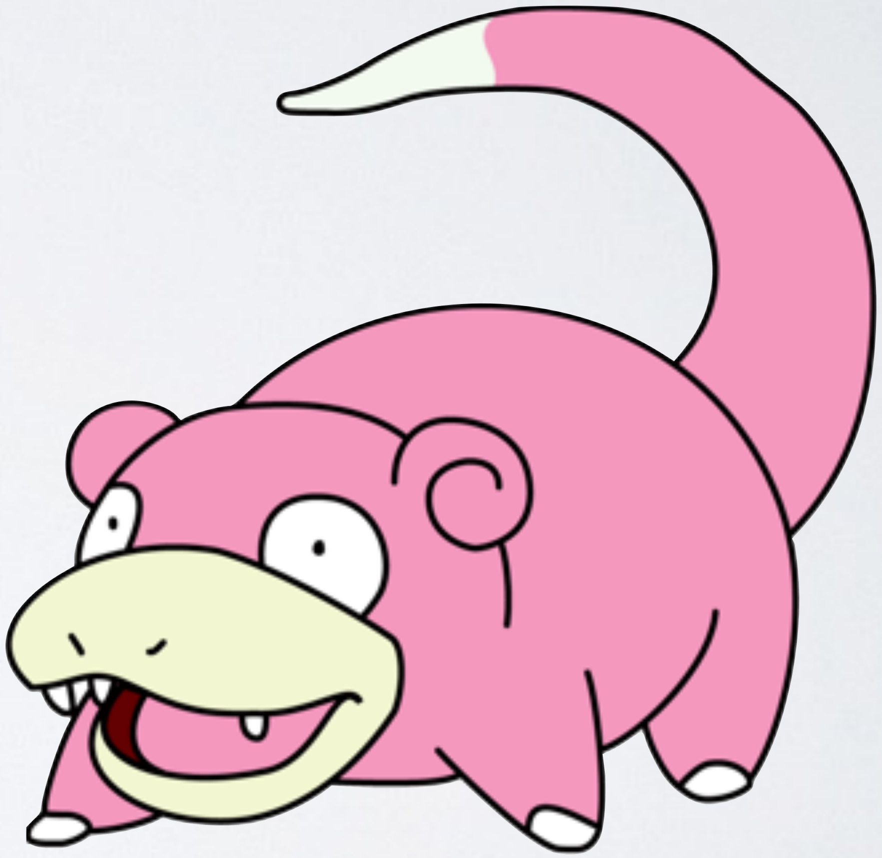
МАСШТАБИРУЕМОСТЬ

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
10	A	B	C	D	E	F
100	A	2B	10C	20D	100E	1000F
10000	A	4B	1000C	4000D	$10^6 \cdot E$	$10^9 \cdot F$

О СКОРОСТИ И ТОРМОЗАХ

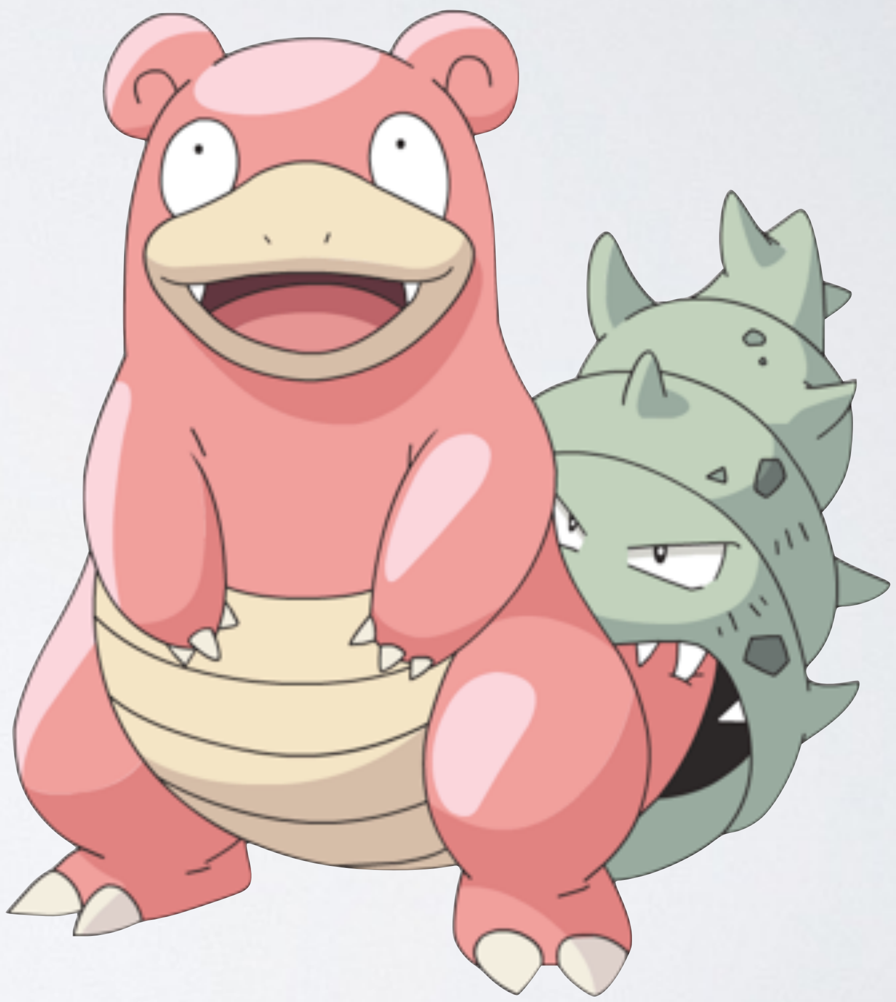
Для n=10 процесс выполняется 1 сек.

Зависимость	Время выполнения для n=1000
O(1)	1 сек
O(log n)	3 сек
O(n)	2 мин
O(n log n)	5 мин
$O(n^2)$	3 часа
$O(n^3)$	12 суток



О СКОРОСТИ И ТОРМОЗАХ

Экспоненциальная сложность
$Q(n)=O(2^n)$



- Для n=10 процесс выполняется:
  1 сек.
- Для n=100 процесс выполняется:
  10^{27} сек. ≈ $4 \cdot 10^{19}$ лет
- Для n=1000 процесс выполняется:
  10^{298} сек. ≈ $3 \cdot 10^{290}$ лет

ПРИМЕРЫ

- Вычисление суммы двух чисел – **O(1)**
  (хотя на самом деле зависит от размера чисел).

- Подсчет длины строки – **O(n)**
  (один цикл).

- Умножение двух матриц – **$O(n^3)$**
  (три вложенных цикла).

- Полный перебор всех **n**-битных чисел – **$O(2^n)$**.

O, Ω, Θ

- $f(n)=O(g(n)) \Leftrightarrow \exists\ n_0, M\ \forall\ n \geq n_0$:
  $|f(n)| \leq M \cdot |g(n)|$.

- $f(n)=\Omega(g(n)) \Leftrightarrow \exists\ n_0, M\ \forall\ n \geq n_0$:
  $|f(n)| \geq M \cdot |g(n)|$.

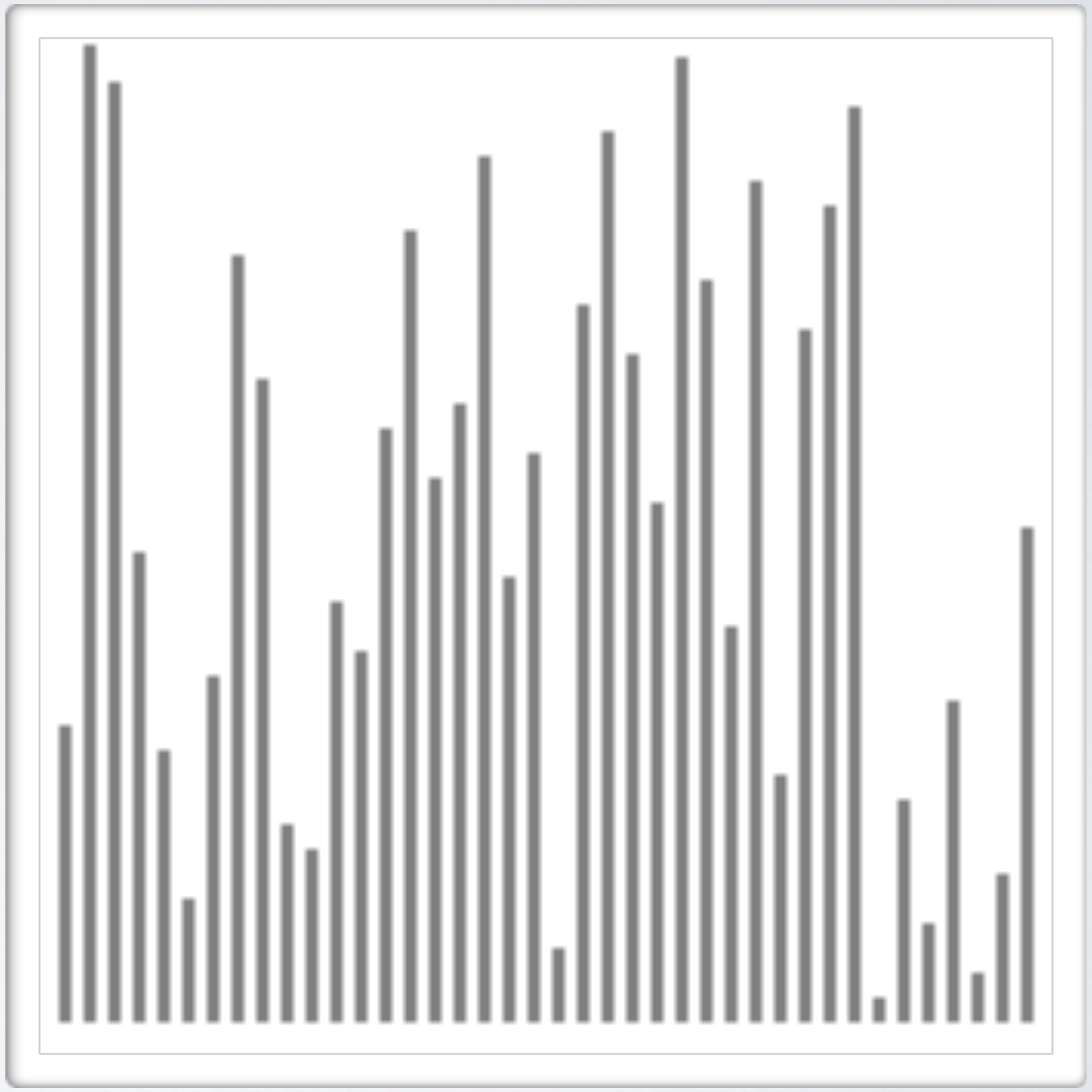
- $f(n)=\Theta(g(n)) \Leftrightarrow \exists\ n_0, M_1, M_2\ \forall\ n \geq n_0$:
  $M_1 \cdot |g(n)| \leq |f(n)| \leq M_2 \cdot |g(n)|$.

СЛУЧАИ

- Сложность в …
  - в лучшем случае (best-case)
  - в среднем случае (average-case)
  - в худшем случае (worst-case) *(вариант по умолчанию)*
- best-case ≤ average-case ≤ worst-case

ЗАДАЧА СОРТИРОВКИ

- Записи **R_1**, **R_2**, …, **R_n**.
- Ключи **K_1**, **K_2**, …, **K_n**.
- Требуется расставить записи так, чтобы ключи шли в неубывающем порядке.



ИНТЕРФЕЙС СОРТИРОВКИ В БИБЛИОТЕКЕ С

```
void sort(void *base, size_t nitems,
          size_t item_size,
          int (*compar)(void *p1, void *p2));
```

- **base** – указатель на начало массива.
- **nitems** – количество сортируемых элементов.
- **item_size** – размер одного элемента в байтах.
- **compar** – функция сравнения. Возвращает **−1**, **0** или **1**.
  p1 и **p2** – указатели на сравниваемые элементы.

ИДИОТСКАЯ СОРТИРОВКА (BOGOSORT)

1. Проверить, является ли массив уже отсортированным. Если да, то алгоритм завершается.

2. Случайным образом перемешать записи в массиве и перейти к шагу 1.

в среднем: **$O(n \cdot n!)$**

ТУПАЯ СОРТИРОВКА (BOZOSORT)

1. Проверить, является ли массив уже отсортированным. Если да, то алгоритм завершается.

2. Случайным образом поменять местами две записи в массиве и перейти к шагу 1.

в среднем: **O(n!)**

СОРТИРОВКА ВЫБОРОМ (SELECTION SORT)

1. На **j**-ой итерации первые **$(j - 1)$** элементов уже на своих местах.

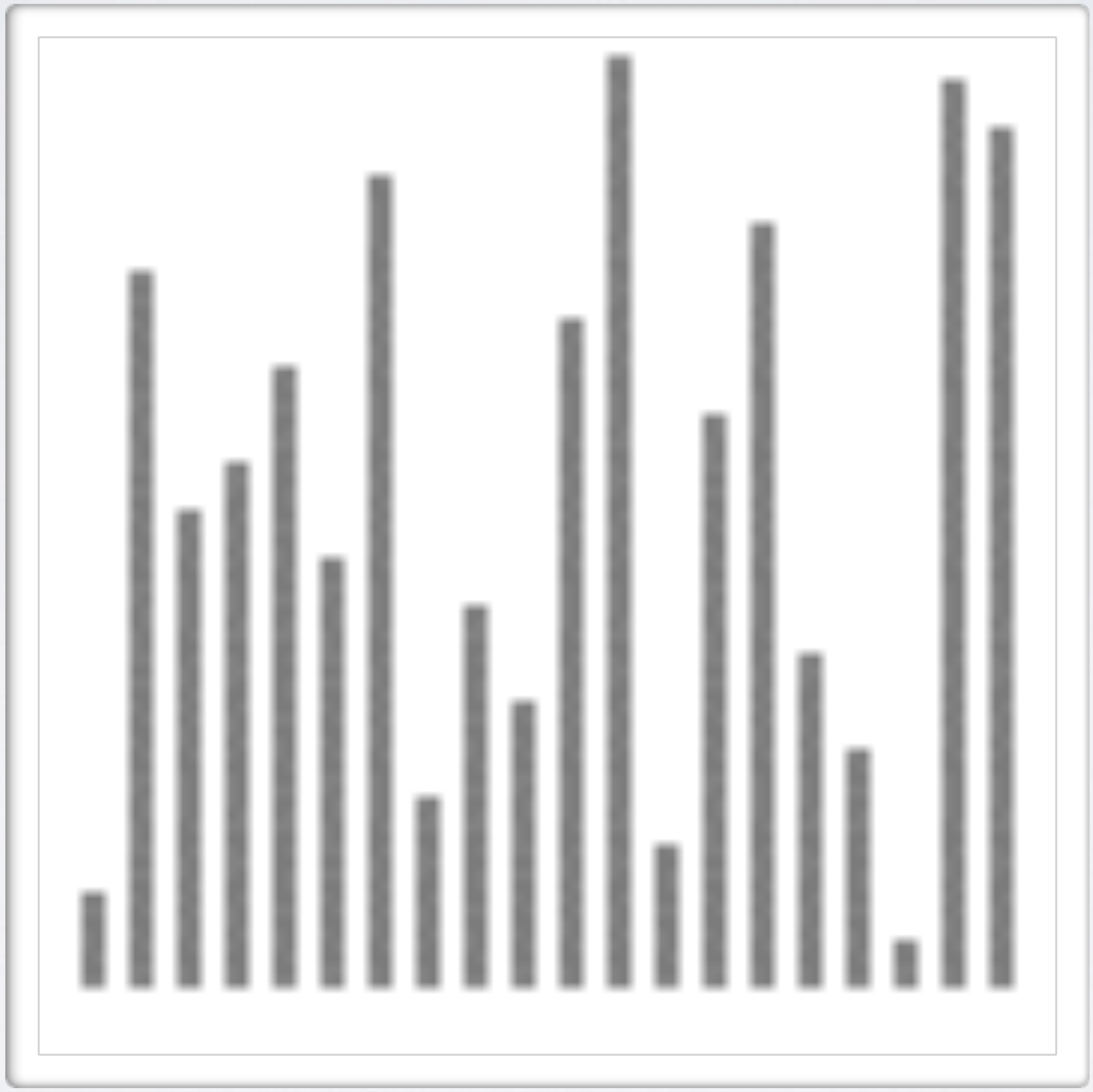
2. Найти наименьший ключ **K_m** из **$K_j, K_{j+1}, \ldots, K_n$**.

3. Поменять местами **R_j** и **R_m**.
   Тогда первые **j** элементов на своих местах.

4. Если **j** равен **n**, то массив отсортирован, иначе переходим на шаг 1.

$O(n^2)$

СОРТИРОВКА ВЫБОРОМ (SELECTION SORT)



http://www.sorting-algorithms.com/selection-sort

«ПУЗЫРЬКОВАЯ» СОРТИРОВКА (BUBBLE SORT)

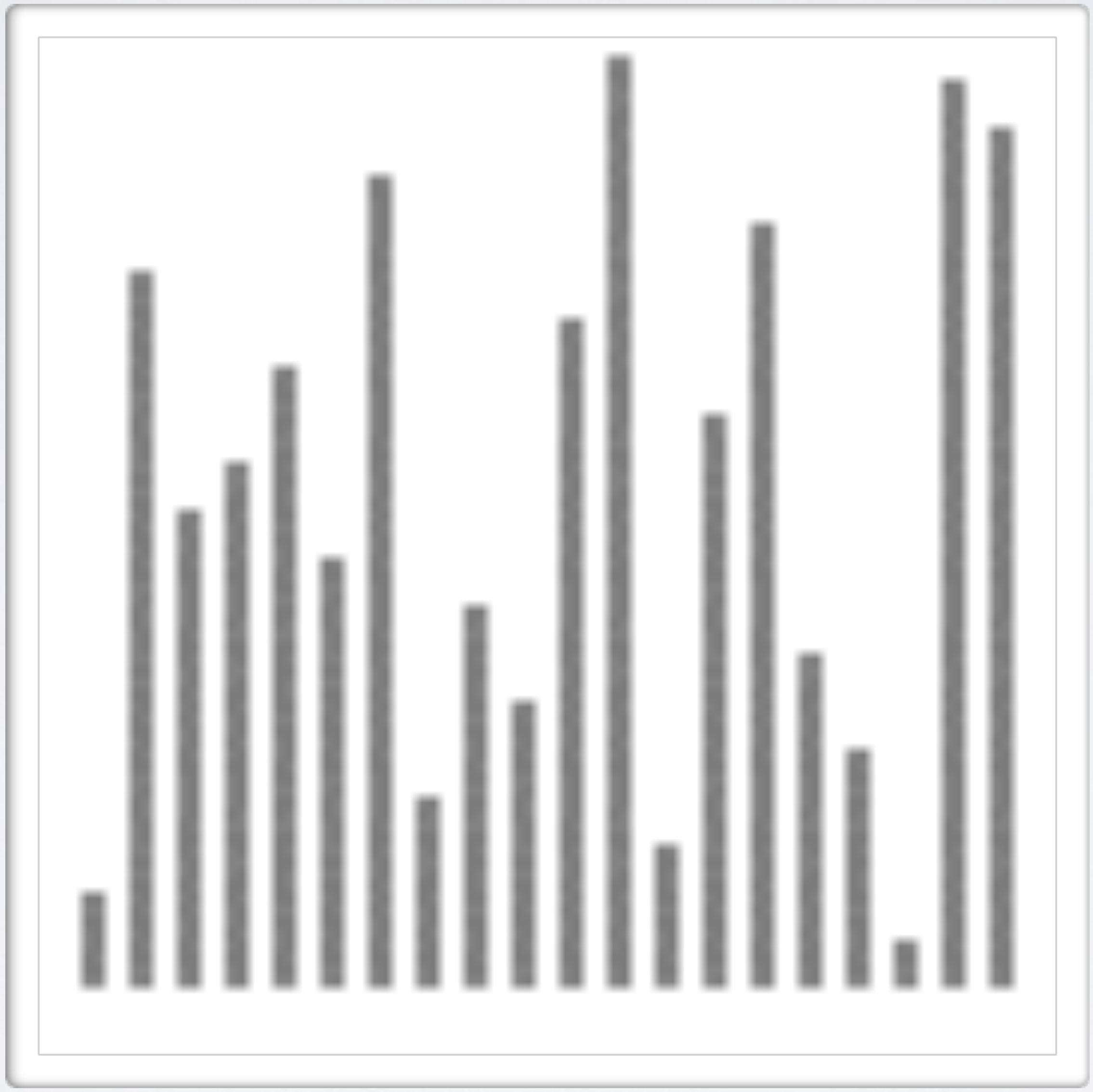
1. Пройти массив от начала к концу, сравнивая на каждом шаге пару соседних ключей $\mathbf{K_i}$ и $\mathbf{K_{i+1}}$. Если $\mathbf{K_i > K_{i+1}}$, то записи $\mathbf{R_i}$ и $\mathbf{R_{i+1}}$ меняются местами.

2. Если на предыдущем шаге была хотя бы одна перестановка, перейти к шагу 1.
   В противном случае — конец, массив отсортирован.

Бонус: чередовать проходы в прямую и обратную стороны (cocktail sort).

$O(n^2)$

«ПУЗЫРЬКОВАЯ» СОРТИРОВКА (BUBBLE SORT)



* анимация для версии сортировки, которая проверяет в обратную сторону
http://www.sorting-algorithms.com/bubble-sort

СОРТИРОВКА ВСТАВКАМИ (INSERTION SORT)

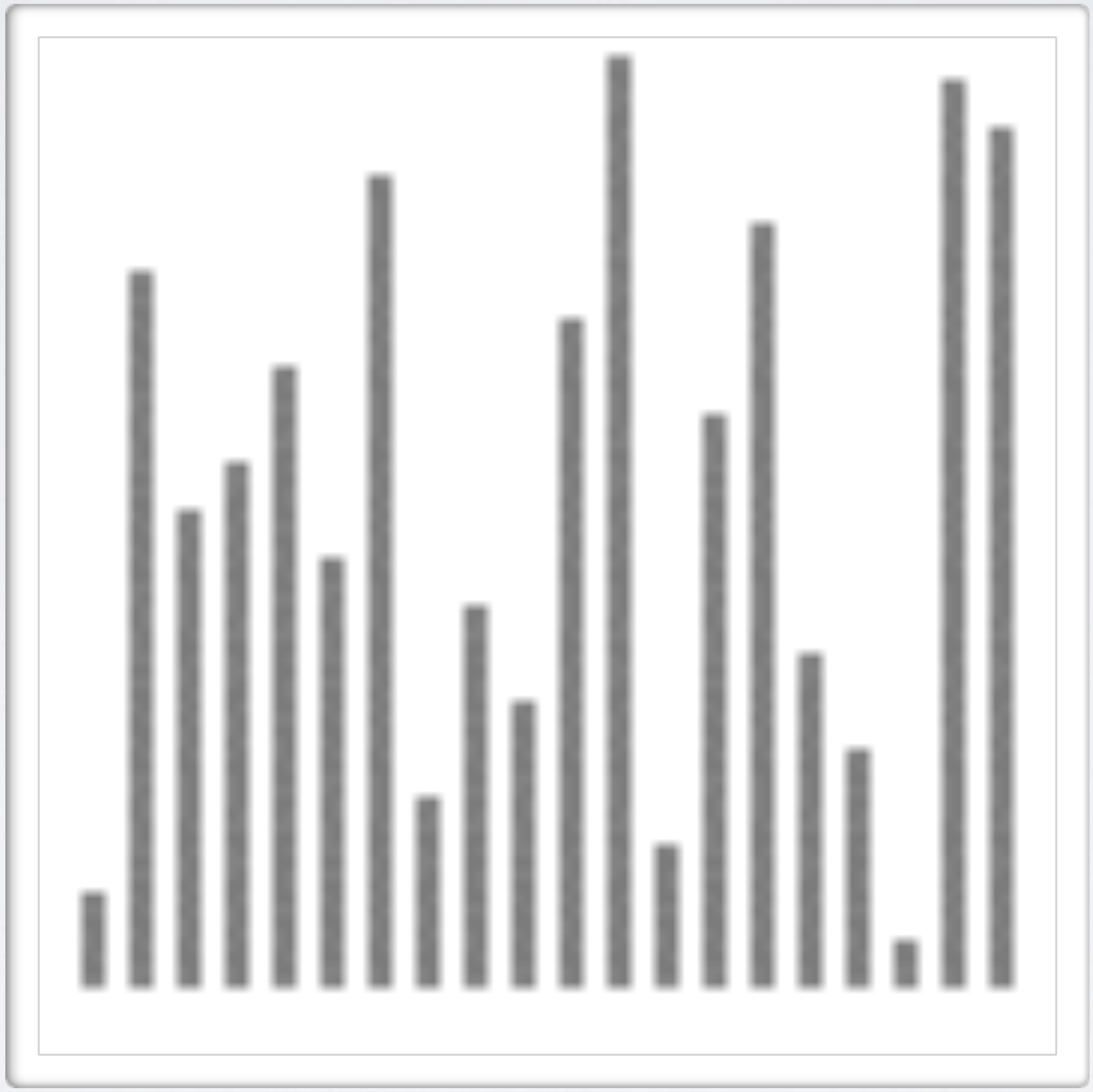
1. На **j**-ой итерации часть записей уже отсортирована: $K_1 \le K_2 \le \ldots \le K_{j-1}$, $1 < j \le n$.

2. Меняем R_j с элементами слева от него, пока не встретим элемент, меньший K_j. После этого последовательность $R_1, R_2, \ldots, R_j$ отсортирована.

3. Если **j** равен **n**, то массив отсортирован, иначе переходим на шаг 1.

$O(n^2)$

СОРТИРОВКА ВСТАВКАМИ (INSERTION SORT)

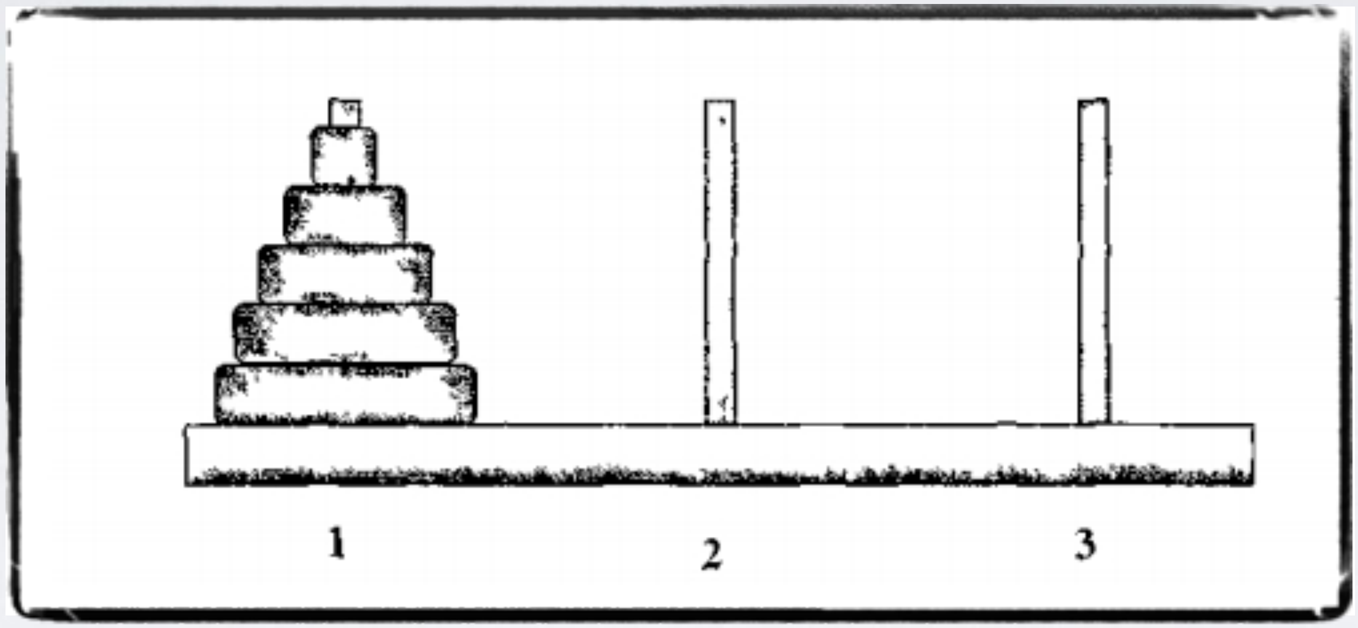


http://www.sorting-algorithms.com/insertion-sort

«РАЗДЕЛЯЙ И ВЛАСТВУЙ»

- **Разделение** задачи на несколько подзадач.
- **Покорение**: решение подзадач.
- **Комбинирование** решения исходной задачи из решений подзадач.

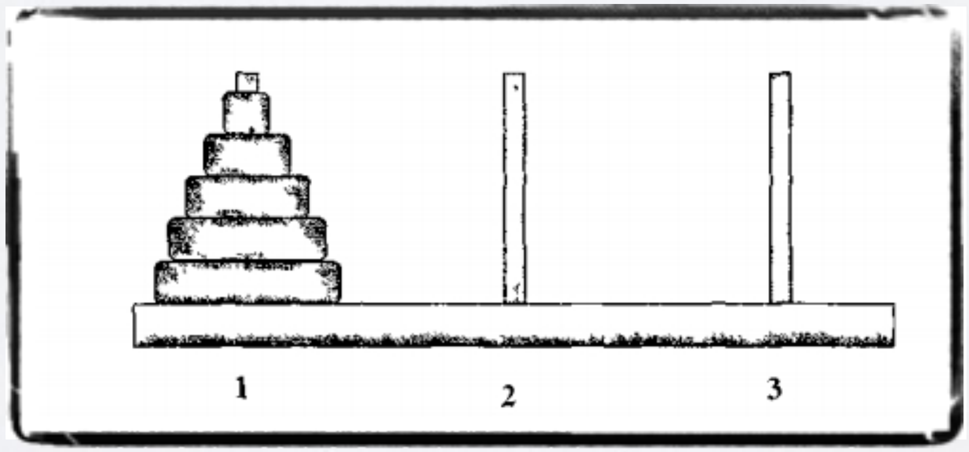
ЗАДАЧА
О ХАНОЙСКОЙ БАШНЕ



- Задача: перенести **n** дисков с палки **1** на палку **3**, используя палку **2** как свободную.
- Можно класть только меньший диск на больший!

РЕШЕНИЕ

1. Перенести **n−1** дисков с **1** на **2**, пользуясь свободной **3**.
2. Перенести самый большой (**n**-й по счету) диск с **1** на **3**.
3. Перенести **n−1** дисков с **2** на **3**, пользуясь свободной **1**.



ПОЛУЧЕНИЕ ВСЕХ ПЕРЕСТАНОВОК

Как получить все перестановки чисел **1…n**?

1. Как-то получаем все перестановки **1…n−1**.
2. Есть **n** способов добавить число **n** к каждой из перестановок, полученных на шаге 1.

АЛГОРИТМ КАРАЦУБЫ ДЛЯ УМНОЖЕНИЯ ДЛИННЫХ ЧИСЕЛ

Есть два длинных числа: $a_{2n}\, a_{2n-1} \ldots a_1$ и $b_{2n}\, b_{2n-1} \ldots b_1$. Нужно получить произведение.

$$C = (10^{2n} a_{2n} + 10^{2n-1} a_{2n-1} + \ldots + a_1) \times$$
$$(10^{2n} b_{2n} + 10^{2n-1} b_{2n-1} + \ldots + b_1)$$

$$C = (10^n A_1 + A_0) \times (10^n B_1 + B_0) =$$
$$10^{2n} A_1 B_1 + 10^n (A_1 B_0 + B_1 A_0) + A_0 B_0,$$
где $A_1 = 10^n a_{2n} + 10^{n-1} a_{2n-1} + \ldots + a_{n+1}$,
$$A_0 = 10^n a_n + 10^{n-1} a_{n-1} + \ldots + a_1,$$
B_0, B_1 – аналогично.

АЛГОРИТМ КАРАЦУБЫ ДЛЯ УМНОЖЕНИЯ ДЛИННЫХ ЧИСЕЛ

Умножение чисел размерности **2n** сведено к четырем умножениям **n**-размерных чисел и комбинированию посредством сдвигов и сложений.

Хитрость! Вычисляем 3 умножения:

$D_1 = A_0B_0$,

$D_2 = A_1B_1$,

$D_3 = (A_0 + A_1)(B_0 + B_1)$.

Тогда $C = (10^{2n} D_2 + 10^n (D_3 - D_2 - D_1) + D_1)$.

ОСНОВНАЯ ТЕОРЕМА О РЕКУРРЕНТНЫХ СООТНОШЕНИЯХ (THE MASTER METHOD)

сложность разделения и объединения

количество подзадач

Пусть $T(n) = a\,T(n/b) + O(n^d)$, $a \geq 1$, $b \geq 2$, $d \geq 0$

размер каждой подзадачи

$T(n) = O(n^d \log n)$, если $a = b^d$

$T(n) = O(n^d)$, если $a < b^d$

$T(n) = O(n^{\log a / \log b})$, если $a > b^d$

ПРИМЕРЫ РЕКУРРЕНТНЫХ СООТНОШЕНИЙ

- Прямолинейное умножение длинных чисел:
  $T(n) = 4\,T(n/2) + O(n) \rightarrow T(n) = O(n^{\log 4 / \log 2}) = O(n^2)$
- Алгоритм Карацубы:
  $T(n) = 3\,T(n/2) + O(n) \rightarrow T(n) = O(n^{\log 3 / \log 2}) \approx O(n^{1{,}58})$

ЛИНЕЙНЫЙ ПОИСК

Дан массив **A** длины **n** и ключ **K**. Требуется определить положение элемента с данным ключом в массиве или установить, что его там нет.

Последовательно сравниваем ключи, пока не найдем совпадающий ключ или массив не кончится.

Сложность: **O(n)**.

ДВОИЧНЫЙ ПОИСК

Дан **отсортированный** массив **A** длины **n** и ключ **K**. Двоичный поиск (он же бинарный поиск, он же поиск делением пополам):

Возьмем серединный элемент массива (**m**-й) и сравним его ключ K_m с **K**:

- Если $K_m = K$, то ключ найден.
- Если $K_m < K$, то продолжаем поиск в правой половине: $A[m+1 \ldots n-1]$.
- Если $K_m > K$, то продолжаем поиск в левой половине: $A[0 \ldots m-1]$.

Сложность: $T(n) = T(n/2) + O(1) = O(\log n)$.

РЕКУРСИЯ

- Рекурсия — см. рекурсия.
- Рекурсия — вызов функцией самой себя.
- Есть опасность бесконечной рекурсии.
- В большинстве случаев расходуется машинный стек.
- Взаимодействие только через аргументы и возвращаемое значение.
- Реализация может быть итеративной (своя реализация стека или оптимизация хвостовой рекурсии).

ХВОСТОВАЯ РЕКУРСИЯ

обычная

```
int fact(int n) {
    return (n == 0) ? 1
                    : n * fact(n-1);
}
```

хвостовая

```
int fact_acc(int n, int acc) {
    return (n == 0) ? acc
                    : fact_acc(n-1, n*acc);
}
int fact(int n) {
    return fact_acc(n, 1);
}
```

ДРУГИЕ ПРИМЕНЕНИЯ?

$$n! = n \cdot (n-1)!$$

$$F_n = F_{n-1} + F_{n-2}$$

Удачно ли?

КОНЕЦ ТРЕТЬЕЙ ЛЕКЦИИ