Заметки и примеры к лекциям

по курсу «Основные алгоритмы»

25 февраля 2020 г.

Содержание

1 Лекция 2

1.1 Расширенный алгоритм Евклида

Для начала рассмотрим конкретный пример, а затем приведем общий алгоритм и обоснование его корректности.

Рассмотрим уравнение $715x+273y = 91$ и найдем его решение в целых числах:

В общем случае для того, чтобы существовало решение уравнения $ax + by = d$ в целых числах, необходимо и достаточно, чтобы d делилось на $\gcd(a, b)$. Проверим это, воспользовавшись для нахождения $\gcd(715, 273)$ обычным алгоритмом Евклида:

$$\gcd(715, 273) = \gcd(273, 715 \pmod{273}) = \gcd(273, 169) = \gcd(169, 104) =$$
$$\gcd(104, 65) = \gcd(65, 39) = \gcd(39, 26) = \gcd(26, 13) = \gcd(13, 13)$$
$$= \gcd(13, 0) = 13$$

Теперь приступим к решению уравнения, будем заполнять таблицу, которую инициализируем следующим образом.

x	y	715x + 273y
1	0	715
0	1	273

На следующих шагах из предпоследней строки таблицы «покомпонентно» вычитаем последнюю строку таблицы, умноженную на целую часть от деления предпоследнего значения в 3-ей колонке на последнее: в данном случае $\frac{715}{273} = 2$. Тогда новая строка в таблице: $(1, 0, 715) - 2(0, 1, 273) = (1, -2, 169)$

x	y	715x + 273y
1	0	715
0	1	273
1	-2	169

Далее результатом взятия остатка фактически будет вычитание из предпоследнего значения 3-го столбца последнего: $273 - 169 = 104$, а значит параметры x и y определяются следующим образом: $(0, 1, 273) - (1, -2, 169) = (-1, 3, 104)$

Будем продолжать выполнять те же операции, пока в правом нижнем углу таблицы не появится $\gcd(715, 273) = 13$.

x	y	715x + 273y
1	0	715
0	1	273
1	-2	169
-1	3	104
2	-5	65
-3	8	39
5	-13	26
-8	21	13

Теперь домножим последнюю строку таблицы на $\frac{91}{13} = 7$
$\rightarrow (7 \cdot 8, 7 \cdot 21, 7 \cdot 13)$
Откуда приходим к равенству: $715 \cdot (-56) + 273 \cdot 147 = 91$
То есть ОДНИМ из решений уравнения является пара: $(-56, 147)$. Обратите внимание, что это лишь одно из решений, нахождение общего будет предложено в курсе ОВАиТК.

Алгоритм

Вход: уравнение $ax + by = d$
Выход: одно из целочисленных решений уравнения

1. Проверяем, что d делится на $\gcd(a, b)$.

2. Инициализируем таблицу следующим образом:

x	y	$ax + by$
1	0	a
0	1	b

3. Затем будем заполнять таблицу в соотвествии с правилом, описанным выше до тех пор, пока в правом нижнем углу не окажется значение $\gcd(a, b)$

x	y	$ax + by$
1	0	a
0	1	b
...	...	...
a_i	b_i	c_i
a_{i+1}	b_{i+1}	c_{i+1}
$a_i - [\frac{c_i}{c_{i+1}}]a_{i+1}$	$b_i - [\frac{c_i}{c_{i+1}}]b_{i+1}$	$c_i - [\frac{c_i}{c_{i+1}}]c_{i+1}$
...	...	...
a_n	b_n	$\gcd(a, b)$

4. Домножаем последнюю строку таблицы на $\frac{d}{\gcd(a,b)}$, как не трудно заметить полученные значения x и y будут являться решением исходного уравнения.

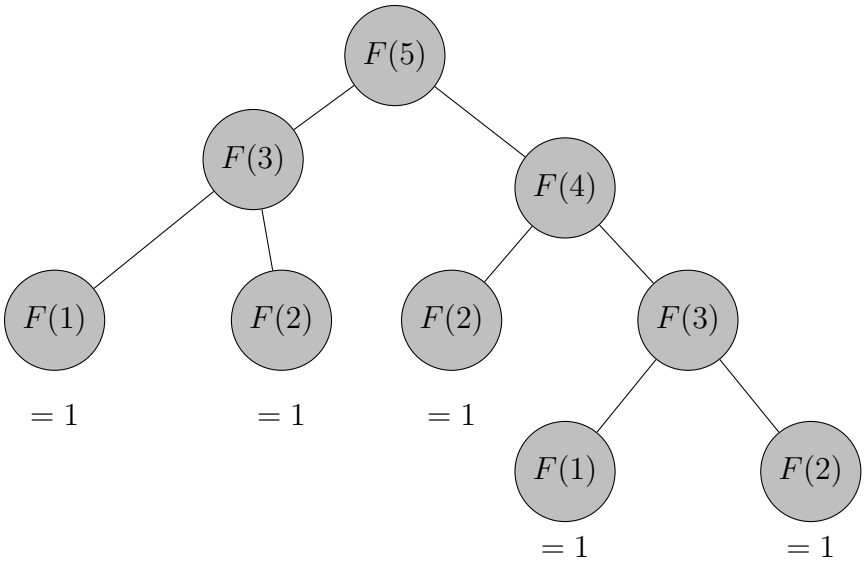
Корректность: в первых двух столбцах мы задаем значения аргументов x и y, в третьем значение линейной функции по этим аргументам, поэтому при сложении двух строк с произвольными коэффициентами получаем новую строку, где значения в соотвествующих столбцах будут линейной комбинацией исходных строк с выбранными коэффициентами. При этом на каждом шаге алгоритма в третьем столбце «идут» шаги обычного алгоритма Евклида, который заканчивается НОДом коэффициентов исходного уравнения. А зная значения параметров x и y для НОД(a, b), по линейности можно найти решение исходного уравнения.

1.2 Числа Фибоначчи

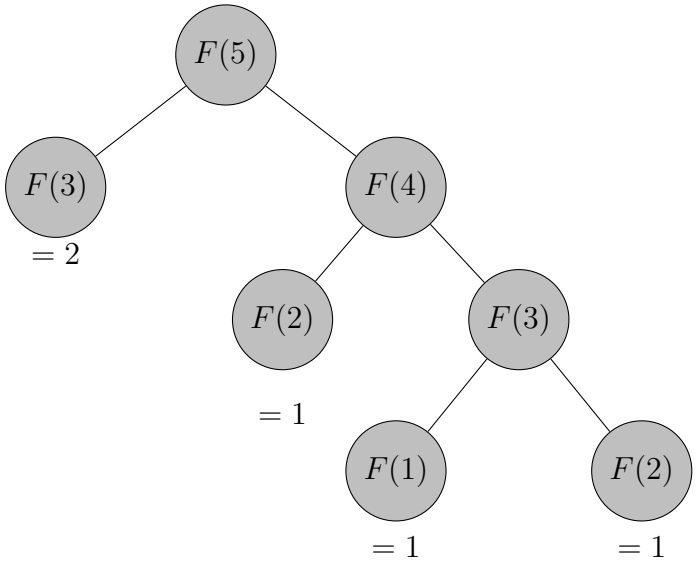
```
1 Function F(n) :
  |  Вход  : n — целое положительное число
  |  Выход: n-ое число Фибоначчи
2 |  if n < 3 then
3 |  |  return 1
4 |  end
5 |  return F(n − 1) + F(n − 2)
6 end
```

Рис. 1. Рекурсивный алгоритм вычисления чисел Фибоначчи

Это экспоненциальный алгоритм.
Рассмотрим для приведенного алгоритма дерево рекурсивных вызовов при $n = 5$:



Как видим, даже при $n = 5$ алгоритм вычисляет одно и то же значение несколько раз. Чтобы решить эту проблему можно использовать модифицированный алгоритм, который будет запоминать уже вычисленные значения и не будет перевычислять их заново, для него дерево рекурсивных вызовов будет выглядеть следующим образом:



Здесь мы полагали, что алгоритм сначала пошел в правое поддерево, где было вычисленно и сохранено значение $F(3)$, тогда попадая затем в

левую часть поддерева значение $F(3)$ будет восстановлено из сохраненного, а не перевычисленно заново. Данный алгоритм является значительно более эффективным на практике, особенно при малых n. Псевдокод для вычисления $n-$го числа Фибоначчи с запоминанием приведен ниже:

```
1 //f[] — глобальный массив, заполненный -1
2 f[1] = 1, f[2] = 1;
3 Function F2(n) :
  |  Вход  : n — целое положительное число
  |  Выход: n-ое число Фибоначчи
4 |  if f[n] ≠ −1 then
5 |  |  return f[n]
6 |  end
7 |  f[n] = F2(n − 1) + F2(n − 2);
8 |  return f[n]
9 end
```

Рис. 2. Вычисление через рекурсию с запоминанием

Псевдополиномиальный алгоритм

```
1 Function F3(n) :
  |  Вход  : n — целое положительное число
  |  Выход: n-ое число Фибоначчи
2 |  a = 1, b = 1
3 |  for i ← 3 to n do
4 |  |  c = a;
5 |  |  a = a + b;
6 |  |  b = c;
7 |  end
8 |  return a
9 end
```

Рис. 3. Итеративный алгоритм

В зависимости от рассматриваемой модели, можно по разному оценивать сложность приведенного выше алгоритма: если считать, что арифметические операции с числами стоят $O(1)$, то время работы равно $O(n)$. Заметим, что даже в этом случае алгоритм не полиномиален, поскольку сложность измеряется от длины входа, то есть от $\lceil \log_2 n \rceil$ — число n

поступает на вход в двоичной записи; такие алгоритмы называют псевдополиномиальными: алгоритм будет полиномиальным, если подать на вход число n в унарной записи, т. е. n единиц.

Если же учитывать размер слагаемых($2^{n-2} \leq F_n \leq 2^n$, несложно показать по индукции), то на сложение двух чисел $a, b \in (2^{\lfloor \log_2 n \rfloor}, 2^{\lceil \log_2 n \rceil})$ потребуется порядка $O(\log n)$ битовых операций далее учитывая оценку на последние, числа фибоначчи получаем, что в конце на каждое сложение приходится $O(n)$ битовых операций, откуда получаем, что время работы пропорцианально $O(n^2)$, что все равно существенно лучше рекурсивного алгоритма.

Однако по длине входа этот алгоритм тоже будет экспоненциальным: длина входа $l = \lceil \log n \rceil$ битов, откуда в соотвествии с приведенными рассуждениями на n, убеждаемся, что зависимость от l экспоненциальная.

2 Лекция 3

2.1 Сортировка слиянием

Основной идеей сортировки слиянием является применение принципа «разделяй и властвуй». В его основе лежит следующая идея: исходная задача разбивается несколько простых подзадач, подобных исходной, которые решаются рекурсивно(для простых подзадач решения получаем непосредственно), полученные решения комбинируются для получения решения исходной задачи.

Алгоритм сортировки слиянием

1. Делим n элементную сортируемую последовательность на две подпоследовательности по $n/2$ элементов.

2. Рекурсивно сортируем эти две подпоследовательности с использованием сортировки слиянием.

3. Соединяем две подпоследовательности для получения отсортированной последовательности.

Рассмотрим пример. Требуется с помощью сортировки слиянием отсортировать массив: $[1, 7, 3, 8, 2, 6, 5, 4]$. На первом шаге массив разбивается на две равных половины:

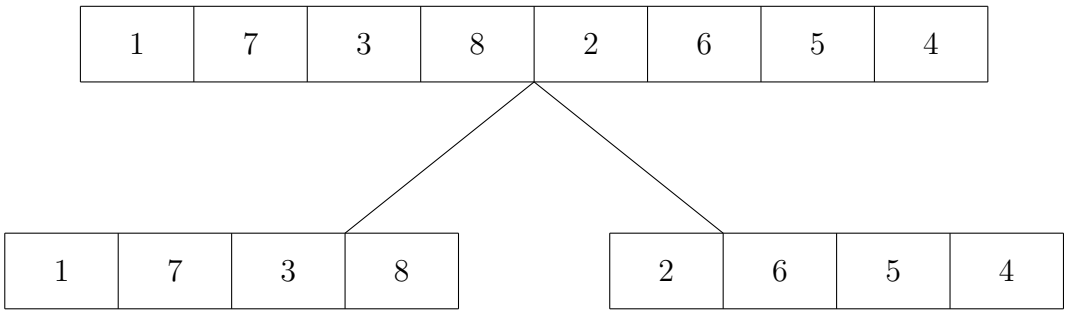
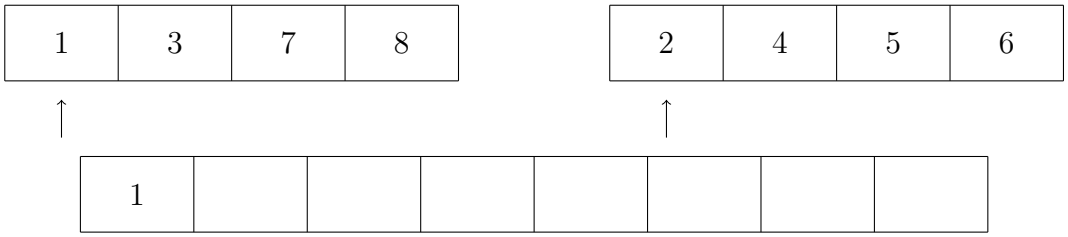


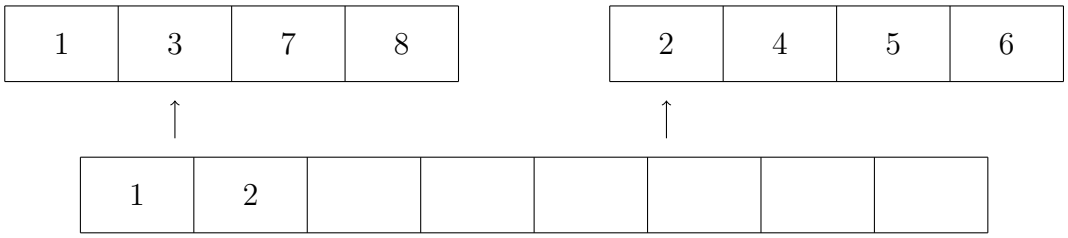
Рис. 4. Первый шаг сортировки слиянием.

Далее мы разберем пример до конца, но сначала продемонстрируем процедуру слияния. Предположим, что мы уже отсортировали половины исходного массива. Для двух отсортированных массивов заводим

указатели, которые сначала будут указывать на крайние левые элементы, затем будем сравнивать значения элементов в массивах, на которые указывают указатели, и меньший заносить в новый массив, а указатель этого элемента сдвигать на одно значения вправо:

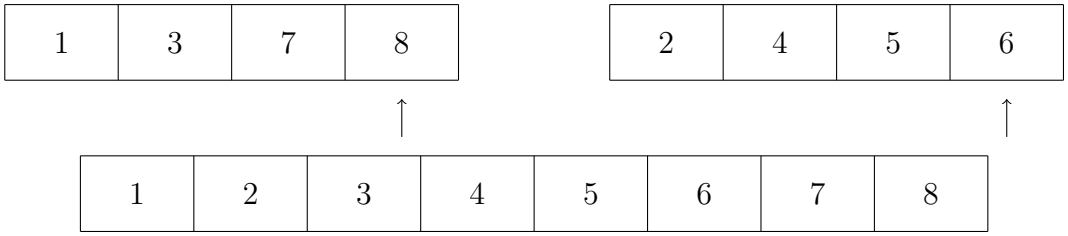


Передвигаем указатель, вновь сравниваем элементы над указателями и заносим меньшее число:



Эта процедура повторяется до тех пор пока весь итоговый массив не будет заполнен.

Конечная конфигурация будет выглядеть так:



Как видно, в результате слияния двух отсортированных половин массива, мы получили отсортированный исходный массив.

Оценить время описанной процедуры, которая называется «Слияние»(англ. «Merge») совсем просто, оно равно кол-ву сдвигов указателей(или числу сравнений элементов) - $O(n)$.

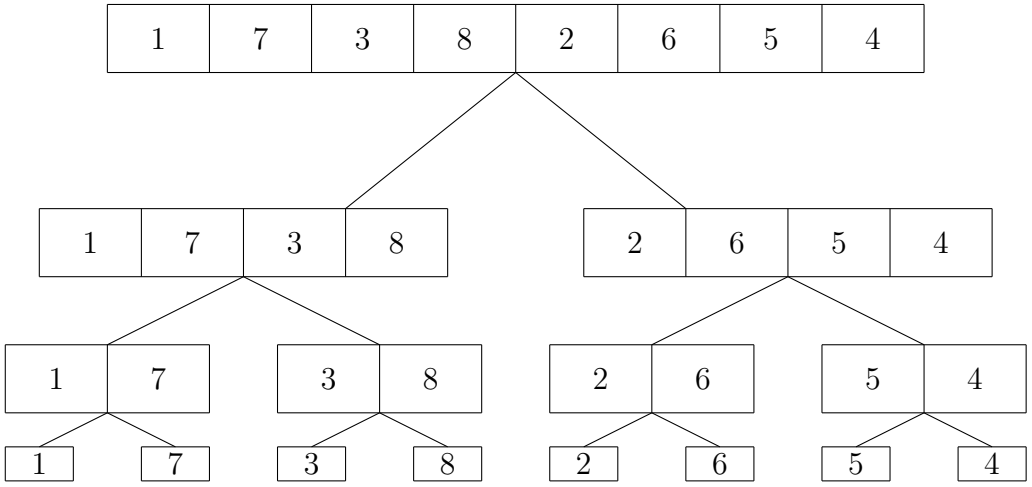


Рис. 5. Конфигурация рекурсивных вызовов на этапе разбиения.

Ознакомившись с процедурой слияния можем вернуться к примеру.

Очевидно, что массивы длины один отсортированы, поэтому можно приступать к процедуре слияния, описанной выше.

Слияние массивов длины один и два оставим для читателя, слияние массивов длины четыре приведено выше.

Корректность: для доказательства корректности алгоритма докажем корректность процедуры «слияния». Покажем, что на k-ом шаге работы процедуры слияния алгоритм выбирает k-ую порядковую статистику массива, состоящего из объедения двух уже отсортированных массивов, полученных на предыдущем уровне рекурсии. (k-ой порядковой статистикой набора элементов линейно упорядоченного множества называется такой его элемент, который является k-ым элементом набора в порядке сортировки). Предположим, утверждение выше верно для $k \leq n$, покажем, что оно верно для $k = n+1$. Пусть в некоторый момент времени на $n+1$ шаге работы фиксированы указатели p, q для соответствующих массивов x, y, по предположению $x[i], y[j], i < p, j < q$ формируют первые k порядковых статистик, следовательно $x[p], y[q]$ больше любого из уже рассмотренных элементов и в то же время в силу отсортированности массивов x, y минимальным элементом из нерассмотренных является $\min(x[p], y[q])$. В итоге получаем, что на $n+1$ шаге алгоритм выбирает $n+1$ порядковую статистику. База полученной индукции очевидна, из вышесказанного получаем корректность процедуры «слияния».

Оценим время работы описанного алгоритма, оно задаётся рекуррентным соотношением

$$T(n) = 2T(n/2) + cn.$$

На каждом шаге массив разбивается на два подмассива меньшей длины, для которых мы рекурсивно вызываем процедуру сортировки слиянием (за это отвечает слагаемое $2T(n/2)$, выполнив сортировку двух подмассивов их нужно слить, откуда в силу времени работы процедуры «слияния» имеем приведенное рекуррентное соотношение. Построим *дерево рекурсии*, чтобы понять как решать подобные соотношения:

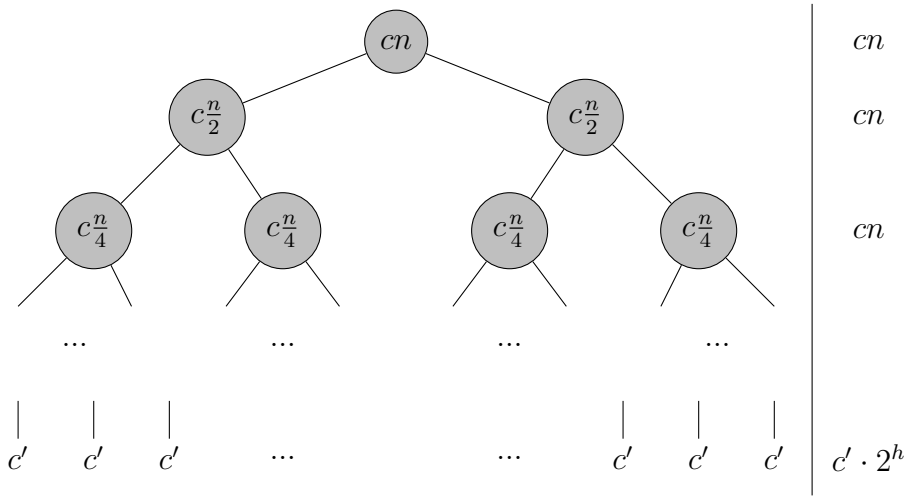


Рис. 6. Дерево рекурсии.

Самый верхний уровень будем считать нулевым. В вершинах дерева рекурсии стоит время затраченное алгоритмом только на операции на этом уровне рекурсии (остальные операции учитываются на тех уровнях, на которых они выполнялись). В случае сортировки слиянием, в вершинах записано время, которое подзадача тратит на дальнейшее разбиение и слияние после окончания работы рекурсивных вызовов (сборки ответа из ответов к подзадачам). В данном случае исходная подзадача разбивается на две подзадачи вдвое меньшего размера, которые в свою очередь разбиваются на две подзадачи вдвое меньшего размера и так далее.

Оценим число операций на каждом уровне: нулевой уровень - cn операций, первый уровень - $cn/2 + cn/2 = cn$, ... так до последнего уровня уровня, на котором для решения одной задачи треубется лишь некоторая

константа c', независящая от n. Обозначим за h — высоту полученного дерева, она очевидно вычисляется из уравнения $\frac{n}{2^h} = 1$, откуда $h = \log_2 n$; тогда общее число операций на последнем уровне $c' \cdot 2^h$. Вершины на последнем уровне будем называть *кроной*, а остальные вершины — *внутренними.*

Теперь подсчитаем общее число операций в дереве, как увидим далее, в зависимости от параметров в рекурентном соотношении основной вклад в число операций могут вносить как внутренние вершины, так и крона, поэтому подсчет для них мы будем проводить раздельно[1].

Внутренние вершны: $cn(h-1) = cn(\log_2 n - 1)$.

Крона: $c' \cdot 2^h = c'n$

Итого: $cn \log_2 n - cn + c'n = \Theta(n \log_2 n)$

2.2 Быстрое умножение - Алгоритм Карацубы

Опишем алгоритм позволяющий выполнять опрецию умножения двух чисел A и B битовой длины n за время $O(n^{\log_2 3})$.

Представим эти числа в виде $A = \overline{ab}$, $B = \overline{cd}$ где a, c - старшие половины двоичной записи, b, d - младшие половины двоичной записи (для числа $11 = 1011_2$ $a = 10_2 = 2$, $b = 11_2 = 3$)

В этом случае:

$$A \times B = \overline{ab} \times \overline{cd} = (a \times 2^{n/2} + b)(c \times 2^{n/2} + d) = ab2^n + bd + 2^{n/2}(ad + bc)$$

Заметим, что $(ad+bc) = (a+b)(c+d) - ac - bd$, и того получаем, что для вычисления искомого произведения n-битовых чисел, необходимо вычислить 3 произведения (ac, bd, $(a+b)(c+d)$) чисел битовой длинны $n/2$, и найти соответсвующую сумму затратив на это $\Theta(n)$ элементарных операций. Таким образом приходим к рекурентной формуле:

$$T(n) = 3T\left(\frac{n}{2}\right) + cn$$

[1]При должном старании можно свести всё к анализу только внутренних вершин. Введённое нами разделение нужно только для удобства в доказательствах.

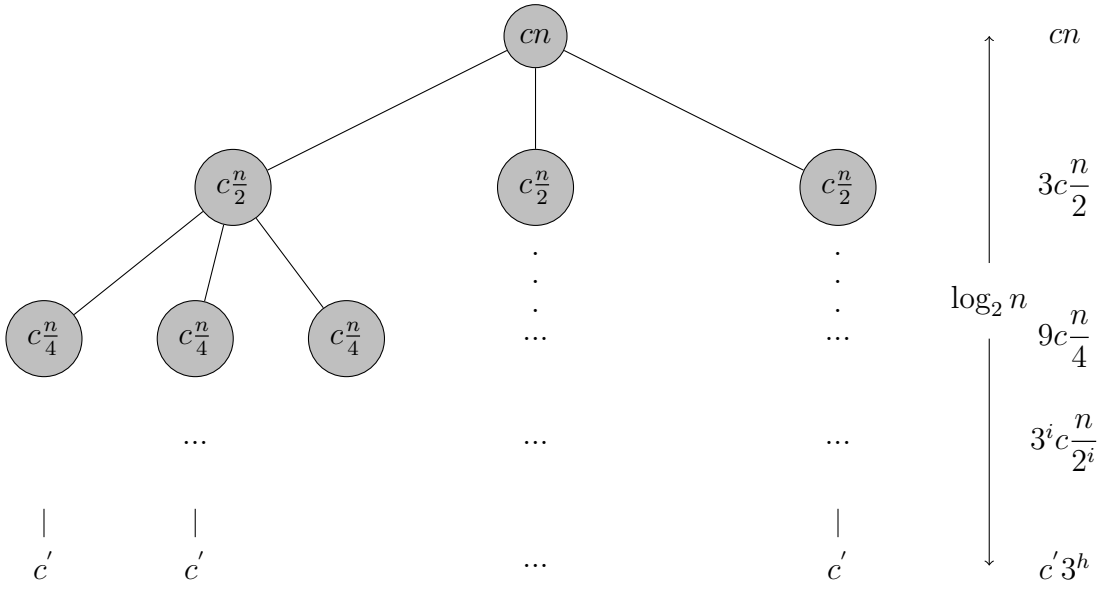


Рис. 7. Дерево рекурсии для алгоритма Карацубы

Найдём с помощью дерева рекурсии на рис. 7 время работы алгоритма; как и раньше оценим по отдельности вклад внутренних вершин и кроны ($h = \log_2 n$):

$$\sum_{i=0}^{h-1} c\left(\frac{3}{2}\right)^i n + c'3^h = cn\frac{1-(3/2)^{h-1}}{1-3/2} + c'3^h =$$

$$= 2cn\left(\frac{n^{\log_2 3}}{n} - 1\right) + c'n^{\log_2 3} = \Theta(n^{\log_2 3}).$$

Далее рассмотрим теорему, позволяющая быстро и удобно решать подобные рекуррентные соотношения.

2.3 Основная теорема(Master theorem)

Имеется рекурентное соотношение

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

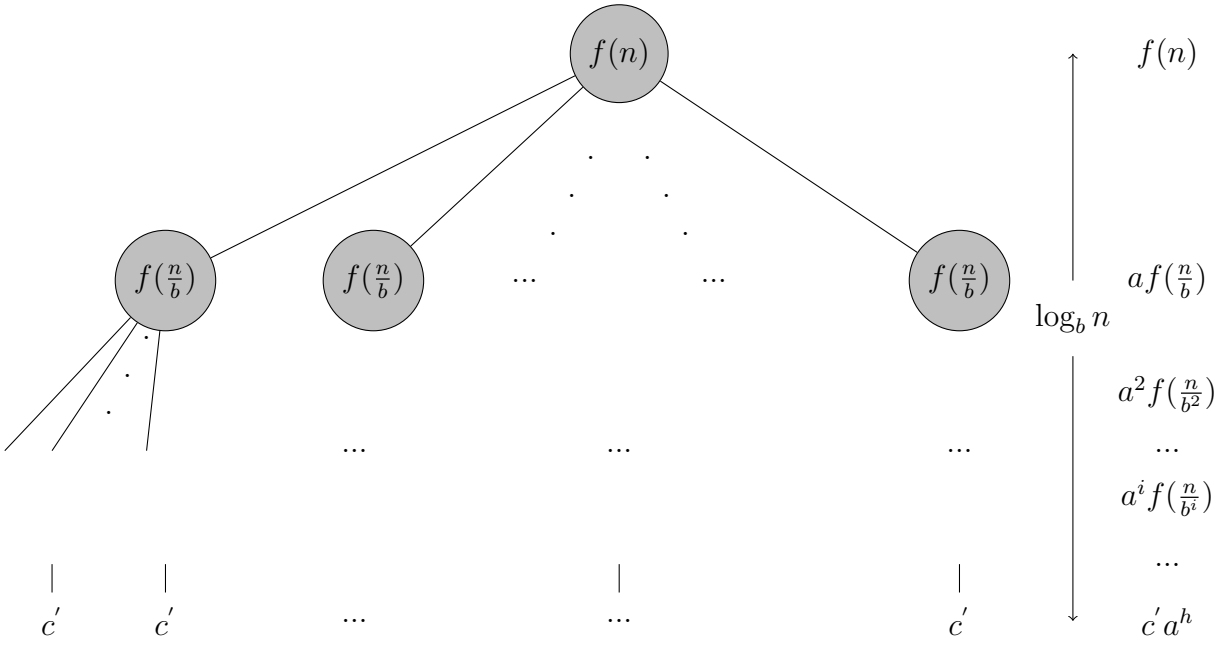
в котором $f(n) = \Theta(1)$ при малых n. Обозначим $d = \log_b a$. Тогда справедливы следующие утверждения.

1. Если $\exists\ \varepsilon > 0 : f(n) = O(n^{d-\varepsilon})$, то $T(n) = \Theta(n^d)$.
2. Если $f(n) = \Theta(n^d)$, то $T(n) = \Theta(n^d \log n)$.
3. Если $\exists\ \varepsilon > 0$ и одновременно выполняются условия
   (a) $f(n) = \Omega(n^{d+\varepsilon})$,
   (b) $\exists c\ \ 0 < c < 1 : af\left(\frac{n}{b}\right) \le cf(n)$,

   то $T(n) = \Theta(f(n))$.

Доказательство

Построим дерево рекурсии для общего случая: исходная подзадача разбивается на a подзадач, каждая размером n/b



Как нетрудно понять, высота дерева $h = \log_b n$. В дальнейшем анализе нам будут часто требоваться следующие соотношения: $a^d = b$ (напомним, что $d = \log_b a$), $a^h = a^{\log_b n} = b^{\log_b a \cdot \log_b n} = n^d$. Выразим $T(n)$ через число операций во внутренних вершинах и в кроне:

$$T(n) = \sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) + c' n^d. \qquad (1)$$

1. Учтем, что $f(n) = O(n^{d-\varepsilon})$ и оценим первое слагаемое суммы (1); примечания к переходам вынесены в сноски.

   $\sum\limits_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) \le \sum\limits_{i=0}^{h-1} a^i c\left(\frac{n}{b^i}\right)^{d-\varepsilon} \le cn^d \sum\limits_{i=0}^{h-1} \frac{a^i}{(b^d)^i} \cdot \frac{n^{-\varepsilon}}{b^{-i\varepsilon}} \overset{2}{=} cn^{d-\varepsilon} \sum\limits_{i=0}^{h-1} (b^\varepsilon)^i \overset{3}{=}$

   $= cn^{d-\varepsilon} \frac{(b^\varepsilon)^h - 1}{b^\varepsilon - 1} \overset{4}{=} \Theta(n^d).$

   Откуда $\sum\limits_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) = O(n^d)$.

   Кроме того, как было показано выше:

   $T(n) = \sum\limits_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) + c'n^d > c'n^d$, а значит $T(n) = \Omega(n^d)$.

   Откуда приходим к утверждению первого пункта теоремы.

2. Также оценим первое слагаемое суммы (1), учитывая теперь, что $f(n) = \Theta(n^d)$.

   $\sum\limits_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) = \sum\limits_{i=0}^{h-1} a^i \left(\frac{n}{b^i}\right)^d = \sum\limits_{i=0}^{h-1} n^d = n^d \cdot h = n^d \cdot \log n = \Theta(n^d \log n).$

   Таким образом, $T(n) = \Theta(n^d \log n) + c'n^d = \Theta(n^d \log n)$.

3. Начнём с условия (b): $af\left(\frac{n}{b}\right) \le cf(n) \overset{5}{\Rightarrow} a^i f(\frac{n}{b^i}) \le c^i f(n)$.

   $\sum\limits_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) \le f(n) \sum\limits_{i=0}^{h-1} c^i \le f(n) \sum\limits_{i=0}^{\infty} c^i \overset{6}{=} \frac{f(n)}{1-c} = O(f(n)).$

   То есть $T(n) = O(f(n)) + c'n^d = O(f(n))$, поскольку $n^d = O(f(n))$ в силу условия (a).

   С другой стороы $T(n) = aT\left(\frac{n}{b}\right) + f(n) \ge f(n)$, т. е. $T(n) = \Omega(f(n))$.

   Откуда следует утверждение третьего пунтка теоремы.

Важно отметить, что вышеприведенные оценки верны и для соотношений:

$$T(n) = af\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n),$$

---

[2] $b^d = a$

[3] воспользуемся формулой для геометрической прогрессии

[4] $b^h = n$

[5] i раз применим это условие

[6] бесконечно убывающая геометрическая прогрессия

$$T(n) = af\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n).$$

Это справедливо потому что при работе с рекуррентными соотношения для алгоритмов мы имеем дело только с монотонными функциями $T(n)$ — если задачу можно решить на входе большей длины за время t, то и на входе меньшей длины её тоже можно решить за время t. Осталось только воспользоваться соотношением

$$\forall b \;\; \forall n \;\; \exists h : b^h \leq n < b^{h+1},$$

которое берётся из свойств систем счисления.