Лекции по предмету

Алгоритмы и структуры данных

Группа лектория ФКН ПМИ 2015-2016
Никита Попов
Тамерлан Таболов
Лёша Хачиянц

2016 год

Содержание

1 Оргмоменты **1**

2 Лекция 1 от 12.01.2016 **2**
2.1 Задача о Ханойских башнях. Три стержня. 2
2.2 Четыре стержня. Обобщение задачи. 3

3 Лекция 2 от 14.01.2016 **5**
3.1 Задача сортировки 5

4 Лекция 3 от 19.01.2016 **7**
4.1 Нотация 7
4.2 Разделяй и властвуй. Быстрая сортировка 8
4.3 Как обеспечить везение? 10

5 Лекция 4 от 21.01.2016 **10**
5.1 Двоичное дерево поиска 10

6 Лекция 5 от 26.01.2016 **13**
6.1 Быстрая сортировка. Продолжение 13
6.2 Поиск медианы 15
6.3 Медиана медиан 15

7 Лекция 7 от 02.02.2016 **17**
7.1 Умножение чисел. Алгоритм Карацубы 17
7.2 Перемножение матриц. Алгоритм Штрассена 18

8 Лекция 8 от 4.02.2016 **19**
8.1 Быстрое возведение в степень 19
8.2 Обратная задача 19
8.3 Обработка текста 19

9 Лекция 9 от 9.02.2016 **21**
9.1 Продажа земли . . . 21
9.2 В общем о динамическом программировании . . . 22
9.3 Задача с прошлой лекции — выравнивание текста . . . 22

10 Лекция 10 от 11.02.2016 **23**
10.1 Расстояние редактирования (расстояние Левенштейна) . . . 23
10.2 Сравнение алгоритмов . . . 25

11 Лекция 11 от 16.02.2016 **26**
11.1 Алгоритмы на графах . . . 26
11.2 Достижимость . . . 26
11.3 Компоненты связности . . . 27

12 Лекция 12 от 18.02.2016 **27**
12.1 Представление графов в памяти компьютера . . . 27
12.2 Оценка BFS . . . 28
12.3 Оценка DFS . . . 28

Оргмоменты

Одна контрольная — контест на реализацию какого-то алгоритма. Пользоваться своим кодом запрещено.

Задача — привести алгоритм, провести теоретический анализ (доказать его корректность, оценить время работы) и запрограммировать. Сначала сдаётся теория, потом практика.

Экзамен устный.

$$O_{\text{итоговая}} = 0.7 \cdot O_{\text{накопленная}} + 0.3 \cdot O_{\text{экзамен}}$$

$$O_{\text{накопленная}} = 0.2 \cdot O_{\text{КР}} + 0.12 \sum_{i=1}^{5} O_{\text{ДЗ i}} + 0.2 \cdot O_{\text{семинары}}$$

Списывание, как обычно, *не поощряется*. ДЗ предполагается не обсуждать.

Здесь (ссылка слева) можно найти ссылки на ДЗ и краткое содержание лекций.

Автоматов *пока* не предусмотрено.

Литература:

- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. — «Алгоритмы. Построение и анализ»
- Дасгупта С., Пападимитриу Х., Вазирани У. — «Алгоритмы»

Лекция 1 от 12.01.2016

Задача о Ханойских башнях. Три стержня.

Есть три стержня. На первый стержень нанизано 64 диска, от самого большого к самому маленькому. Задача: переложить все диски на второй стержень. Ограничения:

- Диски можно переносить только по одному.
- Нельзя класть диск большего диаметра на диск меньшего диаметра.

Какой может быть алгоритм?
Варианты из аудитории:

1. Полный перебор
2. Рекурсивный алгоритм.

Рассмотрим такой рекурсивный алгоритм:

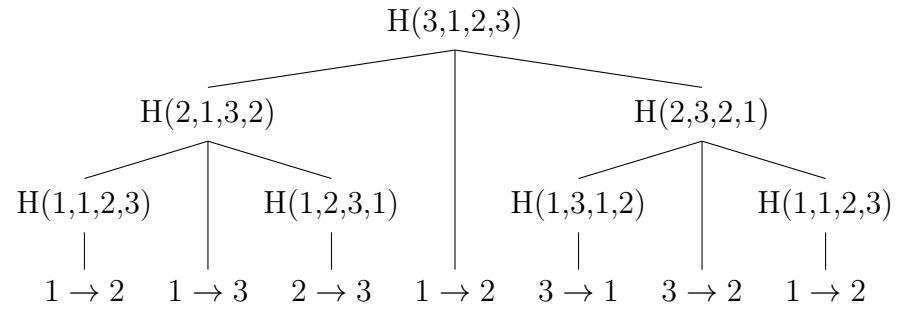
1. Переложим все диски, кроме n-ного, на третий стержень;
2. Переложим n-ный диск с первого на второй стержень;
3. Переложим все остальные с третьего стержня на второй.

Запишем этот алгоритм с помощью псевдокода:

Algorithm 1 Рекурсивный алгоритм решения задачи о Ханойской башне

```
1: function HANOI3(n,i,j,k)        ▷ n — количество дисков, i,j,k — номера стержней
2:     if n > 0 then
3:         HANOI3(n − 1,i,k,j)
4:         move i → j
5:         HANOI3(n − 1,k,j,i)
```

Нарисуем дерево операций для $n = 3$:



Алгоритм, по сути, обходит это дерево в глубину и при этом слева направо, выполняя все перемещения, что встретятся.

Это дерево можно рассматривать, как полное бинарное дерево глубины n, если перемещения учитывать не в отдельных листьях, а в родительских узлах. Тогда в каждом узле мы выполняем одно действие, а в полном бинарном дереве $2^n - 1$ узлов. Следовательно, выполняется $2^n - 1$ перемещение.

Пусть число перемещений для n дисков равно $f(n)$. Тогда верно следующее:

$$f(n) = \begin{cases} 0, & n = 0 \\ 2f(n-1) + 1, & n > 0 \end{cases}$$

Свойство: $f(n) = 2^n - 1$

Доказательство. Докажем это по индукции. База верна, так как $f(0) = 0 = 2^0 - 1$. Теперь пусть предположение верно для $n-1$, то есть $f(n-1) = 2^{n-1} - 1$. Тогда $f(n) = 2f(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$, что и требовалось доказать. □

Можно ли улучшить время работы? Оказывается, что нет.

Рассмотрим некоторый алгоритм. Он рано или поздно должен переложить наибольший диск на второй стержень. Для этого ничего не должно быть на нём и на втором, т.е. все на третьем. А как получить эту конфигурацию? Оптимальным алгоритмом на $n-1$ шаг, что приводит к нашим вычислениям и уже полученному минимальному результату в 2^n-1.

Утверждение. *Задачу о Ханойских башнях нельзя решить за меньшее число шагов, причём решение с таким числом шагов ровно одно.*

Доказательство. По индукции.

База ($n=0$)**:** очевидно, решить быстрее, чем за 0 шагов нельзя и последовательность такая ровно одна.

Переход ($n-1 \to n$)**:** предположим, что мы доказали это утверждение для $n-1$. Рассмотрим утверждение для n. Рано или поздно алгоритму понадобится освободить первые два стержня, чтобы переложить первый диск на второй стержень. Необходимо сделать это одно перекладывание и после вернуть все оптимальным алгоритмом. Итого, опираясь на предположение индукции шагов в оптимальном и единственном решении для $n-1$, нам понадобится $2(2^{n-1}-1)+1$ шаг, что и равно 2^n-1. □

Изменим задачу:

Четыре стержня. Обобщение задачи.

Условие в остальном ровно то же. Стала ли задача проще?

Сложнее она точно не стала, т.к. четвёртым можно просто не пользоваться.

Рассмотрев переход от двух к трём, кажется, что должно быть проще; как можно воспользоваться четвёртым?

Предложение: переложить предпоследний отдельно на четвёртый и сэкономить на перекладывании башни из $n-1$, перекладывая вместо неё башню из $n-2$

Algorithm 2 Рекурсивный алгоритм решения задачи о Ханойской башне на 4-х стержнях, версия 1

```
1: function HANOI4(n,i,j,k,l)        ▷ n — количество дисков, i,j,k,l — номера стержней
2:     if n > 0 then
3:         HANOI4(n − 1,i,l,k,j)
4:         move i → k
5:         move i → j
6:         move k → j
7:         HANOI4(n − 1,l,j,i,k)
```

Построив аналогичное дерево, получим, что в каждом узле три перемещения, а узлов $2^{\lfloor \frac{n}{2} \rfloor}-1$ — экономия, но не очень большая.

Построим другой алгоритм:

Заметим, что число шагов зависит от m. Пусть $n_m = \frac{m(m+1)}{2}$ (если n другое, то на первом шаге выберем такой m, чтобы $n-m$ было таким, а дальше на вход будет поступать число такого вида).

Построим дерево алгоритма. Оно также будет полным бинарным деревом. Заметим, что в нём m уровней, так как при каждом шаге m уменьшается на единицу. Это связано с тем, что $n_m - m = \frac{m(m+1)}{2} - m = \frac{(m-1)m}{2} = n_{m-1}$. При этом в узле на i-м уровне (нумерация с нуля) проводится $2^{m-i}-1$ операция, так как мы пользуемся доказанным ранее алгоритмом для трёх

Algorithm 3 Рекурсивный алгоритм решения задачи о Ханойской башне на 4-х стержнях, версия 2

```
1: function HANOI4(n,i,j,k,l)          ▷ n — количество дисков, i,j,k,l — номера стержней
2:     if n > 0 then
3:         HANOI4(n − m,i,l,k,j)
4:         HANOI3(m,i,j,k)
5:         HANOI4(n − m,l,j,i,k)
```

стержней. Тогда на каждом уровне выполняется $2^m - 2^i$ операций. Тогда всего выполняется $\sum\limits_{i=0}^{m-1}(2^m - 2^i) = m2^m - \sum\limits_{i=0}^{m-1} 2^i = (m-1)2^m + 1$ операций.

Пусть число перемещений для n дисков равно $g(n)$. Тогда

$$g(n_m) = \begin{cases} 0, & m = 0 \\ 2g(n_{m-1}) + 2^m - 1, & m > 0 \end{cases}$$

Предложение: $g(n_m) = (m-1)2^m + 1$

Доказательство. По индукции. База верна, так как $g(n_0) = 0 = (0-1)2^0 + 1 = -1 + 1$. Теперь допустим, что предположение верно для n_{m-1}, то есть $g(n_{m-1}) = (m-2)2^{m-1} + 1$. Тогда $g(n_m) = 2g(n_{m-1}) + 2^m - 1 = (m-2)2^m + 2 + 2^m - 1 = (m-1)2^m + 1$. □

Заметим, что при достаточно больших n верно, что $m \approx \sqrt{2n}$. Тогда $g(n) \approx \sqrt{2n} \cdot 2^{\sqrt{2n}}$. Тогда $g(n) = \Theta(\sqrt{n} \cdot 2^{\sqrt{2n}})$. Θ означает (грубо говоря), что функция растёт примерно так же.

Попробуем обобщить этот алгоритм для любого числа стержней:

Algorithm 4 Рекурсивный алгоритм решения задачи о Ханойской башне, общий случай

```
1: function HANOI(n,i,j,P)             ▷ n — количество дисков, i,j, — основные стержни
2:                                     ▷ P —множество вспомогательных стержней
3:     if n > 0 then
4:         choose p ∈ P
5:         R := P \ p
6:         if R = ∅ then
7:             HANOI3(n,i,j,p)
8:         else
9:             HANOI(n − m,i,p,R ∪ {j})
10:            HANOI(m,i,j,R)
11:            HANOI(n − m,p,j,R ∪ {i})
```

Пусть для перемещения n дисков с помощью алгоритма на k стержнях ($k \geq 3$) нужно $h(n,k)$ операций. Тогда верно следующее:

$$h(n_m, k) = \begin{cases} 0 & n = 0 \\ 2^{n_m} - 1 & n > 0, k = 3 \\ 2h(n_{m-1}, k) + h(m, k-1) & n > 0, k > 3 \end{cases}$$

Лекция 2 от 14.01.2016

Задача сортировки

Вход: последовательность чисел (строго говоря, может быть что угодно с полным порядком) $(a_1, a_2, \ldots, a_n)$.

Выход: $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$, где $a_{i_k} \leq a_{i_{k+1}}$. Другими словами, на выходе получается отсортированная по возрастанию последовательность.

Рассмотрим неэффективный алгоритм:

Algorithm 5 Неэффективный алгоритм сортировки

```
1:  function RECURSIVE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:      n := |a|
3:      if n > 1 then
4:          RECURSIVE_SORT(a[1 : n − 1])
5:          k := a_n
6:          for i := n − 1 downto 1 do
7:              if a_i > k then
8:                  a_{i+1} := a_i
9:              else
10:                 break
11:             a_{i+1} := k
```

[6,8,3,4] → [3,6,8,4] $\xrightarrow{8>4}$ [3,6, ,8] $\xrightarrow{6>4}$ [3, ,6,8] $\xrightarrow{3<4}$ [3,4,6,8]

По сути, мы идём слева направо и каждому элементу находим место среди прошлых уже отсортированных элементов.

Теперь рассмотрим алгоритм *сортировки вставками.*

Algorithm 6 Алгоритм сортировки вставками

```
1:  function RECURSIVE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:      n := |a|
3:      for j := 2 to n do
4:          k := a_j
5:          for i := j − 1 downto 1 do
6:              if a_i > k then
7:                  a_{i+1} = a_i
8:              else
9:                  break
10:             a_{i+1} := k
```

Докажем корректность алгоритма формально. Для этого найдём *инвариант.*

Инвариант: в начале каждой итерации цикла по j массив с 1 по $j-1$ индекс уже отсортирован. При этом он состоит из тех же элементов, что и раньше.

Если это условие выполняется, то после выполнения алгоритма, весь массив (с 1-го по n-ый индексы) будет отсортирован.

Доказательство. По индукции:

База: $j = 2$ — $a[1:1]$ отсортирован

Переход Всё до j-го отсортировано; Поставим a_j на нужное место. Тогда полученный массив также будет отсортирован. □

Насколько эффективно он работает? Понятно, что это зависит от входных данных. Ясно, что чем больше элементов, тем дольше он работает. Понятно также, что если массив уже отсортирован, то работать он будет быстрее.

$$T(n) \text{ — время работы на входе длины } n \text{ в худшем случае.} \quad (1)$$

$$\text{в среднем случае.} \quad (2)$$

$$\text{в лучшем случае.} \quad (3)$$

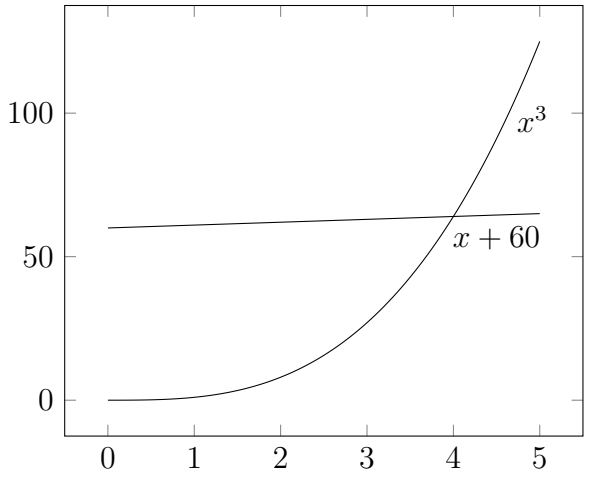
Однако оценка в лучшем случае, вообще говоря, бесполезна. Ведь любой алгоритм можно модифицировать так, чтобы в каком-то случае он работал очень быстро.

Асимптотический анализ: как меняется $T(n)$ при $n \to \infty$? Для исследования этого обычно применяют O-нотацию или Θ-нотацию.

$$\Theta(g(n)) = \{f(n) \mid \exists\, c_1, c_2 > 0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$

Например, пусть задана функция $f(n) = 3n^2 + 2n - 6$. Тогда $f(n) \in \Theta(n^2)$.

Асимптотика — это хорошо, но на константы тоже стоит обращать внимание: для маленьких n вполне может быть, что n^3 работает быстрее, чем n.



Оценим *худший случай* нашего алгоритма (когда на каждом шагу приходится совершать максимальное число перемещений): $T(n) = \sum\limits_{j=2}^{n} \sum\limits_{i=j-1}^{1} \Theta(1) = \sum\limits_{j=2}^{n} \Theta(j) = \Theta(n^2)$

Средний случай: Предположим, что все входы равновероятны. Тогда будет выполняться примерно половина сравнений и $T(n) = \sum \Theta(\frac{j}{2}) = \Theta(n^2)$

Лучший случай — это случай, когда массив уже отсортирован. Тогда $T(n) = \Theta(n)$.

Рассмотрим другой алгоритм — *сортировку слиянием.*

Algorithm 7 Алгоритм сортировки слиянием

```
1: function MERGE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:     n := |a|
3:     if n > 1 then
4:         b_1 := MERGE_SORT(a[1 : n/2])
5:         b_2 := MERGE_SORT(a[n/2 + 1 : n])
6:         a := MERGE(b_1, b_2)          ▷ сливаем два отсортированных массива в один
7:     return a
```

Рассмотрим, как может работать MERGE(b_1, b_2) на примере. Пусть даны массивы $b_1 :=$ [2,5,6,8] и $b_2 :=$ [1,3,7,9].

Будем сливать элементы из массивов в результирующий массив b, сравнивая поочерёдно минимальные элементы, которые ещё не вошли в результирующий массив.

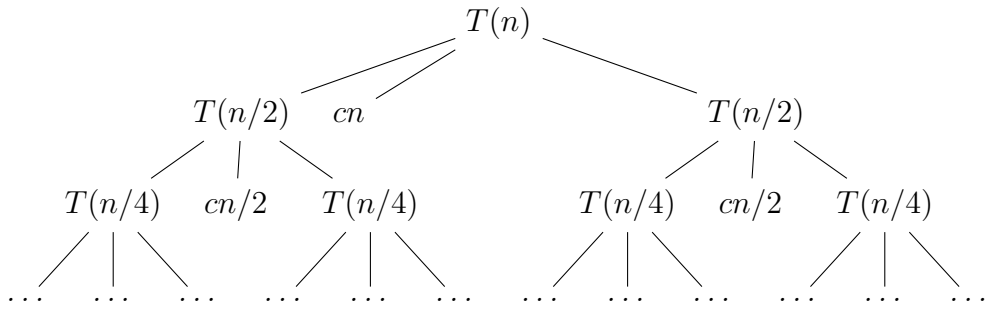
- $2 > 1$. Тогда $b[1] := b_2[1] = 1$ (будем считать, что нумерация идёт с единицы).
- $2 < 3$. Тогда $b[2] := b_1[1] = 2$.
- Аналогично продолжаем для всех остальных элементов массивов.

Очевидно, что алгоритм корректен, а его сложность — линейная, так как мы один раз проходим по массивам, то есть $\Theta(n)$.

Пусть худшее время для MERGE_SORT — $T(n)$. Тогда

$$T(n) = \begin{cases} \Theta(1), n = 1 \\ 2T(n/2) + \Theta(n) \end{cases}$$

Построим дерево рекурсии:



На каждом уровне cn работы, а высота дерева — $\log_2 n$. Общее время работы — $n\Theta(1) + cn \log n = \Theta(n \log n)$.

Лекция 3 от 19.01.2016

Нотация

$$\Theta(g(n)) = \{f(n) \mid \exists c_1 > 0, c_2 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\},$$

Θ — *асимптотическое* $=$. Например, $2n = \Theta(n)$. По определению, $c_1 n \leqslant 2n \leqslant c_2 n$. Тогда $c_1 = 1, c_2 = 2$.

$$O(g(n)) = \{f(n) \mid \exists c_2 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant f(n) \leqslant c_2 g(n)\}$$

O — *асимптотическое* $\leqslant$. Например, по этому определению $n = O(n \log n)$, так как при достаточно больших n_0 $\log n > 1$. Тогда $c_2 = 1$.

$$\Omega(g(n)) = \{f(n) \mid \exists c_1 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n)\}$$

Ω — *асимптотическое* $\geqslant$. Например, $n \log n = \Omega(n \log n)$ и $n \log n = \Omega(n)$. В обоих случаях подходит $c_1 = 1$.

$$o(g(n)) = \{f(n) \mid \forall c_2 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant f(n) \leqslant c_2 g(n)\}$$

o — *асимптотическое* $<$. Например, $n = o(n \log n)$. Покажем это. Пусть $n < c_2 n \log n \iff 1 < c_2 \log n \iff n > 2^{1/c_2}$. Тогда $n_0 = [2^{1/c_2} + 1]$

$$\omega(g(n)) = \{f(n) \mid \forall c_1 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n)\}$$

ω — *асимптотическое* $>$. Например, нельзя сказать, что $n \log n = \omega(n \log n)$. Но можно сказать, что $n \log n = \omega(n)$.

Когда мы пишем такую нотацию, мы подразумеваем функции, а не числа. Если же указывать функции явно, то это можно сделать с помощью λ-нотации:

$$\lambda n.n \in o(\lambda n.n \log_2 n)$$

Примечание: данная нотация очень похожа на лямбда-функции в Python:

$$\texttt{lambda x: x * x} \iff \lambda x.x^2$$

Заметим, что в логарифмах можно свободно менять основание: $\log_c n = \frac{\log_2 n}{\log_2 c}$. Именно поэтому не пишут основание логарифма.

Разделяй и властвуй. Быстрая сортировка

Ход действий при алгоритме **"разделяй и властвуй"**:

1. Разбить задачу на подзадачи.
2. Каждую подзадачу решить рекурсивно.
3. Объединяем решения подзадач некоторым образом.

Этот алгоритм даст решение общей задачи.

Вернёмся к *сортировке слиянием*. Алгоритм состоит из трёх шагов:

1. Раделить массив напополам — $\Theta(1)$
2. Рекурсивно решить подзадачи — $2T(\frac{n}{2})$
3. Слияние уже отсортированных массивов — $\Theta(n)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \implies \Theta(n \log n)$$

Задача та же — отсортировать массив.

Воспользуемся методом “Разделяй и властвуй”. Разобьем по-другому:

Выберем в массиве опорный элемент x (как угодно). Выбор важен, от него может много зависить. Пройдем по всем элементам и запишем те элементы, что меньше x до него, а те, что больше — после.

Две подзадачи: сортировка двух подмассивов.

Третий шаг — соединить их.

Рассмотрим работу алгоритма на примере массива $\{6, 3, 8, 7, 5, 1\}$:

1. $j = 1$. Так как $6 > 3$, то запускается тело цикла. Тогда $i = 1$ и 3 остаётся на месте.
2. $j = 2$. Так как $6 < 8$, то ничего не изменяется.

Algorithm 8 Разбитие массива на подмассивы

```
1: function PARTITION(a, p, q)   ▷ a — массив, p и q — индексы начала и конца соотвественно
2:     i := p
3:     for j := p + 1 to q do
4:         if a[j] < a[p] then
5:             i := i + 1
6:             SWAP(a[i], a[j])
7:     return i
```

3. $j = 3$. Так как $6 < 7$, то ничего не изменяется.

4. $j = 4$. Так как $6 > 5$, то запускается тело цикла. Тогда $i = 2$ и числа 5 и 8 меняются местами.

6	3	**8**	7	**5**	1	$\longrightarrow$	6	3	**5**	7	**8**	1

5. $j = 5$. Так как $6 > 1$, то запускается тело цикла. Тогда $i = 3$ и 7 и 1 меняются местами.

6	3	5	**7**	8	**1**	$\longrightarrow$	6	3	5	**1**	8	**7**

6. Последний шаг — переставить опорный элемент на место i:

6	3	5	**1**	8	7	$\longrightarrow$	**1**	3	5	**6**	8	7

Теперь рассмотрим скорость работы алгоритма.

1. Разбить задачу на подзадачи — $\Theta(n)$
2. Рекурсивно решить подзадачи. Пусть индекс опорного элемента равен r. Тогда на выполнение уйдёт $T(r-1) + T(n-r)$.
3. Объединить решения задач в одно глобальное — 0 (уже сделано).

Тогда скорость работы алгоритма задаётся следующим рекуррентным соотношением:

$$T(n) = T(r-1) + T(n-r) + \Theta(n)$$

Рассмотрим возможные случаи:

1. **Оптимальный вариант** — r всегда посередине:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \implies T(n) = \Theta(n \log n)$$

2. **Худший случай** — r всегда минимален/максимален (массив уже «почти» отсортирован):

$$T(n) = T(n-1) + \Theta(n) \implies T(n) = \Theta(n^2)$$

3. **Средний случай.** Пусть каждый раз обе части не меньше четверти.

   Подзадачей типа j называется задача такая, что размер входного массива n' соответствует следующему неравенству:

$$n\left(\frac{3}{4}\right)^{j+1} < n' \leqslant n\left(\frac{3}{4}\right)^{j}$$

Не считая рекурсии, на каждую подзадачу типа j уходит $O\left(\left(\frac{3}{4}\right)^j n\right)$.

Стоит заметить, что подзадачи типа j не пересекаются по разбиению. При этом из них получаются подзадачи типа не меньше $j+1$.

При этом количество подзадач типа j не больше $\left(\frac{4}{3}\right)^{j+1}$. Отсюда получаем, что на все подзадачи типа j нужно $O\left(\left(\frac{4}{3}\right)^{j+1}\left(\frac{3}{4}\right)^j n\right) = O(n)$.

Так как максимальный тип подзадачи можно ограничить сверху $\log_{4/3} n$, то оценка работы в среднем случае равна $O(n \log n)$. При условии, что «везёт» всегда.

Как обеспечить везение?

Мы хотим, чтобы опорный элемент был близок к середине (в отсортированном массиве). Условно, в пределах средних двух четвертей. Если выбирать наугад, вероятность 50%.

Предположим, что мы выбираем случайный элемент. Распределим все прочие, и если одна из частей меньше четверти, забудем про этот элемент и выберем другой. Повторим, пока не получим хороший элемент. В среднем на это уйдёт две попытки $\left(\frac{1}{P}\right)$. На сложности алгоритма это не сказывается никак, т.к. меняется только константа. Зато теперь так не только в лучшем, но и в среднем случае.

Лекция 4 от 21.01.2016

Двоичное дерево поиска

Так как курс всё же называется «Алгоритмы и *структуры данных*», рассмотрим такую структуру данных, как **двоичное дерево поиска**.

Во-первых, это двоичное дерево — есть узлы и связи между ними, при этом у каждого узла не более двух детей. В вершинах дерева — числа, при этом в левом поддереве узла все числа не больше, чем в самом узле; в правом же поддереве, наоборот, не меньше.

Введём обозначения: пусть x — некоторый узел. Тогда

- x.key — число в узле;
- x.left — левый потомок;
- x.right — правый потомок;
- x.p — родитель.

При этом считаем, что у каждого узла есть оба наследника, но некоторые могут узлами не являться и иметь значение NULL.

Указанные выше свойства двоичного дерева поиска можно записать так:
$y \in \text{Tree}(x.\text{left}) \implies y.\text{key} \leqslant x.\text{key}$
$y \in \text{Tree}(x.\text{right}) \implies y.\text{key} \geqslant x.\text{key}$

Что с этим деревом можно делать? Для начала, это дерево можно обойти так, чтобы перечислить элементы в порядке возрастания:

```
inorder_tree_walk(x)
    if x != NULL then
        inorder_tree_walk(x.left)
```

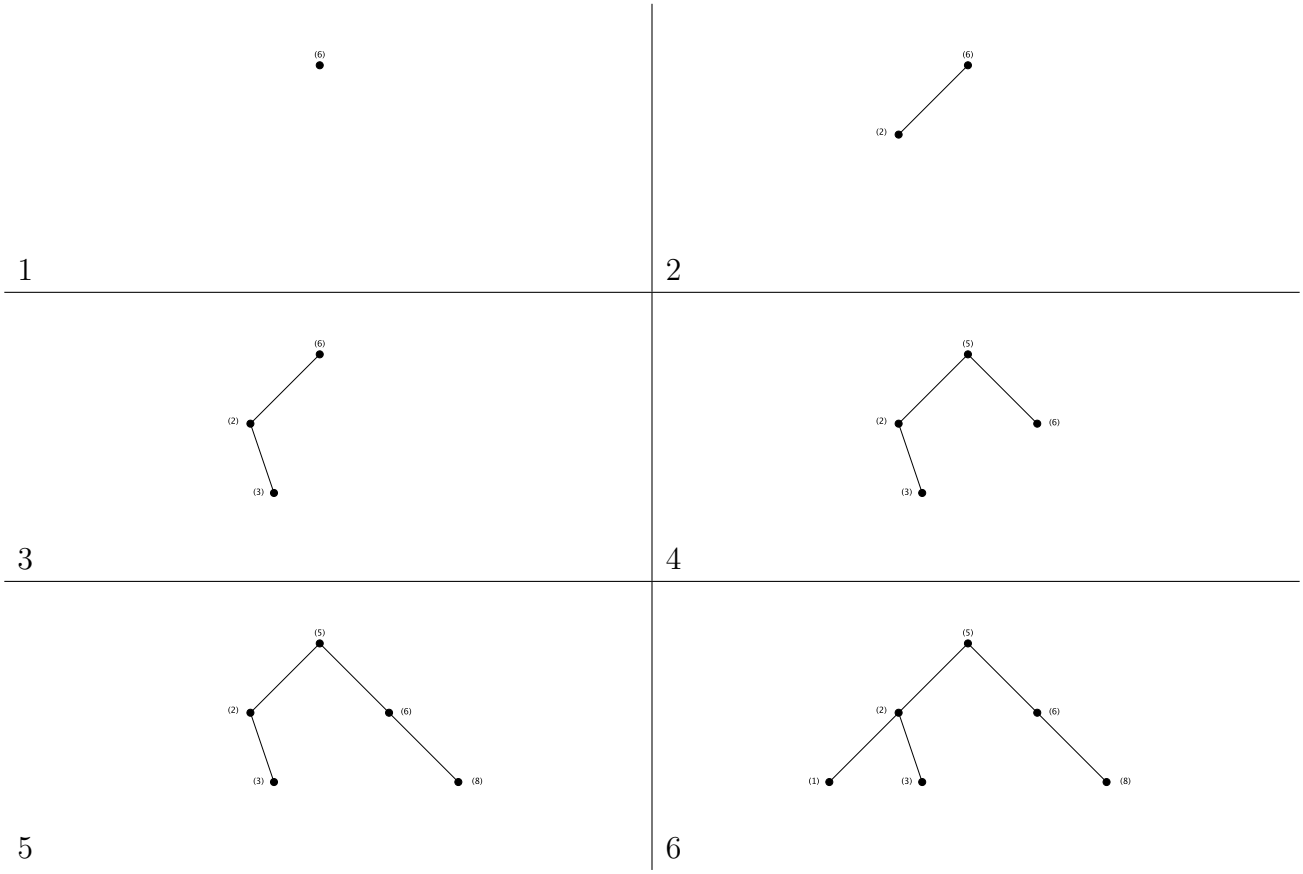
```
        output x.key
        inorder_tree_walk(x.right)
```

Сложность алгоритма — $\Theta(n)$; каждый узел мы посещаем не более одного раза, при этом операции внутри узла занимают константное время.

Запишем алгоритм сортировки с помощью дерева:

```
tree_sort(a)
    t := Tree()
    for x in a do
        tree_insert(x, t)
    inorder_tree_walk(t)
```

Пусть $a = [6,2,3,5,8,1]$. Тогда построение дерева может выглядеть так (может и по-другому, дерево далеко не всегда строится однозначно):



Заметим, что в общем и целом, это довольно похоже на алгоритм быстрой сортировки (особенно, если не двигать корневой элемент при вставке новых), а корень (под-)дерева — опорный элемент.

Время работы влгоритма: $T(n) = 2T(\frac{n}{2}) + cn \implies \Theta(n \log n)$

Предположим, что слияние дорогое:

$T(n) = 2T(\frac{n}{2}) + cn^2 \implies O(n^2 \log n)$

HERE BE KARTINKA

i-ый уровень:

- Число подзадач $= 2^i$
- Размер подзадачи $= \frac{n}{2^i}$
- Время на решение подзадачи $= c\left(\frac{n}{2^i}\right)^2$
- Всего работы $= \frac{cn^2}{2^{2i}} \cdot 2^i = \frac{cn^2}{2^i}$

$T(n) \leqslant \sum\limits_{i=0}^{\log_2 n-1} \frac{cn^2}{2^i} = cn^2 \sum \frac{1}{2^i} \leqslant 2cn^2 = O(n^2)$
$T(n) = kn^d$
Удобно начинать индукцию с шага:
Пусть $T(m) \leqslant km^d$ для $m < d$.
$T(n) \leqslant 2T(\frac{n}{2}) + cn^2 \leqslant 2k\left(\frac{n}{2}\right)^d + cn^2 = 2k\frac{n^d}{2^d} + cn^2 = \{d = 2\} = 2k\frac{n^2}{4} + cn^2 = \frac{k}{2}n^2 + cn^2 = \{k = 2c\} = kn^2$
База: $T(2) \leqslant c \leqslant 2c \cdot 2^2$
я перестал понимать происходящее и переписываю формулы
$T(n) = aT(\frac{n}{2}) + cn \implies O(n^{\log_2 a})$
$T(n) \leqslant kn^d$
$T(n) = aT(\frac{n}{2}) + cn \leqslant ak\frac{n^d}{2^d} + cn = \{d = \log_2 a\} = kn^d + cn$
$T(n) \leqslant kn^d - ln$
$T(n) = aT(\frac{n}{2}) + cn \leqslant a\left(k\frac{n^d}{2^d} - l\frac{n}{2}\right) + cn = kn^d - al\frac{n}{2} + cn = kn^d - \left(\frac{al}{2} - c\right)n = \left\{l = \frac{al}{2} - c\right\} = kn^d - ln$
База: $T(2) \leqslant c \leqslant k \cdot 2^d - 2l$
$T(n) = T\left(\frac{n}{2}\right) + cn$
$T(n) = \sum\limits_{i=0}^{\log_2 n} c\frac{n}{2^i} = cn \sum \frac{1}{2^i} \leqslant 2cn$
Или с помощью метода частичной подстановки:
$T(n) \leqslant kn^d$
$T(n) = T\left(\frac{n}{2}\right) + cn \leqslant \frac{kn}{2} + cn = kn^d$ aaaaaaaaaaaa

$T(n) = aT\left(\frac{n}{b}\right) + cn^d$

HERE BE KARTINKA

i-ый уровень:

- Число подзадач $= a^i$
- Размер подзадачи $= \frac{n}{b^i}$
- Время на решение подзадачи $= c\left(\frac{n}{b^i}\right)^b$
- Всего работы $= c\left(\frac{n}{b^i}\right)^d \cdot a^i = cn^d\left(\frac{a}{b^d}\right)^i$

Всего $\leqslant cn^d \sum\limits_{i=0}^{log_b n} \left(\frac{a}{b^d}\right)^i$

1. $a = b^d$: $O(n^d \log n)$

2. $a < b^d$: $O(n^d)$

3. $a > b^d$: $\sum \left(\frac{a}{b^d}\right)^i = O\left(\left(\frac{a}{b^d}\right)^{log_b n}\right) \ldots\ldots\ldots O\left(n^{\log_b n}\right)$

Лекция 5 от 26.01.2016

Быстрая сортировка. Продолжение

Говоря об алгоритме быстрой сортировки (QSort), мы рассматривали только случаи, когда все элементы различны. Однако это далеко не всегда так. Если в входном массиве есть равные элементы, то алгоритм может застопориться. Для того, чтобы избежать этого, изменим алгоритм Partition. Попытаемся преобразовывать массив таким образом, чтобы в левой части стояли элементы строго меньшие опорного, в правой — строго большие, а в середине — равные ему:

	...	x	...		$\longrightarrow$	$< x$	$= x$	$> x$

Обозначим за опорный элемент последний. Будем проходиться по массиву от начала до конца, выставляя элементы в нужном порядке (? — ещё не просмотренные элементы):

$< x$	$> x$	?	$= x$	x
$[1, i)$	$[i, j)$	$[j, k)$	$[k, n)$	n

Algorithm 9 Модифицированный алгоритм Partition

```
1:  function Partition(a)
2:      i := 1
3:      j := 1
4:      k := n − 1
5:      while j < k do
6:          if a[j] = a[n] then
7:              k := k − 1
8:              a[j], a[k] := a[k], a[j]
9:          else
10:             if a[j] < a[n] then
11:                 a[i], a[j] := a[j], a[i]
12:                 j := j + 1
13:                 i := i + 1
14:             else
15:                 j := j + 1
```

Заметим, что $j = k$ (так как алгоритм не закончит работу до тех пор, пока это не станет верно). Тогда на выходе получится массив вида:

$< x$	$> x$	$= x$
$[1, i)$	$[i, j)$	$[k, n]$

Остаётся только переставить части массива:

B: Самая быстрая из наших сортировок — $O(n \log n)$. А можно ли быстрее?
O: На основе только сравнений — нет.

```
1: j := n
2: while i < k and j ⩾ n do
3:     a[i], a[j] := a[j], a[i]
4:     i := i + 1
5:     j := j − 1
```

Использовать разобранные нами сортировки можно на любых сущностях, для которых определена операция сравнения.

Предположим теперь, что мы сортируем натуральные числа, не превосходящие некоторого числа C.

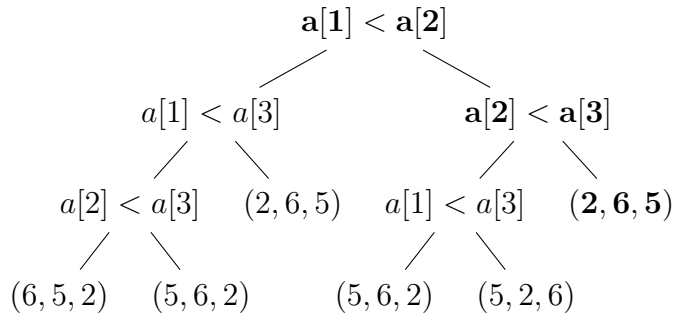
Создадим массив b размера C, заполненный нулями. Будем проходить по исходному массиву a и на каждом шаге будем добавлять 1 к соответствующему элементу массива b:

$$b[a[i]] := b[a[i]] + 1$$

Потом, проходя по получившемуся массиву b, будем восстанавливать исходный массив уже в отсортированном виде.

Такая сортировка будет работать за $O(n)$, однако, она не универсальна.

Вернёмся к универсальным сортировкам. Рассмотрим дерево для массива $a = [6, 5, 2]$:



Подобное дерево можно составить для любого детерминированного[1] алгоритма сортировки, зафиксировав n. Сложность алгоритма будет являться высота h дерева. Посчитаем это h:

- Так как алгоритм должен работать на любой перестановке из n элементов, то у дерева не может быть меньше, чем $n!$ листов.
- Так как сравнение — бинарная операция, то у каждой вершины не более двух потомков. Тогда в дереве не может быть больше, чем 2^h листьев.
- Тогда $2^h \geqslant n! \iff h \geqslant \log_2 n!$. Заметим, что:

$$n! = 1 \cdot 2 \cdot \ldots \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \underbrace{\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \cdot \ldots (n-1) \cdot n}_{\text{каждый из } \frac{n}{2} \text{ элементов не меньше } \frac{n}{2}} \geqslant \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

  Тогда $h \geqslant \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$. Из этого следует, что отсортировать произвольный массив с помощью только сравнений меньше, чем за $\Omega(n \log n)$ операций, невозможно.

[1] Детерминированный алгоритм — алгоритмический процесс, который выдаёт предопределённый результат для заданных входных данных. Например, QSort, выбирающий опорный элемент случайным образом, не является детерминированным.

Поиск медианы

Медиана — такой элемент массива, что не меньше половины элементов меньше неё, и не меньше половины — больше.

Для отсортированного массива размера n медиана будет находиться под номером $\frac{n+1}{2}$ для нечётных n и $\frac{n}{2}$ для чётных n. Пример: для массива (8, 1, 3, 5, 6, 9) медианой будет являться 5.

Как же найти медиану? Очевидно, что можно отсортировать и взять средний — $\Theta(n)$.

А можно ли найти медиану ли за линейное время? Можно. Напишем алгоритм, находящий элемент, стоящий на k-ом месте в массиве, получающемся из входного после сортировки. Это называется поиском k-ой порядковой статистики. Составим этот алгоритм, немного модифицировав QSort:

Algorithm 10 Поиск k-ой порядковой статистики

```
1: function SELECT(a, k)
2:     choose pivot a[p]
3:     i := PARTITION(a, p)
4:     if i := k then
5:         return a[i]
6:     if i > k then
7:         return SELECT(a[1 ... i − 1], k)
8:     else
9:         return SELECT(a[i + 1 ... n], k − i)
```

Как и в быстрой сортировке, неправильно выбранный опорный элемент портит скорость до n^2. Будем выбирать опорный элемент случайным образом. Попробуем посчитать время работы в среднем случае.

j-подзадача размера n'. $\left(\frac{3}{4}\right)^{j+1} n < n' \leqslant \left(\frac{3}{4}\right)^{j} n$

Как и в QSort, в среднем мы потратим две попытки на переход к следующему j.

Максимальное j — $O(\log_{\frac{4}{3}} n)$

$T(n) \leqslant \sum\limits_{j=0}^{\log_{\frac{4}{3}} n} 2 \cdot c \cdot \left(\frac{3}{4}\right)^{j} n = 2cn \sum\limits_{j=0}^{\log_{\frac{4}{3}} n} \left(\frac{3}{4}\right)^{j} \leqslant 2cn$

Время работы алгоритма в худшем случае всё ещё $O(n^2)$. Худший случай — когда на каждом шаге мы отщеплем всего один элемент. Для достижения лучшего случая, на каждом шаге нужно выбирать в качестве опорного элемента медиану.

Медиана медиан

Попробуем несколько модифицировать наш алгоритм. Разобьём входной массив на группы по 5 элементов. Отсортируем каждую такую группу. Так как размер каждой группы зафиксирован, время сортировки не зависит от n. Зависит только количество сортировок. Возьмём медиану в каждой группе и применим алгоритм нахождения медианы к получившемуся массиву медиан. Выберем её в качестве опорного элемента.

$T(n) \leqslant cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$

$T(n) \leqslant ln$ для некоторого l

$T(n) \leqslant cn + T(\frac{n}{5}) + T(\frac{7}{10}) \leqslant cn + \frac{ln}{5} + \frac{7}{10}ln$

Вход — множество точек (x_i, y_i). Выход — $i, j : d((x_i, y_i), (x_j, y_j))$ — минимально.

“Разделяй и властвуй”:

```
closest_pair_rec(P_x, P_y)
    if n < 4 then
```

Algorithm 11 Поиск k-ой порядковой статистики 2

```
1: function SELECT(a, k)
2:     Divide a into groups of 5
3:     Choose medians m_1, ... m_{n/5}
4:     x = SELECT([m_1, ..., m_{n/5}], n/10)
5:     choose x as pivot a[p]
6:     i := PARTITION(a, p)
7:     if i := k then
8:         return a[i]
9:     if i > k then
10:        return SELECT(a[1 ... i - 1], k)
11:    else
12:        return SELECT(a[i + 1 ... n], k - i)
```

```
        solve directly
    L_x := P_x[1..ceil(n/2)]. L_y = ...
    R_x := P_x[ceil(n/2)+1..n], R_y = ...
    (l_1, l_2) := closest_pair_rec(L_x, L_y)
    (r_1, r_2) := closest_pair_rec(R_x, R_y)
    delta := min(d(l_1, l_2), d(r_1, r_2))

    S_x^l := {(x, y) \in L_x | x^*-x < delta}
    S_x^r := {(x, y) \in L_x | x-x^* < delta}

    S_y := {(x, y) \in P_y | |x-x^*| \leqslant delta}

    for i := 1 to |S_y| do
        c_i := argmin_{i-15\leqslant j\leqslant i+15}(d(S_y[i], S_y[j]))
        if d(S_y[i], s_y[c]) < delta then

        WAAAAAAAAT

closest_pair(P)
    P_x := sort P by x
    P_y := sort P by y
    closest_pair_rec(P_x, P_y)
```

Теперь нужно объединить. Сколько времени у нас есть?

Для $n \log n$ $T(n) = 2T(\frac{n}{2}) + O(n)$, то есть у нас остаётся $O(n)$

Теперь нам нужно проверить, нет ли пары ближе, не по рзные стороны от границы. Нам интересна ближайшая такая пара, но только если расстояние меньше delta.

Пусть $x^* = max\{x \mid (x, y) \in L_x\}$

Тут картинка

Рассмотрим точки, попавшие в полосу. Для всех точек в L_x нам не обязательно рассматривать все точки в полосе в R_y. Рассмотрим только те, что по y лежат в delta-окрестности

Возьмём полосу и поделим на квадраты со стороной $\delta/2$. Сколько точек в каждом таком квадрате? Если там есть две точки, то расстояние между ними меньше δ, а это невозможно.

Понятно, что между точками на расстоянии $< \delta$ не более трёх рядов квадратов. Значит, всегда будет достаточно рассмотреть 15 ближайших точек.

Лекция 7 от 02.02.2016

Умножение чисел. Алгоритм Карацубы

Пусть $x = \overline{x_1 x_2 \dots x_n}$ и $y = \overline{y_1 y_2 \dots y_n}$. Распишем их умножение в столбик:

$$\begin{array}{r} \times\, x_1\, x_2 \dots x_n \\ \underline{y_1\, y_2 \dots y_n} \\ z_{11}\, z_{12} \dots z_{1n} \\ z_{21}\, z_{22} \dots z_{2n} \\ + \quad \dots\dots\dots\dots \\ \underline{z_{n1}\, z_{n2} \dots z_{nn}} \\ z_{11}\, z_{12} \dots\dots\dots\dots z_{2n}\, z_{2n+1} \end{array}$$

Какова сложность такого умножения? Всего n строк. На получение каждой строки тратится $O(n)$ операций. Тогда сложность этого алгоритма — $nO(n) = O(n^2)$. Теперь вопрос: *а можно ли быстрее?* Один из величайших математиков XX века, А.Н. Колмогоров, считал, что это невозможно.

Попробуем воспользоваться стратегией «Разделяй и властвуй». Разобьём числа в разрядной записи пополам. Тогда

$$\times \begin{cases} x = 10^{n/2}a + b \\ y = 10^{n/2}c + d \end{cases}$$
$$\Downarrow$$
$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd$$

Как видно, получается 4 умножения чисел размера $\frac{n}{2}$. Так как сложение имеет сложность $\Omega(n)$, то

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$

Чему равно $T(n)$? Воспользуемся основной теоремой. Напомним: в общем виде неравенство имеет вид:

$$T(n) \leqslant aT\left(\frac{n}{b}\right) + cn^d$$

В нашем случае $a = 4, b = 2, d = 1$. Заметим, что $4 > 2^1 \implies a > b^d$. Тогда $T(n) = O(n^{\log_2 4}) = O(n^2)$.

Как видно, it's not very effective. Хотелось бы свести число умножений на каждом этапе к трём, так как это понизит сложность до $O(n^{\log_2 3}) \approx O(n^{1.58})$Но как?

Вернёмся к началу. Разложим $(a + b)(c + d)$

$$(a + b)(c + d) = ac + (ad + bc) + bd \implies ad + bc = (a + b)(c + d) - ac - bd$$

Подставим это в начальное выражение для xy:

$$xy = 10^n ac + 10^{n/2}((a + b)(c + d) - ac - bd) + bd$$

Отсюда видно, что достаточно посчитать три числа размера $\frac{n}{2}$: $(a+b)(c+d), ac$ и bd. Тогда:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) \implies T(n) = O(n^{\log_2 3})$$

Полученный алгоритм называется *алгоритмом Карацубы*. На данный момент доказано, что для любого $\varepsilon > 0$ существует алгоритм, который совершает умножение двух чисел с сложностью $O(n^{1+\varepsilon})$. Также стоит упомянуть *алгоритм Шёнхаге-Штрассена*, работающий за $O(n \log n \log\log n)$

Перемножение матриц. Алгоритм Штрассена

Пусть у нас есть квадратные матрицы

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \text{ и } B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}$$

Сколько операций нужно для умножения матриц? Умножим их по определению. Матрицу $C = AB$ заполним следующим образом:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Всего в матрице n^2 элементов. На получение каждого элемента уходит $O(n)$ операций (умножение за константное время и сложение n элементов). Тогда умножение требует $n^2 O(n) = O(n^3)$ операций.

А можно ли быстрее? Попробуем применить стратегию «Разделяй и властвуй». Представим матрицы A и B в виде:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ и } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

где каждая матрица имеет размер $\frac{n}{2}$. Тогда матрица C будет иметь вид:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Как видно, получаем 8 перемножений матриц порядка $\frac{n}{2}$. Тогда

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

По основной теореме получаем, что $T(n) = O\left(n^{\log_2 8}\right) = O(n^3)$.

Можно ли уменьшить число умножений до 7? *Алгоритм Штрассена* утверждает, что можно. Он предлагает ввести следующие матрицы (даже не спрашивайте, как до них дошли):

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}); \\ M_2 = (A_{21} + A_{22})B_{11}; \\ M_3 = A_{11}(B_{12} - B_{22}); \\ M_4 = A_{22}(B_{21} + B_{11}); \\ M_5 = (A_{11} + A_{12})B_{22}; \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}); \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}); \end{cases}$$

Тогда

$$\begin{cases} C_1 &= M_1 + M_4 - M_5 + M_7; \\ C_2 &= M_3 + M_5; \\ C_3 &= M_2 + M_4; \\ C_4 &= M_1 - M_2 + M_5 + M_6; \end{cases}$$

Можно проверить что всё верно (оставим это как ~~наказание~~ упражнение читателю). Сложность алгоритма:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \implies T(n) = O\left(n^{\log_2 7}\right)$$

На данный момент один из самых быстрых алгоритмов имеет сложность $\approx O(n^{2.3})$ (*алгоритм Виноградова*). Но этот алгоритм быстрее только в теории — из-за астрономически огромной константы.

Лекция 8 от 4.02.2016

Быстрое возведение в степень

Пусть $x = \overline{x_1 \dots x_n}$, $y = \overline{y_1 \dots y_n}$. Можно ли за полиномиальное время возвести число x в степень y?

Тупо умножать x на себя y совершенно неоптимально — несложно показать, что сложность алгоритма будет $O(2^n)$ (где n — число цифр в числе). При этом само число x^y содержит $10^n n$ цифр. Получается, что один только размер результата экспоненциален, то есть полиномиальной сложности не хватит даже на вывод результата.

А если по модулю (т.е. результатом будет $x^y \pmod p$ для некоторого указанного p)? Прямое умножение всё равно достаточно медленно. Можно ли быстрее? Оказывается, что да.

Algorithm 12 Быстрое возведение в степень

```
1: function POWER(x, y, p)                          ▷ алгоритм считает x^y (mod p)
2:     if y = 0 then
3:         return 1
4:     t := POWER(x, ⌊y/2⌋, p)
5:     if y ≡ 0 (mod 2) then
6:         return t^2
7:     else
8:         return xt^2
```

Легко понять, что глубина рекурсии для данного алгоритма равна $O(\log y) = O(n)$.

Покажем, как он работает на примере $x = 4, y = 33$:

$$x^{33} = x(x^{16})^2 = x((x^8)^2)^2 = x(((x^4)^2)^2)^2 = x((((x^2)^2)^2)^2)^2$$

. Как видно, для возведения числа в 33-ю степень достаточно 7 умножений.

Обратная задача

Пусть нам известны числа x, z, p, каждое по n цифр. Можно ли за полиномиальное время найти число y такое, что $x^y = z \pmod p$.

Сказать сложно — с одной стороны, такой алгоритм ещё не смогли придумать. С другой стороны, не могут доказать того, что его нет. Это всё, вообще говоря, висит на известной проблеме $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ и подробнее мы об этом поговорим ближе к концу курса.

Обработка текста

Предположим, у нас есть n слов, и эти слова мы хотим разместить на странице (порядок, разумеется, не меняя — это же, в конце концов, текст). При этом, шрифт моноширинный, а

ширина строки ограничена. Что мы хотим — разместить текст так, чтобы он был выровнен по обоим краям. При этом хотелось бы, чтобы пробелы были примерно одинаковы по ширине.

Введём такую ??? (меру? хз): $\varepsilon(i,j) = L - \sum\limits_{t=i}^{j} |w_t| - (j-i)$ — число дополнительных пробелов в строке с i-го по j-ое слово.

Также введём $c(i,j)$ — стоимость размещения.

$$c(i,j) = \begin{cases} +\infty, \varepsilon(i,j) < 0 \\ \left(\frac{\varepsilon(i,j)}{j-i}\right)^3, \varepsilon(i,j) \geqslant 0 \end{cases}$$

И как это решать? Можно попробовать жадным алгоритмом — просто "впихивать" слова в строку, пока влезают. Он тут не работает, так как он вообще не учитывает стоимость.

Попробуем наш извечный "разделяй и властвуй". Базовый случай — слова помещаются в одну строку, а если не помещаются — переносим и повторяем. Но тут тоже не учитывается стоимость, так что вряд ли будет сильно лучше.

Вход: $w_1, \ldots, w_n; c(i,j)$.

Выход: $j_0, \ldots, j_{l+1}$, такие что $j_0 = 1$, $j_{l+1} = n$, $\sum c(j_i, j_{i+1})$ минимальна.

Сколько всего таких наборов? Мест, где в принципе может оказаться разрыв строки — $n-1$, в каждом можно поставить или не поставить — итого 2^{n-1} разбиений.

Пусть $OPT(j)$ — стоимость оптимального размещения слов с j-ого по n-ное. Наша задача — вычислить $OPT(1)$. А как?

$$OPT(1) = \min_{i \leqslant n} \{c(1,i) + OPT(i+1)\}$$

```
OPT(j):
    if j = n+1 then return 0
    f:= +inf
    for i:= j to n do
     f:= min(f, c(i, j)+OPT(i+1))
```

$$(*)\ \mathrm{OPT}(3) = \begin{cases} 0, & j > n \\ \min\limits_{i=j\ldots n} \{c(j,i) + \mathrm{OPT}(3)(i+1)\}, & \text{иначе} \end{cases}$$

А сложность? Построив дерево, заметим, что OPT(3) вычисляется два раза; OPT(4) – три раза и так далее.

Будем сохранять результаты:

```
function OPT_CACHE(t)
    if M[j] ≠ NULL then
    else
        M[j] := OPT(j)
    return M[j]
```

Такая методика называется *динамическим программированием.*

Основная идея — каждая задача зависит от полиномиального числа других задач.

Лекция 9 от 9.02.2016

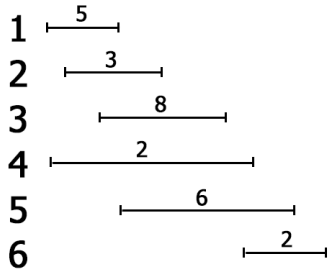
Продажа земли

Предположим, что у нас есть участок земли у берега и мы хотим его продать. При этом у нас есть несколько покупателей и каждый из них готов отдать некоторую сумму за некоторый фрагмент участка. Как максимизировать выгоду?

Формализуем: у нас есть n предложений и каждое характеризуется тремя числами — началом s, концом f и весом w. Таким образом, вход выглядит так:

$s_1, \ldots, s_n$ — начала
$f_1, \ldots, f_n$ — концы
$w_1, \ldots, w_n$ — веса

Пример:



На выходе мы хотим получить максимальную сумму весов непересекающихся интервалов:

$$\max_{T \subseteq \{1,\ldots,n\}} \sum_{i \in T} w_i$$

$$T : \forall i < j \implies f_i \leqslant s_j \vee f_j \leqslant s_i$$

Перебором задача решается за $O(2^n)$
Давайте в качестве первого шага отсортируем по правым концам за $O(n \log n)$.
Введём обозначения:
O — оптимальное решение
$OPT(i)$ — максимальаня стоимость оптимального решения для первых i интервалов
$OPT(n)$ — Максимальная стоимость оптимального решения длв всех интервалов

Пример:

$$\mathrm{OPT}(6) = \max \begin{cases} \mathrm{OPT}(5), 6 \notin O; \\ 2 + \mathrm{OPT}(3), 6 \notin O; (\text{так как все до третьего не пересекаются с шестым}) \end{cases}$$

Пусть $p(j) = \max \{i < j \mid f_i \leqslant s_j\}$ — первый интервал до j-го, совместимый с ним (то есть не пересекающийся).

Эффективное вычисление p остается в качестве упражнения.

Тогда общая формула для OPT такова:

$$\mathrm{OPT}(i) = \max \begin{cases} \mathrm{OPT}(i-1); \\ w_i + \mathrm{OPT}(p(i)); \end{cases}$$

Считая, что данные уже отсортированы, получим что сложность равна $O(n)$, но только если мы сохраняем результаты вычислений. Иначе мы делаем много лишних вычислений, и

Algorithm 13 Подсчёт $OPT(i)$

```
1: function ComputeOpt(i)
2:     if i = 0 then
3:         return 0
4:     return max{ComputeOpt(i − 1), w_i + ComputeOpt(p(i))}
```

время будет таким: $T(n) = T(n-1) + T(n-2) + c$. Очень похоже на числа Фибоначчи, а они растут экспоненциально; это выражение — тоже.

Значит надо сохранять вычисления в некоторый массив OPT. Инициализируем его так:

$$\text{OPT} = [0, -1, \ldots, -1]$$

Algorithm 14 Модифицированный подсчёт $OPT(i)$

```
1: function ComputeOpt(i)
2:     if OPT(I) < 0 then
3:         OPT[i] = max{ComputeOpt(i − 1), w_i + ComputeOpt(p(i))}
4:     return OPT[i]
```

Вот теперь сложность алгоритма — $O(n)$

Так как мы вычисляем 1 раз каждый элемент OPT, мы можем избавиться от рекурсии:

Algorithm 15 Модифицированный подсчёт $OPT(i)$ без рекурсии

```
1: function ComputeOpt(i)
2:     OPT = [0, −1, ..., −1]
3:     for OPT(I) < 0 to n do
4:         OPT[i] = max{OPT[i − 1], w_i + OPT[p(i)]}
5:     return OPT[n]
```

Как теперь определить, какие именно участки нужно продать? Можно хранить в OPT на i-ом месте необходимые участки, но это замедлит нашу программу (не асимптотически), так как придётся тратить на запись не константное время, а некоторое $O(n)$. Однако, можно восстановить номера участков по массиву OPT:

В общем о динамическом программировании

Чем оно отличается от “Разделяй и властвуй”? А тем, что задачи могут пересекаться. Ведь при использовании классического “разделяй и властвуй” мы бы получили экспоненциальное решение.

Для эффективного использования этого принципа необходимы следующие условия:

- Небольшое число задач; например, полиномиальное;
- Возможность их упорядочить и выразить решения следующих через предыдущие.

Задача с прошлой лекции — выравнивание текста

Дано:

$w_1, \ldots, w_n$ — длины слов.

Algorithm 16 Восстановление решения

```
function FindSolution(OPT)
    T = ∅
    i = n
    while i > 0 do
        if OPT[i − 1] > w_i + OPT[p(i)] then
            i = i − 1
        else
            T = T ∪ i
            i = p(i)
    return p(i)
```

$c(i, j)$ — штраф за размещение $w_i, \ldots, w_j$ на одной строке.

Преобразуем наше рекурсивное решение в итеративное.
OPT(i) — оптимальное размещение $w_i, \ldots, w_n$.
OPT(i) = $\min\limits_{i \leqslant j \leqslant n} \{c(i, j) + \text{OPT}(j + 1))\}$.

Запишем итеративный алгоритм для этой формулы. Так как i-ая задача зависит от задач с большим индексом, будем заполнять массив с конца.

Algorithm 17 Выравнивание текста

```
function ComputeOpt(w_1, ..., w_n)
    best = [0] × n
    OPT = [+∞] × (n + 1)
    OPT[n + 1] = 0
    for i := n downto 1 do
        for j := i to n do
            if c(i, j) + OPT[j + i] ⩽ OPT[i] then
                OPT[i] = c(i, j) + OPT[j + i]
                best[i] = j
    return OPT[i]
```

Как видно, в данном алгоритме решение строится по ходу, потому что в данном случае это допустимо.

Лекция 10 от 11.02.2016

Расстояние редактирования (расстояние Левенштейна)

Пусть у нас есть два слова и мы хотим из первого сделать второе. При этом мы можем произвольно расставлять пробелы и заменять буквы. Например:

	п е р в о е	и ⁄ е ⁄ р ⁄ в ⁄ о ⁄ е ⁄	п е р ⁄ в о е
enewcommand10.7	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓ ↓ ↓ ↓	↓ ↓ ↓
	в т о р о е	в т о р о е	в т о р о е

Применяется это почти повсеместно:

- Проверка орфографии. Введённое слово сравнивается с теми, которые есть в словаре. Если в слове есть ошибка, то программа предложит заменить слово на наиболее похожее верное слово.

- Проверка работ на плагиат. Опять же, идёт сравнение фраз с фразами из ранее написанных работ.

- Также его используют для изучения и сравнения последовательностей ДНК.

Теперь формализуем ранее сказанное. Пусть даны строки $s = s_1 \dots s_m$ и $t = t_1 \dots t_n$. Тогда *выравнивание* M — множество пар (i,j) таких, что $1 \leqslant i \leqslant m, 1 \leqslant i \leqslant m$, при этом:

$$(i_1, j_1) \in M, (i_2, j_2) \in M, i_1 = i_2 \implies j_1 = j_2$$

$$(i_1, j_1) \in M, (i_2, j_2) \in M, i_1 < i_2 \implies j_1 < j_2$$

Расстояние редактирования (оно же расстояние Левенштейна) — минимальное число операций, переводящее s в t. Доступные операции — удаление букв, добавление букв и замена одной буквы на другую.

Сколько вообще существует возможных выравниваний?

Для простоты положим, что $n = m$.

- Так как каждую букву слова можно либо заменить, либо убрать, то их явно не меньше, чем 2^n.

- Пусть мы выбрали k букв первого слова, которые будут сопоставлены k буквам другого слова. Тогда получаем, что есть $(C_n^k)^2$ выравниваний для фиксированного k (так как мы вольны в выборе этих букв). Следовательно, всего существует $\sum\limits_{k=0}^{n} (C_n^k)^2$ выравниваний.

- Или по-другому: есть $2n$ букв. Из них надо выбрать k букв первого слова для замены и $n - k$ букв второго слова для вставки. Тогда существует C_{2n}^n выравниваний.

Пусть $d(s,t)$ — расстояние редактирования между s и t.

Рассмотрим первые буквы слов. У нас есть три варианта

1. Удалить s_1 и как-то превратить остаток первого слова во второе;

2. Удалить t_1 и превратить первое слово в остаток второго;

3. Сопоставить s_1 и t_1 и превратить то, что осталось, друг в друга.

Логично, что нужно выбрать наиболее оптимальный вариант. Тогда значение расстояния будет равно минимальному из этих трёх.

Базовый случай — когда одно из слов пустое. Тогда достаточно вставить все буквы из непустого слова. Следовательно, $d(w, \text{“”}) = d(\text{“”}, w) = |w|$

В итоге получаем, что расстояние Левенштейна определяется следующим образом:

$$d(s_1 \dots s_m, t_1 \dots t_n) = \begin{cases} |t|, & s = \text{“ ”} \\ |s|, & t = \text{“ ”} \\ \min \begin{cases} d(s_1 \dots s_{m-1}, t_1 \dots t_n) + 1 \\ d(s_1 \dots s_m, t_1 \dots t_{n-1}) + 1 \\ d(s_1 \dots s_{m-1}, t_1 \dots t_{n-1}) + (s_1 = t_1) \end{cases} & \text{иначе} \end{cases}$$

Составим таблицу размером $(m+1) \times (n+1)$, где $T[i][j] = d(s_i \dots s_m, t_j \dots t_n)$. Пользуясь базовыми случаями заполним верхнюю строку и правый столбец. Теперь заполним $T[6][6]$, как минимум из $1 + T[6][7], 1 + T[7][6]$ и $T[7][7] + (s_6 \neq t_6)$. Будем продолжать так до тех пор, пока не дойдём до ячейки $T[1][1]$, в которой и будет записано искомое расстояние Левенштейна:

“”	6	5	4	3	2	1	0
е	5	4	3	2	1	0	1
о	4	3	2	1	0	1	2
в	3	3	2	1	1	2	3
р	4	3	2	2	2	3	4
е	4	3	3	3	3	4	5
п	**4**	4	4	4	4	5	6
	в	т	о	р	о	е	“”

Из данной таблицы видно, что d(“первое”,“второе”) = 4.

Напишем псевдокод для этого алгоритма:

Algorithm 18 Нахождение расстояния Левенштейна

```
1: function EditDistance(s, t)                                   ▷ p и q — слова
2:     create T[1..(m + 1), 1..(n + 1)]
3:     for k := 1 to m + 1 do
4:         T[k][n + 1] = m + 1 − k
5:     for k := 1 to n + 1 do
6:         T[m + 1][k] = n + 1 − k
7:     for j := n downto 1 do
8:         for i := m downto 1 do
9:             T[i][j] := min(1 + T[i + 1][j], 1 + T[i][j + 1], T[i + 1][j + 1] + (s[i] ≠ t[i]))
10:    return T[1][1]
```

Данный алгоритм можно улучшить, если рассматривать отдельно случаи, когда две соседние буквы переставлены местами. Тогда

$$d(s_1 \dots s_m, t_1 \dots t_n) = \begin{cases} |t|, & s = \text{“ ”} \\ |s|, & t = \text{“ ”} \\ \min \begin{cases} d(s_1 \dots s_{m-1}, t_1 \dots t_n) + 1 \\ d(s_1 \dots s_m, t_1 \dots t_{n-1}) + 1 \\ d(s_1 \dots s_{m-1}, t_1 \dots t_{n-1}) + (s_1 = t_1) \\ d(s_1 \dots s_{m-2}, t_1 \dots t_{n-2}) + 1 \end{cases} & \text{если } s_i = t_{i-1} \text{ и } s_{i-1} = t_i \\ \min \begin{cases} d(s_1 \dots s_{m-1}, t_1 \dots t_n) + 1 \\ d(s_1 \dots s_m, t_1 \dots t_{n-1}) + 1 \\ d(s_1 \dots s_{m-1}, t_1 \dots t_{n-1}) + (s_1 = t_1) \end{cases} & \text{иначе} \end{cases}$$

Данная функция называется *расстоянием Дамерау-Левенштейна.*

Сравнение алгоритмов

	Интервалы	Ширина	Редактирование
Число подзадач	$O(n)$	$O(n)$	$O(n^2)$
Число подзадач, от которых зависит задача	2	$O(n)$	3
Время	$O(n)$	$O(n^2)$	$O(n^2)$

Лекция 11 от 16.02.2016

Алгоритмы на графах

Графы бывают ориентированными и неориентированными, при этом неориентированные — частный случай ориентированных. Сегодня мы будем говорить исключительно о неориентированных.

Достижимость

Вход: $G(V, E); s, t \in V$. Вопрос: есть ли путь из s в t в G?

Небольшое отступление: ещё бывают взвешенные графы, где каждому ребру сопоставлен его вес (например, длина).

А можно поставить задачу поиска *кратчайшего пути*: Вход: $G(V, E); s, t \in V, W_i$. Выход: длина кратчайшего пути из s в t.

Но о взвешенных графах мы тоде говорит сегодня не будем.

Вообще говоря, многие задачи на графах не сразу бросаются в глаза как задачи, собственно, на графах — например, та же задача о расстоянии редактирования; если рассмотреть слова как вершины графа, а рёбра провести между теми вершинами, которые можно перевести одну в другую одой операцией; а хотим мы найти кратчайшее расстояние между dthibyjq s и t. Понятно, что такое решение не очень эффективно просто потому, что граф получается бесконечный, но тем не менее, как решение вполне годится.

Итак, задача достижимости. Что можно сделать в самом начале? Посмотреть на соседей s. Если среди них есть t, то мы выиграли; если нет, то посмотрим на соседей этих соседей и так далее. Запишем алгоритм, который обойдёт всевершины, достижимые из s и пометит их:

Пусть Q — множество тех, кого мы уже нашли, но чьих соседей ещё не проверили;

```
Explore(G, s)
    Q := {s}
    while Q != {} do
        extract u from Q
        mark u
        for v such that (u, v) in E do
            if v is not marked then
             add v to Q
```

Однако, алгоритм ещё не идеален; мы не уточнили, какую именно вершину мы извлекаем из Q; проверим, что будет, если Q работает как стек.

(тут неплохо бы как-то это описать)

Заметим, что такой алгоритм будет “углубляться” в граф на каждой итерации, двигаясь по неокоторому ациклическому пути пока ему есть кда идти; как только идти стало некуда, он начнёт “отступать”, пока не окажется в вершине, из которой можно попасть в некоторую ещё не исследованную. То есть, попав в вершину f, мы не вернёмся назад, пока не обойдём всех соседей f. Такой алгоритм называется *поиск в глубину* (*depth-first search*).

Можно записать его рекурсивно:

```
DFS(G, S)
    mark s
    for v such that (s, v) in E do
        if v is not marked
            DSF(G, v)
```

А теперь попробуем сделать Q очередью. Пройдясь по графу вручную, становится понятно, что для улучшения алгоритма стоит помечать вершины перед их добавлением в очередь. Порядок обхода это не меняет, впрочем; наш алгоритм обходит алгоритм как бы "по слоям":

Пусть $L_0 = \{s\}$. определим слои рекуррентно: $L_{j+1} = \{v \mid \forall i \leqslant j : v \notin L_i, \exists u \in L_j : (u, v) \in S\}$. Заметим, что номер слоя, в который входит вершина, есть длина кратчайшего пути из s в неё, то есть *поиск в ширину* (*breadth-first search*) можно использовать для поиска кратчайшего пути. Перепишем его для этого:

Пусть d — список расстояний.

```
BFS(G, s)
    for v in V do
        d[v] := infinity
    d[s] := 0
    Q := [s]
    while Q is not empty do
        u := dequeue(Q)
        for v such that (u, v) in E do
        if d[v] = infinity then
            enqueue(v)
            d[v] = d[u]+1
```

При этом, мы получим расстояния, но не получим пути; для этого, впрочем, достаточно завести массив P и сохранять в него родителя текущей вершины.

Компоненты связности

Для поиска компонент связности, вообще говоря, можно использовать любой алгоритм обхода графа:

Лекция 12 от 18.02.2016

Представление графов в памяти компьютера

Пусть у нас есть граф $G = (V, E)$. Пусть $n = |V|, m = |E|$. Если мы говорим о линейном времени работы, то подразумеваем $\Theta(n + m)$.

Можем заметить, что если наш граф ориентированный, да ещё и с петлями, то $m \leqslant n^2$. В более привычном случае неориентированного графа без петель, $m \leqslant \frac{n(n-1)}{2}$, но это всё равно $\Theta(n^2)$.

Минимальная же граница для связного графа — $n - 1$, но как правило, для большинства графов можно считать, что $m = \Omega(n)$.

Как можно представить граф в памяти? Самый известный способ — *матрица смежности*. Представляет она собой матрицу размера $n \times n$, где на пересечении i-ой строки и j-го столбца стоит 1 тогдjа и только тогда, когда в графе есть ребро (i, j): $A_{ij} = (i, j) \in E$. Проверить существование ребра можно за константное время, просто найдя нужную ячейку таблицы.

Также существует представление *списками смежности* — используется n списков, где каждой вершине сопоставлен один из них, и в список $N[i]$ входят только те вершины, что связаны с i-ой. Заметим интересную особенности таких списков — каждое ребро входит ровно в два списка (в один, если мы говорим об ориентированном графе); плюс нужна память на хранение пустых списков, если таковые имеются.

Выигрыш по памяти у списков смежности, по сравнению с матрицей, очень существенен для *разреженных графов* (граф, где $m = \Theta(n)$ *или около того* (цитата)). И действительно — веб-граф, представление всех связей между веб-страницами, имеет миллиарды и миллиарды вершин. Построить и обрабатывать матрицу такого размера — вообще довольно нетривиальная задача, но тут это ещё и оверкилл — большая часть ячеек будет равна 0 — в среднем на странице несколько десятков ссылок, но никак не миллиарды.

	Память	$(i,j) \in E$?	for $v \in E$
Матрица смежности	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n)$
Списки смежности	$\Theta(m+n)$	$\Theta(\deg(i))$	$\Theta(\deg(i))$

А ещё бывают мультиграфы, где может быть несколько рёбер в одной паре вершин, и взвешенные графы, но оба представления несложно модифицируются для их поддержки.

Оценка BFS

Запишем несколько отличающийся псевдокод:

```
BFS(G, s)
    discovered[s] := true
    for v in V\setminus{s} do
        discovered[v] := false
    i := 0
    T := {}
    L[0] := {s}
    while L[i] != {} do
        L[i+1] := {}
        for u in L[i] do
            for (u, v) in E do
                if not discovered[v] then
                    discovered[v] := true
                    T := T\cup{(u, v)}
                    L[i+1] := L[i+1]\cup{v}
        i := i+1
```

тут снова офигительно долгий пример работы алгоритма

Алгоритм работает примерно так же, как тот, что мы уже рассмотрели, но теперь он явно демонстрирует "послойность" BFS. Сложность алгоритма — $O(n+m)$; n как минимум потому, что мы создаём n списков; m же берётся из того, что мы рано или поздно посмотрим на все рёбра. Однако эта оценка верна для списков смежности; матрица смежности медленнее перебирает соседей вершины и с ней мы получили бы $O(n^2)$.

Оценка DFS

Тут тоже запишем несколько другой алгоритм:

```
DFS(G, s)
    for v in V do
        explored[v] := false
    S := [s] // stack
    while not empty(S) do
        u := pop(s)
        if not explored[u] then
```

```
        explored[u] := true
        for (u, v) in E do
            if not explored[v] then
                push(v, S)
                parent[v] := u
```

и ещё одна десятиминутная демонстрация *А, нет, не успели*
Сложно