

Алгоритмы и структуры данных

Лекция 1: Введение

А. Куликов

Академия современного программирования

План лекции

- 1 Опрос
- 2 Числа Фибоначчи
- 3 Задача сортировки
- 4 Скорость роста функций

План лекции

- 1 Опрос
- 2 Числа Фибоначчи
- 3 Задача сортировки
- 4 Скорость роста функций

Какая из функций растет быстрее?

Какая из функций растет быстрее?

Варианты ответов: $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$.

	$f(n)$	$g(n)$
1.	$n - 100$	$n - 200$
2.	$n^{1/2}$	$n^{2/3}$
3.	$100n + \log n$	$n + (\log n)^2$
4.	$n \log n$	$10n \log 10n$
5.	$\log 2n$	$\log 3n$
6.	$10 \log n$	$\log n^2$
7.	$n^{1.01}$	$n \log^2 n$
8.	$n^2 / \log n$	$n(\log n)^2$
9.	$n^{0.1}$	$(\log n)^{10}$

Какая из функций растет быстрее?

Какая из функций растет быстрее?

Варианты ответов: $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$.

	$f(n)$	$g(n)$
10.	$(\log n)^{\log n}$	$n/\log n$
11.	\sqrt{n}	$(\log n)^3$
12.	$n^{1/2}$	$5^{\log_2 n}$
13.	$n2^n$	3^n
14.	2^n	2^{n+1}
15.	$n!$	2^n
16.	$(\log n)^{\log n}$	$2^{(\log n)^2}$
17.	$\sum_{i=1}^n i^k$	n^{k+1}

Рекуррентные соотношения

Рекуррентные соотношения

Определите скорость роста функции по рекуррентному неравенству, которому она удовлетворяет (считаем, что $f(1) = 1$).

- 1 $f(n) \leq 2f(n/2) + 10n$
- 2 $f(n) \leq f(n/3) + f(2n/3) + n$
- 3 $f(n) \leq f(n-1) + 5n$
- 4 $f(n) \leq 2f(n-1) + n^2$

Типы сортировки

Типы сортировки

Кратко опишите каждый из перечисленных типов сортировки и предъявите оценку на время работы в худшем случае.

- Сортировка вставками.
- Сортировка слиянием.
- Быстрая сортировка.

Алгоритмы на графах

Алгоритмы на графах

- Что такое поиск в ширину? Что такое поиск в глубину? Каково время работы?
- Что такое минимальное покрывающее дерево? За какое время оно может быть построено?

Структуры данных

Структуры данных

- Что такое двоичное дерево поиска? Для чего оно используется?
- Что такое красно-чёрное дерево? Чем оно лучше обычного бинарного дерева?

План лекции

- 1 Опрос
- 2 Числа Фибоначчи
- 3 Задача сортировки
- 4 Скорость роста функций

Числа Фибоначчи

Числа Фибоначчи

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- формально:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & n > 1 \\ 1, & n = 1 \\ 0, & n = 0 \end{cases}$$

- скорость роста **экспоненциальна**: $F_{30} > 1000000$, десятичная запись числа F_{100} состоит из 21-й цифры
- $F_n \approx 2^{0.694n}$

Экспоненциальный алгоритм

Алгоритм

Алгоритм $FIB1(n)$

- **if** $n = 0$: **return** 0
- **if** $n = 1$: **return** 1
- **return** $FIB1(n-1) + FIB1(n-2)$

Анализ алгоритмов

Для каждого алгоритма мы должны ответить на следующие вопросы:

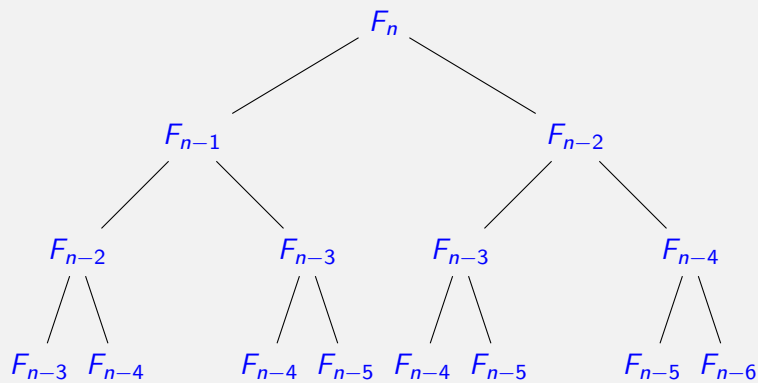
1. Корректен ли он?
2. Каково время его работы (как функция от длины входа)?
3. Существует ли алгоритм быстрее?

Анализ алгоритма FIB1

Ответы:

1. Конечно, корректен. Он просто реализует определение.
2. Пусть $T(n)$ — время работы (количество операций) алгоритма FIB1 на входе n .
 - ▶ Ясно, что $T(n) = T(n-1) + T(n-2) + 3$.
 - ▶ Значит, $T(n) \geq F_n!$ Время работы алгоритма экспоненциально, что означает, что он практически бесполезен на практике.
 - ▶ Например, для вычисления F_{200} понадобится около 2^{138} операций. Тогда даже суперкомпьютеру, выполняющему 40 триллионов операций в секунду, понадобится 2^{92} секунд.
 - ▶ По закону Мура компьютеры становятся быстрее примерно в 1.6 раз каждый год. Время работы алгоритма FIB1 как раз примерно равно 1.6^n . Итак, если современные компьютеры позволяют вычислить в этом году F_{100} , то в следующем году мы вычислим F_{101} , через два года — F_{102} .
3. Да, к счастью, более быстрый алгоритм существует.

Почему же алгоритм FIB1 такой медленный?



Потому что он много раз вычисляет одно и то же.

Полиномиальный алгоритм

Алгоритм

Алгоритм FIB2(n)

- if $n = 0$ return 0
- создать массив $f[0..n]$
- $f[0] = 0, f[1] = 1$
- for $i = 2..n$:
 - ▶ $f[i] = f[i-1] + f[i-2]$
- return $f[n]$

Время работы

Ясно, что время работы такого алгоритма линейно.

Более детальный анализ

Более детальный анализ

- В анализе предыдущих алгоритмов мы подсчитывали **количество элементарных операций**, предполагая, что каждая такая операция занимает константное время.
- Однако является ли сложение чисел элементарной операцией?
- Если бы все рассматриваемые числа были, скажем, не более чем 32-битными, то можно было бы так считать.
- Но количество бит в записи F_n примерно равно $0.694n$ и, следовательно, может быть и больше 32.
- Два k -битных числа можно сложить за линейное по k время (в столбик, например).
- Таким образом, количество операций алгоритма FIB1 примерно равно nF_n , а алгоритма FIB2 — n^2 .

План лекции

- 1 Опрос
- 2 Числа Фибоначчи
- 3 **Задача сортировки**
- 4 Скорость роста функций

Задача сортировки

Определение

Задача сортировки:

- Вход: последовательность n чисел a_1, a_2, \dots, a_n .
- Выход: перестановка a'_1, a'_2, \dots, a'_n исходной последовательности, для которой $a'_1 \leq a'_2 \leq \dots \leq a'_n$

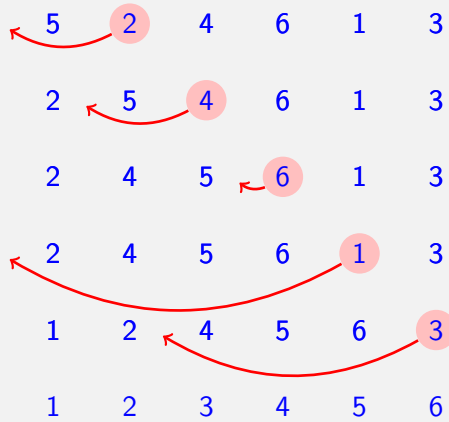
Сортировка вставками

Сортировка вставками

Алгоритм INSERTION-SORT(A)

- **for** $j = 2..length[A]$
 - ▶ $key = A[j]$
 - ▶ $i = j - 1$
 - ▶ **while** $i > 0$ и $A[i] > key$
 - ★ $A[i + 1] = A[i]$
 - ★ $i = i - 1$
 - ▶ $A[i + 1] = key$

Пример работы алгоритма



Анализ алгоритмов

Анализ алгоритмов

- Ясно, что чем больше массив, тем больше нужно времени на его сортировку.
- Однако важен и порядок элементов входного массива.
- Как же измерять **размер входа**?
- В разных задачах по-разному: иногда рассматривается количество элементов, иногда общее количество битов в записи входных данных, иногда используется несколько параметров входа (например, количество вершин и ребер графа).
- Под размером входа задачи сортировки мы будем понимать количество элементов во входном массиве.

Анализ сортировки вставками

Анализ сортировки вставками

- Если массив отсортирован, то алгоритм работает линейной время.
- Если же массив отсортирован в обратном порядке, то время работы сортировки вставками будет максимальным, поскольку каждый элемент $A[j]$ придется сравнить со всеми его предшественниками $A[1], \dots, A[j-1]$.
- Время работы в таком случае составит

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}.$$

Время работы в худшем и среднем случае

Время работы в худшем и среднем случае

Как мы установили, время работы в лучшем и худшем случае могут сильно различаться. Мы будем изучать **время работы в худшем случае**, которое равно максимальному времени работы на всех входах данной размера. Причины:

- Зная время работы в худшем случае, можно гарантировать, что алгоритм завершится за некоторое время на любом входе данного размера.
- На практике “плохие” входы могут попадаться часто.
- Время работы в среднем может быть близко ко времени работы в худшем случае.

План лекции

- 1 Опрос
- 2 Числа Фибоначчи
- 3 Задача сортировки
- 4 Скорость роста функций

Скорость роста функций

Определение

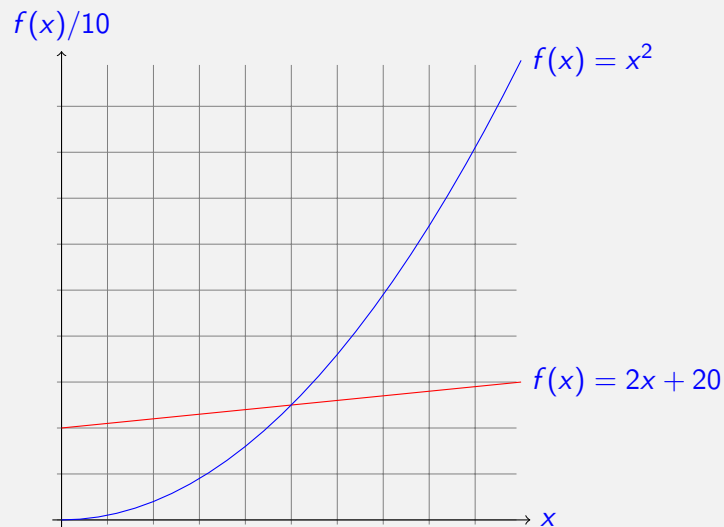
Рассмотрим функции $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ ($f(n)$ и $g(n)$ можно представлять себе как время работы двух алгоритмов на входах размера n).

- Говорим, что f **растет не быстрее** g , и пишем $f = O(g)$, если найдутся такие константы $c > 0$ и n_0 , что

$$\forall n \geq n_0, f(n) \leq cg(n).$$

- Говорим, что f **растет не медленнее** g , и пишем $f = \Omega(g)$, если $g = O(f)$.
- Говорим, что f и g **имеют одинаковый порядок роста**, и пишем $f = \Theta(g)$, если $f = O(g)$ и $g = O(f)$.

Пример: x^2 против $2x + 20$



Какими бы ни были коэффициенты, квадратичная функция всегда обгонит линейную.

Формальный анализ: x^2 , $2x + 20$, $x + 1$

Формальный анализ

- Пусть $f_1(x) = x^2$, $f_2(x) = 2x + 20$, $f_3(x) = x + 1$.
- Ясно, что $f_2 = O(f_1)$, поскольку

$$\frac{f_2(x)}{f_1(x)} = \frac{2x + 20}{x^2} \leq 22.$$

- Однако $f_1 \neq O(f_2)$, поскольку $\frac{x^2}{2x+20}$ принимает сколь угодно большие значения.
- Ясно также, что $f_2 = O(f_3)$ и $f_3 = O(f_2)$, из чего следует, что $f_2 = \Theta(f_3)$.

Общие правила

Общие правила

- Мультипликативные константы могут быть опущены: $14n^2$ заменяется на n^2 .
- n^a растет быстрее n^b для $a > b$: n^2 доминирует n .
- Любая экспонента доминирует любой полином: 3^n растет быстрее n^5 .
- Аналогично любой полином доминирует любой логарифм: n растет быстрее $(\log n)^3$. Из этого также следует, например, что n^2 доминирует $n \log n$.