Московский Государственный Университет имени М. В. Ломоносова

Факультет Вычислительной Математики и Кибернетики

Кафедра Алгоритмических Языков

НЕ ЯВЛЯЕТСЯ ОФИЦИАЛЬНЫМ УЧЕБНЫМ ПОСОБИЕМ

Вычислительная сложность алгоритмов

(V семестр)

лектор — профессор С. А. Абрамов

(составлено по конспекту лекций за осенний семестр 2005/06 учебного года)

Москва 2005

Содержание

Сложность алгоритмов как функций числовых аргументов

A — алгоритм: на вход x, результат — y.

$C_A^T(x)$ — временные затраты (цена)

$C_A^S(x)$ — пространственные затраты (по памяти)

$\|x\|$ — размер входа: $\|x\| \geq 0$

$\lfloor x \rfloor$ — пол: округление в меньшую сторону

$\lceil x \rceil$ — потолок: округление в большую сторону

Примеры

$$\lfloor 3{,}14 \rfloor = 3; \qquad \lfloor -3{,}14 \rfloor = -4$$

$$\lceil -3{,}14 \rceil = -3; \lceil 3{,}14 \rceil = 4$$

Рассмотрим следующие алгоритмы:

1. (MM) Перемножение 2-х квадратных матриц порядка n: требуется n^3 умножений.
2. (TD) Проверка на простоту числа n — пробные деления на 2, 3, …, $\lfloor \sqrt{n} \rfloor$: требуется от 1 до $\lfloor \sqrt{n} \rfloor - 1$ делений.
3. (I) Сортировка простыми вставками массива из n элементов: требуется от $n-1$ до $\frac{n(n-1)}{2}$ сравнений.

Определение. Временной сложностью (сложностью в худшем случае по времени) алгоритма A называется функция от размера входа n следующего вида: $T_A(n) = \max\limits_{\|x\|=n} C_A^T(x)$.

Определение. Пространственной сложностью (сложностью в худшем случае по памяти) алгоритма A называется функция от размера входа n следующего вида: $S_A(n) = \max\limits_{\|x\|=n} C_A^S(x)$.

Замечание. Если алгоритм A представляет собой суперпозицию двух алгоритмов U и V: $A = U \cdot V$, то вообще говоря, $T_A(n) \neq T_U(n) + T_V(n)$.

Примеры.

$$T_{MM}(n) = n^3; \; T_I(n) = \frac{n(n-1)}{2}; \; T_{TD}(n) \leq \lfloor \sqrt{n} \rfloor - 1$$

Рассмотрим $T_{TD}(n)$ при конкретных значениях n:

n	111	112	113	114	115	116	117	118	119	120
$T_{TD}(n)$	2	1	9	1	4	1	2	1	6	1

Такое поведение сложности связано с неверным выбором размера входа.

$$S_{TD}(n) = O(1); \; S_{MM}(n) = n^2 + O(1)$$

Рассмотрим детальнее алгоритм сортировки простыми вставками (I). Имеется массив из n элементов: $x_1, x_2, ..., x_n$. Считая на i-м шаге, что первые $(i-1)$ элементов упорядочены и

нужно поставить $(i+1)$-й элемент на место. Для этого представляются 2 возможности: 1) пока $(i+1)$-й элемент больше соседнего левого, менять их местами; 2) начиная с первого элемента и до *i*-го, сравнивать каждый элемент с $(i+1)$-м, для определения правильного места $(i+1)$-го элемента, после чего начать перестановки, аналогичные пункту 1, чтобы $(i+1)$-й элемент встал на нужное место. Оказывается, что сложности в этих двух случаях будут различны: $T_{I_1}(n) = n(n-1)$; $T_{I_1}(n) = \dfrac{(n-1)(n+2)}{2}$. Тем не менее $T_I(n) = O(n^2)$, но $T_{I_1}(n) = n^2 + O(n)$, а $T_{I_2}(n) = \dfrac{n^2}{2} + O(n)$.

Определение. Функции $f(n)$ и $g(n)$ являются функциями одного порядка, если найдутся такие числа $c_1, c_2 > 0$ и такое $N > 0$, что $\forall n > N$ верно $c_1|g(n)| < |f(n)| < c_2|g(n)|$.

Обозначение: $f(n) \asymp g(n);\ f(n) = \Theta(g(n))$

Пример. $T_I(n) = \Theta(n)$

Замечание. Ограничение $\forall n > N$ в определении не существенно и от него можно отказаться, рассматривая $\forall n > 0$. Действительно, пусть условие $c_1|g(n)| < |f(n)| < c_2|g(n)|$ выполняется для всех *n*, больших некоторого *N*. Обозначим

$$m = \min_{1 \le n \le N} \frac{|f(n)|}{|g(n)|},\ c_1' = \min\{c_1, m\}$$

$$M = \max_{1 \le n \le N} \frac{|f(n)|}{|g(n)|},\ c_2' = \min\{c_2, M\}.$$

Тогда для таких c_1' и c_2' будет верно $c_1'|g(n)| < |f(n)| < c_2'|g(n)|$ для всех $n > 0$.

Определение. $f(n) = \Omega(g(n)) \Leftrightarrow \exists c, N > 0 : |f(n)| > c|g(n)|, \forall n > N$.

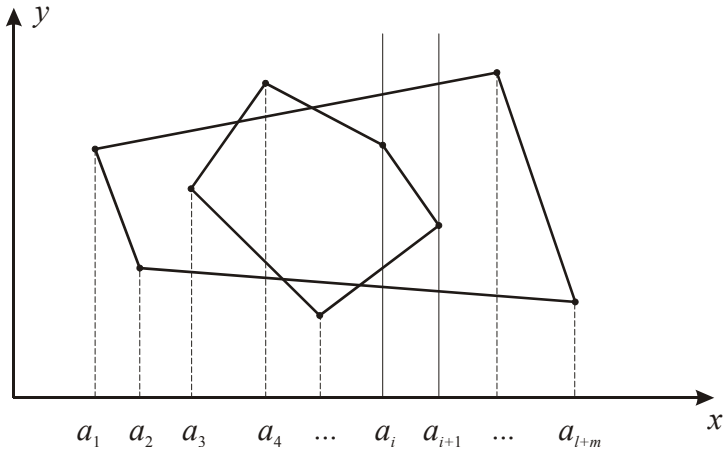
Определение. Оценка $f(n) = O(g(n))$ называется точной, если можно выбрать подпоследовательность $\{n_k\}$ такую, что для функций $\varphi(k) = f(n_k)$ и $\psi(k) = g(n_k)$ выполнено $\varphi(k) = \Omega(\psi(k))$ или $\varphi(k) = \Theta(\psi(k))$.

Определение. Оценка $f(n) = O(g(n))$ называется точной, если $f(n) = O(g(n))$ верно, а $f(n) = o(g(n))$ — не верно.

Определение. Функция $f(n)$, определённая для достаточно больших *n*, называется функцией, определённой финально.

Рассмотрим алгоритм построения пересечения 2-х выпуклых многоугольников (алгоритм Шеймоса (SH)). Имеются два выпуклых многоугольника, заданные координатами своих вершин на плоскости: $P_1, P_2, \ldots, P_l$ и $Q_1, Q_2, \ldots, Q_m$. Упорядочим массивы вершин по возрастанию *x*. Для этого воспользуемся выпуклостью многоугольников. Найдем вершину с минимальной координатой *x* — в упорядоченном массиве она будет первой. Далее выбираем вершину, смежную с найденной и имеющую минимальную координату *x* — она будет второй. Далее рассматриваются вершины, смежные с уже упорядоченными, но еще не рассмотренными: таких вершин не больше двух. Среди них выбираем ту, у которой меньшая координата *x* и добавляем в конец формируемого упорядоченного массива. Процедура повторяется до тех пор, пока все вершины не будут рассмотрены алгоритмом и упорядочены.

Используя указанный алгоритм можно упорядочить массив вершин многоугольника P, выполнив $\le c_0 l$ операций сравнения. Массив вершин многоугольника Q тем же алгоритмом упорядочивается за $\le c_0 m$ операций сравнения. Далее «сольём» полученные два массива, формируя новый массив длинной $l+m$, выбирая на каждом шаге минимальный из первых нерассмотренных элементов массивов. Для этого потребуется $\le c_1(l+m)$ операций сравнения.



Итоговая сложность не превзойдёт $c_2(l+m)$. Для каждой полосы $a_i a_{i+1}$ строим пересечение двух трапеций (в некоторых случаях трапеция вырождается в треугольник). Затем объединяем пересечения и получаем требуемый результат, причём количество необходимых операций не превзойдёт $c_3(l+m)$. В результате получаем оценки:

$$l+m < C_{HS}(P,Q) < c(l+m)$$

Если за размер входа взять величину $r = l+m$, тогда не трудно видеть, что $T_{SH}(r) = \Theta(r)$. Для размера входа $s = \max\{m,l\}$ имеем:

$$s < m+l \le 2s \Rightarrow s \le C_{SH}(P,Q) \le 2cs \Rightarrow T'_{SH}(s) = \Theta(s)$$

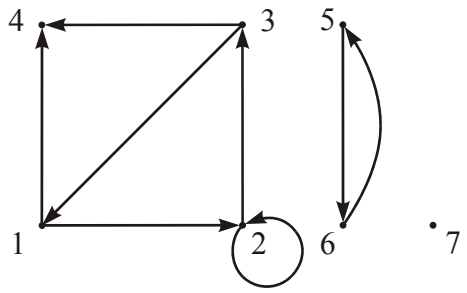
Если рассматривать функцию сложности как функцию двух аргументов l и m, то не трудно видеть, что $T''_{SH}(l,m) = \Theta(l+m)$.

Определение. $f(n_1,n_2) = \Theta\big(g(n_1,n_2)\big) \Leftrightarrow \exists c_1, c_2, N > 0$ такие, что $\forall n_1, n_2 > N$ верно $c_1\big|g(n_1,n_2)\big| \le \big|f(n_1,n_2)\big| \le c_2\big|g(n_1,n_2)\big|$.

Пусть $G = (V, E)$ — ориентированный граф. Вояжем в графе G назовем путь, удовлетворяющий 3 условиям: 1) он начинается в какой-то вершине графа G; 2) не проходит по одной дуге дважды и 3) заканчивается в вершине, откуда нет путей.

Вояж не обязательно охватывает все дуги графа.

Пример.



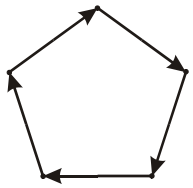
Вояж: (1, 2, 2, 3, 1, 4).

Рассмотрим алгоритм нахождения вояжа в графе. Для этого сопоставим с графом квадратную матрицу, порядок которой совпадает с числом вершин в графе. Элементы матрицы a_{ij} показывают, сколькими ребрами соединяются вершины i и j. Для графа из примера матрица выглядит следующим образом:

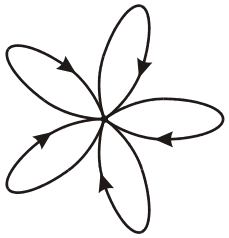
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Алгоритм заключается в следующем: начинаем с любого ненулевого элемента матрицы a_{ij}, получаем начальную вершину вояжа i — строка матрицы с выбранным элементом. Далее, так как элемент матрицы отличен от нуля, существует путь из этой вершины в вершину, соответствующую номеру столбца матрицы j. Полученная вершина будет второй в искомом вояже. Для продолжения алгоритма нужно уменьшить на единицу текущий элемент матрицы a_{ij} и перейти к рассмотрению строки j матрицы. Если эта строка не содержит отличных от нуля элементов, то построение вояжа окончено, так как из текущей вершины нет путей. Если найдётся хотя бы один отличный от нуля элемент a_{jk}, то в вояж добавляется вершина k, элемент a_{jk} уменьшается на единицу и построение вояжа продолжается поиском ненулевого элемента в строке k.

Рассмотрим сложность такого алгоритма. Не трудно видеть, что для заданного числа вершин существуют графы в виде «кольца» и «цветка»:



«кольцо»



«цветок»

на которых будет достигаться максимальное число операций. Матрицы для таких графов имеют следующий вид:

$$\begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

для «кольца»

$$\lceil n \rceil$$

для «цветка»,

где n — число дуг в графе

Для «кольца» с n вершинами получаем сложность: $2+3+\dots+n+1=\frac{n(n+1)}{2}$, тем самым $T(|E|)=c|E|^2$.

Другой алгоритм для поиска вояжа основан на представлении графа в виде набора вершин, за каждой из которых закреплён список вершин, куда идут дуги из данной. Для графа из примера это представление имеет вид:

$$a_1 = (2,4);\ a_2 = (2,3);\ a_3 = (1,4);\ a_4 = ();\ a_5 = (6);\ a_6 = (5);\ a_7 = ()$$

Для «кольца» имеем следующее представление:

$$a_1 = (2);\ a_2 = (3);\ a_3 = (4);\ \ldots;\ a_{|V|-1} = (|V|);\ a_{|V|} = (1)$$

Алгоритм для поиска вояжа по такому представлению приведём на псевдокоде:

```
l := cons(v, nil); i := v;
while not null(aᵢ) do
  i := car(aᵢ);
  l := cons(i, l);
  aᵢ := cdr(aᵢ);
od
```

где `car` — первый элемент списка;
`cdr` — конец списка;
`cons` — вставка.

Не трудно видеть, что сложность представленного алгоритма составляет $c|E|$, причём $T(|E|) = \Theta(|E|)$, так как для любого числа рёбер $|E|$ существует граф в виде «цветка».

Битовая длина числа как размер входа.

Рассмотрим функцию $\nu(n)$, определённую следующим образом:

$$\nu(n) = \begin{cases} 1, \text{ если } n = 0 \\ \lfloor \log_2 n \rfloor + 1, \text{ если } n > 0 \end{cases}$$

Определение. $\nu(n)$ называется битовой длиной n.

Рассмотрим сложность алгоритма пробных делений (TD), взяв в качестве размера входа битовую длину n (сложность, для которой в качестве размера входа выбрана битовая n, будем обозначать T_A^*).

$T_{TD}^*(m = \nu(n))$:

m	2	3	4	5	6	7
n	2-3	4-7	8-15	16-31	32-63	64-127
$\hat{n}$	3	5	13	31	59	127
$T_{TD}^*(m)$	1	1	2	4	6	10

где $\hat{n}$ — некоторое конкретное число из указанного интервала, на котором достигается максимум делений. Таких чисел в конкретном интервале может быть больше одного (например, для интервала 64-127 число 121 тоже требует 10 делений для определения простоты). Как видно из приведённой таблице, сложность алгоритма пробных делений растёт вместе с ростом величины размера входа, что даёт возможность вычислить её асимптотику.

Для этого используем постулат Бертрана, который приведём без доказательства.

Теорема (постулат Бертрана). $\forall N > 1$ в интервале $(N, 2N)$ найдётся простое число.

Используя это утверждение, легко показать, что $T^*_{TD}\left(m=\nu(n)\right)=\Theta\left(2^{\frac{m}{2}}\right)$ (см. задачу 6).

Лемма 1. Пусть $f(x)$ — финально неубывающая. Если для некоторого алгоритма A $T^*_A(m)\le f(m)$, то $T_A(n)\le f(\log_2 n+1)$.

Доказательство: $2^{m-1}\le n<2^m$; $\exists\hat{n}: 2^{m-1}\le\hat{n}<2^m$ и такое, что $T_A(\hat{n})=T^*_A(m)$.

$$T_A(n)\le T_A(\hat{n})=T^*_A(m)\le f(m)=f\left(\lfloor\log_2 n\rfloor+1\right)\le f(\log_2 n+1).$$

Лемма 2. Пусть $g(x)$ не убывает. Тогда

1) если $T_A(n)\le g(n)$, то $T^*_A(m)\le g\left(2^m\right)$;
2) если $T_A(n)\ge g(n)$, то $T^*_A(m)\ge g\left(2^{m-1}\right)$.

Доказательство: $2^{m-1}\le n<2^m$; $\exists\hat{n}\in\left[2^{m-1},\,2^m\right)$, такое, что на нём достигается максимум $T_A(n)$, т.е. $T_A(\hat{n})=\max\limits_{2^{m-1}\le n<2^m}T_A(n)$.

1) $T^*_A(m)=T_A(\hat{n})\le g(\hat{n})\le g\left(2^m\right)$;
2) $T^*_A(m)=T_A(\hat{n})\ge g(\hat{n})\ge g\left(2^{m-1}\right)$.

Указанные леммы можно обобщить на асимптотики, тогда получим:

Лемма 1*. Если $f(x)$ — неубывающая функция, $T^*_A(m)=O\left(f(m)\right)$, то $T_A(n)=$ $=O\left(f(\log_2 n+1)\right)$. Если дополнительно $f(x+1)=O\left(f(x)\right)$, то $T_A(n)=O\left(f(\log_2 n)\right)$.

Лемма 2*. Пусть $g(x)$ не убывает. Тогда

1) если $T_A(n)=O\left(g(n)\right)$, то $T^*_A(m)=O\left(g\left(2^m\right)\right)$;
2) если $T_A(n)=\Omega\left(g(n)\right)$, то $T^*_A(m)=\Omega\left(g\left(2^{m-1}\right)\right)$.

Если дополнительно $g\left(\frac{x}{2}\right)=\Omega\left(g(x)\right)$, то $T^*_A(m)=\Omega\left(g\left(2^m\right)\right)$.

Алгоритм возведения в степень: a^n.

Рассмотрим алгоритм возведения в натуральную степень методом повторных квадратов (RS = repeating squaring). Для этого запишем n в двоичной системе счисления: $n=\beta_k\beta_{k-1}\ldots\beta_1\beta_0$, $\beta_i\in\{0,1\}$. Приведём сам алгоритм на псевдокоде:

```
q := a; u := 1;
for i = 0 to k do
  if β_i = 1 then u := u * q fi;
  if i < k then q := q^2 fi;
od
```

Легко видеть, что, взяв в качестве размера входа битовую длину n, получим сложность этого алгоритма по числу умножений в следующем виде: $T^*_{RS}(m)=2m-2$. Если в качестве входа взять значение n, то сложность перепишется в следующем виде: $T_{RS}(n)=\nu(n)+\nu^*(n)-2$, где $\nu^*(n)$ — количество единиц в двоичной записи n.

Сложность в среднем

Рассмотрим множество $I_s = \{x : \|x\| = s\}$. Введём на этом множестве классическую вероятность $P_s(x) = \frac{1}{N_s}$, $\forall x \in I_s$, где $N_s = \operatorname{card} I_s = |I_s|$ (количество элементов множества). Не трудно видеть, что так введённая вероятность удовлетворяет условию: $\sum_{x \in I_s} P_s(x) = 1$.

Определение. Величина $\overline{T}_A(s) = \sum_{x \in I_s} C_A^T(x) P_s(x)$ называется временно́й сложностью в среднем алгоритма A.

Определение. Величина $\overline{S}_A(s) = \sum_{x \in I_s} C_A^S(x) P_s(x)$ называется пространственной сложностью в среднем алгоритма A.

Утверждение. Сложность в среднем не превосходит сложность в худшем случае:

$$\overline{T}_A(s) \le T_A(s) \text{ и } \overline{S}_A(s) \le S_A(s).$$

Доказательство проведём для временной сложности (для пространственной сложности доказательство проводится аналогично): $\overline{T}_A(s) = \sum_{x \in I_s} C_A^T(x) P_s(x) \le \sum_{x \in I_s} \left(\max_{x \in I_s} C_A^T(x) \right) P_s(x) =$

$$= \sum_{x \in s} T_A(s) P_s(x) = T_A(s) \underbrace{\sum_{x \in s} P_s(x)}_{=1} = T_A(s).$$

Пример. Найдём сложность в среднем алгоритма возведения в степень (RS), описанного ранее. Рассмотрим множество $I_m = \{n : 2^{m-1} \le n < 2^m\}$. Вероятность каждого элемента этого множества положим равной $\frac{1}{|I_m|} = \frac{1}{2^m - 2^{m-1}} = \frac{1}{2^{m-1}}$. Тогда

$$\overline{T}_{RS}(m) = \frac{1}{2^{m-1}} \sum_{n=2^{m-1}}^{2^m-1} \left(\underbrace{\nu(n)}_{=m} + \nu^*(n) - 2 \right) = (m-2) + \frac{1}{2^{m-1}} \sum_{n=2^{m-1}}^{2^m-1} \nu^*(n) = m - 2 + \frac{1}{2^{m-1}} \left(2^{m-1} + \sum_{k=1}^{m-1} k C_{m-1}^k \right) =$$

$$= \left\{ k C_{m-1}^k = (m-1) C_{m-2}^{k-1} \right\} = m - 2 + \frac{1}{2^{m-1}} \left(2^{m-1} + (m-1) \sum_{k=1}^{m-1} C_{m-2}^{k-1} \right) = \{ k = l + 1 \} =$$

$$= m - 2 + \frac{1}{2^{m-1}} \left(2^{m-1} + (m-1) \underbrace{\sum_{l=0}^{m-2} C_{m-2}^l}_{=(1+1)^{m-2}} \right) = m - 1 + \frac{m-1}{2^{m-1}} \cdot 2^{m-2} = \frac{3}{2}(m-1) < 2m - 2 = T_{RS}^*(m)$$

$$= m - 2 + \frac{1}{2^{m-1}} \left(2^{m-1} + (m-1) \underbrace{\sum_{l=0}^{m-2} C_{m-2}^l}_{=(1+1)^{m-2}} \right) = m - 1 + \frac{m-1}{2^{m-1}} \cdot 2^{m-2} = \frac{3}{2}(m-1) < 2m - 2 = T_{RS}^*(m)$$

Асимптотический закон распределения простых чисел.

Рассмотрим функцию Эйлера $\pi(n)$, равную количеству простых чисел, меньших n. Для этой

функции известна асимптотика: $\pi(n) \sim \frac{n}{\ln n}$. Рассмотрим ряд 1, 2, …, *N*. Вероятность того, что наперёд взятое число из этого ряда простое, согласно асимптотике $\pi(n)$, близка к $\frac{1}{\ln N}$.

Предпринимались многие попытки доказать асимптотический закон $\pi(n)$. Так, в 1850 г. Чебышёвым было доказано, что $\exists c, C > 0 : c\frac{n}{\ln n} < \pi(n) < C\frac{n}{\ln n}$. Позднее было получено, что $c = 0{,}92129$, $C = 1{,}10555$. В 1896 г. асимптотический закон для $\pi(n)$ был доказан Адамаром.

Вернёмся к алгоритму пробных делений. Известно, что $T^*_{TD}\left(m = \nu(n)\right) = \Theta\left(2^{\frac{m}{2}}\right)$ (см. задачу 6).

Определение. Алгоритм называют алгоритмом с полиномиально ограниченной сложностью, если его сложность составляет $O(m^d)$.

Определение. Алгоритм называют полиномиальным, если его сложность полиномиально ограничена.

Не используя предыдущую оценку, покажем, что алгоритм пробных делений (TD) не является полиномиально ограниченным. Для этого рассмотрим функцию $V(m)$ — количество простых чисел из полуинтервала $2^{m-1} \le n < 2^m$. При $m \to \infty$, $\pi\left(2^m\right) \sim \frac{2^m}{m \ln 2}$. Тогда $V(m) = \pi\left(2^m\right) - \pi\left(2^{m-1}\right) \sim \frac{m-2}{2m(m-1)\ln 2} \cdot 2^m$. Пусть $D(n)$ — количество делений, необходимых алгоритму при работе с *n*, тогда если $p \ge 2^{m-1}$ — простое число, то $D(p) = \left\lfloor \sqrt{2^{m-1}} \right\rfloor - 1 \ge 2^{\frac{m}{2}-1}$ (для $m \ge 4$). Используя эту оценку, для $\overline{T}_{TD}(m)$ получаем:

$$\overline{T}_{TD}(m) = \frac{1}{2^{m-1}} \sum_{n=2^{m-1}}^{2^m-1} D(n) \ge \frac{1}{2^{m-1}} \sum_{\substack{2^{m-1} \le p < 2^m \\ p \text{ - простое}}} D(p) \ge \frac{1}{2^{m-1}} \cdot 2^{\frac{m}{2}-1} \cdot V(m) = \frac{m-2}{2m(m-1)\ln 2} \cdot 2^{\frac{m}{2}} = \Omega\left(\frac{1}{m} \cdot 2^{\frac{m}{2}}\right),$$

откуда, в силу утверждения $T_A \ge \overline{T}_A$, получаем, что сложность алгоритма пробных делений не является полиномиально ограниченной.

Сложность в среднем алгоритмов сортировки.

Пусть $(a_1, ..., a_n)$ — некоторая перестановка ($a_i \in [1, n], a_i \ne a_j$ при $i \ne j$). Рассмотрим функцию $\psi_n(t_1, ..., t_n)$, которая для любого набора попарно различных чисел будет возвращать перестановку $(a_1, ..., a_n)$, отражающую порядок чисел $t_1, ..., t_n$.

Пример. $\psi_3\left(-\frac{3}{4}, -10, 17\right) = (2, 1, 3)$.

Не трудно видеть, что сложности сортировок исходного массива $t_1, \dots, t_n$ и полученной перестановки $a_1, \dots, a_n$ совпадают, поэтому можно ограничится рассмотрением сложности алгоритмов сортировки, применяемых к перестановкам.

Рассмотрим множество всевозможных перестановок их n элементов Π_n, оно же будет являться вероятностным пространством, с вероятностью каждого элемента $\frac{1}{n!}$ (т.к. количество всевозможных перестановок из n элементов равно $n!$).

Для дальнейшего нам понадобится факт из теории вероятностей, получивший название *формула полного математического ожидания*.

Пусть пространство событий W может быть разложено на сумму попарно несовместных событий W_i: $W = W_1 + ... + W_l$, $W_i \cap W_j = \varnothing$ при $i \neq j$. Пусть ξ — случайная величина, определённая на W. Тогда по определению $\mathbf{E}\xi = \sum_{\omega \in W} \xi(\omega) P(\omega)$. Пусть известны условные математические ожидания $\mathbf{E}(\xi \mid W_k)$, $k = 1, ..., l$. Тогда *формула полного математического ожидания* выглядит следующим образом:

$$\mathbf{E}\xi = \sum_{k=1}^{l} \mathbf{E}(\xi | W_k) P(W_k).$$

Утверждение.

(i) Пусть $1 \le u \le n$, $1 \le v \le n$ и событие $F_n^{u,v}$ заключается в том, что v-й элемент перестановки $a_1,...,a_n$ равен u, т.е. $a_v = u$. Тогда вероятность такого события $P_n\left(F_n^{u,v}\right) = \frac{1}{n}$.

(ii) Пусть $1 \le u \le n$ и p — некоторая перестановка чисел $1, ..., u-1$. Пусть событие $G_n^{u,p}$ заключается в том, что порядок элементов, меньших u, совпадает с p. Тогда вероятность такого события $P_n\left(G_n^{u,p}\right) = \frac{1}{(u-1)!}$.

(iii) Пусть $0 < u < v \le n$ и событие $H_n^{u,v}$ заключается в том, что существует ровно u элементов перестановки $a_1, ..., a_n$, для которых верно $a_i < a_v \quad \forall 1 \le i < v$. Тогда вероятность такого события $P_n\left(H_n^{u,v}\right) = \frac{1}{v}$.

Доказательство:

(i) очевидно: всего перестановок $n!$, перестановок, у которых один элемент фиксирован $(n-1)! \Rightarrow P_n\left(F_n^{u,v}\right) = \frac{(n-1)!}{n!} = \frac{(n-1)!}{n(n-1)!} = \frac{1}{n}$.

(ii) всего перестановок $n!$, из них перестановок, в которых порядок элементов меньших u фиксирован, $C_n^{u-1}(n-u+1)! = \frac{n!(n-u+1)!}{(u-1)!(n-u+1)!} = \frac{n!}{(u-1)!}$, откуда $P_n\left(G_n^{u,p}\right) = \frac{1}{(u-1)!}$.

(iii) если p — некоторая перестановка $1, ..., v$, то число перестановок $(a_1, ..., a_n) \in \Pi_n$, для которых $\psi_n(a_1, ..., a_v) = p$, равно $C_n^v(n-v)! = \frac{n!}{v!}$. Число перестановок p, для

которых последний элемент равен $u-1$, составляет $(v-1)!$, откуда общее число событий равно $\frac{n!}{v} \Rightarrow P_n\left(H_n^{u,v}\right)=\frac{1}{v}$.

Сортировка простыми вставками (I_1).

На i-й итерации алгоритма первые i элементов массива упорядочены: $\overbrace{x_1,...,x_i}^{\text{упорядочены}},x_{i+1},...$ и требуется поставить на место x_{i+1}. Для этого x_{i+1} сравнивается подряд со всеми элементами слева, начиная с x_i, до тех пор, пока не будет найден элемент, меньший x_{i+1}, или не будет достигнута граница массива. После определения правильного места для элемента x_{i+1} начинаются обмены элемента x_{i+1} с элементом, стоящим справа до тех пор, пока элемент x_{i+1} не встанет на место.

Найдём сложность в среднем этого алгоритма. Для этого введём следующие случайные величины $\xi_n^1,...,\xi_n^{n-1}$ такие, что $\forall a \in \Pi_n \quad \xi_n^i(a)$ показывает затраты на i-м шаге алгоритма. Тогда $\overline{T}_{I_1}(n)=\mathbf{E}\left(\xi_n^1+...+\xi_n^{n-1}\right)=\mathbf{E}\xi_n^1+...+\mathbf{E}\xi_n^{n-1}$. Рассмотрим события $H_n^{k,i+1}$, $k=0,1,...,i$, тогда $\Pi_n=H_n^{0,i+1}+H_n^{1,i+1}+...+H_n^{i,i+1}$.

По формуле полного математического ожидания получаем:

$$\mathbf{E}\xi_n^i=\frac{1}{i+1}\sum_{k=0}^{i}\mathbf{E}\left(\xi_n^i\middle|H_n^{k,i+1}\right).$$

Допустим, что событие $H_n^{k,i+1}$ выполнено, тогда

$$\xi_n^i(a)=\begin{cases} i-k+1, & k>0 \\ i, & k=0 \end{cases}$$

$$\xi_n^i(a)=\mathbf{E}\left(\xi_n^i\middle|H_n^{k,i+1}\right) \Rightarrow \mathbf{E}\xi_n^i=\frac{1}{i+1}\left(i+\sum_{k=1}^{i}\left(i-k+1\right)\right)=\frac{i}{2}+1-\frac{1}{i+1}$$

Откуда получаем $\overline{T}_{I_1}(n)=\sum_{i=1}^{n-1}\left(\frac{i}{2}+1-\frac{1}{i+1}\right)=n-1+\sum_{i=1}^{n-1}\left(\frac{i}{2}-\frac{1}{i+1}\right)$. Из курса математического анализа известно, что $\sum_{i=1}^{n}\frac{1}{i}=\ln n+O(1)$, тогда для $\overline{T}_{I_1}(n)$ получаем:

$$\overline{T}_{I_1}(n)=\frac{(n+4)(n-1)}{4}-\ln n+O(1)=\frac{n^2}{4}+O(n).$$

По числу перемещений (обменов) оценка останется верной, т.к. число перемещений отличается от числа сравнений не более, чем на 1.

Пусть ξ — затраты на сравнения, а η — затраты на перемещения. Тогда суммарная сложность в среднем будет равна: $\mathbf{E}(\xi+\eta)=\mathbf{E}\xi+\mathbf{E}\eta$.

Сортировка «пузырьком».

Имеется массив $x_1,x_2,...,x_n$, который требуется отсортировать. Сравниваем парами x_i и x_{i+1}, если $x_i>x_{i+1}$, то меняем их местами и продолжаем с x_{i+1}. После первого прохода последний (наибольший) элемент окажется на своем месте. Второй проход идёт до $n-1$, третий до

$n-2$ и т.д. Заканчиваем, когда ни одной перестановки не было или остался массив из одного элемента.

Сложность в среднем такого алгоритма составит $\overline{T}(n) = n - \sqrt{\frac{\pi n}{2}}$.

Быстрая сортировка (QS = quick sort).

Есть массив $x_1, x_2, ..., x_n$, который нужно отсортировать. Для этого выбираем разбивающий элемент (не важно какой, например, первый). Разбиваем элементы массива на 2 группы: те элементы, которые меньше разбивающего, и те, которые больше. Далее применяем ту же процедуру рекурсивно к полученным после разбиения двум частям.

В худшем случае заведомо $T_{QS}(n) = \Omega\left(n^2\right)$, т.к. придётся произвести сравнение каждого элемента с каждым в случае, если изначальный массив упорядочен. Найдём сложность в среднем $\overline{T}_{QS}(n)$ алгоритма быстрой сортировки. Для этого введём случайную величину $\xi_n(a)$, отражающую общее число сравнений, необходимых алгоритму для упорядочения перестановки *a*. Не трудно видеть, что $\overline{T}_{QS}(n)$ в этом случае будет выражаться формулой $\overline{T}_{QS}(n) = \mathbf{E}\xi_n = \frac{1}{n!}\sum_{a\in\Pi_n}\xi_n(a)$.

Разложим пространство Π_n следующим образом: $\Pi_n = K_n^1 + ... + K_n^n$, где события K_n^i заключаются в том, что разбивающий элемент после разбиения занимает *i*-ю позицию. Так как в качестве разбивающего элемента всегда берём первый, то события K_n^i будут совпадать с событиями $F_n^{i,1}$, откуда $P_n\left(K_n^i\right) = P_n\left(F_n^{i,1}\right) = \frac{1}{n}$, $i = 1,...,n$. После разбиения исходная перестановка *a* разбивается на a' и a'', причём $a' \in \Pi_{i-1}$, $a'' \in \Pi_{n-i}$.

Введём случайные величины $\xi'_n(a) = \xi_{i-1}(a')$ и $\xi''_n(a) = \xi_{n-i}(a'')$, тем самым мы останемся в том же вероятностном пространстве.

$$\overline{T}_{QS}(n) = \mathbf{E}\xi_n = \sum_{i=1}^{n}\mathbf{E}\left(\xi_n\middle|K_n^i\right)P_n\left(K_n^i\right) = \sum_{i=1}^{n}\left(\overbrace{n-1}^{\text{на разбиение}} + \mathbf{E}\left(\xi'_n\middle|K_n^i\right) + \mathbf{E}\left(\xi''_n\middle|K_n^i\right)\right)\cdot\frac{1}{n} =$$

$$= n - 1 + \frac{1}{n}\sum_{i=1}^{n}\left[\mathbf{E}\left(\xi'_n\middle|K_n^i\right) + \mathbf{E}\left(\xi''_n\middle|K_n^i\right)\right]$$

Используя очевидный факт $K_n^i = \sum_{p\in\Pi_{i-1}} G_n^{i,p}$, получаем:

$$\mathbf{E}\left(\xi'_n\middle|K_n^i\right) = \sum_{p\in\Pi_{i-1}}\xi_{i-1}(p)\cdot\frac{1}{(i-1)!} = \mathbf{E}\xi_{i-1},$$

$$\mathbf{E}\left(\xi''_n\middle|K_n^i\right) = \sum_{p\in\Pi_{n-i}}\xi_{n-i}(p)\cdot\frac{1}{(n-i)!} = \mathbf{E}\xi_{n-i}.$$

Откуда получаем $\mathbf{E}\xi_n = n - 1 + \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{E}\xi_{i-1} + \mathbf{E}\xi_{n-i}\right)$. Непосредственной проверкой устанавливается, что $\sum_{i=1}^{n}\mathbf{E}\xi_{i-1} = \sum_{i=1}^{n}\mathbf{E}\xi_{n-i} = \sum_{k=0}^{n-1}\mathbf{E}\xi_k \quad\Rightarrow\quad \overline{T}_{QS}(n) = \mathbf{E}\xi_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1}\mathbf{E}\xi_k$, что можно переписать в следующем виде:

$$\overline{T}_{QS}(n) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}_{QS}(k), \qquad (*)$$

причём, как не трудно видеть, $\overline{T}_{QS}(0) = 0$ (для упорядочения массива нулевой длины не требуется ни одного сравнения). Тогда, для $n \geq 1$, получаем

$$\overline{T}_{QS}(n-1) = n - 2 + \frac{2}{n}\sum_{k=0}^{n-2}\overline{T}_{QS}(k). \qquad (**)$$

Домножим (*) на n и вычтем (**), умноженное на $(n-1)$, тогда получим:

$$n\overline{T}_{QS}(n) - (n-1)\overline{T}_{QS}(n-1) = \underbrace{n(n-1) - (n-1)(n-2)}_{2(n-1)} + 2\overline{T}_{QS}(n-1) \Leftrightarrow$$

$$n\overline{T}_{QS}(n) - (n+1)\overline{T}_{QS}(n-1) = 2(n-1)$$

$$\Rightarrow \frac{\overline{T}_{QS}(n)}{n+1} - \frac{\overline{T}_{QS}(n-1)}{n} = \frac{2(n-1)}{n(n+1)}.$$

Введём обозначение $t(n) = \frac{\overline{T}_{QS}(n)}{n+1}$, тогда предыдущее равенство перепишется в виде: $t(n) - t(n-1) = \frac{2(n-1)}{n(n+1)}$, причём $t(0) = 0$. Известно, что если $t(n) - t(n-1) = \varphi(n)$ для всех $n \geq n_0$, тогда $t(n) = \varphi(n_0) + \sum_{k=n_0+1}^{n}\varphi(k)$ (доказывается по индукции) $\Rightarrow$ для $t(n)$ получаем:

$$t(n) = 2\sum_{k=1}^{n}\frac{k-1}{k(k+1)} = 2\sum_{k=1}^{n}\frac{1}{k+1} - 2\sum_{k=1}^{n}\frac{1}{k(k+1)}.$$

Для первого ряда из математического анализа известна оценка: $\sum_{i=1}^{n}\frac{1}{i} = \ln n + O(1)$, второй ряд сходится, а значит, может быть оценён через $O(1)$, в итоге получаем:

$$t(n) = 2\big(\ln n + O(1)\big) - O(1) = 2\ln n + O(1) \Rightarrow \overline{T}_{QS}(n) = 2n\ln n + O(n).$$

Получилась довольно приятная оценка, хоть и с участием логарифма. Но, как говорил Ландау, «Курица не птица, логарифм не бесконечность».

Для пространственной сложности верно $S_{QS}(n) = \overline{S}_{QS}(n) = O(1)$.

Рассмотрим пространственные затраты стека: т.к. алгоритм работает с отложенными задачами, то для его реализации потребуется стек, в который будут помещаться пары (p,q) — сегменты массива, которые необходимо сортировать. Оказывается (что будет доказано ниже), что если из двух получающихся после разбиения частей для обработки брать меньшую, откладывая бóльшую, то затраты не превзойдут $\log_2 n$.

Утверждение. Если бинарная рекурсивная сортировка такова, что после разбиения для сортировки выбирается меньшая часть, то количество отложенных в стеке задач не превзойдёт $\log_2 n$.

Доказательство: после разбиения длина короткого массива $\le \frac{n}{2}$, при этом длинна длинного $\le n-1$. Пусть x' — меньшая часть, длина которой равна $n' \le \frac{n}{2}$, x'' — большая, длинной $n'' \le n-1$. Для сортировки выбираем x', x'' откладывая в стек, тогда в стек попадёт $\max\{\log_2 n'+1, \log_2 n''\} \le \log_2 n$, что и требовалось доказать.

Приведём алгоритм быстрой сортировки на псевдокоде:

```
qsort(k,l):
if k<l-1 then
  xk выбирается в качестве разбивающего элемента,
  и выполняется разбиение xk,…,xl;
  пусть i – индекс, получаемый разбивающим
  элементом после разбиения;
  if i-k-1<l-i then qsort(k,i-1); qsort(i+1,l)
               else qsort(i+1,l); qsort(k,i-1)
  fi
fi
```

Рандомизированные алгоритмы.

Рассмотрим функцию `random(N)`, возвращающую случайное число от 0 до $N-1$, с вероятностью выпадения каждого $\frac{1}{N}$. Введём элемент рандомизации в алгоритм быстрой сортировки, тогда на псевдокоде он перепишется следующим образом:

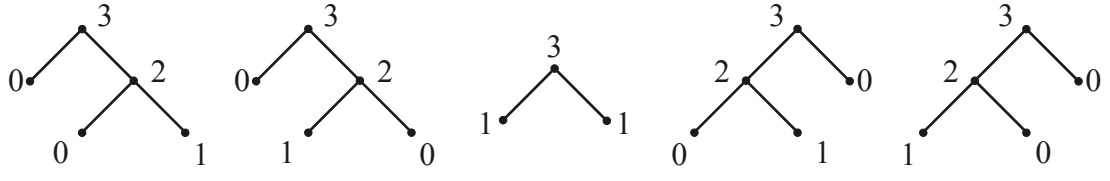
```
qsort(k,l):
if k<l-1 then
  m:=k+random(l-k+1);
  xm выбирается в качестве разбивающего элемента,
  и выполняется разбиение xk,…,xl;
  пусть i – индекс, получаемый разбивающим
  элементом после разбиения;
  if i-k-1<l-i then qsort(k,i-1); qsort(i+1,l)
               else qsort(i+1,l); qsort(k,i-1)
  fi
fi
```

Дадим другую формулировку алгоритма быстрой сортировки:

Пусть имеется полоска клетчатой бумаги размером $1\times n$ клеток, которую требуется разрезать на отдельные клетки. Разрезать может только специальный разрезальщик, причем за одну операцию он может вырезать одну любую клетку (с края или из середины). При вырезании клетки из середины полоска разбивается на две. За одно отрезание разрезальщик берёт $(n-1)$ рубль, где n — длинна полоски, из которой вырезается клетка. Цель — порезать всю полоску на клетки.

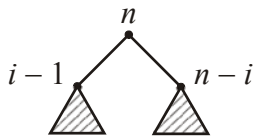
В такой формулировке сложностью алгоритма является суммарное количество рублей, необходимых для работы разрезальщику.

Сценарий для разрезаний удобно изобразить в виде двоичного дерева. Для $n=3$ (полоска из 3-х клеток) получаем следующие варианты сценариев (3-сценарии):



Смежные вершины здесь обозначены числами, соответствующими количествам остающихся клеток с одной и другой стороны, после того как одна из исходных будет вырезана.

Любой n-сценарий получается из левого $(i-1)$-сценария и правого $(n-i)$-сценария:



Обозначим через S_n множество всех n-сценариев. При фиксированном n всем n-сценариям приписываются следующие вероятности:

- если $n=0$ или $n=1$, то $P_n(s)=1$;
- если $n>1$, то $P_n(s)=\frac{1}{n}P_{i-1}(s')P_{n-i}(s'')$.

Покажем, что так введённая вероятность удовлетворяет условию $\sum_{s\in S_n}P_n(s)=1$. Доказательство проведём по индукции: для $n=0$ и $n=1$ утверждение верно по определению; пусть для всех для $k<n$ выполнено $\sum_{s\in S_k}P_k(s)=1$, покажем что для n утверждение тоже верно: $\sum_{s\in S_n}P_n(s)=\{\text{по определению}\}=\sum_{i=1}^{n}\sum_{\substack{s'\in S_{i-1}\\ s''\in S_{n-i}}}\frac{1}{n}P_{i-1}(s')P_{n-i}(s'')=$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{\left(\sum_{s'\in S_{i-1}}P_{i-1}(s')\right)}_{=1\text{ по предполо-жению индукции}}\cdot\underbrace{\left(\sum_{s''\in S_{n-i}}P_{n-i}(s'')\right)}_{=1\text{ по предполо-жению индукции}}\right)=\frac{1}{n}\sum_{i=1}^{n}1\cdot 1=1.$$

Введём случайную величину $\chi_n(s)$ — затраты алгоритма для сценария s. Наряду с ней рассмотрим также случайные величины $\chi'_n(s)=\chi_{i-1}(s')$ и $\chi''_n(s)=\chi_{n-i}(s'')$, определённые на том же вероятностном пространстве, что и $\chi_n(s)$.

Из формулировки алгоритма следует, что $\chi_n(s)=n-1+\chi'_n(s)+\chi''_n(s)$. Разложим вероятностное пространство S_n следующим образом: $S_n=S_n^1+\ldots+S_n^n$, где S_n^i соответствует тому, что левый сценарий является $(i-1)$-сценарий, а правый — $(n-i)$-сценарий. Не трудно проверить, что $P_n(S_n^i)=\frac{1}{n}$, откуда получаем:

$$\mathbf{E}\chi_n=\sum_{i=1}^{n}\mathbf{E}\left(\chi_n\middle|S_n^i\right)\cdot\frac{1}{n}=n-1+\frac{1}{n}\sum_{i=1}^{n}\left[\mathbf{E}\left(\chi'_n\middle|S_n^i\right)+\mathbf{E}\left(\chi''_n\middle|S_n^i\right)\right]=n-1+\frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{E}\chi_{i-1}+\mathbf{E}\chi_{n-i}\right)=$$
$$=n-1+\frac{2}{n}\sum_{i=1}^{n}\mathbf{E}\chi_{n-i}=n-1+\frac{2}{n}\sum_{k=0}^{n-1}\mathbf{E}\chi_k$$

т.е. получаем, что $\mathbf{E}\chi_n=n-1+\frac{2}{n}\sum_{k=0}^{n-1}\mathbf{E}\chi_k$, $\mathbf{E}\chi_0=0$. Откуда, аналогично предыдущему случаю, получаем: $\widetilde{T}_{QS}(n)=\mathbf{E}\chi_n=2n\ln n+O(n)$. В итоге получилось, что введение элемента рандомизации никак не отразилось на сложности в среднем алгоритма быстрой сортировки.

Способы ускорения алгоритма быстрой сортировки.

Если на каждой итерации выбирать 3 случайных элемента и брать серединный в качестве разбивающего, то в итоге сложность в среднем так модифицированного алгоритма составит $\frac{12}{7} n \ln n + O(n)$.

Алгоритм Евклида (Е).

Рассмотрим алгоритм Евклида нахождения наибольшего общего делителя (НОД) двух целых положительных чисел $a_0 \geq a_1$, которые будут являться входом алгоритма. Алгоритм заключается в последовательных делениях с остатком:

$$\begin{aligned} a_0 &= q_1 a_1 + a_2 \\ a_1 &= q_2 a_2 + a_3 \\ &\ldots \\ a_{n-3} &= q_{n-2} a_{n-2} + a_{n-1} \\ a_{n-2} &= q_{n-1} a_{n-1} + a_n \\ a_{n-1} &= q_n a_n \end{aligned}$$

Тогда a_n будет являться НОД, что легко показать по индукции:

$$\text{НОД}(a_0, a_1) = \text{НОД}(a_1, a_2) = \ldots = \text{НОД}(a_n, 0) = a_n .$$

Затраты алгоритма будем определять числом необходимых делений с остатком $C_E(a_0, a_1)$.

Не трудно видеть, что $C_E(a_0, a_1) \leq a_1$, т.к. после первого деления остаток остаётся меньше a_1, а при каждом следующем делении остаток будет меньше предыдущего как минимум на 1. В связи с этим в качестве входа можно взять только второе меньшее число a_1, тогда $T_E(a_1) \leq a_1$.

Один шаг алгоритма Евклида переводит пару a_{i-1}, a_i в пару a_i, a_{i+1}. Для оценки количества шагов алгоритма Евклида, которое совпадает с количеством необходимых делений с остатком, хорошо было бы найти такую функцию $\lambda(a_{i-1}, a_i)$, для которой выполнялись бы следующие условия:

1. $\lambda(a_{i-1}, a_i) \geq 0$;
2. $C_E(a_{i-1}, a_i) \leq \lambda(a_{i-1}, a_i)$;
3. на каждом шаге алгоритма Евклида она уменьшается хотя бы на 1.

Рассмотрим функцию $\lambda(k, l) = \nu(k) + \nu(l) - 2$, где $\nu(x)$ соответствует битовой длине x. Очевидно, что она удовлетворяет первым двум сформулированным условиям. Убедимся в справедливости третьего. Справедливо следующее

Утверждение. Пусть $k, l \in \mathbf{N}$, $k > l$, r — остаток от деления $\frac{k}{l}$. Тогда

1. $\nu(k) > \nu(r)$;
2. $\lambda(k, l) > \lambda(l, r)$.

Доказательство:

1. $k = ql + r$, $q \geq 1 \Rightarrow k \geq l + r > 2r \Rightarrow \nu(k) > \nu(l)$;

2. $\nu(k) > \nu(r) \Rightarrow \nu(k)+\nu(l)-2 > \nu(l)+\nu(r)-2 \Leftrightarrow \lambda(k,l) > \lambda(l,r)$, что и требовалось доказать.

$$C_E(a_0,a_1) \le \lambda(a_0,a_1) = \nu(a_0)+\nu(a_1)-2$$

$$C_E(a_0,a_1) = C_E(a_1,a_2)+1 \le \nu(a_1)+\underbrace{\nu(a_2)}_{\le \nu(a_1)}-1 \le 2\nu(a_1)-1$$

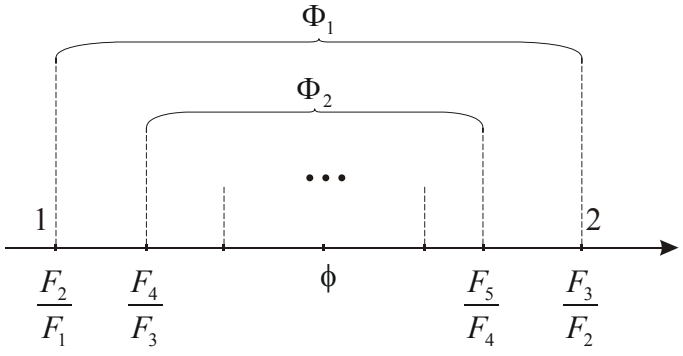
$$T_E(a_1) = \max_{a_0 \ge a_1} C_E(a_0,a_1) \le 2\nu(a_1)-1 = 2\lfloor \log_2 a_1 \rfloor + 1$$

Итак, для сложности алгоритма Евклида была получена оценка: $T_E(a_1) = O(\log a_1)$. Возникает вопрос: можем ли мы улучшить эту оценку? Ответ: нет, эта оценка точная. Для доказательства точности оценки построим последовательность входов $\left(a_0^{(1)}, a_1^{(1)}\right)$, $\left(a_0^{(2)}, a_1^{(2)}\right)$, $\left(a_0^{(3)}, a_1^{(3)}\right)$, … такую, что для неё будет выполняться $C_E(a_0^{(n)}, a_1^{(n)}) = \Omega\left(\log a_1^{(n)}\right)$ при $n \to \infty$. Для наших целей хорошо подойдут числа Фибоначчи: $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$, т.е. последовательность 0, 1, 1, 2, 3, 5, 8, … Применяя алгоритм Евклида к паре (F_{n+1}, F_n) получим ровно *n* делений: $C_E(F_{n+1}, F_n) = n$. Используем формулу Бене:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - (-\phi)^{-n}\right), \text{ где } \phi = \frac{1+\sqrt{5}}{2} = 1{,}61803... \text{ («золотое сечение»).}$$

Откуда получаем, что $\phi^n = \sqrt{5}F_n(1+o(1))$ при $n \to \infty \Rightarrow n = \log_\phi F_n + O(1)$. Тогда для затрат алгоритма получаем: $C_E(F_{n+1}, F_n) = n = \log_\phi F_n + O(1) = \Omega(\log F_n)$. Следовательно, оценка $T_E(a_1) = O(\log a_1)$ точна. Более того, можно показать, что $T_E(a_1) = \Theta(\log a_1)$. Для этого докажем, что $T_E(a_1) > \frac{1}{4}\log_\phi a_1 + \gamma$.

Рассмотрим систему вложенных сегментов $\Phi_1 \supset \Phi_2 \supset \Phi_3 \supset \ldots$, где $\Phi_n = \left[\frac{F_{2n}}{F_{2n-1}}, \frac{F_{2n+1}}{F_{2n}}\right]$.



Для чисел Фибоначчи справедливо равенство:

$$F_n^2 - F_{n-1}F_{n+1} = (-1)^n, \; n = 1,2,...$$

которое легко устанавливается по индукции (см. задачу 24). Используя это равенство, для длины сегмента Φ_n получаем выражение:

$$|\Phi_n| = \left|\frac{F_{2n+1}}{F_{2n}} - \frac{F_{2n}}{F_{2n-1}}\right| = \left|\frac{F_{2n+1}F_{2n-1} - F_{2n}^2}{F_{2n}F_{2n-1}}\right| = \frac{1}{F_{2n}F_{2n-1}}$$

Так как $|\Phi_n| \to 0$ при $n \to \infty$, $a_1 > 1 \Rightarrow \exists N : \forall n \le N$ выполняется $\frac{1}{a_1} \le |\Phi_n| = \frac{1}{F_{2n}F_{2n-1}}$, а для $N+1$ неравенство уже не выполняется, т.е. $\frac{1}{a_1} > |\Phi_{N+1}| = \frac{1}{F_{2N+1}F_{2N+2}} \Leftrightarrow a_1 < F_{2N+1}F_{2N+2}$.

Используем формулу Бене и получим:

$$a_1 < F_{2N+1}F_{2N+2} = \frac{1}{\sqrt{5}}\left(\phi^{2N+1} - (-\phi)^{-2N-1}\right) \cdot \frac{1}{\sqrt{5}}\left(\phi^{2N+2} - (-\phi)^{-2N-2}\right) < c\phi^{4N}$$

$\Rightarrow N > \frac{1}{4}\log_\phi a_1 - \log_\phi c$, что доказывает оценку $T_E(a_1) = \Theta(\log a_1)$. Для сложности в среднем можно получить оценку $\overline{T}_E(a_1) = \frac{12\ln 2}{\pi^2}\ln a_1 + O(1)$.

Расширенный алгоритм Евклида (EE).

Можно показать, что $\exists s,t \in \mathbf{Z} : sa_0 + ta_1 = \text{НОД}(a_0, a_1)$. В частности, если a_0 и a_1 взаимно простые, то $\exists s,t \in \mathbf{Z} : sa_0 + ta_1 = 1$.

Пусть в алгоритме Евклида уже найдены $a_0, a_1, ..., a_i$, $1 \le i \le n-1$ и пусть найдены s_0, t_0; s_1, t_1; … ; s_i, t_i такие, что $s_0a_0 + t_0a_1 = a_0$; $s_1a_0 + t_1a_1 = a_1$; … ; $s_{i-1}a_0 + t_{i-1}a_1 = a_{i-1}$; $s_ia_0 + t_ia_1 = a_i$.
$a_{i-1} = q_ia_i + a_{i+1} \Rightarrow a_{i+1} = a_{i-1} - q_ia_i = s_{i-1}a_0 + t_{i-1}a_1 - q_i(s_ia_0 + t_ia_1) = \underbrace{(s_{i-1} - q_is_i)}_{s_{i+1}}a_0 + \underbrace{(t_{i-1} - q_it_i)}_{t_{i+1}}a_1$, т.е.

$s_{i+1} = s_{i-1} - q_is_i$; $t_{i+1} = t_{i-1} - q_it_i$, при этом $s_0 = 1$, $t_0 = 0$, $s_1 = 0$, $t_1 = 1$.

Пример. $a_0 = 39; a_1 = 15$

Остатки, получаемые алгоритмом Евклида: 9, 6, 3, 0.

s, *t*: 1,-2; -1,-3; 2,-5; $\Rightarrow 2 \cdot 39 + (-5) \cdot 15 = 3 = \text{НОД}(39,15)$.

Алгоритм Евклида, в котором дополнительно определяются числа t_i, s_i буден называть расширенным алгоритмом Евклида (EE). Не трудно видеть, что для его сложности справедлива оценка: ё $T_{EE}(a_1) < 6\log_2 a_1 + 3$, т.е. всё утроилось.

Можно также найти числа $s_{n+1}, t_{n+1} : s_{n+1}a_0 + t_{n+1}a_1 = 0$ (для примера, рассмотренного выше, $s_{n+1} = s_5 = -5; t_5 = 13$).

Для чисел s_i, t_i можно доказать следующий факт: $|s_1| < |s_2| < ... < |s_{n+1}|$, $|t_1| < |t_2| < ... < |t_{n+1}|$.
Кроме того, $|s_{n+1}| = \frac{a_1}{\text{НОД}(a_0, a_1)}$, $|t_{n+1}| = \frac{a_0}{\text{НОД}(a_0, a_1)}$, тогда $|s_k| < a_1, |t_k| < a_0$, $k = 1,2,...,n$.

Бинарный поиск (BS = binary search).

Есть массив $x_1, ..., x_n$, элементы которого упорядочены по возрастанию. И есть ещё один элемент *y*, для которого нужно найти место в этом массиве такое, что после вставки элемента *y* полученный массив был упорядоченным. Существуют следующие варианты для вставки *y*:

$$\underset{1}{y} \le x_1, \quad x_1 < \underset{2}{y} \le x_2, \quad x_2 < \underset{3}{y} \le x_3, \quad \ldots \quad x_{n-1} < \underset{n}{y} \le x_n, \quad x_n \underset{n+1}{<} y$$

Задачей алгоритма является выдача числа от 1 до $n+1$, соответствующего варианту правильного расположения *y*.

На псевдокоде алгоритм выглядит следующим образом:

```
p:=1; q:=n-1;
while p<q do
  r:=⌊(p+q)/2⌋;
  if x_r<y then p:=r+1 else q:=r fi
od
```

Не трудно видеть, что каждый шаг алгоритма переводит рассматриваемую задачу для массива длины k к задаче для массива длинны $\left\lfloor \frac{k}{2} \right\rfloor$ или $\left\lfloor \frac{k}{2} \right\rfloor - 1$. Введём функцию $\lambda(k) = \nu(k)$, тогда на каждом шаге алгоритма $\lambda(k)$ будет убывать хотя бы на 1. Если $y \le x_1$, то $\lambda(n) = \lfloor \log_2 n \rfloor + 1 = T_{BS}(n) = \lceil \log_2 (n+1) \rceil$.

Сортировка бинарными вставками.

Рассмотрим алгоритм сортировки массива, основанный на алгоритме бинарного поиска. Алгоритм заключается в следующем: создается новый массив, который формируется из элементов исходного, каждый новый элемент занимает место, определяемое алгоритмом бинарного поиска. Тем сам на *i*-м шаге уже имеется упорядоченный массив из $i-1$ элемента, в который вставляется *i*-й элемент исходного массива.

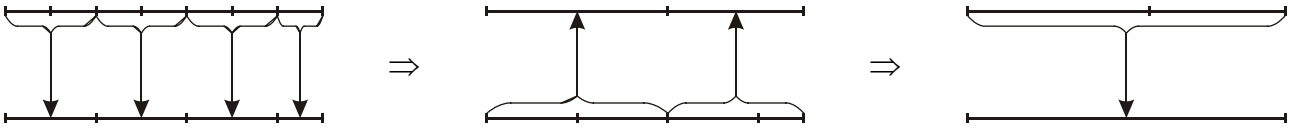
Для сложности в худшем случае имеем: $T_B(n) = \sum_{l=0}^{n-1} \lceil \log_2 (l+1) \rceil = \sum_{k=1}^{n} \lceil \log_2 k \rceil = \sum_{k=1}^{n} \log_2 n + O(1)$.

Из курса математического анализа известна формула Стирлинга: $n! = n^n e^{-n} \sqrt{2\pi n}\left(1 + o(1)\right) \Rightarrow$

$\sum_{k=1}^{n} \log_2 n = \log_2 n! = n \log_2 n + O(n) \Rightarrow T_B(n) = n \log_2 n + O(n)$.

Сортировка фон Неймана (vN).

Имеется массив $x_1, ..., x_n$, который требуется упорядочить. Пусть на каком-то шаге алгоритма массив разбивается на некоторое число упорядоченных сегментов. Тогда на следующем шаге сегменты объединяются парами и элементы внутри новых сегментов упорядочиваются. Далее опять происходит слияние сегментов по два и т.д. Причём для слияния сегментов используется вспомогательный массив такой же длины, что и исходный. Схематично действие алгоритма можно изобразить следующим образом:



При слиянии сегментов используется информация о том, что каждый из сливаемых сегментов сам по себе уже упорядочен, поэтому для слияния и одновременного упорядочивания двух сегментов длины *k* необходимо менее $2k$ операций сравнения, т.к. новый сегмент длины $2k$ формируется в новом массиве (новый сегмент формируется поэлементно из меньшего из наименьших нерассмотренных элементов исходных сегментов). На первом шаге исходный массив делится на сегменты длины 2 (за исключением, быть может, одного сегмента длины 1 в случае, если исходный массив имеет нечётное количество элементов), на втором шаге — длины 4 (если не было парного какому-то сегменту, то ещё не более одного сегмента длины 1, 2 или 3) и т.д. Таким образом, число этапов (перебросок) в сортировке фон Неймана будет равно $\lceil \log_2 n \rceil$. Т.к. число присваиваний на каждом этапе

равно n, то общее число присваиваний, необходимых алгоритму, будет равно $n\lceil \log_2 n \rceil$. Для числа сравнений получаем следующую оценку: $T_{vN}(n) < n\lceil \log_2 n \rceil$.

Завершимость алгоритмов.

Рассмотрим следующий пример.

Пример. (блуждание частицы по целым числам) В начале координат имеется частица, которая может двигаться на единицу влево или вправо с одинаковой вероятностью. Оказывается, что $\forall m \in \mathbf{Z}$ частица рано или поздно попадёт в точку m, причём с вероятностью 1. Но среднее необходимое для этого время (даже если $m = 1$) равно ∞.

Алгоритм кратных карт.

Имеется колода карточек, на каждой из которых с одной стороны пишутся вопросы, с другой — ответы. Все вопросы разной сложности. Обучаемый перебирая по очереди все карточки должен отвечать на вопросы, если ответ неверный, то можно посмотреть ответ на обороте. Если обучаемый отвечает на вопрос карточки, то она изымается, если не отвечает, то карточка возвращается в колоду, плюс добавляется еще одна такая же.

Рассмотрим стратегию «двоечника», когда обучаемый всегда отвечает неправильно.

Пусть все вопросы в колоде имеют кратность >1 кроме одной. Если m — суммарная кратность карт, то вероятность вытащить первый вопрос (вопрос кратности 1) на первом шаге равна $\frac{1}{m}$. Вероятность вытащить первый вопрос на i-м шаге равна:

$$\frac{m-1}{m} \cdot \frac{m}{m+1} \cdot \ldots \cdot \frac{m+i-2}{m+i-1} \cdot \frac{1}{m+i} = \frac{m-1}{(m+i-1)(m+i)}$$

Тогда вероятность того, что рано или поздно вытащат первый вопрос, будет равна:

$$P = \frac{1}{m} + (m-1)\sum_{i=1}^{\infty} \frac{1}{(m+i-1)(m+i)}$$

Покажем, что полученный ряд сходится:

$$\frac{1}{(m+i-1)(m+i)} = \frac{1}{m+i-1} - \frac{1}{m+i} \Rightarrow S_n = \sum_{i=1}^{n} \frac{1}{(m+i-1)(m+i)} = \frac{1}{m} - \frac{1}{m+n} \xrightarrow{n\to\infty} \frac{1}{m}$$

$$\Rightarrow P = \frac{1}{m} + (m-1)\frac{1}{m} = 1$$

Что соответствует тому, что первый вопрос рано или поздно будет вытащен с вероятностью 1. Посмотрим, какое время для этого потребуется:

$$T = (m-1)\sum_{i=1}^{\infty} \frac{i}{(m+i-1)(m+i)}$$

Не трудно видеть, что полученный ряд расходится. Это соответствует тому, что для вытаскивания первого вопроса потребуется бесконечное время.

Пусть теперь $u \geq 2$ карточки имеют кратность 1. Тогда для вытаскивания вопроса с кратностью 1 потребуется времени

$$T = \sum_{i=1}^{\infty} \frac{i+u+1}{(i+m)(i+m+1)\ldots(i+m+u)}$$

Полученный ряд, как не трудно видеть, сходится, а значит, мы увидим вопрос с кратностью 1. Далее по индукции легко установить, что пока $u \neq 1$ мы увидим все вопросы.

Допустим, что программа содержит какие-то цифры и её нужно исследовать на сложность. Рассмотрим пример.

Пример. Транспонирование матрицы:

```
for i=1 to n-1 do
  for j=i+1 to n do a_ij ↔ a_ji od
od
```

Если хотим посчитать число обменов, то оно равно $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n^2-n}{2}$.

Рассмотрим ещё один алгоритм:

```
l:=0;
for i=1 to ⌊√(n^3+1)⌋ (φ(n)) do
  k:=i;
  while k>1 do l:=l+k; k:=⌊k/3⌋ od
od
```

Требуется оценить сложность приведённого алгоритма.

Очевидно, что внутренний цикл `while` можно оценить через $\Theta(\log i)$, т.к. количество его итераций совпадает с $\log_3 i$. Поэтому для сложности алгоритма в целом справедлива оценка: $T(n) = \Theta\left(\sum_{i=1}^{\varphi(n)} \log i\right) = \Theta\big(\log \varphi(n)!\big) = \Theta\big(\varphi(n)\log\varphi(n)\big)$ и с учётом $\varphi(n) = \lfloor\sqrt{n^3+1}\rfloor$ получаем:

$$T(n) = \Theta\Big(\lfloor\sqrt{n^3+1}\rfloor \log\sqrt{n^3+1}\Big) = \Theta\big(n^{3/2}\log n\big).$$

До сих пор в качестве размера входа мы использовали целые числа, но определение размера входа, вообще говоря, допускает использование и рациональных, и иррациональных чисел.

Пример. В алгоритме Евклида в качестве входа возьмём рациональное число $r = \frac{a_0}{a_1}$. Ранее было получено, что $T_E(n) = \Theta\big(\log a_1\big)$. Рассмотрим следующую последовательность размеров входа: $r_n = \frac{F_{n+1}}{F_n}$, где F_i — числа Фибоначчи. Из формулы Бене следует, что $\frac{F_{n+1}}{F_n} \to \phi = \frac{1+\sqrt{5}}{2}$, при этом сложность с ростом n становится всё больше и больше. Если и удастся установить оценку $T'_E(r) < f(r)$, то, очевидно, что $f(r)$ будет разрывная. Более того, если берём интервал (a, b), то для любого наперёд заданного N, найдётся рациональное число r такое, что $r = \frac{a_0}{a_1}$ и алгоритму Евклида для чисел (a_0, a_1) потребуется N шагов. Для целых размеров входа таких фокусов нет.

Следует отметить, что существуют алгоритмы, для которых целесообразно выбрать в качестве размера входа нецелое число. Например, алгоритм поиска решения уравнения на отрезке методом деления пополам. Для поиска решения с точностью ε потребуется

$\log_2 \frac{1}{\varepsilon} + O(1)$ операций, и в качестве размера входа здесь целесообразно взять величину $\frac{1}{\varepsilon}$. Для метода касательных сложность составит $O\left(\log\log\frac{1}{\varepsilon}\right)$.

Нижняя граница сложности алгоритмов некоторых классов. Оптимальные алгоритмы

Пример. Рассмотрим задачу поиска максимального элемента массива. Очевидно, что решить эту задачу менее чем за $(n-1)$ сравнение нельзя.

Определение. Пусть $\mathscr{A}$ — класс алгоритмов, решающих некоторую задачу. n — размер входа. Функция $f(n)$ называется нижней границей сложности принадлежащих $\mathscr{A}$ алгоритмов, если $\forall A \in \mathscr{A} \Rightarrow T_A(n) \geq f(n)\ \forall n$.

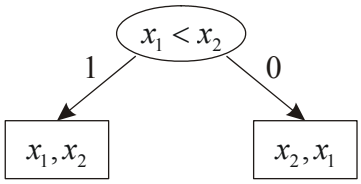
Если сложность зависит от нескольких параметров размеров входа $n_1,...,n_k$, то нижняя граница — функция нескольких переменных.

Для рассмотренного выше примера (поиск максимума) $f(n) = n-1$.

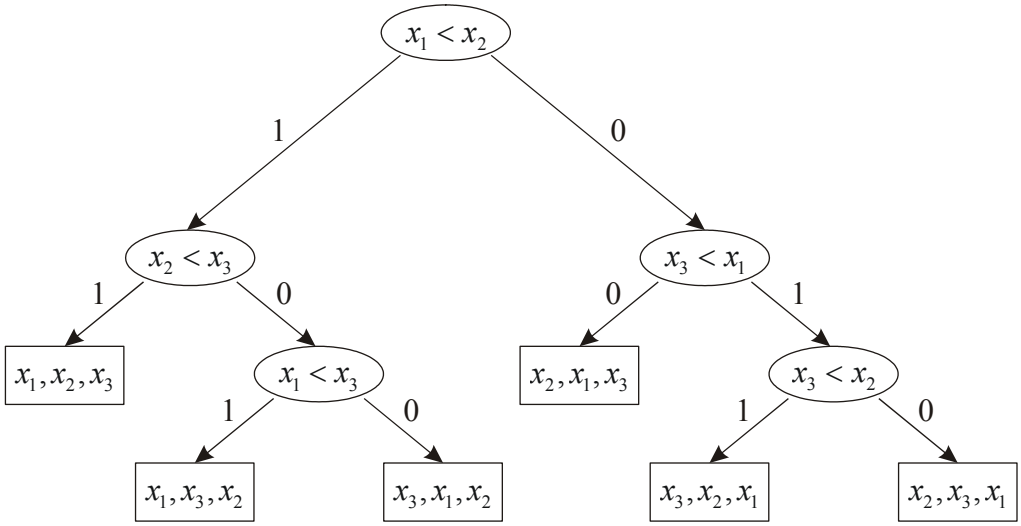
Какой именно класс $\mathscr{A}$ в определении нижней границы имеет большое значение. Например, если имеется функция поиска максимума и сложность алгоритма измеряется в обращениях к этой функции, то и нижняя граница будет иной.

Рассмотрим алгоритмы сортировки. Такие алгоритмы можно изобразить в виде дерева:

$n = 2$:



$n = 3$:



Если высота двоичного дерева h, то число листьев на нём $\leq 2^h$ (т.к. число листьев 2^h достигается на полном двоичном дереве, т.е. дереве, у которого заполнены все уровни). Очевидно, что h в данном случае соответствует сложности алгоритма. В дереве сортировки $n!$ листьев, откуда получаем нижнюю границу сложности для алгоритмов сортировки: $2^{T(n)} \geq n! \Rightarrow T(n) \geq \lceil \log_2 n! \rceil$.

Вспомним алгоритм бинарного поиска: есть массив $x_1 < x_2 < ... < x_n$ и некоторое заданное число y, для которого требуется найти место в массиве, чтобы сохранилась упорядоченность. Для вставки y существует $(n+1)$ возможность: $y \leq x_1$; $x_1 < y \leq x_2$; …; $x_n < y$. Задача алгоритма заключается в определении номера возможности. Опять же можно строить дерево,

но в данном случае листьев на дереве будет $(n+1)$ листьев, и оценка для сложности примет вид: $T(n) \geq \lceil \log_2(n+1) \rceil$.

Вспомним алгоритм вычисления степени a^n с помощью умножений. Каждый этап алгоритма может быть охарактеризован тем, какие степени мы имеем на текущий момент: $a^{i_1},...,a^{i_m}$ и более того, какой максимальный порядок встречается в этом наборе. На каждом шаге максимальный порядок может увеличиться не более чем в 2 раза. Определим на наборе $a^{i_1},...,a^{i_m}$ функцию

$$\lambda = \underbrace{\log_2 n}_{const} - \log_2 i$$

где i — максимум из $i_1,...,i_m$. Тогда функция на каждом шаге может уменьшить своё значение не более чем на 1. Очевидно, что в этих обозначениях алгоритм заканчивает работу, как только λ станет ≤ 0. Отсюда получаем нижнюю границу сложности алгоритма: $\log_2 n$.

Определение. Если в $\mathscr{A}$ $\exists A : T_A(n)$ является нижней границей сложности для $\mathscr{A}$, то A является оптимальным в этом классе и $\forall B \in \mathscr{A} \Rightarrow T_B(n) \geq T_A(n)$.

Найти оптимальный алгоритм значит найти нижнюю границу для алгоритмов данного класса и предъявить алгоритм с такой сложностью.

Рассмотрим задачу: имеется массив, требуется найти и минимальный, и максимальный элемент. Покажем, что нижняя граница сложности составляет $f(n) = \left\lceil \frac{3n}{2} \right\rceil - 2$ и приведём оптимальный алгоритм.

Каждый этап произвольного алгоритма *V*, решающего эту задачу, характеризуется следующей четвёркой множеств элементов массива: (A,B,C,D), где A — множество элементов, не участвовавших в сравнениях; B — множество элементов, которые во всех сравнениях оказывались большими; C — множество элементов, которые во всех сравнениях оказывались меньшими; D — множество элементов, которые в некоторых сравнениях были больше, а в других — меньше. Начальная ситуация в ведённых обозначениях выглядит как $(n, 0, 0, 0)$, конечная — $(0, 1, 1, n-2)$, т.е. множества B и C состоят из одного элемента, которые и будут максимальным и минимальным соответственно.

Рассмотрим функцию $\lambda(a,b,c) = \frac{3}{2}a + b + c - 2$, где *a*, *b* и *c* соответствуют числу элементов в соответствующих множествах *A*, *B* и *C*. Возможны следующие сравнения и соответствующие изменения функции λ:

сравнение	(A,B,C,D)	изменение λ
AA	$(a-2,b+1,c+1,d)$	-1
AB	$(a-1,b,c+1,d) \mid (a-1,b,c,d+1)$	$-\frac{1}{2} \mid -\frac{3}{2}$
AC	$(a-1,b+1,c,d) \mid (a-1,b,c,d+1)$	$-\frac{1}{2} \mid -\frac{3}{2}$
AD	$(a-1,b+1,c,d) \mid (a-1,b,c+1,d)$	$-\frac{1}{2} \mid -\frac{1}{2}$
BB	$(a,b-1,c,d+1)$	-1
BC	$(a,b-1,c-1,d+2) \mid (a,b,c,d)$	$-2 \mid 0$
BD	$(a,b-1,c,d+1) \mid (a,b,c,d)$	$-1 \mid 0$

CC	$(a,b,c-1,d+1)$	-1
CD	$(a,b,c-1,d+1) \mid (a,b,c,d)$	$-1 \mid 0$
DD	(a,b,c,d)	0

Здесь *AA* означает сравнение элемента из *A* с элементом из *A*; *AB* — сравнение элемента из *A* с элементом из *B* и т.д.

В худшем случае шаг от шага λ уменьшается не более, чем на 1, откуда получаем нижнюю границу сложности $f(n) = \left\lceil \frac{3n}{2} \right\rceil - 2$.

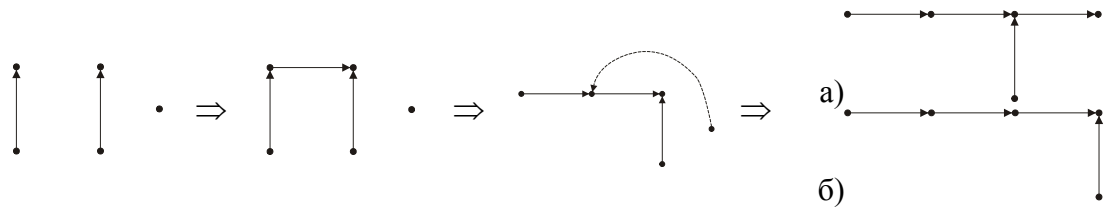
Приведём оптимальный алгоритм, решающий поставленную задачу: изначально имеется массив из *n* элементов $x_1,...,x_n$. Разобьём исходный массив на пары x_1, x_2; x_3, x_4; … В каждой паре найдём минимум и максимум за одно сравнение. Тогда минимальные элементы из пар образуют множество $m_1, m_2,...$, а максимальные — $M_1, M_2,...$ Среди $m_1, m_2,...$ найдём минимальный, среди $M_1, M_2,...$ — максимальный. Если на первом шаге был непарный элемент (*n* — нечётное), то на него потребуется ещё два сравнения с найденными минимумом и максимумом. В итоге на каждую пару тратится 3 сравнения.

Надо добавить, что оптимальный алгоритм не обязан существовать. Например, в рассматриваемом классе алгоритмов имеется всего 2 алгоритма, один из которых быстро работает для чётных *n*, второй — для нечётных.

Оптимальные алгоритмы сортировки.

Для построения оптимального алгоритма сортировки можно, конечно, построить все возможные бинарные деревья с *n*! листьями, выбрать дерево с наименьшей высотой и предъявить.

Считалось, что алгоритм бинарными вставками является оптимальный. Но для $n = 5$ ему понадобиться 8 сравнений, в то время как полученная выше нижняя граница даёт $\lceil \log_2 n! \rceil = \lceil \log_2 5! \rceil = 7$.



сравнений: 2 3 3 + 2 5

Очевидно, что для полного упорядочения потребуется ещё не более 2-х сравнений, в итоге получим 7, т.е. алгоритм для сортировки массива длины 5, сложность которого равна 7, существует. Интересно, что уже для $n = 12$ оптимальный алгоритм потребует $\lceil \log_2 12! \rceil + 1$ сравнений.

Бинарный алгоритм возведения в степень не является оптимальным. Оптимальным является алгоритм, называемый схемой Горнера, который по заданным числам $a_0, a_1,..., a_n, x$ вычисляет значение выражения $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ и требует *n* умножений.

Нужно отметить, что оптимальных алгоритмов человечество знает не много. Поэтому понятие оптимальности хорошо было бы расширить.

Определение. Асимптотической нижней границей сложности алгоритмов некоторого класса $\mathscr{A}$ называется такая функция $f(n)$, что $\forall A \in \mathscr{A} \Rightarrow T_A(n) = \Omega(f(n))$.

Определение. Алгоритм называется асимптотически оптимальным (оптимальным по порядку сложности), если $T_A(n)$ является асимптотической нижней границей сложности и $\forall B \in \mathscr{A} \Rightarrow T_B(n) = \Omega\big(T_A(n)\big)$.

Оптимальных по порядку сложности алгоритмов в определённом класс может быть несколько, но оптимальная асимптотическая сложность определена однозначно.

Пусть A и B — оптимальные алгоритмы по порядку сложности. Тогда справедливо $T_B(n) = \Omega\big(T_A(n)\big)$, что эквивалентно тому, что $T_A(n) = O\big(T_B(n)\big)$.

Утверждение. $T_A(n) = \Theta\big(T_B(n)\big)$.

Доказательство:

$$\left.\begin{aligned} T_B(n) &= \Omega\big(T_A(n)\big) \\ T_A(n) &= \Omega\big(T_B(n)\big) \end{aligned}\right\} \Rightarrow T_A(n) = \Theta\big(T_B(n)\big).$$

Утверждение. Если $f(n)$ является асимптотической нижней границей сложности алгоритмов класса $\mathscr{A}$, $A \in \mathscr{A}$ и $T_A(n) = O\big(f(n)\big)$, то A — оптимальный по порядку сложности и более того $T_A(n) = \Theta\big(f(n)\big)$.

Доказательство: $f(n)$ — асимптотическая нижняя граница сложности $\Rightarrow T_A(n) = \Omega\big(f(n)\big)$, но $T_A(n) = O\big(f(n)\big)$ по условию $\Rightarrow T_A(n) = \Theta\big(f(n)\big) \Rightarrow A$ — оптимальный по порядку сложности.

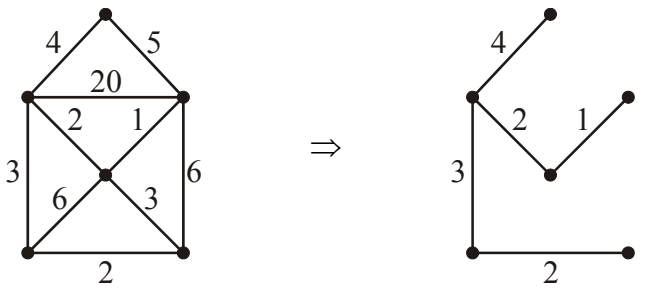
Алгоритм возведения в степень (RS), рассмотренный ранее, не является оптимальным, но является асимптотически оптимальным, т.к. ранее было показано, что $T_{RS}(n) = \Omega\big(\log n\big)$ и $T_{RS}(n) < 2\log_2 n$, а следовательно, он является асимптотически оптимальным.

Если мы обнаруживаем, что для некоторого алгоритма сортировки его сложность может быть оценена через $O(\log n!)$ или, используя формулу Стирлинга, $O(n \log n)$, то он является оптимальным по порядку сложности. Поэтому сортировка фон Неймана и сортировка бинарными вставками являются оптимальными по порядку сложности. Позднее в этот список добавим ещё алгоритм сортировки слияниями. А равны ли $T_{vN}(n)$ и $T_B(n)$? Ответ: нет, это не одно и то же, т.к. хотя бы для $n = 5$ имеем $T_{vN}(n) = 9$, $T_B(n) = 8$.

Для алгоритма построения Эйлерова цикла (частный случай вояжа) была получена оценка $O\big(|E|\big)$, но очевидно, что меньше, чем за $|E|$ шагов, его построить нельзя. Поэтому алгоритм является оптимальным по порядку сложности.

Рассмотрим взвешенный связный граф без кратных рёбер. Остовным деревом будем называть подграф, который: 1) охватывает все вершины; 2) ребрами его являются рёбра исходного графа; 3) сумма весов ребер минимальна.

Пример.



Известен алгоритм Прима построения остовного дерева со сложностью $O(|E|+|V|\log|V|)$. Но если в качестве размера входа рассматривать только $|V|$, то алгоритм будет оптимальным по порядку сложности.

Среди всех графов имеющих $|V|$ вершин наибольшее количество рёбер имеет полный граф: $|E|=\frac{|V|(|V|-1)}{2}$, следовательно, алгоритм Прима допускает оценку $O(|V|^2)$, но с другой стороны существуют полные графы, и следовательно, $\Omega(|V|^2)$. Поэтому он оптимален и допускает оценку $\Theta(|V|^2)$.

Следует отметить, что не всегда существует оптимальный алгоритм по порядку сложности. Например, в классе всего два алгоритма, один из которых быстро работает для чётных n, а второй — для нечётных. Но, тем не менее, сложности алгоритмов, оптимальных по порядку сложности, являются величинами одного порядка.

Пока рассматривались оптимальные алгоритмы по порядку сложности, и сложность была сложностью в худшем случае. Можно также рассматривать по сложности в среднем.

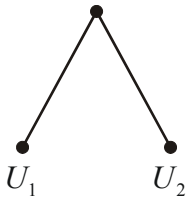
Утверждение. $\log_2 n!$ является нижней границей сложности в среднем для алгоритмов сортировки.

Доказательству утверждения предпошлём лемму.

Лемма. В любом двоичном дереве с m листьями сумма высот всех листьев $\geq m\log_2 m$.

Доказательство леммы проведём от противного: пусть сделанное утверждение не верно. Тогда из всех деревьев, для которых это не так, возьмём дерево U с минимальной суммой высот. Это дерево не может состоять из одной вершины, т.к. $0 \geq 1\cdot\log_2 1 = 0$. Следовательно, из корня этого дерева исходит одно или два ребра. Пусть исходит одно ребро:

Но этого быть не может, т.к. для U' утверждение должно быть не выполнено, иначе добавление ребра лишь увеличит сумму высот и получим, что для U утверждение выполнено, но это не так. А если для U' утверждение не выполнено, тогда U не является деревом с минимальной суммой высот (сумма высот у U' меньше, чем у U). Поэтому из корня может исходить только 2 ребра:



Обозначим сумму высот всех листьев дерева через $H(U)$, тогда $H(U)=H(U_1)+H(U_2)+m$, т.к. к каждой высоте деревьев U_1 и U_2 прибавится единица, а сумма листьев в деревьях U_1 и U_2 равно m (в дереве U m листьев). Обозначим через m_1, m_2 количества листьев в деревьях U_1 и U_2 соответственно. Для деревьев U_1 и U_2 утверждение должно быть выполнено, т.к. U — дерево с минимальной суммой высот, для которого утверждение не выполнено. Тогда $H(U_1)\geq m_1\log_2 m_1$ и $H(U_2)\geq m_2\log_2 m_2$.

$$H(U) \geq m_1 \log_2 m_1 + m_2 \log_2 m_2 + m$$

С учётом $m = m_1 + m_2$ получаем:

$$H(U) \geq m_1 \log_2 m_1 + (m - m_1)\log_2 (m - m_1) + m,\ 1 \leq m_1 \leq m - 1$$

Рассмотрим функцию $f(x) = x \log_2 x + (m - x)\log_2 (m - x)$. Найдём минимум этой функции на отрезке $[1, m-1]$. Проведя стандартный анализ с привлечением аппарата производных, получаем, что минимум достигается в точке $x = \frac{m}{2}$ $\Rightarrow$

$$H(U) \geq \frac{m}{2}\log_2 \frac{m}{2} + \frac{m}{2}\log_2 \frac{m}{2} + m = m \log_2 m,$$

что противоречит сделанному предположению. Лемма доказана.

Доказательство утверждения: рассматривается пространство перестановок Π_n. Дерево сортировки имеет $n!$ листьев, и нам нужно посчитать математическое ожидание суммарных затрат. Суммарные затраты совпадают с суммой высот всех листьев и, согласно лемме, они $\geq n! \log_2 n!$. Чтобы посчитать математическое ожидание, достаточно суммарные затраты умножить на $\frac{1}{n!}$, откуда получаем нижнюю границу сложности в среднем: $\frac{1}{n!} \cdot n! \log_2 n! = \log_2 n!$.

Согласно этому утверждению, сортировка фон Неймана и сортировка бинарными вставками будут оптимальными по порядку сложности в среднем.

Ранее нам для определения числа шагов алгоритма часто помогал приём рассмотрения функции λ, которая в ходе алгоритма убывала. Для асимптотической сложности в среднем тоже могут применяться подобные методы.

Пусть на некотором вероятностном пространстве задана последовательность случайных величин $\xi_1, \xi_2, \ldots$ и числовая последовательность $h_1, h_2, \ldots$ такая, что выполнено неравенство: $\mathbf{E}(\xi_k | \xi_1, \ldots, \xi_{k-1}) \leq h_k$. Зафиксируем $q \geq 0$ и рассмотрим цело число $\tau = \min\left\{ n : \sum_{k=1}^{n} \xi_k \geq q \right\}$.

Тогда $\mathbf{E}\left(\sum_{i=1}^{\tau} n_k \right) \geq q$.

Этот факт позволяет доказывать, что число шагов алгоритма $\geq q$.

Для алгоритма поиска максимального и минимального элемента массива асимптотическая оценка снизу сложности в среднем равна:

$$f(n) = \begin{cases} \frac{3}{2}n - 2, \text{ если } n \text{ - чётное} \\ \frac{3}{2}n - 2 + \frac{1}{2n}, \text{ если } n \text{ - нечётное} \end{cases}$$

Битовая сложность

Когда мы занимались алгебраической сложностью, при её исследовании мы считали битовую длину операндов не важной, за исключением случаев, когда в качестве размера входа выбиралась битовая длина. В случае, когда алгоритмы предназначены для исполнения на вычислительных машинах, битовая длина операндов становится существенной.

При исследовании алгоритма на битовую сложность числа представляются строками («словами») бит (но не «битов») и операции над числами рассматриваются как операции над отдельными битами, которые в свою очередь считаются равнозатратными. В битовой модели (машине) все биты считаются равнодоступными (в отличие от машины Тьюринга, где ячейки ленты нельзя считать равнодоступными).

Мы бы хотели исследовать битовую сложность алгоритмов и интересуемся количеством битовых операций. Работа эта более кропотливая и детальная, поэтому ограничимся арифметическими операциями.

На первый взгляд кажется, что если мы хотим исследовать битовую сложность какого-то алгоритма, то нам нужно переписать его в битовых операциях. Но никто так не делает. Вместо этого полагают наличие соответствующих арифметических процедур (сложение, вычитание и т.д.).

Рассмотрим целую арифметику. Будем считать, что числа заданы в двоичном виде.

Итак, у арифметической операции 2 аргумента: a и b. Что в этом случае считать размером входа? Возможны следующие варианты: 1) размером входа можно считать сами числа a и b; 2) битовую длину операндов: $m_a = \nu(a)$; $m_b = \nu(b)$; 3) $m = \max\{m_a, m_b\}$; 4) $m_a + m_b$.

Операция сложения (Add).

Для затрат битового сложения справедлива оценка:

$$\min\{m_a, m_b\} \le C^T_{Add}(a, b) \le c\big(\max\{m_a, m_b\}+1\big)$$

где c — некоторая постоянная, не зависящая от a и b. Действительно, для сложения столбиком на каждом этапе, исходя их 3-х бит на входе (слагаемые и возможный перенос), нужно получить 2 бита (результат и перенос), откуда следует оценка сверху. Оценка снизу тоже понятна, т.к. мы должны поработать с каждым битом меньшего числа.

Получим асимптотику для сложности в следующем виде: $T^*_{Add}(m) = \Theta(m)$. Для этого достаточно доказать соответствующие оценки снизу и сверху. Оценка сверху следует из вышеприведённой оценки для затрат, а для оценки снизу можно взять два числа с одинаковой длинной: $m_a = m_b$. Тогда $\min\{m_a, m_b\} = \max\{m_a, m_b\}$, и в силу оценки для затрат получаем требуемую оценку снизу для сложности.

Для пространственной сложности получаем следующие оценки: в случае формирования результата на месте большего числа — $S^*_{Add}(m) = O(1)$; в случае формирования результата на новом месте — $S^*_{Add}(m) = m + O(1)$.

Алгоритмы умножения через сложение и деления через вычитание будем называть «наивными».

Пример.

> «Сверхнаивное» умножение. Для вычисления $a \cdot b$ вычисляются следующие суммы: $a + a$, $(a + a) + a$, …, $\underbrace{(a + \ldots + a)}_{b-1} + a$. В качестве размера входа возьмём величину

$m = \max\{m_a, m_b\}$. Рассмотрим случай, когда $m_a = m_b = m$, тогда $b > 2^{m-1}$ и общее число битовых операций допускает оценку $\Omega(2^m m)$. $ab < 2^{2m}$ и, следовательно, результат каждого сложения имеет не превосходящую $2m$ битовую длину, а значит число битовых операций на каждом шаге $\le c \cdot 2m$. Поскольку число шагов $< 2^m$, мы можем написать оценку $O(2^m m)$. В итоге поучаем оценку $T^*_{NM}(m) = \Theta(2^m m)$.

Посмотрим на сложность сложения в случае, когда в качестве размера входа выбираются 2 параметра m_a и m_b:

$$T^{**}_{Add}(m_a, m_b) = \Theta(\max\{m_a, m_b\}).$$

Заметим, что это не просто переписывание оценки через *m*. Покажем, откуда следует $T^{**}_{Add}(m_a, m_b) = \Omega(\max\{m_a, m_b\})$. Доказательство оценки Ω заключается в описание «неудобных» данных для алгоритма:

$$a = \overbrace{1\underbrace{1\ldots1}_{m_b}0\underbrace{1\ldots1}_{m_b}}^{m_a} \qquad b = \underbrace{10\ldots01}_{m_b}$$

Для так выбранных входных данных становиться очевидным то, что для их сложения придётся поработать со всеми битами более длинного числа, что доказывает оценку с Ω.

Для алгоритма «сверхнаивного» умножения из примера допустимы следующие оценки сложности: $\Omega(2^{m_b} m_a)$ и $O(2^{m_b}(m_a + m_b))$, но в то же время, оценка $\Theta(2^{\max\{m_a, m_b\}} \max\{m_a, m_b\})$ будет неверна.

Для алгоритма «наивного» умножения (NM) (обычное умножение столбиком) справедливы следующие оценки сложности: $T^*_{NM}(m) = \Theta(m^2)$, $T^{**}_{NM}(m_a, m_b) = \Theta(m_a m_b)$. Они легко следуют из определения: требуется суммировать m_b чисел $n_1, \ldots, n_{m_b}$, где n_i либо 0, либо $a \cdot 2^{i-1}$. Обозначим $s_i = n_1 + \ldots + n_i$, $i = 1, \ldots, m_b$. Несложно индукцией показывается, что $\nu(s_i) \le m_a + i$. Прибавляем $n_{i+1} \ne 0$. Последние *i* цифр этого числа равны нулю — мы их игнорируем: $s_{i+1} = s_i + n_{i+1}$.

Число битовых операций для преобразования s_i в s_{i+1} не превосходит $c(m_a + 1)$. Если все цифры числа *b* равны 1, то число битовых операций для вычисления произведения не меньше, чем $m_a m_b$. Откуда получаем оценку для сложности: $T^{**}_{NM}(m_a, m_b) = \Theta(m_a m_b)$.

Если будем рассматривать $m = \max\{m_a, m_b\}$ как размер входа, то затраты не превзойдут $c(m+1)m$, откуда следует оценка для сложности $O(m^2)$. Оценку снизу выведем из $T^{**}_{NM}(m_a, m_b)$, рассмотрев случай $m_a = m_b = m \Rightarrow$ затраты будут не меньше, чем $m_a m_b = m^2$. Откуда получаем $T^*_{NM}(m) = \Theta(m^2)$.

Для пространственной сложности будут верны оценки: $S^*_{NM}(m) = 2m + O(1)$, $S^{**}_{NM}(m_a, m_b) = m_a + m_b + O(1)$.

Иногда удобно иметь оценки сложности, выраженные в самих *a* и *b*. Возникает вопрос: можно ли в вышеприведённых рассуждениях заменить $m_a m_b$ на $\log a \log b$? Для перехода в оценках сверху от $\nu(c)$ к $\log_2 c$ удобно такое неравенство: $\lceil \log_2(c+1) \rceil < 2\log_2 c$, $c \ge 2$. Тем самым неравенства $m_a \le 2\log_2 a$, $m_b \le 2\log_2 b$ дают нам возможность написать следующую оценку:

$$C_{NM}^{T}(a,b) \le T_{NM}^{**}(m_a, m_b) \le cm_a m_b \le 4c \log_2 a \log_2 b \Rightarrow C_{NM}^{T}(a,b) = O(\log a \log b)$$

Функция затрат в этом случае будет соответствовать сложности (т.к. размер входа — a и b). Тем самым доказано, что $T_{NM}^{**}(a,b) = O(\log a \log b)$. Можно ли то же самое проделать для установления оценки Θ? Ответ: нет, т.к. уже для $a = 2^{k_1}$, $b = 2^{k_2}$ оценка будет линейной.

Рассмотрим задачу вычисления $n!$ следующим образом: $2 \cdot 3$; $(2 \cdot 3) \cdot 4$; … ; $(2 \cdot \ldots \cdot (n-1)) \cdot n$. Докажем, что затраты допускают оценку $O\left((n \log n)^2\right)$.

Очевидно, что затраты не превосходят $c \cdot f(n)$, где $f(n) = \log_2 2 \log_2 3 + \log_2 (2 \cdot 3) \log_2 4 + \ldots + \log_2 (2 \cdot 3 \cdot \ldots \cdot (n-1)) \log_2 n < \log_2 (2 \cdot 3 \cdot \ldots \cdot (n-1)) [\log_2 3 + \log_2 4 + \ldots + \log_2 n] < \log_2 n! \log_2 n!$.

Откуда следует оценка для сложности $O(\log^2 n!)$, что эквивалентно, согласно формуле Стирлинга, $O\left((n \log n)^2\right)$.

Пример. Пусть имеются числа $a_1, \ldots, a_n > 0$, которые перемножаются наивным способом: $a_1 a_2$, $(a_1 a_2) a_3$, … , $(a_1 \cdot \ldots \cdot a_{n-1}) a_n$. Обозначим суммарную битовую длину через $M = \sum_{i=1}^{n} \nu(a_i)$. Докажем, что битовая сложность такого алгоритма допускает оценку $O(M^2)$.

Проведём доказательство, предполагая, что $a_i \ne 1$ (если есть единицы, то оценка тем более верна). Очевидно, что затраты алгоритма не превзойдут $c \cdot F(a_1, \ldots, a_n)$, где

$$F(a_1, \ldots, a_n) = \log_2 a_1 \log_2 a_2 + \log_2 (a_1 a_2) \log_2 a_3 + \ldots + \log_2 (a_1 \cdot \ldots \cdot a_{n-1}) \log_2 a_n \le$$

$$\le \log_2 (a_1 a_2 \ldots a_{n-1}) [\log_2 a_2 + \ldots + \log_2 a_n] \le (\log_2 a_1 + \log_2 a_2 + \ldots + \log_2 a_n)^2 \le \{\log_2 a_i \le \nu(a_i)\} \le$$

$$\le (\nu(a_1) + \nu(a_2) + \ldots + \nu(a_n))^2 = M^2.$$

Деление с остатком.

Пусть даны два числа $a \ge b > 1$. Покажем, что для алгоритма «наивного» деления (ND) (деление столбиком) имеют место следующие оценки:

$$T_{ND}^{*}(m) = \Theta(m^2),\ T_{ND}^{**}(m_a, m_b) = \Theta((m_a - m_b + 1) m_b).$$

На каждое вычитание уходит $c(m_b + 1)$, количество вычитаний не превосходит $(m_a - m_b + 1)$, откуда получаем $T_{ND}^{**}(m_a, m_b) = O((m_a - m_b + 1) m_b)$ (единицей пренебречь нельзя, т.к. вполне возможно, что $m_a = m_b$, и выражение под знаком O превратится в нуль).

Что касается нижней оценки, то можно рассмотреть числа $a = 2^{m_a} - 1$ и $b = 2^{m_b - 1}$, для которых алгоритму потребуется $(m_a - m_b + 1) m_b$ операций, что доказывает оценку снизу. Это даёт возможность написать оценку через Θ.

Что касается оценки $T_{ND}^{*}(m) = O(m^2)$, то она следует из того, что $m^2 \ge m_a m_b \ge (m_a - m_b + 1) m_b$.

Для доказательства нижней оценки, рассмотрим $a = 2^m - 1$ и $b = 2^{\lfloor \frac{m}{2} \rfloor}$, которые являются самыми неудобными для алгоритма. Это позволяет написать нижнюю оценку, что завершает доказательство оценки $T_{ND}^{*}(m) = \Theta(m^2)$.

Когда a и b рассматриваются в качестве размера входа, то битовая сложность допускает оценку $O((\log a - \log b + a) \log b) \Rightarrow O(\log a \log b)$. Если брать только a, то $O(\log^2 a)$.

Пример. Пусть на входе имеется n и $k \geq 2$. Требуется перевести n из двоичной системы счисления к системе с основанием k. Покажем, что битовая сложность такого алгоритма допускает оценку $O(\log^2 n)$.

n в k-ичной записи формируется из остатков от деления чисел $q_0, \ldots, q_t$, $t = \lfloor \log_k n \rfloor + 1$ на k, где $q_0 = n$, $q_i = \left\lfloor \frac{q_{i-1}}{k} \right\rfloor$, $i = 1, \ldots, t$. Все $q_i \leq n$, и общие затраты на всех шагах допускают оценку $O(\underbrace{\log n \log k}_{\substack{\text{затраты на} \\ \text{одном шаге}}} \underbrace{\log_k n}_{\substack{\text{число} \\ \text{шагов } (t)}})$, $\log_2 k \log_k n = \log_2 n \Rightarrow O(\log^2 n)$.

Интересно, что если известен некоторый алгоритм умножения со сложностью $T_M(m)$, то по нему можно построить алгоритм деления со сложностью не хуже, чем $O(T_M(m))$.

Исследуем алгоритм Евклида на битовую сложность. На входе два числа: $a_0 \geq a_1 > 0$. Ранее было сказано, что величина a_0 не сильно влияет на алгебраическую сложность, т.к. после первого же деления с остатком получим число, меньшее a_1. Но в случае битовой сложности это не так. Если в качестве размера входа взять a_1, то сложность будет бесконечной. Чтобы получить информативную оценку битов можно в качестве размера входа взять a_0 или $m = \nu(a_0)$. Возьмём m, тогда можно показать, что для битовой сложности алгоритма Евклида имеет место оценка $O(m^2)$. Доказательство этого факта не очень сложное: очевидно, что затраты алгоритма не превзойдут величины

$$c \cdot \sum_{i=1}^{n} (\nu(a_{i-1}) - \nu(a_i) + 1)\nu(a_i).$$

Здесь под знаком суммы написана битовая сложность деления a_{i-1} на a_i, причём число шагов алгоритма Евклида равно n. Функция $\nu(a_i)$ с ростом i убывает, поэтому мы можем написать следующую оценку:

$$c \cdot \sum_{i=1}^{n} (\nu(a_{i-1}) - \nu(a_i) + 1)\nu(a_i) \leq cn\,\nu(a_0) + c\underbrace{\nu(a_0)}_{\geq \nu(a_1)} \sum_{i=1}^{n} (\nu(a_{i-1}) - \nu(a_i))$$

дальше эта сумма вычисляется легко:

$$cn\,\nu(a_0) + c\,\nu(a_0)(\nu(a_0) - \nu(a_n)) \leq c\,\nu(a_0)(n + \nu(a_0)).$$

Всё хорошо, но в оценку влезло n. Но n — это число шагов алгоритма Евклида и, как было получено ранее, является величиной логарифмической: $n \leq 2\log_2 a_1 + 1 \Rightarrow n \leq 2\log_2 a_0 + 1$, откуда следует заявленная оценка $O(m^2)$.

Т.к. эта оценка является верхней, то можно перейти к оценке через a_0: $O(\log^2 a_0)$. Эта оценка получена на базе алгоритма «наивного» деления и возникает вопрос: если использовать более эффективный алгоритм деления, можно ли уменьшить оценку? Оказывается, что это не так, и оценка точная. Для доказательства достаточно положить $a_0 = F_n$; $a_1 = F_{n-1}$, и затраты алгоритма будут примерно равны $\log_2^2 a_0$, т.к. для чисел Фибоначчи каким бы не был алгоритм деления, он будет эквивалентен вычитанию.

Более того, можно доказать теорему о нижней границе класса алгоритмов Евклида (отличающихся алгоритмом деления) и показать, что она равна $\log_2^2 a_0$.

Модулярная арифметика.

Все вычисления производятся по модулю некоторого числа.

Определение. Число a сравнимо с b по модулю k, если число k делит разность $(a-b)$.

Обозначение: $a \equiv b (\mathrm{mod}\, k)$.

Утверждение.

(i) Отношение сравнимости по модулю k является отношением эквивалентности.

(ii) Если $a_1 \equiv b_1 (\mathrm{mod}\, k)$, $a_2 \equiv b_2 (\mathrm{mod}\, k)$, то

$$a_1 + a_2 \equiv b_1 + b_2 (\mathrm{mod}\, k),\ a_1 - a_2 \equiv b_1 - b_2 (\mathrm{mod}\, k),\ a_1 a_2 \equiv b_1 b_2 (\mathrm{mod}\, k).$$

(iii) Каждое число a сравнимо по модулю k с одним и только одним числом из множества $\{0, 1, \dots, k-1\}$.

Отношение эквивалентности разбивает множество целых чисел на непересекающиеся классы, причём количество этих классов равно k. Возьмём по одному представителю из каждого класса: $\{a_0, a_1, \dots, a_{k-1}\}$ — полная система вычетов по модулю k. Система $\mathbf{I} = \{0, 1, \dots, k-1\}$ — каноническая система вычетов. $\mathbf{S} = \left\{ \left\lfloor -\frac{k}{2} \right\rfloor, \dots, -1, 0, 1, \dots, \left\lfloor \frac{k}{2} \right\rfloor \right\}$ — симметричная система вычетов. В курсе линейной алгебры доказывалось, что $\mathbf{Z}_k$ является кольцом вычетов по модулю k. При этом вычеты называются изображениями. При сложении складываем изображения, если остаёмся в рамках канонической системы, то всё в порядке, если получаем число $\geq k$, то вычитаем k. Противоположный a элемент определяется как $k - a$. Поэтому для битовых затрат операций сложения и вычитания имеем оценку $O(m)$, где $m = \nu(k)$.

Если для умножения используем алгоритм «наивное» умножения, то затраты допускают оценку $O(m^2)$.

Если k — простое, то $\mathbf{Z}_k$ является полем, если k не является простым, то $\mathbf{Z}_k$ — это кольцо с делителями нуля.

Сосредоточимся на случае $k = p$ — простое число. Покажем, откуда следует существование обратного элемента. Рассмотрим каноническую систему вычетов: $\{0, 1, \dots, p-1\}$. Из расширенного алгоритма Евклида следует, что найдутся такие целые s и t, что будет верно $sp + ta = 1 \Leftrightarrow ta \equiv 1 (\mathrm{mod}\, p)$. Следовательно, t будет обратным к a. Но t является целым и, возможно, не входит в систему. Было показано, что $|t| < p$, и t может быть отрицательным. Поэтому, если $t \geq 0$, то всё в порядке и t входит в рассматриваемую систему вычетов. Если $t < 0$, то прибавим p и попадём в рамки системы (ещё не более одной операции). Затраты на каждом шаге расширенного алгоритма Евклида для перехода от s_{i-1} к s_i и от t_{i-1} к t_i сравнимы с затратами на деление с остатком. Но расширенный алгоритм Евклида позволяет вычислять только t и не тратить операции на вычисление s. Но для $O(\log^2 p)$ это не имеет значения.

Пример. $a = 5$, $p = 13$

$2 \cdot 13 + (-5) \cdot 5 = 1 \Rightarrow$ для числа 5 обратным будет $(-5) \Rightarrow$ добавим 13 и получим $(-5) + 13 = 8$ и имеет место сравнение $8 \cdot 5 \equiv 1 (\mathrm{mod}\, 13)$.

Как следует из расширенного алгоритма Евклида, обращать можно не только по простому модулю, лишь бы a и p были взаимно простые.

Теорема (малая теорема Ферма). Пусть p — простое, u — целое, $u \neq 0 \Rightarrow u^{p-1} \equiv 1 (\mathrm{mod}\ p)$, при условии, что p не делит u.

Условие p — простое является существенным: Например, $u = 5$, $p = 6 \Rightarrow 5^5 \not\equiv 1 (\mathrm{mod}\ 6)$. Так что расширенный алгоритм Евклида в этом плане лучше.

Проблемой является распознавание простоты числа такое, чтоб его сложность была полиноминально ограниченной: $O\left(m^d\right)$, $m = \nu(n)$ или, что то же самое, $O\left(\log^d n\right)$. Можно попробовать зацепиться за малую теорему Ферма: $u^p \equiv u (\mathrm{mod}\ p) \Rightarrow p$ — простое. Т.е. можно для возведения в степень воспользоваться бинарным алгоритмом возведения в степень, но для сокращения затрат брать результат по модулю. Далее проверить условия теоремы и дать ответ. Но такой алгоритм, вообще говоря, неверный, т.к. существуют составные числа Кармайкла (бесконечно много), которые не ловятся таким алгоритмом.

Тем не менее, в 2002 году три индийских математика — Агравал, Кайал и Саксена — разработали алгоритм определения простоты числа с полиномиальной сложностью. Этот алгоритм базируется на следующем утверждении:

Утверждение. Пусть $u, n \in \mathbf{N}$, $u \perp n$ (взаимно простые). Тогда n является простым, если и только если $(x-u)^n \equiv x^n - u (\mathrm{mod}\ n)$.

Доказательство. ***Необходимость.*** Пусть n — простое и, для определённости, нечётное (для $n = 2$ проверяется непосредственно). Сравним коэффициенты при степенях x^k для $1 < k < n$. В правой части очевидно все коэффициенты равны нулю, а в правой из формулы бинома Ньютона получаем $(-1)^k C_n^k a^{n-k}$. Т.к. для числа сочетаний верно $C_n^k \equiv 0 (\mathrm{mod}\ n)$ (см. задачу 42), то получаем, что $(-1)^k C_n^k a^{n-k} \equiv 0 (\mathrm{mod}\ n)$, т.е. коэффициенты в левой и правой части сравнимы с нулём по модулю n. Для $k = n$ доказывать нечего, проверим свободный член: надо показать, что $(-u)^n \equiv -u (\mathrm{mod}\ n)$, но это следует непосредственно из малой теоремы Ферма.

Достаточность докажем от противного. Пусть n — составное, тогда найдётся такое q, которое будет являться делителем n. Рассмотрим ещё k такое, что q^k ещё делит n, а q^{k+1} уже нет. Рассмотрим число сочетаний из n по q:

$$C_n^q = \frac{n\left[(n-1)\cdot \ldots \cdot (n-q+1)\right]}{q!}$$

Ни один из множителей в квадратных скобках не может делиться на q. В итоге после сокращения должно получиться целое число, которое на q^k уже не делится (т.к. одно q из знаменателя сократится с n). $q^k \perp u^{n-q}$, и коэффициент в правой части $(-1)^{n-q} u^{n-q} C_n^q x^q$ не сравним с нулём по модулю n, что противоречит утверждению. Полученное противоречие завершает доказательство.

Просто построить алгоритм определения простоты числа на основе доказанного утверждения нельзя, но использовать его как основу вместо малой теоремы Ферма вполне реально.

Итак, получаем следующий способ определения простоты числа: возьмём $u = 1$, раскроем скобки и посчитаем остаток от деления на n. Но после раскрытия скобок получим $n+1$ слагаемое, и нужно будет поработать со всеми ими. Откуда для сложности алгоритма будет верна оценка $\Omega(n)$, а $n \approx 2^m$ и всё хорошо, но сложность экспоненциальная.

Посмотрим, что предлагают авторы. Рассмотрим сравнение: $(x-u)^n \equiv x^n - u \left(\mathrm{mod}\ x^r - 1, n\right)$, которое означает следующее: $(x-u)^n - x^n + u = Q(x)\left(x^r - 1\right) + S(x)$ и все коэффициенты делятся на n.

Чтобы двигаться к полиномиальности, нужно чтобы r было маленьким. Возводить в степень надо бинарным алгоритмом возведения в степень, каждый раз беря результат по модулю. Во-первых, r берётся не «с потолка», а совершенно конкретным, допуская оценку $r = O\left(m^6\right)$. Во-вторых, u тоже не является произвольным. Авторы показывают, что достаточно проверить сравнительно небольшой набор, допускающий оценку $O\left(m\sqrt{r}\right) \sim m^4$. Тогда, если использовать эти r и u, то по необходимому и достаточному условию можно распознать простоту числа, и сложность будет полиномиальной.

Об одном классе рекуррентных соотношений («разделяй и властвуй»)

Мы будем рассматривать рекурсивные алгоритмы. Рекурсивные соотношения для исследования сложности таких алгоритмов являются эффективным методом. Основой будут линейные рекурренции с постоянными коэффициентами и, возможно, с правыми частями.

Для начала рассмотрим «игрушечный» алгоритм построения множества V_n, содержащего все положительные целые числа, десятичные записи которых таковы, что сумма их цифр равна *n*. $V_1 = \{1\}$. Упростим задачу, рассматривая только такие цифры, десятичные записи которых состоят только из единиц и двоек: $V_2 = \{11, 2\}$, $V_3 = \{111, 21, 12\}$. Задача ставится следующим образом: дано *n*, и надо построить V_n.

Множество V_n можно разделить на 2 непересекающихся подмножеств: на конце чисел первого подмножества стоят единицы, на конце цифр второго — двойки. Если откинуть последнюю цифру, то получим V_{n-1} и V_{n-2} ($n > 2$). Тем самым получаем алгоритм построения V_n: к числам из V_{n-1} нужно приписать «1», к числам из V_{n-2} — «2» и объединить полученные множества.

Обозначим через v_n количество элементов в V_n, тогда $v_1 = 1$, $v_2 = 2$, $v_n = v_{n-1} + v_{n-2}$. Замечаем, что v_n совпадает с $(n+1)$-м числом Фибоначчи, т.е. $v_n = v_{n-1} + v_{n-2} = F_{n+1}$.

Попробуем определить $y(n)$ — число операций над числами (приписывание «1» или «2»). Для $y(n)$ мы можем написать следующее рекуррентное соотношение:

$$y(n) = y(n-1) + y(n-2) + \underbrace{F_n + F_{n-1}}_{F_{n+1}} \Rightarrow$$

$$y(n) - y(n-1) - y(n-2) = F_{n+1}, \text{ причём } y(1) = y(2) = 0$$

Обозначим затраты на объединение двух множеств через $z(n)$ и, аналогично, получим для $z(n)$ следующее рекуррентное соотношение:

$$z(n) - z(n-1) - z(n-2) = 1, \qquad (*)$$

$z(1) = z(2) = 0$ (предполагаем, что V_1 и V_2 нам даны).

Непосредственно видно, что -1 является частным решением соотношения (*). Частные решения можно угадывать, но можно воспользоваться специальным методом их нахождения, который очень сильно похож на метод нахождения частных решений линейных дифференциальных уравнений с постоянными коэффициентами. Суть метода заключается в следующем: рассмотрим однородное рекуррентное соотношение

$$a_d y(n) + a_{d-1} y(n-1) + \ldots + a_0 y(n-d) = 0$$

Составим по нему характеристическое уравнение, которое имеет вид:

$$a_d \lambda^d + a_{d-1} \lambda^{d-1} + \ldots + a_0 = 0,$$

которое, в свою очередь, будет иметь корни $\lambda_1, \lambda_2, \ldots, \lambda_k$ (включая комплексные) с кратностями $m_1, m_2, \ldots, m_k$ ($\sum_{i=1}^{k} m_i = d$). Аналогично дифференциальным уравнениям, в случае простого корня (кратность единица) будем иметь просто константу, в случае кратного

корня появляется многочлен (в общем случае, константу тоже можно рассматривать как многочлен нулевой степени). Итого, общее решение однородного рекуррентного соотношения будет иметь вид:

$$\sum_{i=1}^{k}\left(C_{i,0}+C_{i,1}n+\ldots+C_{i,m_i-1}n^{m_i-1}\right)\lambda_i^n$$

Рассмотрим теперь неоднородное рекуррентное соотношение. Когда в правой части стоит квазиполином (выражение вида $p(n)\mu^n$, где $p(n)$ — полином) или сумма квазиполиномов, то в линейной алгебре доказывается, что частное решение в этом случае, тоже будет квазиполиномом. Для нахождения частного решения (как и в случае дифференциальных уравнений) смотрим, сколько раз μ встречается среди корней характеристического уравнения. Пусть среди корней характеристического уравнения μ встречается l раз, тогда частным решением будет квазиполином $q(n)\mu^n$, где $\deg q(n)=l+\deg p(n)$.

Если в правой части стоит сумма квазиполиномов, то для построения частного решения нужно получить квазиполином для каждого слагаемого и просуммировать.

Таким образом, для (*) характеристическим уравнением будет следующее:

$$\lambda^2-\lambda-1=0$$

корнями которого будут $\lambda_1=\dfrac{1+\sqrt{5}}{2}=\phi$ и $\lambda_2=\dfrac{1-\sqrt{5}}{2}=\widetilde{\phi}$. Исходя из этого, получаем, что общем решение однородного рекуррентного соотношения будет иметь вид:

$$C_1\phi^n+C_2\widetilde{\phi}^n$$

В правой части стоит «1», но никто не мешает рассматривать её как 1^n, т.е. она является квазиполиномом ($\mu=1$; $p(n)=1$). 1 не встречается среди корней характеристического уравнения, поэтому частное решение можно искать в виде константы, откуда получаем, что -1 будет являться частным решением (*). Общее решение неоднородного соотношение будет складываться из общего решения однородного и частного решения неоднородного:

$$z(n)=C_1\phi^n+C_2\widetilde{\phi}^n-1$$

Из начальных условий ($z(1)=z(2)=0$), находим константы: $C_1=\dfrac{1}{\sqrt{5}}$, $C_2=-\dfrac{1}{\sqrt{5}}$. Таким образом $z(n)=\dfrac{1}{\sqrt{5}}\left(\phi^n-\widetilde{\phi}^n\right)-1=\dfrac{1}{\sqrt{5}}\phi^n+O(1)$. Можно выписать более простую (и более грубую) оценку: $z(n)=\Theta\left(\phi^n\right)$.

Обратимся к соотношению для $y(n)$: очевидно, что $-\dfrac{1}{\phi}=\widetilde{\phi}$ и по формуле Бене получаем, что правая часть $F_{n+1}=C_1\phi^n+C_2\widetilde{\phi}^n$, где C_1 и C_2 — полиномы нулевой степени, т.е. имеет вид суммы двух квазиполиномов, причем имеет место совпадение с корнями характеристического уравнения. Это указывает на то, что наше соотношение будет обладать частным решением следующего вида: $p_1(n)\phi^n+p_2(n)\widetilde{\phi}^n$, где $p_1(n)$ и $p_2(n)$ — полиномы 1-й степени.

Общее решение неоднородного соотношения будет иметь вид:

$$y(n)=C_1\phi^n+C_2\widetilde{\phi}^n+p_1(n)\phi^n+p_2(n)\widetilde{\phi}^n=q_1(n)\phi^n+q_2(n)\widetilde{\phi}^n,$$

где $q_1(n)$ и $q_2(n)$ — полиномы, имеющие первую степень. Их можно найти, но в данном случае это не нужно, т.к. нам нужна оценка для $y(n)$. Для $y(n)$ будет верна следующая оценка: $y(n) = \Theta\left(n\phi^n\right)$. Если нужны более точные оценки, то их можно вычислить, найдя точные представления для $q_1(n)$ и $q_2(n)$.

Отметим, что представленный алгоритм вычисления V_n не является эффективным. Действительно, V_n вычисляется, исходя из V_{n-1} и V_{n-2}. V_{n-1}, в свою очередь, вычисляется, исходя из V_{n-2} и V_{n-3}. Но V_{n-2} и в том и в другом случае будут вычисляться независимо (уж такой алгоритм). Поэтому напрашивается следующая модернизация: нужно вычислять множества парами: (V_n, V_{n-1}), (V_{n-1}, V_{n-2}), …, где V_{n-1} просто переходит. Тогда, для числа операций над числами получим такое соотношение:

$$y^*(n) - y^*(n-1) = F_{n+1}$$
$$y^*(1) = 0$$

Характеристическое уравнение такого соотношения будет уравнением первой степени, имеющее единственный корень $\lambda = 1 \Rightarrow y^*(n) = \Theta\left(\phi^n\right)$.

Для числа объединений получаем следующее рекуррентное соотношение:

$$\begin{matrix} z^*(n) - z^*(n-1) = 1 \\ z^*(1) = 0 \end{matrix} \Rightarrow z^*(n) = n - 2$$

Рассмотрим ещё более «неприятную» рекурсию:

$$Y_n = U\left(Y_{n-1}, Y_{n-2}, \ldots, Y_{n-k}\right)$$

Сколько раз при обращении к Y_n нужно будет обратиться к U ? Возникает такое рекуррентное соотношение:

$$y(n) - y(n-1) - \ldots - y(n-k) = 1$$
$$y(0) = \ldots = y(k-1) = 0$$

потому что можно считать, что $Y_0, Y_1, \ldots, Y_{k-1}$ нам известны.

Чтобы разобраться, как решение будет себя вести, надо исследовать характеристическое уравнение:

$$\lambda^k - \lambda^{k-1} - \ldots - \lambda - 1 = 0$$

Может быть показано следующее: все корни, кроме одного, этого характеристического уравнения лежат внутри единичного круга на комплексной плоскости, а для последнего корня верно $1 < |\lambda_k| < 2$, причём при $k \to \infty$ $|\lambda_k| \to 2$. Таким образом, когда $k \to \infty$ для сложности получаем оценку $\Theta\left(2^n\right)$.

Стратегия «разделяй и властвуй» заключается в следующем: пусть имеется задача, размер входа равен *n*; разбиваем исходную задачу на несколько подзадач, размеры хода которых будут меньше *n* (если *c* задач, то размеры входа $\frac{n}{c}$ с соответствующим округлением). В случае размера входа $n = 0$ или $n = 1$, предполагаем, что решение задачи известно.

Обратимся к какому-нибудь известному алгоритму, чтобы увидеть, как всё это работает. Для примера рассмотрим алгоритм сортировки слияниями (MS = merged sort): пусть дан массив

$x_1, \dots, x_n$, разделяем его на две части длинами $\left\lfloor \frac{n}{2} \right\rfloor$ и $\left\lceil \frac{n}{2} \right\rceil$, упорядочиваем полученные части (применяя алгоритм рекурсивно) и сливаем. Если длина массива равна 1, то ничего делать не надо.

Если мы попробуем исследовать сложность (в худшем случае) этого алгоритма по числу сравнений, то

$$T_{MS}(n) \le \begin{cases} 0, n = 1 \\ T_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{n-1}_{\text{на слияние}}, n > 1 \end{cases}$$

Пока не знаем, что равенство достигается, поэтому ставим $\le$.

Разные книги по-разному показывают, как работать с этим соотношением. Мы сведём его к рекуррентному соотношению с постоянными коэффициентами.

Предложение. Пусть функция $f : \mathbf{N} \to \mathbf{R}$ такая, что

$$f(n) \le \begin{cases} u, n = 1 \\ v f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + w f\left(\left\lceil \frac{n}{2} \right\rceil\right) + \varphi(n), n > 1, \end{cases}$$

где u, v, w — целые неотрицательные числа и v, w не равны нулю одновременно, функция $\varphi : \mathbf{N} \to \mathbf{R}$ неотрицательная и неубывающая. Тогда $f(n) \le t(\lceil \log_2 n \rceil)$, где $t(n)$ — решение рекуррентного уравнения

$$t(k) = \begin{cases} u, k = 0 \\ (v+w) \cdot t(k-1) + \varphi\left(2^k\right), k > 0 \end{cases}$$

Доказательство: Введём $F(n) = \begin{cases} u, n = 1 \\ vF\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + wF\left(\left\lceil \frac{n}{2} \right\rceil\right) + \varphi(n), n > 1 \end{cases}$ (то же, что и $f(n)$, но с «=»). Можно доказать, что $F(n)$ является неубывающей, т.е.

$$\forall n \ge 2 \ \Rightarrow \ F(n) \ge F(n-1) \qquad (*)$$

Доказательство этого факта проведём по индукции: для $n = 2$ имеем:

$$F(2) = \underbrace{(v+w)}_{\ne 0} \overbrace{u}^{F(1)} + \varphi(2) \ge u = F(1)$$

Если неравенство (*) выполнено для $2,3,\dots,n-1$, то $n > \left\lfloor \frac{n}{2} \right\rfloor \ge \left\lfloor \frac{n-1}{2} \right\rfloor \ge 1$, $n > \left\lceil \frac{n}{2} \right\rceil \ge \left\lceil \frac{n-1}{2} \right\rceil \ge 1$

$$\Rightarrow \ F(n) = vF\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + wF\left(\left\lceil \frac{n}{2} \right\rceil\right) + \varphi(n) \ge vF\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + wF\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + \underbrace{\varphi(n-1)}_{\text{не убывает}} = F(n-1)$$

Аналогично доказывается индукцией, что $f(n) \le F(n) \ \Rightarrow \ f(n) \le F(n) \le F\left(2^{\lceil \log_2 n \rceil}\right)$.

$$F\left(2^k\right) = \begin{cases} u, k = 0 \\ (v+w)F\left(2^{k-1}\right) + \varphi\left(2^k\right), k \ge 1 \end{cases}$$

т.е. $t(k) = F\left(2^k\right)$, и предложение доказано.

Теперь мы можем вернуться к алгоритму сортировки слияниями. В этом случае для $t(k)$ имеем следующее рекуррентное уравнение:

$$t(k) = \begin{cases} 0, k = 0 \\ 2t(k-1) + 2^k - 1, k > 0 \end{cases}$$

Если его записать в более привычном для рекуррентных уравнений виде, то получим

$$t(k) - 2t(k-1) = 2^k - 1$$
$$t(0) = 0$$

Корнем соответствующего характеристического уравнения будет $\lambda = 2$. Правая часть, в свою очередь, содержит слагаемое 2^k, поэтому частным решением будет $p(k)2^k$, где $\deg p(k) = 1$. «1» не является корнем характеристического уравнения, и вклад этого слагаемого будет весьма скромным. В итоге, общее решение рекуррентного соотношения будет иметь вид $t(k) = c_1 k 2^k + c_2 2^k + c_3$. Определив константы из начальных условий, получим: $t(k) = k2^k - 2^k + 1 \quad \Rightarrow \quad T_{MS}(n) \le \lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil} + 1 \le 2n \log_2 n + 1 \quad \Rightarrow$ $T_{MS}(n) = O(n \log n)$.

Задержимся на некоторое время всём этом. Чувствуется, что для асимптотических оценок не обязательно решать рекуррентное соотношение до конца. Но если мы будем удовлетворять асимптотическим верхним оценкам, то всего этого делать не нужно.

Определение. Рекуррентное соотношение, которое сразу следует из условий, будем называть *ассоциированным*.

Если $t(k) = O(\psi(k))$, то $f(k)$ (из предложения) представляет собой $f(n) = O(\psi(\lceil \log_2 n \rceil))$. Если дополнительно ψ такая, что $\psi(n+1) = O(\psi(n))$ (ψ растёт не слишком быстро), то для $f(n)$ можно написать $f(n) = O(\psi(\log_2 n))$.

Возвращаясь к примеру с алгоритмом сортировки слияниями, замечаем что $\lambda = 2$ является корнем характеристического уравнения, $p(k)2^k$, где $\deg p(k) = 1$, является частным решением, а остальные слагаемые решения не будут по скорости роста дотягивать до него $\Rightarrow t(k) = \Theta\left(k2^k\right)$.

Упомянем следующее: мы нашли оценку сверху для алгоритма *MS* ($T_{MS} \le 2n \log_2 n + 1$), но тщательный (кропотливый) анализ предполагает дополнительно нахождение оценки снизу. Для алгоритмов сортировки мы уже знаем, что оценка $\Omega(n \log n)$ имеет место, поэтом в данном конкретном примере мы можем остановиться.

Можно рассмотреть оценку для T_{MS} снизу, т.е. рассмотреть аналогичное рекуррентное неравенство с $\ge$. В этом случае будет справедливо предложение, аналогичное доказанному, за исключением того, что в этом случае $f(n) \ge t(\lfloor \log_2 n \rfloor)$ (доказывается аналогично).

В случае, когда для сложности задаётся рекуррентное равенство ($T(n) = \ldots$), то его можно рассматривать как систему неравенств $T(n) \le \ldots$ и $T(n) \ge \ldots$. После чего будут получены соотношения вида

$$t(\lfloor \log_2 n \rfloor) \le f(n) \le t(\lceil \log_2 n \rceil)$$

Границы очень тесные, откуда получаем хорошую оценку.

Попробуем описать класс массивов, на которых алгоритм (MS) будет работать дольше всего. Для этого достаточно рассмотреть действие алгоритма на перестановках. Назовём перестановку критичной, если на ней достигается максимум сложности алгоритма слияния.

Пусть $x_1, \dots, x_{\lfloor \frac{n}{2} \rfloor}$ и $y_1, \dots, y_{\lceil \frac{n}{2} \rceil}$ — критичные перестановки. Тогда $z_1, \dots, z_n$ равные

$$2x_1, \dots, 2x_{\lfloor \frac{n}{2} \rfloor}, 2y_1 - 1, \dots, 2y_{\lceil \frac{n}{2} \rceil} - 1 \qquad (*)$$

является перестановкой (доказать этот факт предлагается читателю) и для неё алгоритм сортировки слияниями потребует ровно $T_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$ сравнений. Отсюда можно вывести, что для сложности алгоритма сортировки слияниями можно писать

$$T_{MS}(n) \geq \begin{cases} 0, n = 1 \\ T_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1, n > 1 \end{cases}$$

Наряду с операциями сравнения мы можем рассмотреть сложность по перемещениям. Для сложности алгоритма сортировки слияниями по числу перемещений будет иметь место следующее соотношение:

$$\widetilde{T}_{MS}(n) = \begin{cases} 0, n = 1 \\ \widetilde{T}_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \widetilde{T}_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{n}_{\text{на слияние}}, n > 1 \end{cases}$$

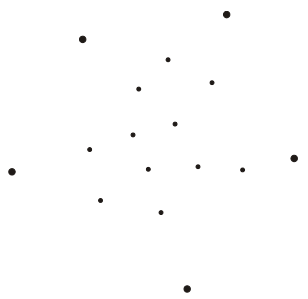
Ассоциированным соотношением в этом случае будет

$$t(k) = \begin{cases} 0, k = 0 \\ 2t(k-1) + 2^k, k > 0 \end{cases}$$

откуда получаем, что $t(k) = \Theta(n \log n)$.

Отметим, что рекурсия предполагает использование стека отложенных задач. Ранее было показано, что при хорошей стратегии выбора текущей задачи можно добиться того, что число отложенных задач (сохранённых в стеке) не превзойдёт $\log_2 n$ (выбирать следует более простую, сложную откладывая в стек).

Нередко $\varphi(n)$ (см. предложение) не известно в явном виде, а известна лишь оценка: $\varphi(n) = O(\xi(n))$. Как тут быть? На самом деле, особой сложности с этим нет. Рассмотрим такой пример: в вычислительной геометрии есть задача о выпуклой оболочке (на плоскости заданы точки и требуется найти такой выпуклый многоугольник, который охватывал бы все точки и обладал при этом наименьшей площадью).



Вообще говоря, здесь присутствуют 2 задачи, т.к. искомый многоугольник можно выдавать как упорядоченный набор вершин (например, с обходом по часовой стрелке начиная с некоторой), либо просто набором точек, хотя в данном контексте это не важно. Существует

множество алгоритмов, решающих поставленную задачу. В основе их лежит алгоритм построения выпуклого пересечения двух выпуклых многоугольников со сложностью $O(n_1 + n_2)$, где n_1 и n_2 — число вершин в данных многоугольниках. Пусть такой алгоритм у нас есть. Будем строить выпуклую оболочку следующим образом: разобьём данные n точек на две группы $\left\lfloor \frac{n}{2} \right\rfloor$ и $\left\lceil \frac{n}{2} \right\rceil$, для каждой из которых построим выпуклую оболочку и применим алгоритм построения пересечения. Тогда для сложности такого алгоритма получим:

$$T(n) = \begin{cases} 0, n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \overbrace{O(n)}^{\substack{\text{сложность алг.} \\ \text{пересечения}}}, n > 1 \end{cases}$$

Распишем $O(n)$ по определению: $\exists C > 0 : O(n) \le Cn$, тогда получим

$$T(n) \le \begin{cases} 0, n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + Cn, n > 1 \end{cases}$$

Соответствующее ассоциированное уравнение будет иметь вид

$$t(k) = \begin{cases} 0, k = 0 \\ 2t(k-1) + C \cdot 2^k, k > 0 \end{cases}$$

Сильно ли нам портит дело то, что мы не знаем значение величины константы C? Оказывается, что не очень, так как в любом случае рекуррентное соотношение имеет частное решение вида $p(k)2^k$, где $p(k)$ — многочлен первой степени, и, следовательно, для $t(k)$ будет верна оценка $t(k) = O\left(k2^k\right)$ и для сложности получаем $T(n) = O(n \log n)$. О том, можно ли строить выпуклую оболочку быстрее, поговорим позже.

Давайте посмотрим на задачи, которые мы знаем, с позиции рассматриваемого метода.

Для алгоритма возведения в степень (RS) можно написать следующую вещь:

$$T_{RS}(n) \le \begin{cases} 0, \text{если } n = 1 \\ T_{RS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 + 1, n > 1 \end{cases}$$

где одна из единиц во второй строке идёт на возведение в квадрат и ещё одна быть может на умножение уже накопленного. Второй единицы может не быть, но из-за знака $\le$ это ничему не противоречит.

Ассоциированное уравнение в этом случае будет иметь вид:

$$t(k) = \begin{cases} 0, k = 0 \\ t(k-1) + 2, k > 0 \end{cases}$$

решением которого будет $t(k) = 2k \;\Rightarrow\; T_{RS}(n) \le 2\lceil \log_2 n \rceil$. Ранее для этого алгоритма мы получали $2\lfloor \log_2 n \rfloor$, но новая оценка не на много хуже прежней.

Рассмотрим ещё один пример: алгоритм бинарного поиска (BS). Для него можно написать

$$T_{BS} \le \begin{cases} 1, n = 1 \\ T_{BS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1, n > 1 \end{cases}$$

Когда мы производим одно сравнение, то мы можем перейти к поиску в массиве длины $\left\lfloor \frac{n}{2} \right\rfloor$ или $\left\lfloor \frac{n}{2} \right\rfloor - 1$, поэтому для справедливости верхнего соотношения нужно быть уверенным, что $T_{BS}(n)$ с ростом n не убывает. Откуда такая уверенность? Конкретно для алгоритма бинарного поиска этот факт можно доказать по индукции, но сложность произвольного алгоритма, вообще говоря, не обладает этим свойством. Алгоритм возведения в степень, к примеру, не обладает таким свойством (в 7-ю степень возвести труднее, чем в 8-ю). Возвращаясь к алгоритму бинарного поиска, если всё сделать по предложенному рецепту, то ассоциированное уравнение будет иметь вид:

$$t(k) = \begin{cases} 1, k = 0 \\ t(k-1) + 1, k > 0 \end{cases}$$

Единица, стоящая в правой части рекуррентного соотношения, является корнем характеристического уравнения, поэтому для сложности будем иметь $T_{BS}(n) \le \lceil \log_2 n \rceil + 1$. Ранее была получена оценка чуть лучше: с полом вместо потолка.

Перейдём к алгоритмам более арифметическим. Иногда, если речь идёт об умножении целых чисел, то бывает удобно, если количество цифр в двоичной записи чисел равно 2^k. При работе с матрицами — хорошо, если их порядок равен 2^k. Оказывается, что если прибегнуть к такому приёму, что приписать к числу некоторое количество незначащих нулей так, что длина в итоге станет равна 2^k, то это позволяет умножать числа очень быстро. Если перемножаем две матрицы A и B, то можно дописать нужное количество нулей:

$$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & 0 \end{bmatrix}$$

чтобы порядок стал 2^k. Рассмотрим 2 алгоритма такого рода.

Алгоритм умножения Карацубы (КМ). Рассмотрим двоичные числа a и b длины $n = 2l$ (одинаковая и чётная): $a = e \cdot 2^l + f$, $b = g \cdot 2^l + h$, где e, f, g, h — двоичные числа длины l. А.Карацубе удалось составить алгоритм умножения двух чисел длины $2l$, использующий 3 умножения:

$$ab = eg \cdot 2^{2l} + \big((e+f)(g+h) - eg - fh\big) \cdot 2^l + fh$$

в то время как обычное раскрытие скобок

$$\left(e \cdot 2^l + f\right)\left(g \cdot 2^l + h\right) = eg \cdot 2^{2l} + \left(eh + fg\right) \cdot 2^l + fh$$

требует 4 умножения.

Если посмотреть на формулу Карацубы, то там нужно вычислить лишь eg, fh и $(e+f)(g+h)$. Рассмотрим внимательнее последнее произведение. Обозначим $e + f = e_1 2^l + f_1$, $g + h = g_1 2^l + h_1$, где e_1 и g_1 — однобитовые числа, а битовая длина f_1 и h_1 не превосходит l.

$$(e+f)(g+h) = e_1 g_1 2^{2l} + (e_1 h_1 + g_1 f_1) 2^l + \underbrace{f_1 h_1}_{\substack{\text{вычисляется} \\ \text{рекурсивно}}}$$

Все остальные операции (сложения, сдвиги) можно оценить через $O(n)$.

Для начала рассмотрим случай $n = 2^k$ (на вход подаются два числа с одинаковой битовой длиной равной 2^k). Тогда для сложности алгоритма Карацубы можно написать

$$T(k) = \begin{cases} 1, \ k = 0 \\ 3T(k-1) + O(2), \ k > 0 \end{cases}$$

Так же, как и в случае с выпуклой оболочкой, можно перейти к рекуррентному неравенству

$$T(k) \le \begin{cases} 1, k = 0 \\ 3T(k-1) + C \cdot 2^k, k > 0 \end{cases}$$

Ассоциированным уравнением будет

$$t(k) = \begin{cases} 1, k = 0 \\ 3t(k-1) + C \cdot 2^k, k \ge 1 \end{cases}$$

$$T(k) \le t(k)$$

Решение ассоциированного уравнения можно записать в виде $t(k) = O\left(3^k\right)$ (беря доминирующий член) $\Rightarrow T(k) = O\left(3^k\right)$.

Теперь перейдём к более общему случаю: пусть нам даны два числа произвольной длины. Пусть m — наибольшая из битовых длин этих чисел. Далее подгоняем длины обоих чисел к $2^{\lceil \log_2 m \rceil}$, приписывая незначащие нули. Тогда для сложности алгоритма получим:

$$T_{KM}(m) = O\left(3^{\lceil \log_2 m \rceil}\right) = O\left(3^{\log_2 m}\right) = \left\{ \log_2 m = \log_2 3 \cdot \log_3 m \right\} = O\left(m^{\log_2 3}\right)$$

При этом $\log_2 3 = 1{,}58... < 2$. Если вспомнить алгоритм «наивного» умножения, то для него была получена оценка $\Theta\left(m^2\right)$, поэтому алгоритм Карацубы в этом плане лучше.

Чтобы иметь возможность сравнивать этот алгоритм с другими, нужно иметь для него оценку снизу. Её можно получить. Давайте подсчитаем общее число умножений однобитовых чисел. Их количество будет

$$\tau(k) \ge \begin{cases} 1, k = 0 \\ 3\tau(k-1), k > 0 \end{cases}$$

$\Rightarrow \tau(k) \ge 3^k$, что даёт нам возможность написать $T_{KM}(m) = \Theta\left(m^{\log_2 3}\right)$.

Далее речь пойдёт об алгоритме Штрассена для умножения матриц (SM). Пусть A и B — квадратные матрицы. Если их порядки чётные ($n = 2l$), то мы можем их «разрезать» пополам:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Далее если мы попробуем их перемножить «в лоб», то нам потребуется 8 умножений матриц порядка l. Штрассен предлагает алгоритм, использующий в этом случае 7 умножений, но число сложений в этом случае будет не меленькое:

$$X_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \qquad X_5 = (A_{11} + A_{12})B_{22}$$
$$X_2 = (A_{21} + A_{22})B_{11} \qquad X_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$X_3 = A_{11}(B_{12} - B_{22}) \qquad X_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$X_4 = A_{22}(B_{21} - B_{11})$$

$$C_{11} = X_1 + X_4 - X_5 + X_7 \qquad C_{12} = X_3 + X_5$$
$$C_{21} = X_2 + X_4 \qquad C_{22} = X_1 + X_3 - X_2 + X_6$$

$$AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$T_{SM}(n) = \Theta\left(7^{\lceil \log_2 n \rceil}\right) = \Theta\left(7^{\log_2 n}\right)$$

Пользуясь тем же приёмом, что и в алгоритме Карацубы, получим

$$T_{SM}(n) = \Theta\left(n^{\log_2 7}\right),\ \log_2 7 = 2{,}807... < 3\,.$$

Вернёмся к алгоритму Карацубы: если разбиваем на 2 части, то нужно 3 умножения. Возникает идея: можно ли разбивать на p частей, чтобы требовалось q умножений чисел меньшей длины ($\Theta\left(n^{\log_p q}\right)$)? В алгоритме Карацубы $p = 2$, $q = 3$. Можно ли подобрать p и q так, чтобы log был ещё меньше? Ответ на этот вопрос дал А.Л.Тоом, который показал, что для любого p можно предложить алгоритм разбиения чисел длины p^k на части p^{k-1} и использующий $2p-1$ умножений: $\Theta\left(n^{\log_p (2p-1)}\right)$. Обратим внимание на следующее:

$$\log_p(2p-1) = \log_p 2p\left(1 - \frac{1}{2p}\right) = \log_p 2p + \log_p\left(1 - \frac{1}{2p}\right) = 1 + \log_p 2 + \log_p\left(1 - \frac{1}{2p}\right)$$

Последние 2 слагаемых приведённого равенства с ростом p стремятся к нулю, поэтому $\log_p(2p-1) \to 1$ при $p \to \infty$, т.е. $\forall \varepsilon > 0\ \exists$ алгоритм умножения со сложностью $\mathrm{O}\left(n^{1+\varepsilon}\right)$.

Отметим, что Штрассену удалось из других соображений предложить метод умножения, допускающий оценку $O(n \log n \log\log n)$, которая лучше полученной.

Тоом в своём методе «резал» числа более чем на 2 части. Казалось бы, что для матриц можно применить такой же подход и получить оценку вроде $O\left(n^{2+\varepsilon}\right)$. Оказывается, что ничего подобного. Но останавливаться на этом факте мы здесь не будем.

Отметим, что в литературе по теории сложности указывается более короткий путь анализа соотношений, получаемых из алгоритмов, действующих по принципу «разделяй и властвуй». На этот счёт формулируется соответствующая теорема, часто называемая master theorem. К примеру, в пособии В.Б.Алексеева эта теорема формулируется следующим образом:

Теорема (о рекуррентном неравенстве). Пусть $f(n)$ — функция натурального аргумента, пусть C — натуральное число, $C \geq 2$, и a, b, α — вещественные константы, причём $\alpha > 0$, и пусть для всех n вида C^k, $k = 1, 2, ...$ выполнено неравенство

$$f(n) \leq af\left(\frac{n}{C}\right) + bn^{\alpha}$$

Пусть при этом $f(n)$ монотонно не убывает на каждом отрезке вида $\left[C^k + 1, C^{k+1}\right]$. Тогда при $n \to \infty$ для всех n выполняется

$$f(n = \begin{cases} O\left(n^{\alpha}\right), \alpha > \log_C a \\ O\left(n^{\log_C a}\right), \alpha < \log_C a \\ O\left(n^{\alpha} \log n\right), \alpha = \log_C a \end{cases}$$

Посмотрим, даёт ли нам эта теорема что-то новое. Пусть $C = 2$, тогда $n = 2^k$ и ассоциированным уравнением для рекуррентного неравенства из условия теоремы будет

$$t(k) = a \cdot t(k-1) + b \cdot \left(2^k\right)^{\alpha}$$

или, что то же самое,

$$t(k) = a \cdot t(k-1) + b \cdot \left(2^{\alpha}\right)^k .$$

Корнем соответствующего характеристического уравнения будет $\lambda = a$ и дальнейшее решение рекуррентного соотношения будет зависеть от того, совпадает ли 2^{α} с a, что эквивалентно равенству $\alpha = \log_2 a$. Так, если $\alpha \neq \log_2 a$, то, учитывая доминанты, получим первые 2 строчки утверждаемого соотношения. Тем самым убеждаемся, что сформулированное ранее предложение весьма схоже с этой теоремой.

Всё это получается для чисел вида 2^k . Если переходить к общему случаю, то, используя монотонность, нужно рассматривать на одном из концов отрезка. Но монотонность не всегда очевидна, поэтому в этом вопросе следует проявлять осторожность (уже рассматривался пример сложности алгоритма возведения в степень RS, которая не обладает монотонностью). И ещё: теорема даёт оценку сверху, а для сравнения алгоритмов, только оценки сверху не достаточно.

Сводимость

Пусть имеются две задачи *P* и *Q*. Под «задачей» будем понимать следующую пару:

1) вычислительное задание;
2) соглашения о размере входа и о том, в чём измеряются вычислительные затраты.

Размер входа будем в дальнейшем всегда считать целым неотрицательным числом, сложность всегда будем считать однопараметрической функцией. Заметим сразу, что сформулированные допущения нас вовсе не обременяют.

Определение. Будем говорить, что *задача P не сложнее Q*, если для любого алгоритма A_Q, решающего задачу Q найдётся алгоритм A_P, решающий задачу *P*, такой, что $T_{A_P}(n) = O\left(T_{A_Q}(n)\right)$.

Обозначение: $P \le Q$.

Отметим сразу, что это бинарное отношение транзитивно. И ещё одно важное дополнение: жизнь не столь проста, как кажется. Определением, сформулированным выше, на практике приходится пользоваться редко. В связи с этим принимаются дополнительные соглашения о сложностях рассматриваемых алгоритмов, которые мы сформулируем ниже, а пока их обозначим. Оговаривается, что сложности растут не слишком медленно и не слишком быстро. Эти соглашения иногда доказуемы, иногда выводятся из правдоподобности. Также не рассматриваются «второсортные» алгоритмы, т.е. «на каждый хороший алгоритм решения задачи *Q* найдётся хороший алгоритм, решающий задачу *P*».

Определение. Задачи *P* и *Q*, для которых одновременно верно $P \le Q$ и $Q \le P$, будем называть *эквивалентными* (по сложности).

Обозначение: $P \bowtie Q$.

Далее рассмотрим пример.

Пример. Раньше про деление мы почти ничего не говорили, кроме того, что его можно реализовать на базе умножения. Рассмотрим следующие задачи:

M: умножение 2-х целых чисел *a* и *b*

D: деление целого *a* битовой длины $\le 2m$ на целое *b* битовой длины *m*

S: возведение в квадрат целого *a*

R: обращение целого *a*

В задаче *R* имеется в виду не точное вычисление, а до *m* двоичных разрядов после запятой. Для задачи *D* — построение *m* двоичных цифр частного, не зависимо от положения запятой.

Покажем, что $M \bowtie D \bowtie S \bowtie R$.

Для начала сформулируем ограничения, которые будем предполагать выполненными:

1) сложность любого из рассматриваемых алгоритмов $f(m)$ является неубывающей функцией;
2) для сложности верно $f(m) \ge m$ (растёт не слишком медленно);
3) для $f(\alpha m)$ верно: $\alpha f(m) \le f(\alpha m) \le \alpha^2 f(m)$, для любого вещественного $\alpha > 1$.

Полное доказательство мы здесь приводить не будем (его можно найти в книге «Построение и анализ вычислительных алгоритмов» из рекомендуемой литературы) и рассмотрим его лишь в общих чертах.

Начнём с того, что докажем $M \le S$. Для этого нужно показать, что для любого алгоритма A_S, решающего задачу S, найдётся алгоритм A_M, решающий задачу M, такой, что будет выполнена оценка: $T_{A_M}(m) = O\left(T_{A_S}(m)\right)$.

Воспользуемся следующим соотношением: $ab = \frac{1}{2}\left((a+b)^2 - a^2 - b^2\right)$. Это соотношение даёт нам алгоритм вычисления произведения, используя операцию возведения в степень. Очевидно, что сложение и вычитание имеет линейную по времени сложность, а деление на 2 вообще выполняется за даром (деление на 2 эквивалентно сдвигу всех разрядов числа на 1 бит вправо). Поэтому для сложности такого алгоритма получаем:

$$T_{A_M}(m) = \underbrace{T_{A_S}(m+1)}_{(a+b)^2} + 2T_{A_S}(m) + \underbrace{O(m)}_{\text{"+" и "–"}}$$

$T_{A_S}(m+1) \overset{1)}{\le} T_{A_S}(2m) \overset{3)}{\le} 4T_{A_S}(m)$ (в силу сформулированных ограничений 1 и 3)

$$\Rightarrow T_{A_M}(m) = O\left(T_{A_S}(m)\right).$$

Теперь докажем, что $S \le R$. Будем считать, что у нас есть алгоритм A_R, решающий задачу R — обращения целого числа. Воспользуемся равенством:

$$a^2 = \frac{1}{\frac{1}{a} - \frac{1}{a+1}} - a$$

Заметим, что в итоге мы должны получить целое число (a^2), в то время как в правой части равенства стоят совсем не целые величины. Дотошные авторы в умных книжках говорят, что в обращении надо брать $3m$ цифр после запятой. Умножаем на 2^{3m}, получая целое число, и обращаем. Затем, глядя на последние 4 цифры, можно получить целый ответ, сделав некоторую поправку. $T_{A_S}(m) = O\left(T_{A_R}(3m)\right)$ и с учётом пункта 3 ограничений, получаем $T_{A_S}(m) = O\left(T_{A_R}(m)\right)$.

Покажем, что $R \le M$. В этом случае одной формулой обойтись не получится и придётся привлечь математический анализ. Построим последовательность $x_0, x_1, x_2, \ldots$, сходящуюся к $\frac{1}{a}$. Такую последовательность, например, можно описать рекуррентной формулой: $x_i = 2x_{i-1} - ax_{i-1}^2$. Соотношений такого рода можно привести не одно, но особенностью такой формулы является то, что она родственна с методом касательных решения уравнения на отрезке (если сходится, то очень быстро: сходимость квадратичная, а не линейная). Кроме того, исследуя несколько первых цифр числа a, можно выбрать x_0 так, что $\frac{1}{2a} \le x_0 \le \frac{1}{a}$ и для элементов последовательности будет выполнено: $x_i = \frac{1}{a}(1-\varepsilon)$, $x_{i+1} = \frac{1}{a}\left(1-\varepsilon^2\right)$. С учётом нулевого приближения, для элементов последовательности будет верно, что $\left|x_i - \frac{1}{a}\right| \le \frac{1}{2^{2^i}}$. У приближений отбрасываем концы (те цифры, которые не заслуживают доверия). Используя тот факт,

что сумма геометрической прогрессии имеет порядок последнего члена, получаем, что $T_{A_R}(m) = O\left(T_{A_M}(m)\right)$.

Тем самым доказано, что $M \le S \le R \le M \ \Rightarrow\ M \bowtie S \bowtie R$.

Следующим этапом покажем, что $D \le M$, используя, что $M \bowtie R$. Будем исходить из того, что $\frac{a}{b} = a \cdot \frac{1}{b}$. Деление в этом случае будет неточным, поэтому надо выполнить деление с запасом ($3m$), после чего умножить. При этом младшие разряды могут оказаться неверными. Дотошные люди посчитали, что их количество не превосходит 3, и «шевелением» этих разрядов можно получить точный ответ. Осталось показать, что $R \le D$, но это очевидно.

Определение. Пусть существуют 2 задачи P и Q и 2 алгоритма U и V такие, что алгоритм U по входу задачи P строит один или несколько входов для задачи Q:

$$x \text{ — вход } P \xrightarrow{U} y_1, \dots, y_k \text{ — входы } Q$$

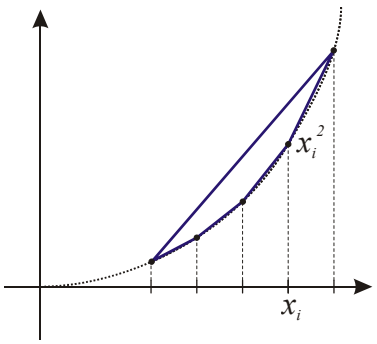
Задача Q переводит $y_1, \dots, y_k$ в $z_1, \dots, z_k$: $Q(y_1, \dots, y_k) \to z_1, \dots, z_k$, а алгоритм V по $z_1, \dots, z_k$ строит решение задачи Q. Тогда пару (U, V) будем называть *сведением*.

Отвлечёмся на минуту от сведений и докажем общее утверждение.

Утверждение. Если P и Q таковы, что $P \le Q$, $\mathscr{P}$ и $\mathscr{Q}$ — классы алгоритмов, решающих задачи P и Q соответственно, и пусть $f(n)$ — асимптотическая нижняя граница сложности для класса $\mathscr{P}$. Тогда $f(n)$ будет являться асимптотической нижней границей сложности для класса $\mathscr{Q}$.

Доказательство: $P \le Q \ \Leftrightarrow\ T_{A_P}(n) = O\left(T_{A_Q}(n)\right) \ \Leftrightarrow\ T_{A_Q}(n) = \Omega\left(T_{A_P}(n)\right)$. С другой стороны, $f(n)$ является асимптотической нижней границей для $\mathscr{P} \Rightarrow\ \forall A \in \mathscr{P}$ (в частности, для $A = A_P$) будет верна оценка: $T_A(n) = \Omega\left(f(n)\right) \ \Rightarrow\ T_{A_P}(n) = \Omega\left(f(n)\right) \ \Rightarrow\ T_{A_Q}(n) = \Omega\left(f(n)\right)$, что доказывает утверждение.

Пример. Вернёмся к задаче построения выпуклой оболочки. Покажем, что если за размера входа взять n — количество данных точек на плоскости, то $\Omega(n \log n)$ будет нижней асимптотической оценкой. Отметим, что «продвинутые» алгоритмы построения выпуклой оболочки имеют оценку сложности $O(n \log n)$, тем самым являются оптимальными по порядку сложности. Как показать $\Omega(n \log n)$? Для этого сведём алгоритм сортировки к задаче о построении выпуклой оболочки. Пусть на входе алгоритма сортировки имеется набор чисел $x_1, x_2, \dots, x_n$. Построим по ним набор точек на плоскости следующим образом: $\left(x_1, x_1^2\right), \left(x_2, x_2^2\right), \dots, \left(x_n, x_n^2\right)$.



Очевидно, что все точки будут принадлежать выпуклой оболочке. Теперь, если мы захотим её выдать в виде упорядоченного массива вершин с обходом, например, по

часовой стрелке, то нам непременно потребуется отсортировать начальные значения $x_1, x_2, \dots, x_n$. Мы уже знаем, что $\Omega(n \log n)$ является асимптотической оценкой снизу для алгоритмов сортировки, поэтому для алгоритмов построения выпуклой оболочки будет верна оценка $\Omega(n \log n)$. Таким образом, если известно, что одна задача решается медленно, то, сведя другую задачу к первой, можно показать, что она тоже решается медленно.

Рассмотрим две задачи: умножение булевых матриц и транзитивно-рефлексивное замыкание графа. Можно показать, что эти задачи являются эквивалентными. Таким образом, если мы научимся находить транзитивное замыкание за $O(n^2)$, то и булевые матрицы мы сможем умножать за $O(n^2)$.

Действительно, если A — матрица смежности исходного графа (A порядка n), тогда матрица смежности транзитивно-рефлексивного замыкания графа будет равна

$$A^* = I \vee A \vee A^2 \vee \dots \vee A^n = (I \vee A)^n \text{ (максимальная длина пути — } n)$$

Пусть у нас есть две матрицы: A и B. Построим по ним ещё одну матрицу по следующему правилу:

$$\begin{bmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{bmatrix}$$

и рассмотрим граф, для которого эта матрица будет являться матрицей смежности. Теперь посмотрим, что будет транзитивно-рефлексивным замыканием этого графа. Замыканием будет граф со следующей матрицей смежности:

$$\begin{bmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$$

Тем самым, для получения произведения AB достаточно прочитать верхний правый угол получившейся матрицы.

Обычно для построения транзитивно-рефлексивного замыкания используют алгоритм Уоршела со сложностью $O(n^3)$, где за O скрывается совсем не большая константа (порядка 3), и ещё одним преимуществом алгоритма является небольшой расход памяти.

Далее нашей целью будет рассмотрение некоторых вопросов алгоритмов полиномиальной сложности. Для таких алгоритмов сложность не обязательно является полиномом, но тем не менее она должна допускать оценку $O(n^c)$, где c — некоторая константа. Вообще говоря, оценка с O предполагает выполнение соответствующего неравенства начиная с некоторого n, но мы будем считать, что для всех n выполняется $T(n) \le p(n)$, где $p(n)$ — полином.

Полиномиальной сложности алгоритмы — это искусство. Вопрос существования полиномиального алгоритма, решающего ту или иную задачу, отнюдь не праздный. От экспоненциальной сложности стараются перейти к полиномиальной. Сторонники получения какого-нибудь алгоритма не стремятся получить алгоритм с полиномиальной сложностью. Основной своей задачей они считают получение алгоритма, а снижение показателя сложности — это вопрос времени.

Рассмотрим следующий пример. В линейном программировании существует симплекс-метод, позволяющий решать задачи, сложность которого не является полиномиальной. Несколько лет назад был найден алгоритм решения задач линейного программирования с

полиномиальной сложностью, но коэффициенты полинома были настолько велики, что никакого продвижения не было. Симплекс-метод обладает субэкспоненциальной сложностью: растёт медленнее, чем d^n, но быстрее, чем любой полином. Например, субэкспоненциальной будет сложность, имеющая вид $n^{\log n}$. Тем не менее, дело-то в том, что сложно, глядя на задачу, определить, имеется ли полиномиальный алгоритм её решения или нет.

Сосредоточимся на задачах распознавания, т.е. задачах, ответом на которые является «да» или «нет». Более конкретно рассмотрим задачу распознавания языков, которая ставится следующим образом: есть некоторый алгоритм A, множество слов в алфавите A обозначим через A^* и выбирается их подмножество $L \subset A^*$ — язык в алфавите A. Оговорим ещё одно допущение: что брать за размер входа? Для массивов мы брали число элементов, для квадратных матриц — порядок (что не то же самое, что число элементов). Здесь входом будут слова x, а за размер входа возьмём его длину $|x|$ — число букв в этом слове (неотрицательное целое). Определимся с операциями. Т.к. любой конечный алфавит можно свести к $A = \{0, 1\}$ (все слова в алфавите A можно кодировать последовательностями из нулей и единиц). Тогда в качестве операций можно рассматривать битовые операции и битовую сложность. Не обязательно рассматривать двухсимвольный конечный алфавит, можно и более богатый, только операции в этом случае будут задаваться соответствующими таблицами. Примечательно, что таким способом суженная задача вызывает проблемы более осязаемые и более привычные.

Пусть алгоритмы распознавания языков, имеющие полиномиальную сложность, образуют класс P. В некоторых изданиях к P относят все алгоритмы с полиномиальной сложностью, что было бы не совсем правильным. Класс полиномиальных алгоритмов правильнее было бы обозначить через *Poly*. Вернёмся к нашему определению класса P, как классу алгоритмов распознавания имеющих полиномиальную сложность. Задача определения принадлежности алгоритма к классу P является достаточно сложной.

Нас будут интересовать предикаты, определённые на словах. Можно говорить не о «задачах» из класса P, а о предикатах.

Рассмотрим наряду с P другой класс алгоритмов — NP.

Определение. Будем говорить, что предикат (задача) $u(x) \in NP$, если существуют такие полиномы p и q и такой предикат $R(x, y)$, что $u(x) = \exists_{y \in A^*, |y| \le p(|x|)} R(x, y)$ и предикат $R(x, y)$ имеет полиномиальную временную сложность, ограниченную $q(|x| + |y|)$.

Соотношение для $u(x)$ можно записать и в другой форме: $\exists_{y \in A^*} (|y| \le p(|x|) \wedge R(x, y))$.

Обозначение NP происходит от non deterministic polynomer. Слово y в этом случае называется сертификатом x (в некоторой литературе — «подсказкой»).

Класс NP замечателен тем, что легко проверить принадлежность к нему. Рассмотрим это на примерах. Но для начала мы должны определится с тем, как будем записывать слова в нашем алфавите. Эти способы, вообще говоря, могут быть весьма разнообразными. Для примера рассмотрим ситуацию, когда алфавитом являются целые положительные числа. В этом случае для их записи мы можем использовать палочную систему (рисовать столько палочек, какое число мы хотим изобразить) или позиционную систему счисления. Сложность в разных случаях будет различной.

Рассмотрим задачу определения, является ли заданный граф гамильтоновым. Гамильтоновым называю граф, если в нём существует путь, проходящий через все вершины ровно по одному разу. Когда мы рассматриваем произвольный граф и хотим определить, является ли он гамильтоновым, мы можем записать граф матрицей смежности (при

необходимости разделяя строчки некоторым разделителем). Не трудно видеть, что сформулированная задача распознавания гамильтонового графа принадлежит классу *NP*. Чтобы доказать это достаточно указать сертификат, в качестве которого в данном случае можно взять сам гамильтонов путь. Уж как-нибудь за не очень большое (полиномиальное) время проверить, является ли он путём и является ли он гамильтоновым, мы вполне сможем.

Задача определения составного числа так же принадлежит классу *NP*. Сертификатом в этом случае будет делитель. Очевидно, что за короткое время мы сможем проверить, является ли он делителем или нет.

Отметим ещё, что на вход алгоритма распознавания наряду с корректными данными могут подаваться некорректные, например какая-нибудь белиберда. Но задача проверки корректности введённой информации решается за полиномиальное время, поэтому, не ограничивая общности, можно считать, что на вход нам всегда подаются корректные данные.

Важным понятием является понятие полиномиальной сводимости.

Определение. Будем говорить, что задача (предикат) u_1 полиномиально сводится к задаче u_2, если существует функция $f : A^* \to A^*$, применимая к любому слову из A^*, время вычисления которой ограничено $p_0(|x|)$, и такая, что истинность $u_1(x)$ эквивалентна истинности $u_2(f(x))$.

Обозначение: $u_1 \le_P u_2$

Необходимо отметить важный момент: если u_1 полиномиально сводится к задаче u_2 и $u_2 \in P$, тогда $u_1 \in P$. В этом утверждение есть одна тонкость: новый аргумент $f(x)$ может иметь другую длину, нежели x, но она тоже будет ограничена полиномиально (т.к. время вычисления ограничено полиномиально).

Возникает проблема: верно ли, что $P = NP$?

Важными понятиями являются *NP*-трудный и *NP*-полный предикаты.

Определение. *NP*-трудным называется предикат, к которому полиномиально сводится любой предикат из *NP*.

Определение. *NP*-полным называется *NP*-трудный предикат, принадлежащий классу *NP*.

Вырисовывается очень удобный путь для решения обозначенной проблемы в положительном смысле: нужно найти *NP*-трудную задачу и предложить для неё полиномиальный алгоритм.

Доказано, что *NP*-полной задачей является задача определения выполнимости булевой формулы. Формула является выполнимой, если существует такой набор значений переменных, входящих в формулу, что результатом формулы является истина. Всего различных наборов для n переменных будет 2^n. До сих пор не известно, существует ли алгоритм, который за полиномиальное время позволяет установить выполнимость, но известно, что задача является *NP*-полной, а значит *NP*-трудной.

Как уже было сказано, задача о гамильтоновом графе является *NP*-полной. Это наводит на мысль, что $P \ne NP$ (что, вообще говоря, не доказано и по сей день).

Если вы трудитесь над какой-то проблемой и пытаетесь найти для её решения полиномиальный алгоритм, но вдруг узнаёте, что она является *NP*-трудной (известная *NP*-задача полиномиально сводится к вашей), тогда лучше считать, что для вашей задачи нет полиномиального алгоритма.

Для задачи распознавания простоты числа, если входом считать $m = \nu(n)$ — битовую длину n, 3 года назад был предложен Индийский алгоритм (разработанный тремя индийскими математиками) с полиномиальной сложностью, допускающий оценку $O\left(m^{13}\right)$. Только для решения означенной проблемы этот результат ровным счётом ничего не даёт. Если бы им удалось доказать обратное, что не существует полиномиального алгоритма распознавания простоты числа, тогда сразу следовало бы, что $P \neq NP$, т.к. задача определения простоты числа полиномиально сводится к задаче распознавания составного числа, которая принадлежит классу *NP*.

Уместно будет заметить, что мы разнесли задачи распознавания простоты числа и распознавания составного числа, хотя эти задачи являются обратными. Если для задачи определения составного числа мы можем показать, что она принадлежит классу *NP*, то для обратной задачи распознавания простоты числа мы этого показать не можем (какой тут сертификат?). Следует отметить, что не обязательно из того, что $u(x) \in NP$ следует, что $\overline{u}$ (обратная задача к u) $\in NP$. Поэтому будем говорить, что предикаты, отрицание которых принадлежит классу *NP*, принадлежат классу *Co-NP*.

Отметим, что многие из утверждения, которые мы сделали недавно, доказываются при помощи машины Тьюринга (МТ). Но какое это имеет отношение к нам? А может быть так, когда мы имеем дело с МТ, полиномиального алгоритма нет из-за перемещений по ленте. В равнодоступной адресной машине (РАМ) таких затрат нет и кажется, что существование полиномиальных алгоритмов на РАМ более вероятно. Но это не так, потому как путешествия по ленте не очень сильно удлиняют вычисления (в худшем случае сложность возведётся в квадрат, но изначально полиномиальная сложность полиномиальной же и останется). Следовательно, если нет полиномиального алгоритма для МТ, то и для РАМ его тоже нет.

Более подробно про алгоритмы полиномиальной сложности и найти более подробные доказательства сформулированных утверждений можно найти в книге М.Гэри и Д.Джонсона «Вычислительные машины и трудно решаемые задачи».

В заключение приведём несколько примеров *NP*-полных задач.

Примеры.

1. «Задача редактирования слова». Имеется алгоритм A, два слова $x, y \in A^*$ и $k \in \mathbf{N}$. Алгоритм заключается в определении, можно ли из x получить y не более, чем за k операций, в качестве которых допустимыми являются вычеркивание символа и перестановка двух соседних символов. Доказано, что эта задача *NP*-полная.
2. Имеется квадратная матрица порядка n из нулей и единиц и натуральное число k. Алгоритм заключается в определении, можно ли все единицы покрыть не более чем k прямоугольниками, не покрывая при этом ни одного нуля. Эта задача тоже является *NP*-полной.
3. «Задача о рюкзаке». Задано конечное множество U. Для каждого $u \in U$ определён размер $s(u)$ и цена $v(u)$. Заданы $b, k \in \mathbf{N}$, и спрашивается, можно ли выбрать подмножество $U' \subset U$ такое, что

$$\sum_{u \in U'} s(u) \le b, \ \sum_{u \in U'} v(u) \ge k.$$

   Эта задача тоже является *NP*-полной.

4. Задан набор пар целых чисел $(a_i, b_i) \in \mathbf{Z}^2$, $i = 1, 2, \ldots, n$, $b_i \geq 0$. Рассматривается полином вида $\sum_{i=1}^{n} a_i z^{b_i}$ и спрашивается, есть ли у него корень в комплексной плоскости, лежащий на единичной окружности ($|z| = 1$). Эта задача является *NP*-трудной, но принадлежность её к классу *NP*, не установлена.

Задачи

1. Доказать равенства:

$$n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil,\ n \in \mathbf{N}$$
$$\lceil a \rceil = -\lfloor -a \rfloor,\ a \in \mathbf{R}$$

Решение. 1) В случае чётного n решение тривиально: $n = 2k \Rightarrow \frac{n}{2} = \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil = k$ $\Rightarrow \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = k + k = 2k = n$. Если n нечётно, т.е. $n = 2k+1 \Rightarrow \left\lfloor \frac{n}{2} \right\rfloor = k; \left\lceil \frac{n}{2} \right\rceil = k+1$ $\Rightarrow \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = k + k + 1 = 2k + 1 = n$.

2) Если $a \in \mathbf{Z}$, то очевидно, что $\lceil a \rceil = a; \lfloor -a \rfloor = -a$, что доказывает равенство. Если a не является целым, то найдется такое целое число n и такое $0 < \varepsilon < 1$, что $a = n + \varepsilon$. Тогда $\lceil a \rceil = \lceil n + \varepsilon \rceil = n + 1$, $\lfloor -a \rfloor = \lfloor -n - \varepsilon \rfloor = -n - 1 \Rightarrow \lceil a \rceil = -\lfloor -a \rfloor$.

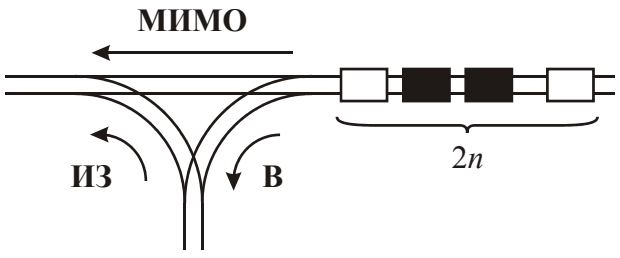
2. Доказать равенство:

$$k, n \in \mathbf{N},\ k > 1 \Rightarrow \lfloor \log_k n \rfloor + 1 = \lceil \log_k (n+1) \rceil$$

Решение. Не трудно видеть, что для данного n всегда найдётся такое целое число m, что выполняется оценка: $k^m \le n < k^{m+1}$. Тогда $\lfloor \log_k n \rfloor = m$. Так как k целое, то $k^m < n + 1 \le k^{m+1}$, откуда $m < \log_k (n+1) \le m + 1 \Rightarrow \lceil \log_k (n+1) \rceil = m + 1$ что и доказывает равенство.

3. Считая в алгоритме сортировки простыми вставками операцию обмена $a \leftrightarrow b$ реализованной следующим образом: `c := a; a := b; b := c`, определить, чему равны T_{I_1} и T_{I_2}.

4. Имеется железнодорожный разъезд с тупиком (см. рис.), в одной части которого находятся $2n$ вагонов двух типов: n белых и n чёрных. Порядок вагонов не определён. Имеется 3 операции перемещения вагонов: 1) **МИМО**, 2) **В** (в тупик) и 3) **ИЗ** (из тупика). Требуется привести алгоритм сортировки вагонов, в результате работы которого с другой стороны будет составлена последовательность из $2n$ вагонов, цвета которых чередуются.



Решение. Требуется переправить вагоны справа на лево, изменив их порядок так, чтобы цвета вагонов чередовались. Предлагаемый алгоритм заключается в следующем: если цвет первого вагона формируемого состава не имеет значения, то первый вагон просто проходит **МИМО**. Если цвет первого вагона важен, то с помощью тупика это всегда можно сделать. Далее, пока имеются вагоны справа, проверяем на совпадение цветов последний вагон слева с первым вагоном справа. Если их цвета не совпадают,

выполняем операцию **МИМО**, после чего проверяем наличие вагонов в тупике. Если в тупике имеются вагоны, то выполняем операцию **ИЗ** и переходим к следующему вагону справа. При совпадении цветов, отправляем очередной вагон в тупик операцией **В**.

На псевдокоде предлагаемый алгоритм выглядит следующим образом: исходный состав справа обозначен массивом a_i, каждый элемент которого принимает значения чёрный или белый. Операция `not` меняет значение цвета с белого на чёрный и наоборот, `z` — текущее количество вагонов в стеке, `k` — цвет последнего вагона формируемого состава слева. Основная идея алгоритма заключается в том, что в тупике в один момент времени не могут находиться вагоны разных цветов и в конце каждой итерации цикла если в тупике есть вагоны, то цвет их совпадает с цветом последнего вагона формируемого состава.

```
z := 0; k := a_1;
for i = 2 to 2n do
  if k = a_i then
    В;
    z := z + 1;
  else
    МИМО;
    k := a_i;
    if z > 0 then
      ИЗ;
      z := z - 1;
      k := not k;
    fi
  fi
od
```

Не трудно видеть, что сложность представленного алгоритма по количеству операций **МИМО**, **В**, **ИЗ** $T = O(n)$, но из постановки задачи ясно, что сложность любого алгоритма, решающего задачу, есть $\Omega(n)$, следовательно, сложность представленного алгоритма $T = \Theta(n)$.

5. Путник стоит перед бесконечной стеной в обе стороны. Известно, что на расстоянии n шагов в одну из сторон находится дверь. Требуется привести алгоритм нахождения двери путником, сложность которого по числу шагов составляла бы $O(n)$.

Решение. Очевидный алгоритм заключается в следующем: путник на первой итерации делает один шаг в одну из сторон, затем возвращается и делает один шаг уже в другую сторону и снова возвращается. На второй итерации ситуация повторяется, но уже совершается по два шага в обе стороны. На третьей итерации — три шага и т.д. пока не будет обнаружена дверь. Сложность такого алгоритма составляет $2 \cdot (1 + 2 + ... + n) = n(n+1) = O(n^2)$, в связи с чем, данный алгоритм не подходит. Предыдущий алгоритм базировался на арифметической прогрессии, что и дало квадратичную сложность. Попробуем оценить сложность подобного алгоритма, базирующегося на геометрической прогрессии. Пусть по прежнему на первой итерации совершается по одному шагу в одну и другую сторону, тем самым первый член прогрессии будет равен 1. Выберем произвольное $q \in \mathbf{N}, q > 1$ и на каждой k-й итерации будем совершать q^{k-1} шагов в обе стороны. Исходя из этих данных, получаем, что количество требуемых алгоритму итераций не превзойдёт $\log_q n + 1$,

тогда суммарное количество шагов не превзойдёт $2\cdot\frac{q^{\log_q n+1}-1}{q-1}=2\cdot\frac{qn-1}{q-1}=O(n)$, что и требовалось.

6. Доказать асимптотику сложности алгоритма пробных делений (TD), приняв за размер входа битовую длину n: $T^*_{TD}\left(m=\nu(n)\right)=\Theta\left(2^{\frac{m}{2}}\right)$.

   Решение. При фиксированной длине числа n, равной m, получаем ограничение на значения n: $2^{m-1}\le n<2^m$. Согласно постулату Бертрана, в интервале $(2^{m-1},2^m)$ найдётся простое число n, для определения простоты которого алгоритму пробных делений потребуется в точности $\lfloor\sqrt{n}\rfloor-1$ делений. Следовательно, получаем оценки на $T^*_{TD}(m)$: $\lfloor\sqrt{2^{m-1}}\rfloor-1<T^*_{TD}(m)<\lfloor\sqrt{2^m}\rfloor-1 \quad\Leftrightarrow\quad \sqrt{2^{m-1}}-2<T^*_{TD}(m)<\sqrt{2^m}-1 \quad\Leftrightarrow$ $\frac{1}{\sqrt{2}}\cdot 2^{\frac{m}{2}}-2<T^*_{TD}(m)<2^{\frac{m}{2}}-1 \Rightarrow T^*_{TD}(m)=\Theta\left(2^{\frac{m}{2}}\right)$.

7. $f(n)=a_m n^m+\ldots+a_1 n+a_0$, $a_m\ne 0$. Доказать утверждения:

   1) $f(n)=O\left(n^k\right)\Leftrightarrow k\ge m$;

   2) $f(n)=\Omega\left(n^k\right)\Leftrightarrow k\le m$;

   3) $f(n)=\Theta\left(n^k\right)\Leftrightarrow k=m$.

   Решение.

   a. $f(n)=O\left(n^k\right)\Leftrightarrow\exists C>0: f(n)\le Cn^k\Leftrightarrow\exists 0\le\lim\limits_{n\to\infty}\frac{f(n)}{n^k}\le C\Leftrightarrow k\ge m$.

   b. $f(n)=\Omega\left(n^k\right)\Leftrightarrow\exists C>0: f(n)\ge Cn^k\Leftrightarrow\exists 0\le\lim\limits_{n\to\infty}\frac{n^k}{f(n)}\le\frac{1}{C}\Leftrightarrow k\le m$.

   c. $f(n)=\Theta\left(n^k\right)\Leftrightarrow\exists c_1,c_2>0: c_1n^k\le f(n)\le c_2n^k\Leftrightarrow\exists c_1\le\lim\limits_{n\to\infty}\frac{f(n)}{n^k}\le c_2\Leftrightarrow k=m$.

8. Аддитивной цепочкой называется последовательность вида $n_1<n_2<\ldots<n_k$, для которой выполнены два условия: 1) $n_1=1$; $n_k=n$ и 2) $\forall i:1<i\le k\ \ \exists s,t:1\le t\le s<i$ такие, что $n_t+n_s=n_i$. Через $l(n)$ обозначим минимальную длину аддитивной цепочки.

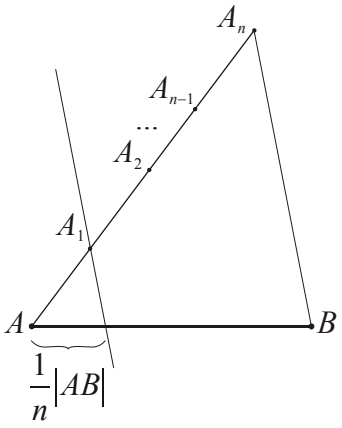
   Доказать: $l(ab)\le l(a)+l(b)-1,\quad a,b\in\mathbf{N}$.

   Решение. Рассмотрим минимальную аддитивную цепочку, последним элементом которой является a: $1=n'_1<\ldots<n'_{l(a)}=a$. Минимальная аддитивная цепочка для b в свою очередь выглядит следующим образом: $1=n''_1<\ldots<n''_{l(b)}=b$. Помножим каждый элемент цепочки для b на a и получим ещё одну цепочку, которая, вообще говоря, не является аддитивной: $a=n'''_1<\ldots<n'''_{l(b)}=ab$. Не трудно видеть, что последний элемент цепочки $\{n'\}$ равен первому элементу цепочки $\{n'''\}$, т.е. $n'_{l(a)}=n'''_1$. Составим из первой и третьей цепочек одну, объединив их по равному элементу: $\underbrace{1=n'_1<\ldots<n'_{l(a)}}_{l(a)}=\overbrace{n'''_1<\ldots<n'''_{l(b)}=ab}^{l(b)}$. Не трудно видеть, что так составленная

последовательность является аддитивной цепочкой, но не обязательно является минимальной. Откуда следует, что $l(ab) \le l(a) + l(b) - 1$.

9. Дан некоторый отрезок и число $n > 1$. Требуется привести алгоритм построения отрезка $\frac{1}{n}$ от данного со сложностью $O(\log n)$.

   Решение. Пусть дан отрезок AB. Для деления отрезка на n воспользуемся теоремой Фалеса. Для этого из одного конца данного отрезка выпустим луч и отложим на нём n раз отрезок произвольной длины (пробный отрезок). Получим набор точек $A_1, A_2, ..., A_n$. После чего проведём прямую, параллельную A_nB, через точку A_2, тогда, согласно теореме, она отсечёт на отрезке AB требуемый отрезок.



   Если откладывать пробный отрезок по одному n раз, то сложность такого алгоритма будет порядка n, и он нам не подойдёт. Чтобы добиться логарифмической сложности следует поступить следующим образом: после отложения 2-х пробных отрезков сразу находим A_4, отложив от A_2 отрезок AA_2. Затем сразу можно найти A_8, отложив два раза отрезок AA_4 и т.д. по степеням двойки. Отсюда получаем логарифмическую сложность, и оценка $O(\log n)$ будет верна.

10. Используя оценку функции Эйлера $\pi(n) > \frac{1}{3}\frac{n}{\ln n}$ доказать, что сложность в среднем алгоритма пробных делений $\overline{T}_{TD}$ не является полиномиальной.

11. Найти сложность в среднем по количеству обменов для алгоритма сортировки выбором, исключая обмены вида $x_k \leftrightarrow x_k$.

12. Дан массив $x_1,...,x_n$. Алгоритм нахождения минимального элемента выглядит следующим образом:

```
m:=x1;
for i=2 to n do
  if xi<m then m:=xi fi
od
```

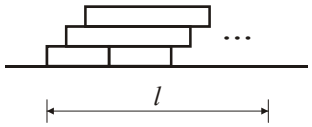
   Найти сложность в среднем по числу присваиваний.

   Решение. Разложим Π_n следующим образом:

$$\Pi_n = A_n^1 + ... + A_n^n,$$

где событие A_n^i заключается в том, что минимальный элемент массива располагается на i-м месте. Не трудно видеть, что $P_n(A_n^i) = \dfrac{(n-1)!}{n!} = \dfrac{1}{n}$. Введём случайную величину ξ_n, показывающую затраты алгоритма, тогда $\overline{T}(n) = \mathbf{E}\xi_n = \sum\limits_{i=1}^{n} \mathbf{E}\left(\xi_n \middle| A_n^i\right) P_n(A_n^i)$. Если событие A_n^i имеет место, то ξ_n при этом принимает значение $\xi_n = i \;\Rightarrow\; \mathbf{E}\left(\xi_n \middle| A_n^i\right) = i \;\Rightarrow$ $\overline{T}(n) = \mathbf{E}\xi_n = \dfrac{1}{n}\sum\limits_{i=1}^{n} i = \dfrac{n+1}{2}$.

13. Имеются кости домино, которые можно класть либо рядом, либо строить навес:



Какой максимальной длины можно построить навес?

14. Дан массив. Описать алгоритм нахождения m-го минимального алгоритма с использованием операции разбиения и исследовать его сложность.

15. Доказать, что количество пар (i, j), попадающих в стек в алгоритме быстрой сортировки (отложенные задачи), меньше длины начального массива n.

16. Доказать, что если считать сложность рандомизированного алгоритма быстрой сортировки в количестве обращений к генератору случайных чисел, то она равна $\Theta(n)$.

Решение. Если воспользоваться формулировкой алгоритма быстрой сортировки с разрезальщиком, то за каждое разрезание он будет брать ровно 1 рубль, и для сложности в среднем получаем: $\overline{T}'_{QS}(n) = \mathbf{E}\chi_n = 1 + \dfrac{1}{n}\sum\limits_{i=1}^{n}\left(\mathbf{E}\chi_{i-1} + \mathbf{E}\chi_{n-i}\right) = 1 + \dfrac{2}{n}\sum\limits_{k=0}^{n-1}\chi_k$.

Обозначим $\overline{T}'_{QS}(n) = \mathbf{E}\chi_n = f(n)$, тогда получим:

$$f(n) = 1 + \frac{2}{n}\sum_{k=0}^{n-1} f(k)$$

$$nf(n) = n + 2\sum_{k=0}^{n-1} f(k) \Rightarrow$$

$$(n-1)f(n-1) = n - 1 + 2\sum_{k=0}^{n-2} f(k)$$

Вычитаем из первого второе и получим: $nf(n) - (n+1)f(n-1) = 1 \;\Leftrightarrow$

$\dfrac{f(n)}{n+1} - \dfrac{f(n-1)}{n} = \dfrac{1}{n(n+1)}$. Введём функцию $\varphi(x) = \dfrac{f(x)}{1+x}$, тогда предыдущее выражение перепишется в виде: $\varphi(n) - \varphi(n-1) = \dfrac{1}{n(n+1)}$, $\varphi(0) = 0 \;\Rightarrow$

$\varphi(n) = \sum\limits_{k=1}^{n} \dfrac{1}{k(k+1)} = \sum\limits_{k=1}^{n}\left(\dfrac{1}{k} - \dfrac{1}{k+1}\right) = 1 - \dfrac{1}{n+1}$, откуда $f(n) = (1+n)\left(1 - \dfrac{1}{n+1}\right) = n = \Theta(n)$.

17. Как реализовать генератор случайных чисел (ГСЧ), чтобы вероятность выпадения числа $0 \le k \le N-1$ была α_k (на базе стандартного ГСЧ выдающего числа из (0,1) с равномерным распределением).

Решение. Так как выпадение чисел от 0 до $N-1$ является полной группой событий, то $\sum_{k=0}^{N-1}\alpha_k = 1$. Пусть стандартный ГСЧ выдаёт число r, тогда если $r \in (0, \alpha_0]$, то конструируемый генератор выдает 0, если $r \in \left(\sum_{k=0}^{n-1}\alpha_k, \sum_{k=0}^{n}\alpha_k\right]$, то конструируемый генератор выдает n.

18. Генератор случайных чисел выдаёт 0 с вероятностью p и 1 с вероятностью $1-p$, где p — неизвестно. Сконструировать такой ГСЧ, чтобы 0 и 1 выпадали с вероятностью $\frac{1}{2}$ на базе первого.

19. Во второй формулировке алгоритма быстрой сортировки разрезальщик берёт 1) n рублей, 2) n^2 рублей. Найти сложность в среднем.

Решение.

1) аналогично задаче 16 введём обозначения: $f(n) = \mathbf{E}\chi_n = \overline{T}(n)$, тогда

$$f(n) = n + \frac{2}{n}\sum_{k=0}^{n-1} f(k) \Rightarrow nf(n) = n^2 + 2\sum_{k=0}^{n-1} f(k) \Rightarrow (n-1)f(n-1) = (n-1)^2 + 2\sum_{k=0}^{n-2} f(k)$$

Вычитаем последнее равенство и предпоследнего и получаем: $nf(n) - (n+1)f(n-1) = 2n-1 \quad \Rightarrow \quad \frac{f(n)}{n+1} - \frac{f(n-1)}{n} = \frac{2n-1}{n(n+1)}$, введя функцию $\varphi(k) = \frac{f(k)}{1+k}$ получим $\varphi(n) - \varphi(n-1) = \frac{2n-1}{n(n+1)}$, $\varphi(0) = 0 \quad \Rightarrow \varphi(n) = \sum_{k=1}^{n}\frac{2k-1}{k(k+1)}$.

Используя факт из математического анализа $\sum_{i=1}^{n}\frac{1}{i} = \ln n + O(1)$, получаем: $\varphi(n) = 2\ln n + O(1) \Rightarrow \overline{T}(n) = f(n) = (1+n)\varphi(n) = 2n\ln n + O(n)$.

2) аналогично пункту 1 получаем:

$$f(n) = n^2 + \frac{2}{n}\sum_{k=0}^{n-1} f(k) \Rightarrow nf(n) = n^3 + 2\sum_{k=0}^{n-1} f(k) \Rightarrow (n-1)f(n-1) = (n-1)^3 + 2\sum_{k=0}^{n-2} f(k)$$

$$\Rightarrow nf(n) - (n+1)f(n-1) = 3n^2 - 3n + 1 \Rightarrow \varphi(n) = \sum_{k=1}^{n}\frac{3k^2 - 3k + 1}{k(k+1)} = 3n - 6\ln n + O(1)$$

$$\Rightarrow \overline{T}(n) = f(n) = (1+n)\varphi(n) = 3n^2 - 6n\ln n + O(n)$$

20. Бинарный алгоритм Евклида: если a_0 и a_1 чётные, то $\text{НОД}(a_0, a_1) = 2\cdot\text{НОД}\left(\frac{a_0}{2}, \frac{a_1}{2}\right)$; если a_0 — чётное, a_1 — нечётное, то $\text{НОД}(a_0, a_1) = \text{НОД}\left(\frac{a_0}{2}, a_1\right)$; если оба нечётные, то максимальное заменяем разностью. Когда одно из чисел станет нулём, тогда второе — НОД. Доказать, что завершается и для количества шагов верна оценка $O(\log a_1)$.

21. При применении алгоритма Евклида к числам $a_0 > a_1$ возникают остатки $a_2, a_3, ..., a_n$, $a_{n+1} = 0$. Доказать, что $a_{n-t} \geq F_{t+1}$, $t = 0, 1, \ldots, n-1$ (теорема Ламе).

22. Из условий предыдущей задачи вывести оценку для числа шагов алгоритма Евклида: $n < \log_\phi a_1 + c$.

23. $m = \nu(a_1)$. Показать, что для числа шагов алгоритма Евклида справедливы оценки: $T_E(m) \le 2m-1$ и $T_{EE}(m) \le 6m+3$.

Решение. Ранее была получена оценка для сложности алгоритма Евклида в следующем виде: $T_E(a_1) = \max\limits_{a_0 \ge a_1} C_E(a_0, a_1) \le 2\nu(a_1) - 1 = 2\lfloor \log_2 a_1 \rfloor + 1$ (см. стр. 18), что в точности соответствует тому, что $T_E(m) \le 2m-1$. Для доказательства $T_{EE}(m) \le 6m+3$ используем полученную ранее оценку $T_{EE}(a_1) < 6\log_2 a_1 + 3$ (см. стр. 19), и очевидный факт $\log_2 a_1 < m = \nu(a_1) = \lfloor \log_2 a_1 \rfloor + 1 \Rightarrow T_{EE}(m) \le 6m+3$.

24. Доказать, что для чисел Фибоначчи справедливо $F_n^2 - F_{n-1}F_{n+1} = (-1)^n$, $n = 1,2,...$

Решение. Докажем по индукции: очевидно, что для $n = 1$ равенство верно; пусть оно верно для $n-1$, т.е. $F_{n-1}^2 - F_{n-2}F_n = (-1)^{n-1}$; докажем для *n*:

$$F_n^2 - F_{n-1}F_{n+1} = \underbrace{\left(F_{n-1} + F_{n-2}\right)}_{F_n}^2 - F_{n-1}\underbrace{\left(F_n + F_{n-1}\right)}_{F_{n+1}} =$$

$$= F_{n-1}^2 + 2F_{n-1}F_{n-2} + F_{n-2}^2 - F_{n-1}F_n - F_{n-1}^2 = 2F_{n-1}F_{n-2} + F_{n-2}^2 - F_{n-1}\underbrace{\left(F_{n-1} + F_{n-2}\right)}_{F_n} =$$

$$= F_{n-1}F_{n-2} + F_{n-2}^2 - F_{n-1}^2 = F_{n-2}\left(F_{n-1} + F_{n-2}\right) - F_{n-1}^2 = F_{n-2}F_n - F_{n-1}^2 = -(-1)^{n-1} = (-1)^n$$

25. Доказать, что уравнение $ax + by = c$ имеет решения в целых числах тогда и только тогда, когда $d = \text{НОД}(a,b)$ делит c . Доказать, что если x_0, y_0 решение, то все решения описываются формулами $x = x_0 + \dfrac{b}{d}t$, $y = y_0 - \dfrac{a}{d}t$, $t = 0,\pm1,\pm2,...$

26. Доказать, что s_i и t_i $\left(i = 0,1,...,n+1\right)$ из расширенного алгоритма Евклида взаимно простые.

Указание:

$$\begin{bmatrix} -q_1 & 1 \\ 1 & 0 \end{bmatrix} \times ... \times \begin{bmatrix} -q_{i-1} & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} t_i & t_{i-1} \\ s_i & s_{i-1} \end{bmatrix}$$

27. Показать, что для бинарного поиска количество сравнений нельзя однозначно определить по *n*, но есть такие *n* (бесконечно много), для которых можно ($\lfloor \log_2 n \rfloor + 1$).

28. Есть плоскость и ортонормированный базис (x, y). $P_1, P_2, ..., P_n$ — заданный *n*-угольник. Требуется привести алгоритм определения, принадлежит ли точка Q многоугольнику, со сложностью $\Theta(\log n)$.

29. Имеется 3^n монет, одна из которых фальшивая. Есть чашечные весы. Известно, что фальшивая монета легче остальных. За *n* взвешиваний требуется найти фальшивую. Алгоритм решения: на каждом этапе алгоритма все монеты делятся на 3 равные части, веса первых двух из которых сравниваются с помощью весов. Если веса равны, то фальшивая монета в третьей части и алгоритм повторяется для неё. Если при взвешивании оказывается, что одна из частей весит меньше, то фальшивая монета необходимо там, следовательно, алгоритм применяется повторно уже к части с фальшивой монетой. Очевидно, что если изначально монет 3^n , то за *n* взвешиваний фальшивая монета определяется однозначно. Требуется обобщить алгоритм на

произвольное число монет: разбиение производится на $\left\lceil \frac{n}{3} \right\rceil$, $\left\lceil \frac{n}{3} \right\rceil$ и $n-2\left\lceil \frac{n}{3} \right\rceil$, и взвешиваются первые две. Показать, что сложность по числу взвешиваний в этом случае составит $\lceil \log_3 n \rceil$.

30. Ситуация как в предыдущей задаче, но разделение происходит на $\left\lfloor \frac{n}{3} \right\rfloor$, $\left\lfloor \frac{n}{3} \right\rfloor$ и $n-2\left\lfloor \frac{n}{3} \right\rfloor$. Показать, что $\exists n$, для которых потребуется $> \lceil \log_3 n \rceil$ взвешиваний.

Решение. Рассмотрим $n = 3^k - 1,\ k \geq 2$. Для таких n получаем $\lceil \log_3 n \rceil = k$. Посмотрим, сколько взвешиваний потребуется алгоритму. На первом шаге все монеты разделятся на три части: $\left(\left\lfloor \frac{n}{3} \right\rfloor, \left\lfloor \frac{n}{3} \right\rfloor, n-2\left\lfloor \frac{n}{3} \right\rfloor \right) = \left(3^{k-1}-1, 3^{k-1}-1, 3^{k-1}+1\right)$. Пусть фальшивая монета оказывается всегда в третьей части, тогда на втором шаге получаем следующие части: $\left(\left\lfloor \frac{3^{k-1}+1}{3} \right\rfloor, \left\lfloor \frac{3^{k-1}+1}{3} \right\rfloor, 3^{k-1}+1-2\left\lfloor \frac{3^{k-1}+1}{3} \right\rfloor \right) = \left(3^{k-2}, 3^{k-2}, 3^{k-2}+1\right)$ и т.д. На k-м шаге монеты разбиваются на следующие части: $(1, 1, 2)$, где фальшивая монета находится в третье группе, а значит необходимо для её выявления ещё одно взвешивание. Итого получаем $k+1 > \lceil \log_3 n \rceil = k$ взвешиваний.

31. Имеется конечная последовательность из нулей и единиц. Разрешается заменять группу 01 на $1\underbrace{0\ldots0}_{\substack{\text{сколько} \\ \text{угодно}}}$. Доказать, что алгоритм завершается.

32. Чему равно r?

```
r:=0;
for i=1 to n do
  for j=i+1 to n do
    for k=i+j to n do
      r:=r+1;
    od
  od
od
```

Решение. Не трудно видеть, что можно изменить правые границы циклов, следующим образом:

```
r:=0;
for i=1 to ⌊(n-1)/2⌋ do
  for j=i+1 to n-i do
    for k=i+j to n do
      r:=r+1;
    od
  od
od
```

Это не изменит семантики программы, но теперь каждый цикл проработает указанное количество раз. Обозначим $x = \left\lfloor \frac{n-1}{2} \right\rfloor$ и составим сумму:

$$r = \sum_{i=1}^{x}\sum_{j=i+1}^{n-i}\sum_{k=i+j}^{n} 1 = \sum_{i=1}^{x}\sum_{j=i+1}^{n-i}\left[n-(i+j)+1\right] = \sum_{i=1}^{x}(n-2i)\frac{n+1-2i}{2} = \sum_{i=1}^{x}\left(\frac{n(n+1)}{2} - i(2n+1) + 2i^2\right)$$

Из математического анализа известна формула: $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$ (устанавливается по индукции). Применим её и окончательно получим:

$$r = \sum_{i=1}^{x}\left(\frac{n(n+1)}{2} - i(2n+1) + 2i^2\right) = x\cdot\frac{n(n+1)}{2} - \frac{1+x}{2}\cdot x(2n+1) + \frac{x(x+1)(2x+1)}{3}$$

33. Для задачи 4 доказать, что $f(n) = 3n-1$ является нижней границей.

    Решение. Худший случай работы алгоритма сортировки заключается в максимальной загруженности тупика. Очевидно, что в отдельно взятый момент времени вагоны в тупике имеют одинаковые цвета (вагон отправляется в тупик, когда его цвет совпадает с цветом последнего вагона формируемого состава, но тогда цвет вагонов в тупике должен совпадать с цветом текущего вагона, т.к. в противном случае была бы возможность операции **ИЗ**). Максимальная загрузка тупика возможна на $n-1$ вагон, т.к. один вагон в любом случае должен быть отправлен **МИМО**. Для доказательства возможности загрузки тупика на $n-1$ вагон приведём пример: исходный состав состоит из n вагонов одного цвета подряд, после которых следуют n вагонов другого цвета. В этой ситуации первый вагон перегоняется **МИМО**, а остальные $n-1$ вагон попадают в тупик. Тогда для нижней границе получаем: 1 операция **МИМО**, $n-1$ операций **В**, каждая из которых в конечном итоге сопровождается операцией **ИЗ**, и оставшиеся n операций **МИМО**. Итого получаем: $1+2\cdot(n-1)+n = 3n-1$.

34. Для задачи 5 доказать, что алгоритм с $O(n)$ является оптимальным по порядку сложности.

    Решение. Т.к. любой алгоритм поиска двери допускает оценку $\Omega(n)$, то алгоритм с оценкой $O(n)$ будет являться оптимальным по порядку сложности.

35. Показать, что для сложности алгоритма «наивного» умножения оценка $\Theta\left(2^{\max\{m_a, m_b\}}\max\{m_a, m_b\}\right)$ не верна.

36. Верна ли для алгоритма «наивного» деления оценка $T^{**}_{ND}(m_a, m_b) = \Theta(m_a m_b)$?

37. Имеется массив из попарно различных элементов и требуется найти медиану — элемент, меньше и больше которого одинаковое число элементов (в случае чётного числа элементов возможно отличие на 1). Доказать, что $n-1$ является нижней границей для сложности алгоритмов, решающих задачу.

38. Бинарный алгоритм возведения в степень (a^n) не является оптимальным, если в качестве размера входа брать n. Будет ли он оптимальным, если в качестве размера входа взять $\nu(n)$.

39. Имеется отрезок и число n. Требуется разбить отрезок на n частей. Показать, что нижняя граница сложности для алгоритмов, решающих эту задачу, составляет $\frac{n-1}{2}$.

40. Алгоритм вычисляет следующую сумму: $1+2+\ldots+n$. n — размер входа. Тщательно исследовать алгоритм на сложность.

41. Пусть алгоритм вычисляет F_n (число Фибоначчи) через $n-1$ сложение. Доказать, что сложность такого алгоритма допускает оценку: $O(n^2)$.

Решение. Битовая длина числа Фибоначчи

$\nu(F_n) = \nu(F_{n-1} + F_{n-2}) \le \{F_{n-1} > F_{n-2}\} \le \nu(F_{n-1} + F_{n-1}) = \nu(2F_{n-1}) = 1 + \nu(F_{n-1})$, $\nu(F_1) = 1 \quad \Rightarrow$ $\nu(F_n) \le n$. Известна оценка битовой сложности для сложения: $T^*_{Add}(m) = \Theta(m)$, поэтому $T(n) = (n-1)O(n) = O(n^2)$.

42. Пусть p — простое, $u \in \mathbf{N}$, $u < p$. Показать, что $C_p^u \equiv 0 (\mathrm{mod}\ p)$.

Решение. $C_p^u = \dfrac{p!}{u!(p-u)!} = \dfrac{p(p-1)\cdot\ldots\cdot(p-u+1)}{u!}$. Т.к. число сочетаний является целым числом, то все множители знаменателя обязаны сократиться с множителями числителя, а т.к. p — простое, то в результате сокращения в числители оно останется $\Rightarrow$ $C_p^u \equiv 0 (\mathrm{mod}\ p)$.

43. Используя задачу 42 и индукцию по u, доказать утверждение: если p — простое, $u \in \mathbf{N}$ $\Rightarrow$ $u^p \equiv u (\mathrm{mod}\ p)$. Если дополнительно p не делит u, то $u^{p-1} \equiv 1 (\mathrm{mod}\ p)$.

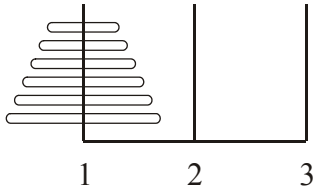
Решение. Проведём индукцию по u: для $u = 1$ очевидно, что $u^p \equiv u (\mathrm{mod}\ p)$. Пусть теперь утверждение верно для $u-1$, т.е. $(u-1)^p \equiv u - 1 (\mathrm{mod}\ p)$, и докажем его для u: $u^p = (1 + (u-1))^p$, согласно формуле бинома Ньютона, последнее перепишется в виде: $(1 + (u-1))^p = 1 + \sum_{i=1}^{p-1} C_p^i (u-1)^i + (u-1)^p$. $C_p^i \equiv 0 (\mathrm{mod}\ p)$ (задача 42), $(u-1)^p \equiv u - 1 (\mathrm{mod}\ p)$ (предположение индукции), откуда следует $u^p \equiv u (\mathrm{mod}\ p)$.

44. Доказать, что если $a, b \in \mathbf{Z}$, p — простое, $k \in \mathbf{N}$, то $(a+b)^{p^k} \equiv a^{p^k} + b^{p^k} (\mathrm{mod}\ p)$.

Решение. $C_{p^k}^i = \dfrac{p^k(p^k-1)\cdot\ldots\cdot(p^k-i+1)}{i!}$, т.к. число сочетаний является целым числом, то все сомножители знаменателя должны необходимо сократится с множителями числителя, но при $i < p^k$ все множители знаменателя $< p^k$; p — простое число, поэтому хотя бы одно p в числителе останется. Тогда $C_{p^k}^i \equiv 0 (\mathrm{mod}\ p)$.

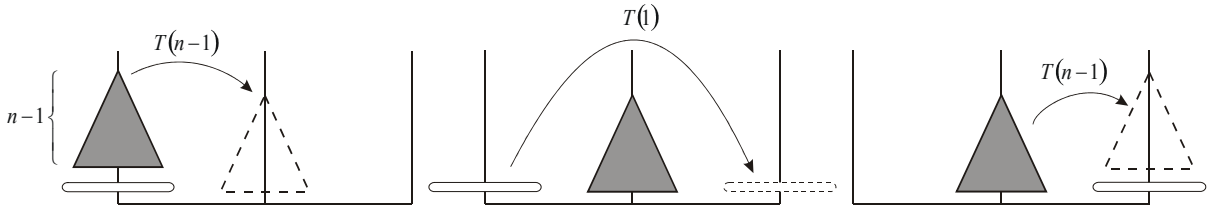
$$(a+b)^{p^k} = a^{p^k} + \sum_{i=1}^{p^k-1} \underbrace{C_{p^k}^i a^i b^{p^k-i}}_{\equiv 0 (\mathrm{mod}\ p)} + b^{p^k} \Rightarrow (a+b)^{p^k} \equiv a^{p^k} + b^{p^k} (\mathrm{mod}\ p).$$

45. *Ханойские башни.* Имеется 3 шеста, на один из которых одето n дисков разного радиуса: в самом низу большой, вверху — самый маленький. Требуется перенести все диски с первого шеста на 2, за раз перекладывая только один диск. Запрещается класть диск большего радиуса на диск меньшего.



Подсчитать цену решения (через рекуррентное соотношение), в случае, если за одно перекладывание мы платим: 1) 1 ед.; 2) i ед. (где i — порядок номера диска считая сверху в начальной пирамиде); 3) i^2; 4) 2^i.

Решение. Алгоритм решения изобразим следующим образом:



$$\Rightarrow \begin{matrix} T(n) = 2T(n-1) + T(1) \\ T(0) = 0 \end{matrix}$$

1) $T(1) = 1 \quad \Rightarrow \quad \begin{cases} T(n) = 2T(n-1) + 1 \\ T(0) = 0 \end{cases}$ или $\begin{cases} T(n) - 2T(n-1) = 1 \\ T(0) = 0 \end{cases}$. Решаем полученное рекуррентное соотношение: характеристическим уравнением для него будет $\lambda - 2 = 0 \Leftrightarrow \lambda = 2$, и общим решением однородного рекуррентного уравнения будет $T^{\text{о.о.}}(n) = C \cdot 2^n$. В правой части стоит квазиполином нулевой степени, поэтому частное решение неоднородного уравнения будем искать в виде $T^{\text{ч.н.}}(n) = a$. Подставляя частное решение в исходное уравнение, определяем константу $a = -1$, следовательно, окончательное решение рекуррентного уравнения будет иметь вид: $T(n) = C \cdot 2^n - 1$. Константу C определяем из начальных условий: $T(0) = C - 1 = 0 \Rightarrow C = 1 \Rightarrow T(n) = 2^n - 1$.

2) $T(1) = i \Rightarrow \begin{cases} T(n) = 2T(n-1) + n \\ T(0) = 0 \end{cases}$. Решение будет аналогично пункту 1), кроме того, что частное решение в этом случае надо будет искать в виде $T^{\text{ч.н.}}(n) = an + b$, т.к. в правой части стоит квазиполином первой степени: $n = n \cdot 1^n$. Подставляя частное решение в исходное уравнение, получаем:

$$an - b - 2\big(a(n-1) - b\big) = n \Leftrightarrow -an - b + 2a = n$$

Приравнивая коэффициенты при одинаковых степенях n, получаем:

$$\begin{matrix} n^1 \\ n^0 \end{matrix} \left| \begin{cases} -a = 1 \\ 2a = b \end{cases} \Leftrightarrow \begin{cases} a = -1 \\ b = -2 \end{cases} \Rightarrow T^{\text{ч.н.}}(n) = -n + 2 \right.$$

Общим решением неоднородного рекуррентного соотношения будет:

$$T(n) = T^{\text{о.о.}}(n) + T^{\text{ч.н.}}(n) = C \cdot 2^n - (n+2)$$

Из начального условия определяем константу C: $T(0) = C - 2 = 0 \Rightarrow C = 2 \quad \Rightarrow$ $T(n) = 2^{n+1} - (n+2)$.

2) $T(1) = i^2 \quad \Rightarrow \quad \begin{cases} T(n) = 2T(n-1) + n^2 \\ T(0) = 0 \end{cases}$. Решается аналогично предыдущему пункту, только на этот раз в правой части стоит квазиполином второй степени, поэтому частное решение будем искать в виде $T^{\text{ч.н.}}(n) = an^2 + bn + c$. Подставляем его в начальное уравнение:

$$an^2 + bn + c - 2\Big[a\big(n-1\big)^2 + b(n-1) + c\Big] = n^2 \Leftrightarrow$$

$$-an^2 + (4a - b)n + 2b - 2a - c = n^2 \Rightarrow \begin{matrix} n^2 \\ n^1 \\ n^0 \end{matrix} \left| \begin{cases} -a = 1 \\ 4a - b = 0 \\ 2b - 2a - c = 0 \end{cases} \Leftrightarrow \begin{cases} a = -1 \\ b = -4 \\ c = -6 \end{cases} \right.$$

$$\Rightarrow T^{\text{ч.н.}}(n) = -(n^2+4n+6) \Rightarrow T(n) = C\cdot 2^n - (n^2+4n+6)$$

$$T(0) = C - 6 = 0 \Rightarrow C = 6 \Rightarrow T(n) = 6\cdot 2^n - (n^2+4n+6).$$

3) $T(1) = 2^i \Rightarrow \begin{cases} T(n) = 2T(n-1) + 2^n \\ T(0) = 0 \end{cases}$. Общее решение неоднородного уравнения ищется аналогично предыдущим пунктам, но поиск частного решения в этом случае будет отличаться, т.к. в этом случае мы имеем в правой части квазиполином 2^n, основание степени которого совпадает с корнем характеристического уравнения. Согласно теории, в этом случае частное решение следует искать в виде $T^{\text{ч.н.}}(n) = an\cdot 2^n$. Подставляя его в исходное уравнение, получаем, что константа a может быть любой. Положим $a = 1$, тогда общее решение неоднородного рекуррентного уравнения запишется в виде: $T(n) = (C+n)\cdot 2^n$. Из начального условия получаем $C = 0$ и $T(n) = n\cdot 2^n$.

46. (Китайская теорема об остатках) Имеется несколько взаимно простых чисел больших единицы: $k_1, k_2, \dots, k_n$, и имеются остатки ($b_1, b_2, \dots, b_n$)от деления некоторого числа f на $k_1, k_2, \dots, k_n$ соответственно. Требуется восстановить число f. Очевидно, что если f — решение, то $f + k_1k_2\dots k_n$ тоже будет решением.

Решение. Пусть $b_1, b_2, \dots, b_n$ — известные остатки, и требуется найти f такое, что $f \equiv b_i \pmod{k_i}$, $i = 1, 2, \dots, n$. Обозначим: $k = k_1k_2\dots k_n$ — произведение модулей, $g_i = \dfrac{k}{k_i}$, $i = 1, 2, \dots, n$ — произведение всех модулей, кроме *i*-го, тогда $g_i \perp k_i$ (взаимно просты) в силу того, что k_i взаимно простые $\Rightarrow$ по расширенному алгоритму Евклида найдём h_i такие, что $h_ig_i \equiv 1 \pmod{k_i}$. Тогда можно положить $f = \sum_{i=1}^{n} b_ig_ih_i$.

47. Найти оценки сложностей, если для них выполняются следующие рекуррентные соотношения:

1) $T(n) = T\left(\left\lfloor \dfrac{n}{2} \right\rfloor\right) + \log_2 n$

2) $T(n) = 2T\left(\left\lfloor \dfrac{n}{2} \right\rfloor\right) + n\log_2 n$

3) $T(n) = 2T\left(\left\lfloor \dfrac{n}{2} \right\rfloor\right) + n^2$

Решение. 1) используя предложение (стр. 40) получаем ассоциированное уравнение

$$t(k) = \begin{cases} 0, k = 0 \\ t(k-1) + k, k > 0 \end{cases}$$

решением которого будет $t(k) = \dfrac{k(k+1)}{2} \Rightarrow$

$$\frac{\lfloor \log_2 n \rfloor (\lfloor \log_2 n \rfloor + 1)}{2} \le T(n) \le \frac{\lceil \log_2 n \rceil (\lceil \log_2 n \rceil + 1)}{2} \Rightarrow T(n) = \Theta\left(\log^2 n\right).$$

2) используя предложение (стр. 40) получаем ассоциированное уравнение

$$t(k) = \begin{cases} 0, k = 0 \\ 2\cdot t(k-1) + k\cdot 2^k, k > 0 \end{cases} \Leftrightarrow \begin{cases} t(k) - 2\cdot t(k-1) = k\cdot 2^k \\ t(0) = 0 \end{cases}$$

Характеристическим уравнением будет $\lambda - 2 = 0 \Rightarrow \lambda = 2$ поэтому общим решением однородного рекуррентного уравнения будет $t^{\text{о.о.}}(k) = C \cdot 2^k$. Частное решение неоднородного рекуррентного уравнения будем искать в виде $t^{\text{ч.н.}}(k) = k^r p(k) \cdot 2^k$, где r соответствует количеству совпадений основания степени из правой части ($k \cdot 2^k$), которое в нашем случае равно 2, с корнями характеристического уравнения. В нашем случае $r = 1$. Степень многочлена $p(k)$ совпадает со степенью многочлена из правой части, в нашем случае $\deg p(k) = \deg k = 1$, т.е. $p(k) = ak + b$ где a и b — константы, подлежат определению. Таким образом, частное решение будем искать в виде $t^{\text{ч.н.}}(k) = k(ak + b) \cdot 2^k$. Подставим его исходное уравнение:

$$k(ak+b)\cdot 2^k - 2\cdot(k-1)\big(a(k-1)+b\big)\cdot 2^{k-1} = k\cdot 2^k \;\Leftrightarrow\; ak^2 + bk - ak^2 + ak - bk + ak - a + b = k$$

$$\Leftrightarrow\; 2ak - a + b = k \;\Rightarrow\; \begin{matrix} k^1: \\ k^0: \end{matrix} \begin{cases} 2a = 1 \\ -a + b = 0 \end{cases} \Leftrightarrow \begin{cases} a = 1/2 \\ b = 1/2 \end{cases} \Rightarrow\; t^{\text{ч.н.}}(k) = k(k+1)\cdot 2^{k-1}$$

Общее решение неоднородного рекуррентного уравнения будет складываться из общего решения однородного и частного неоднородного: $t(k) = t^{\text{о.о.}}(k) + t^{\text{ч.н.}}(k) \;\Rightarrow$ $t(k) = C \cdot 2^k + k(k+1) \cdot 2^{k-1}$. Из начальных условий определяем C: $t(0) = 0 \;\Rightarrow\; C = 0 \;\Rightarrow$ $t(k) = k(k+1) \cdot 2^{k-1}$. Откуда для сложности, согласно предложению, получаем оценку:

$$\frac{\lfloor \log_2 n \rfloor \left(1 + \lfloor \log_2 n \rfloor\right)}{2} \cdot 2^{\lfloor \log_2 n \rfloor} \le T(n) \le \frac{\lceil \log_2 n \rceil \left(1 + \lceil \log_2 n \rceil\right)}{2} \cdot 2^{\lceil \log_2 n \rceil} \;\Rightarrow\; T(n) = \Theta\left(n \log^2 n\right).$$

3) ассоциированное уравнение

$$\begin{cases} t(k) - 2 \cdot t(k-1) = 2^{2k} \\ t(0) = 0 \end{cases}$$

Корнем характеристического уравнения будет $\lambda = 2 \;\Rightarrow\; t^{\text{о.о.}}(k) = C \cdot 2^k$. $\lambda = 2$ не совпадает с основанием степени в правой части: $2^{2k} = \left(2^2\right)^k = 4^k$, поэтому частное решение будем искать в виде $t^{\text{ч.н.}}(k) = p(k) \cdot 4^k$, где $\deg p(k) = 0 \Leftrightarrow p(k) = a$, т.е. $t^{\text{ч.н.}}(k) = a \cdot 4^k$. Для определения константы a, подставляем частное решение в исходное уравнение:

$$a \cdot 4^k - 2a \cdot 4^{k-1} = 4^k \;\Leftrightarrow\; a - \frac{a}{2} = 1 \;\Rightarrow\; a = 2 \;\Rightarrow\; t^{\text{ч.н.}}(k) = 2 \cdot 4^k = 2^{2k+1}$$

Общим решением рекуррентного уравнения будет $t(k) = C \cdot 2^k + 2^{2k+1}$. Из начального условия получаем: $t(0) = 0 \;\Rightarrow\; C + 2 = 0 \;\Rightarrow\; C = -2 \;\Rightarrow\; t(k) = 2^{2k+1} - 2^{k+1}$. Поэтому для сложности получаем оценку: $t\left(\lfloor \log_2 n \rfloor\right) \le T(n) \le t\left(\lceil \log_2 n \rceil\right) \;\Leftrightarrow$

$$2 \cdot \left(2^{\lfloor \log_2 n \rfloor}\right)^2 - 2 \cdot 2^{\lfloor \log_2 n \rfloor} \le T(n) \le 2 \cdot \left(2^{\lceil \log_2 n \rceil}\right)^2 - 2 \cdot 2^{\lceil \log_2 n \rceil} \;\Rightarrow\; T(n) = \Theta\left(n^2\right).$$