Оглавление

Предисловие

Эта книжка написана по мотивам материалов одноименного курса кафедры моу на Физтехе, записанных Димой Голубенко, Лешей Крошниным и Эдом Горбуновым. Курс вел Сергей Тарасов, и он же придумал концепцию, сильно отличающую «Алгоритмы и модели вычисления» от других аналогичных курсов. Стандартный курс по алгоритмам — обзор основных алгоритмов(быстрая сортировка, поиск медианы, обходы графов) и структур данных, необходимых каждому программисту. Нетрудно видеть, что почти все эти основные алгоритмы возникли во время решения некоторых математических задач. Читая «Алгоритмы и модели вычисления», Сергей в первую очередь стремился продемонстрировать связь алгоритмов и математики, какие алгоритмы и как позволяют решать математические задачи (из самых разных областей, будь то теория чисел или топология) и какая математика лежит в основе тех или иных алгоритмов. Теория алгоритмов возникла естественным образом из вычислительных задач в разных областях математики и по сей день остается живой областью, в которой некоторые тривиально сформулированные вопросы до сих пор открыты. В этой книжке мы расскажем математическим языком об основных алгоритмах сортировки, алгебры, теории чисел и теории графов.

Многие школьники изучают алгоритмы, готовясь к олимпиадам по программированию. Возможно, что эта книга поможет математикам-олимпиадникам, не занимавшимся алгоритмами, заинтересоваться олимпиадным программированием, а может быть — и теорией алгоритмов.

Авторы благодарят Сергея Тарасова, собравшего необычную комнаду семиинаристов и придумавшего оригинальную концепцию курса, а также сотрудников Центра Развития ИТ-Образования и лично Татьяну Бабичеву, чьи доброта, отзывчивость и усилия сделали существование этой книжки возможным.

Часть 1

Введение

Со школы каждому знакомо интуитивное определение алгоритма — «набор инструкций, описывающих порядок действий исполнителя для достижения некоторого результата». Считается, что алгоритм принимает на вход некоторый *конструктивный объект*, выполняет определенную последовательность действий и возвращает (если останавливается) новый конструктивный объект. Конструктивный объект — тот, для которого существует конструктивный способ задать его, например, натуральное число, матрица или граф.

Такое понятие алгоритма довольно расплывчато и не позволяет конкретно ответить на такие вопросы:

- *любая ли задача разрешима алгоритмически?*
- *если некоторая задача не разрешается никаким алгоритмом, как это доказать?* Иными словами, как показать, что любой алгоритм разрешает задачу, отличную от данной?
- *как вычислить время, совершенных алгоритмом, или требуемую память?* Нельзя вычислить время работы алгоритма, просто рассмотрев некоторую его реализацию (например, на языке C), и сложив количества выполений каждой операции. Команды не унитарны ни по времени, ни по ресурсоемкости, время выполнения одной и той же команды может быть разным. Лучшее, что можно сделать в данном случае — оценить (если возможно) время работы команды (в худшем случае) сверху некоторой константой. Но тут важно знать, какие операции мы считаем элементарными и у каких операций время можно оценить сверху константой (в частности, сложение чисел нельзя a priori считать операцией, выполянемой за не более чем постоянное время — если мы не считаем, например, что сами слагаемые ограничены сверху).
- *как показать, что данная задача не может быть решена точно за относительно малое время?* Само по себе существование некоторого алгоритма не гарантирует, что нельзя построить более быстрый алгоритм, разрешающий ту же задачу.

Четкие ответы на эти вопросы требуют наличия логического понятия алгоритма. В этой книге мы дадим точное понятие алгоритма в разделе, посвященном теории сложности. Для практических нужд обычно достаточно интуитивного определения алгоритма; каждый раз мы будем оговаривать, что является единицей памяти и какие операции *элементарны*, то есть выполняются за не более чем постоянное время.

Большинство алгоритмов, с которыми мы будем иметь дело, являются *детерминированными*, то есть шаги, которые они делают, а так же результат, который они выдают, зависят только от входа алгоритма. Однако в ряде случаев можно использовать *вероятностные* процедуры, вместо детерминированных. Делается это прежде всего для того, чтобы уменьшить время работы алгоритма. Конечно, за это приходится платить тем, что

решение задачи получается неточным или правильный ответ даётся с некоторой вероятностью. Часто бывает, что приходится делать больше итераций, но стоимость одной итерации становится гораздо «дешевле». Стоит отметить, что вероятностные алгоритмы важны не только с теоретической точки зрения, но и с практической, так как некоторые вероятностные процедуры очень хорошо себя зарекомендовали на практике. Например, некоторые люди, которые занимаются глубоким обучением (Deep Learning), под словами «градиентный спуск» (что бы это не значило) часто понимают его стохастическую версию, ввиду её высокой практической ценности. Так что полезно буквально с первых шагов привыкнуть к языку теории вероятностей, а при разработке и анализе алгоритмов к случайности следует относиться как к потенциально важному ресурсу. Но осознать, в чем, собственно, этот ресурс заключается, можно только решая задачи.

Как оценивать время вероятностных процедур? Существует 2 основных подхода.

(1) Говорят, что алгоритм на входе размера n работает *в худшем* случае время $T(n)$, если на любом входе размера n он работает время $\leqslant T(n)$, причём равенство достигается.
(2) Говорят, что алгоритм на входе размера n работает *в среднем* время $T(n)$, если математическое ожидание времени его работы на входе размера n равняется $T(n)$.

Рассмотрим следующую задачу. На вход подаётся массив из $2N$ букв, причём половину из них составляют буквы a, а другую половину — буквы b. Требуется найти номер любой ячейки, в которой лежит буква a. Рассмотрим 2 идеологии вероятностного решения такой задачи.

Алгоритмом типа Лас-Вегас называется вероятностный алгоритм, который всегда на выходе даёт корректный результат. Рассмотрим для примера следующий алгоритм.

Require: Массив букв A размера N
1: **repeat**
2: Случайно равновероятно выбрать индекс $i \in \{1, 2, \ldots, N\}$
3: **until** $A[i] \neq a$
4: **return** i

Время работы в худшем случае этого алгоритма равно бесконечности. Действительно, есть ненулевая вероятность, что на каждом шаге алгоритм будет выбирать ячейку, в которой лежит буква b, а значит, мы не можем ограничить количество итераций. Однако алгоритм останавливается за конечное число шагов с вероятностью 1. Более того, среднее число выполненных итераций равно

$$A \stackrel{\text{def}}{=} \sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=1}^{\infty} \frac{i+1}{2^i} - \sum_{i=1}^{\infty} \frac{1}{2^i} = 2\left(\sum_{i=1}^{\infty} \frac{i}{2^i} - \frac{1}{2}\right) - 1 = 2\left(A - \tfrac{1}{2}\right) - 1 = 2A - 2 \Rightarrow A = 2,$$

то есть алгоритм *в среднем делает 2 итерации.*

Алгоритмом типа Монте-Карло называется вероятностный алгоритм, который может давать на выходе неправильный результат с некоторой (как правило маленькой) вероятностью. Рассмотрим для примера следующий алгоритм.

Require: Массив букв A размера N; натуральное число k
1: $i := 0$
2: **repeat**
3: Случайно равновероятно выбрать индекс $i \in \{1, 2, \ldots, N\}$
4: **until** $k = i$ или $A[i] \neq a$
5: **return** i

Время работы в худшем случае этого алгоритма равно k. Вероятность получить правильный ответ равна $1 - \frac{1}{2^k}$. Более того, среднее число выполненных итераций равно

$$B \stackrel{\text{def}}{=} \sum_{i=1}^{k} \frac{i}{2^i} = \sum_{i=1}^{k} \frac{i+1}{2^i} - \sum_{i=1}^{k} \frac{1}{2^i} = 2\left(\sum_{i=1}^{k} \frac{i}{2^i} - \frac{1}{2} - \frac{k}{2^k}\right) - 1 = 2\left(B - \frac{1}{2} + \frac{k+1}{2^{k+1}}\right) - 1,$$

следовательно,

$$B = 2 - \frac{k+1}{2^{k+1}},$$

то есть алгоритм *в среднем делает меньше 2-х итераций.*

Псевдослучайные числа. Отдельного внимания заслуживает вопрос эмуляции генератора случайных чисел на практике. Компьютер — детерминированная арифметическая машина, и любой алгоритм на нем даст некоторую определенную, не случайную, последовательность чисел. Лучшее, что можно сделать — сгенерировать последовательность чисел, которая внешне «похожа» на случайную. Отдельная проблема — дать формальное определение псевдослучайной числовой последовательности: нужно определиться с тем, какое мерило случайности мы используем.

Определение 1.1. Пусть $(x_n)_{n\in\mathbb{N}}$ — последовательность остатков по модулю m. Будем говорить, что $(x_n)_{n\in\mathbb{N}}$ *k-распределена*, если для любой последовательности $a_0, \ldots a_{k-1} \in \mathbb{Z}/m\mathbb{Z}$ и для любого $n \in \mathbb{N}$ выполнено

$$\mathbb{P}(x_n = a_0, x_{n+1} = a_1 \ldots x_{n+k-1} = a_{k-1}) = \frac{1}{m^k}$$

Последовательность $(x_n)_{n\in\mathbb{N}}$ называется *∞-распределенной*, если она k-распределена для любого натурального k.

Это интуитивное понятие псевдослучайной последовательности не совсем точно, подробности можно посмотреть в секции 3.5 книги [31].

Простейший способ генерировать псевдослучайные числа — использовать линейные реккуренты, то есть условия вида

$$f_n = a_1 f_{n-1} + \ldots + a_k f_{n-k} \bmod m$$

При некоторых значениях $a_1, \dots a_k$ и m полученные f_n действительно будут «случайными». Функция `rand` в C — реализация именно такого генератора случайных чисел.

Пример 1.2. Рассмотрим $f_n = F_n \bmod m$ — числа Фибоначчи по модулю некоторого натурального m. Нетрудно заметить, что эта последовательность периодична: поскольку $f_n = (f_{n-1} + f_{n-2}) \bmod m$, то различных f_n не больше, чем возможных пар остатков по модулю m, то есть не более, чем m^2. На самом деле верна более точная оценка *периода Пизано* $\pi(m)$ — периода последовательности $F_n \bmod m$: можно убедиться в том, что $\pi(n) \leqslant 6n$.

Для $m = 2$, например, последовательность $(f_n)_{n\in\mathbb{N}}$ есть просто $0, 1, 1, 0, 1, 1, \dots$, период имеет длину 3. Для $m = 11$ получим

$$0, 1, 1, 2, 3, 5, 8, 2, 10, 1, 0, 1, 1, \dots,$$

период имеет длину 10, а при $m = 10$ последовательность имеет вид

$$0, 1, 1, 2, 3, 5, 8, 3, 1, 4, 5, 9, 4, 3, 7, 0, 7, 7, 4, 1, 5, 6, 1, 7, 8, 5, 3, 8, 1, 9, 0, 9, 9, 8, 7, 5, 2, 7, 9, 6, 5, 1, 6,$$
$$7, 3, 0, 3, 3, 6, 9, 5, 4, 9, 3, 2, 5, 7, 2, 9, 1, 0, 1, \dots$$

то есть период имеет длину 60.

Еще одно важное требование к генератору псевдослучайных чисел — криптоустойчивость, то есть устойчивость к предсказанию механизма этого генератора. Генератор псевдослучайных чисел, генерирующий последовательность по формуле $f_n = af_{n-1} + b \bmod m$ для взаимнопростых a и m, не является криптоустойчивым: поскольку

$$f_{n+1} - f_n = a(f_n - f_{n-1}) \bmod m, \quad f_{n+2} - f_{n+1} = a(f_{n+1} - f_n) \bmod m,$$

то

$$m | (f_{n+1} - f_n)^2 - (f_n - f_{n-1})(f_{n+2} - f_{n+1})$$

Далее можно построить алгоритм, угадывающий a, b и m. По формуле выше можно угадать m, а затем подобрать некоторым образом a и b, а затем далее запустить генератор. Вполне возможно, что сами a, b и m не будут угаданы, но полученный генератор будет угадывать ту же самую последовательность. Бояр (см. [5]) построила полиномиальный алгоритм, позволяющий таким образом взламывать генератор.

В наши дни наиболее популярным генератором псевдослучайных чисел является *твистер Мерсенна*.

Короче говоря, как говорил Ричард Ковейю (Richard Coveyu): «Генерирование псевдослучайных чисел слишком важная вещь, чтобы предоставить ее случаю». Подробнее о псевдослучайных числах можно почитать в [31] и [12].

Асимптотические оценки. Метод Акра-Баззи

При теоретическом анализе сложности алгоритма, как правило, нас не интересуют конкретные константы в функциях, а только их асимптотики при длине входа $|x| \to \infty$.

- Функция натурального аргумента $g(n) = O\big(f(n)\big)$, если существует такая константа $C > 0$ и число n_0, что для всех $n \geq n_0$ выполняется $|g(n)| \leq C|f(n)|$.
- $g(n) = o\big(f(n)\big)$, если $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.
- $g(n) = \Omega\big(f(n)\big)$, если $f(n) = O\big(g(n)\big)$.
- $g(n) = \omega\big(f(n)\big)$, если $f(n) = o\big(g(n)\big)$.
- $g(n) = \Theta\big(f(n)\big)$, если $g(n) = O\big(f(n)\big)$ и $f(n) = O\big(g(n)\big)$.
- $g(n) = \widetilde{O}\big(f(n)\big)$, если $f(n) = O\big(g(n)\log^k g(n)\big)$ для некоторого $k \in \mathbb{N}$.

В примере выше мы столкнулись с реккурентным неравеством. Забегая слегка вперед, отметим, что нам и впоследствии попадутся реккурентные уравнения и неравенства. Как правило, реккурентные уравнения будут появляться при оценке сложности процедур, в ходе которых они сами вызываются реккурентно (как, например, стандартная процедура вычисления факториала). Поэтому сейчас мы разберем некоторые способы решения реккурентных уравнений, включая теорему Акра-Баззи.

Самый простой способ — построить *дерево рекурсии*. Рассмотрим дерево, каждая вершина которого соответствует вызову экземпляра функции. Внутренние вершины помечены числом операций, которые потребовались для обработки результатов следующих рекурсивно вызванных функций (в нашем случае — для слияния двух массивов), а листы — числом операций на минимальной подзадаче, для которой уже не использовалась рекурсия. Кроме того, каждой вершине соответствует уровень, или глубина, n — размер задачи, для которой была вызвана эта функция. Очевидно, время решения задачи (в худшем случае), зависит только от ее размера, поэтому все вершины на одном уровне эквивалентны. Суммарное время работы программы складывается из времени работы в каждой вершине.

Пример 1.3. Рассмотрим реккуренту

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn,\ n \geq 2, \tag{1}$$

$$T(1) = C. \tag{2}$$

ПИКЧА Будем для удобства считать, что $n = 2^k$ — как мы увидим позже, это не помешает правильно оценить сложность. Нас интересует асимптотическая оценка $T(n)$. Можно циклически подставлять рекуррентное соотношение, пока не дойдем до нижнего уровня

$(n = 1)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn = 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn = \dots \tag{3}$$
$$\dots = 2^k T\left(\frac{n}{2^k}\right) + kcn = 2^k T(1) + cnk = Cn + cn\log n = \Theta(n\log n).$$

Покажем, что это верно для любых n, не только степеней двойки: очевидно, $T(n)$ неубывающая функция, так что

$$T(n) = O\left(2^{\lceil\log n\rceil}\lceil\log n\rceil\right) = O\left(2^{\log n+1}(\log n + 1)\right) = O(n\log n), \tag{4}$$

$$T(n) = \Omega\left(2^{\lfloor\log n\rfloor}\lfloor\log n\rfloor\right) = \Omega\left(2^{\log n-1}(\log n - 1)\right) = \Omega(n\log n). \tag{5}$$

Таким образом, $T(n) = \Theta(n\log n)$ для всех n.

При анализе сложности алгоритмов нам будут встречаться рекурренты вида

$$T(n) = \sum_{i=1}^{k} a_i T(b_i n) + f(n).$$

Оказывается, существует общий метод, позволяющий во многих случаях найти аналитическое выражение для асимптотики $T(n)$.

Теорема 1.4 (Akra–Bazzi). *Пусть дано следующее рекуррентное уравнение:*

$$T(x) = \begin{cases} \Theta(1), & 1 \le x \le x_0, \\ \sum_{i=1}^{k} a_i T\big(b_i x + h_i(x)\big) + g(x), & x > x_0, \end{cases}$$

где $x \in [1, +\infty)$; $a_i > 0$, $0 < b_i < 1$ — константы;

$$h_i(x) = O\left(\frac{x}{\log^2 x}\right);$$

$g(\cdot)$ — функция полиномиального роста, т.е. для любого $x \ge 1$, $1 \le i \le k$ и $u \in [b_i x, x]$ выполняется

$$c_1 g(x) \le g(u) \le c_2 g(x),$$

где $c_1, c_2 > 0$ — некоторые константы; x_0 — некоторое достаточно большое число.

Пусть $p \in \mathbb{Q}$ — решение уравнения $\sum_{i=1}^{k} a_i b_i^p = 1$. Тогда $T(x)$ имеет следующую асимптотику:

$$T(x) = \Theta\left(x^p\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}du\right)\right).$$

Идея доказательства. Угадав асимптотику, можно ее проверить по индукции, используя все условия теоремы. Но как угадать вид асимптотики? Для этого рассмотрим упрощенное уравнение:

$$T(x) = \sum_{i=1}^{k} a_i T(b_i x) + g(x),\ x \geq 1.$$

Зафиксируем x. Сначала оценим "плотность числа вершин" $f(t)$ в дереве рекурсии на уровне $\frac{x}{e^t}$ (в логарифмическом масштабе), т.е. между уровнями $\frac{x}{e^{t+\delta}}$ и $\frac{x}{e^{t-\delta}}$ находится примерно $2\delta e^{f(t)}$ вершин. Предположим, что эта функция гладкая. Тогда для любого $0 < t < \ln x$ получаем

$$e^{f(t)} \approx \sum_{i=1}^{k} a_i \exp\big(f(t + \ln b_i)\big) \approx \sum_{i=1}^{k} a_i \exp\big(f(t) + f'(t) \ln b_i)\big) = e^{f(t)} \left(\sum_{i=1}^{k} a_i b_i^{f'(t)}\right).$$

Следовательно, $\sum_{i=1}^{k} a_i b_i^{f'(t)} \equiv 1$, поэтому $f'(t) \equiv p$ и $f(t) = pt$.

Таким образом, $e^{f(\ln x)} = e^{p \ln x} = x^p$ — число минимальных подзадач. Вспомогательные вычисления занимают примерно

$$\int_0^{\ln x} e^{f(t)} g\left(\frac{x}{e^t}\right) dt = \int_1^x \left(\frac{x}{u}\right)^p g(u) \frac{du}{u} = x^p \int_1^x \frac{g(u)}{u^{p+1}} du.$$

В итоге получаем то, что нужно:

$$T(x) = \Theta\left(x^p + x^p \int_1^x \frac{g(u)}{u^{p+1}} du\right).$$

□

Добавочные члены $h_i(\cdot)$ в условии теоремы позволяют учитывать, например, округление чисел, прибавление константы и так далее. Заметим, что если заменить все равенства на ограничения сверху, то есть рассматривать рекурренту $T(x) \leq \sum_{i=1}^{k} a_i T\big(b_i x + h_i(x)\big) + O\big(g(x)\big)$, то

$$T(x) = O\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right).$$

Пример 1.5. Пусть дано рекуррентное уравнение

$$T(n) = 2T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + T\left(\left\lceil \frac{n}{3} \right\rceil\right) + \frac{1}{3} T\left(\left\lfloor \frac{4n}{5} \right\rfloor\right) + \Theta\left(n \log^2 n\right).$$

(Откуда мог взяться множитель $\frac{1}{3}$? Например, мы рассматривали *вероятностный* алгоритм, и с вероятность $\frac{2}{3}$ соответствующей подзадачи не возникает.) Т.к.

$$2\frac{1}{5} + \frac{1}{3} + \frac{1}{3}\frac{4}{5} = 1,$$

то $p = 1$. Тогда

$$\int_1^n \frac{u \ln^2 u}{u^2} du = \int_1^n \ln^2 u d \ln u = \frac{\ln^3 n}{3} = \Theta\left(\log^3 n\right).$$

Следовательно, $T(n) = \Theta\left(n + n \log^3 n\right) = \Theta\left(n \log^3 n\right)$.

Амортизационный анализ. Напоследок рассмотрим еще одно средство анализа алгоритмов, производящих последовательность однотипных операций — *амортизационный анализ*. При амортизационном анализе каждой операции присваивается некоторая учётная стоимость (amortized cost), которая может быть больше или меньше реальной длительности операции. При этом должно выполняться следующее условие: для любой последовательности операций фактическая суммарная длительность всех операций (предполагается, что до выполнения операций структура данных находится в начальном состоянии — например, стек пуст) не превосходит суммы их учётных стоимостей. Если это условие выполнено, то говорят, что учётные стоимости присвоены корректно. Оценка, даваемая амортизационным анализом, не является вероятностной: это оценка среднего времени выполнения одной операции для худшего случая.

Рассмотрим следующий пример амортизационного анализа. Рассматривается структура данных, состоящая из

- числа $n \in \mathbb{N}$ в двоичной записи ($\lceil \log_2 n \rceil$ битов),
- операции $Add : \mathbb{N} \to \mathbb{N}$, $Add(n) = n + 1$.

При инициализации $n = 1$, всякий вызов Add увеличивает хранимое число на 1. Так как

$$\underbrace{1 \ldots 1}_{k \text{ нулей}} + 1 = 1 \underbrace{0 \ldots 0}_{k \text{ нулей}},$$

то Add совершает в худшем случае $\Theta(\log n)$ побитовых операций.

Утверждение 1.6. *Рассмотрим процедуру, совершающую большое число N запросов к процедуре Add. Суммарное время работы всех выполнений процедуры оценивается как $\Theta(N)$ (а не $\Theta(N \log N)$).*

Доказательство. Нужно честно просуммировать сложность всех применений Add и оценить ее как $\Theta(N)$. Для начала заметим, что все числа от 1 до N можно разбить в дизъюнктное объединение множеств A_k чисел, оканчивающихся на ровно k единиц (возможно, ноль); иными словами,

$$[1; N] = \bigsqcup_{k=0}^{N} A_k, \quad A_k = \{1 \ldots 0 \underbrace{1 \ldots 1}_{k \text{ единиц}}\}.$$

Теперь заметим, что для любого $x \in A_k$ сложность операции *Add* равна $k+1$ битовых операций:

$$\underbrace{1\ldots1}_{k \text{ нулей}} + 1 = 1\,\underbrace{0\ldots0}_{k \text{ нулей}},$$

тогда для получения N надо заменить последние $k+1$ цифру на противоположную, 0 на 1 и наоборот. Осталось только честно просуммировать сложности каждой операции *Add*, примененной N раз, то есть $T(N)$. Очевидно, $T(N) \leqslant T(2^{\lceil \log_2 N \rceil})$, а $T(2^n)$ можно честно вычислить. Заметим, что

$$|A_k| = \begin{cases} 2^{n-k-1}, & k < n \\ 1, & k = n \end{cases},$$

ведь при $k \neq 0$ все числа A_k можно представлять как конкатенацию последовательности 0 и 1 длины $n-k$ и слова $01\ldots1$ [в старших разрядах могут стоять нули — у нас получится просто число, меньше некоторой степени], а слов, заканчивающихся на 0, ровно половина среди положительных не больших 2^n чисел. Тогда

$$T(2^n) = \sum_{k=1}^{n}(k+1)|A_k| = n + \sum_{k=1}^{n-1}(k+1)2^{n-k-1} = n + 2^{n-1}\sum_{k=1}^{n-1}\frac{k+1}{2^k} \leqslant$$

$$\leqslant n + 2^{n-1}\sum_{k=1}^{\infty}\frac{k+1}{2^k} = n + 2^{n-1}\left(\left(\frac{1}{1-z}\right)' - 1 - z\right)\Big|_{z=\frac{1}{2}} = n + \frac{5}{2}\cdot 2^{n-1},$$

и, таким образом

$$T(N) \leqslant T(2^{\lceil \log_2 N \rceil}) \leqslant \log_2(N) + 3N.$$

Аналогичную оценку снизу сделать совсем просто: достаточно лишь сказать, что $T(N) \geqslant T(2^{\lfloor \log_2 N \rfloor})$ и оценить $T(2^{\lfloor \log_2 N \rfloor})$ снизу как $\Theta(N)$, что совершенно очевидно. Тогда $T(N) = \Theta(N)$. □

В прошлой задаче мы могли сказать, что среднее, или *амортизированное* время работы операции *Add* есть $\Theta(1)$. Теперь рассмотрим еще одну задачу, решающуюся с помощью амортизационного анализа. Мы собираемся получить нижнюю оценку на минимальное число $T_{\min}$ попарных сравнений, которое необходимо сделать для нахождения минимального элемента n-элементного массива.

Утверждение 1.7. *Число шагов любого устойчивого (не зависящего от значений элементов входного массива) алгоритма нахождения минимального элемента $A[1\,..\,n]$ не меньше $n-1$.*

Доказательство. Запишем шаги алгоритма в формате конфигураций (a, b, c), где

- a элементов пока не сравнивались,

- b элементов никогда не были больше в сравнениях,
- c элементов хотя бы раз оказались больше.

Начальная конфигурация суть $Init = (n, 0, 0)$. Введем «потенциальную функцию» на конфигурациях: $f[(a, b, c)] = a + b$. Покажем, что при любом сравнении потенциал $f(\cdot)$ может уменьшиться не больше, чем на единицу, отсюда вытекает, что число шагов любого такого алгоритма не меньше $n - 1$. Алгоритм находит минимальный элемент массива, базируясь на результатах попарных сравнений элементов. Продемонстрируем, как могут меняться параметры a, b, c и значение функции f. Мы оценим трудоемкость алгоритма, просто поделив «разность потенциалов» между начальной и конечной конфигурациями на максимальное изменения потенциала за один шаг алгоритма. Для доказательства утверждения задачи нужно аккуратно разобрать шесть случаев [в зависимости от того, как сравниваемые на данном шаге x и y участвовали в предыдущих сравнениях].

- Если оба элемента не участвовали в сравнении, то либо один из них в предстоящем сравнении станет больше, а другой — меньше, тогда значение f уменьшится на один — a уменьшилось на два, а b увеличилось на один, либо оба элемента будут равны, то есть b увеличилось на два, а a уменьшилось на два, тогда f не изменилась;
- Если x не участвовал в сравнении, а y хотя бы однажды оказался больше, то либо x окажется меньше, что не изменит потенциал f [a уменьшилось на 1, c увеличилось на 1], либо y, что уменьшит потенциал на 1 [a уменьшилось на 1, b не изменилось];
- Если x не участвовал в сравнении, а y во всех сравнениях был не больше, то либо x окажется меньше, что уменьшит значение f на один [a уменьшилось на 1, c не изменилось — выкинули y из множества «всегда меньших элементов» и добавили x], либо y, что уменьшит потенциал на 1 [a уменьшилось на 1, b не изменилось];
- Если x был во всех предыдущих сравнениях не больше, а y уже участвовал в сравнениях, то x будет либо больше [и тогда b уменьшится на 1], либо будет меньше или равен [тогда b не изменится], значение f уменьшается не более чем на один;
- Если оба элемента были больше хотя бы в одном сравнениии, то независимо от итога сравнения a и b не поменяются, ведь в сравнении не участвуют «всегда меньшие» или неиспользованные элементы и новых таких появится, значение f не изменилось.

Все случаи разобраны, осталось только вывести, что сравнений должно быть хотя бы $n - 1$. Это тривиально: $a + b$ уменьшается с каждым сравнением не более чем на единицу, а алгоритм сможет выдать верный ответ, только лишь просмотрев все элементы [в противном случае можно построить пару массивов, на которых алгоритм делает сравнения тех же (по номеру) пар элементов и выдает в одном случае верный, а в другом

— неверный ответ; подробнее об этом — в пункте **(b)** этой же задачи]. Тогда a должно стать равным 0, а b — не большим 1 [все элементы массива просмотрены, единственный минимальный из них найден либо не существует]. На последнем шаге $f[(0, b, c)] \leqslant 1$, и алгоритм должен проделать хотя бы $n-1$ сравнение. □

Линейные рекурренты

Определение 1.8. Будем говорить, что последовательность $(X_n)_{n\in\mathbb{N}}$ задана *линейным рекуррентным уравнением*

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_k X_{n-k},$$

если даны $X_0, \dots X_{k-1}$, а формула выше верна для всех $n \geqslant k$. *Характеристическим уравнением* такой линейной реккуренты будем называть уравнение

$$\lambda^k - a_1\lambda^{k-1} - \cdots - a_k = 0,$$

где λ — переменная.

Пусть $\lambda_1, \dots, \lambda_k$ — корни характеристического уравнения. Оказывается, они тесно связаны с пространством решений линейной рекурренты, а именно: если они все различны, то общее решение рекурренты имеет вид

$$X_n = C_1\lambda_1^n + C_2\lambda_2^n + \cdots + C_k\lambda_k^n.$$

Константы $C_1, \dots, C_n$ находятся из начальных условий $X_0, \dots, X_{k-1}$.

Если же какой-то корень λ_j имеет кратность $s_j > 1$, то в общее решение входит слагаемое вида

$$C_{j,0}\lambda_j^n + C_{j,1}n\lambda_j^n + \cdots + C_{j,s_j-1}n^{s_j-1}\lambda_j^n.$$

Несложно проверить (например, по индукции), что такие последовательности действительно удовлетворяют данному рекуррентному уравнению. Например, можно ввести «повышающий оператор» $\square X_n := X_{n+1}$ (он действует, вообще говоря, на всю последовательность $\{X_n\}_{n\in\mathbb{N}}$). Тогда рекуррентное уравнение записывается в виде

$$\square^k X_n - a_1\square^{k-1}X_n - \cdots - a_k X_n = \prod_{i=1}^{k}(\square - \lambda_i)X_n = 0.$$

Если λ_j имеет кратность s_j, то по индукции устанавливается, что для $0 \leqslant l \leqslant s_j - 1$ выполняется

$$(6)\quad (\square-\lambda_j)^{l+1}n^l\lambda_j^n = (\square-\lambda_j)^l\big((n+1)^l\lambda_j^{n+1} - n^l\lambda_j^{n+1}\big) =$$
$$= (\square-\lambda_j)^l\sum_{i=0}^{l-1}C_l^i n^i\lambda_j^{n+1} = \lambda_j\sum_{i=0}^{l-1}C_l^i(\square-\lambda_j)^{l-i-1}\big((\square-\lambda_j)^{i+1}n^i\lambda_j^n\big) = 0.$$

Почему любое решение имеет такой вид? Заметим, что пространство решений k-мерное: они полностью определяются начальными условиями, т.е. k числами $X_0, \ldots, X_{k-1}$. Можно рассмотреть последовательности X_n^j, удовлетворяющие рекуррентному уравнению и $X_i^j = \delta_{ij}$, $0 \leqslant i < k$. Они образуют базис в пространстве решений. С другой стороны, решения вида $n^l \lambda_j^n$ линейно независимы, т.к. они имеют разный порядок роста. Сумма кратностей всех корней равна k, поэтому эти решения тоже образуют базис, т.е. любое решение можно представить в виде

$$X_n = \sum_j \sum_{l=0}^{s_j - 1} C_{j,l} n^l \lambda_j^n,$$

что и требовалось.

Пример 1.9. Рассмотрим числа Фибоначчи: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$. Характеристическое уравнение для этой рекурренты:

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

Его корни $\lambda_1 = \frac{1-\sqrt{5}}{2}$, $\lambda_2 = \frac{1+\sqrt{5}}{2}$. Следовательно, общее решение имеет вид

$$F_n = C_1 \left(\frac{1-\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1+\sqrt{5}}{2}\right)^n.$$

Найдем константы C_1 и C_2:

$$F_0 = C_1 + C_2 = 0, \tag{7}$$

$$F_1 = C_1 \frac{1-\sqrt{5}}{2} + C_2 \frac{1+\sqrt{5}}{2} = 1. \tag{8}$$

Следовательно, $C_2 = -C_1 = \frac{1}{\sqrt{5}}$. Таким образом, числа Фибоначчи равны

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right).$$

До этого мы рассматривали однородное линейное рекуррентное уравнение. Теперь рассмотрим *неоднородное*:

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_k X_{n-k} + f(n).$$

Его общее решение имеет вид

$$X_n = X_n^{\text{о.о.}} + X_n^{\text{ч.н.}},$$

где $X_n^{\text{о.о.}}$ — общее решение соответствующего однородного уравнения, а $X_n^{\text{ч.н.}}$ — некоторое частное решение неоднородного.

Пример 1.10. Рассмотрим числа Фибоначчи, но с дополнительным слагаемым: $\tilde{F}_0 = 0$, $\tilde{F}_1 = 1$, $\tilde{F}_n = \tilde{F}_{n-1} + \tilde{F}_{n-2} + n$, $n \geq 2$. Общее решение однородного уравнения, как мы знаем, имеет вид

$$F_n = C_1 \left(\frac{1-\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1+\sqrt{5}}{2}\right)^n.$$

Частное решение неоднородного можно угадать. Т.к. $f(n)$ линейна, то решение будем искать тоже линейное: $F_n = an + b$. Тогда

$$an + b = a(n-1) + b + a(n-2) + b + n = (2a+1)n + 2b - 3a.$$

Следовательно, $a = -1$, $b = 3a = -3$, и решение имеет вид

$$\tilde{F}_n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n - an - 3.$$

Константы C_1 и C_2 находятся из системы уравнений:

$$\tilde{F}_0 = C_1 + C_2 - 3 = 0, \tag{9}$$

$$\tilde{F}_1 = C_1 \frac{1-\sqrt{5}}{2} + C_2 \frac{1+\sqrt{5}}{2} - 4 = 1. \tag{10}$$

Вероятность: введение

На всякий случай здесь приведено краткое напоминание дискретной версии теории вероятности. Для более подробного ознакомления советуем книги [38, 16, 22], а также задачник [14].

Определение 1.11. *Вероятностным пространством* называется множество Ω, элементы которого называются возможными или элементарными исходами $\omega \in \Omega$. На вероятностном пространстве задана функция $\mathbb{P} : \Omega \to [0, 1]$, называемая *вероятностным распределением*, для которой выполнено равенство $\sum\limits_{\omega\in\Omega} \mathbb{P}(\omega) = 1$. Число $\mathbb{P}(\omega)$ понимается как *вероятность* ω.

Определение 1.12. *Событием* называется произвольное подмножество Ω. Соответственно, вероятность события $A \subseteq \Omega$ определяется формулой $\mathbb{P}(A) \stackrel{def}{=} \sum_{\omega\in A} \mathbb{P}(\omega)$.

В качестве простейшего примера можно рассмотреть классическую, или наивную, теорию вероятности, когда все исходы полагаются равновероятными: например, при бросании симметричной монетки (то есть вероятность любого исхода есть $\frac{1}{2}$) или игральной кости (выпадает число от 1 до 6, вероятность каждого исхода есть $\frac{1}{6}$).

Определение 1.13. События A и B называют *независимыми*, если

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

События $A_1, \ldots, A_n$ называются *попарно независимыми*, если любые два из них являются независимыми. События $A_1, \ldots, A_n$ называются *независимыми в совокупности*, если для любого набора из этих событий $A_{i_1}, \ldots, A_{i_k}$ выполняется

$$\mathbb{P}(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}) = \mathbb{P}(A_{i_1})\mathbb{P}(A_{i_2}) \ldots \mathbb{P}(A_{i_k}).$$

Определение 1.14. *Условная вероятность* $\mathbb{P}(A \mid B)$ события A при условии B определяется из формулы

$$\mathbb{P}(A \cap B) = \mathbb{P}(A \mid B) \cdot \mathbb{P}(B).$$

В частности, если $\mathbb{P}(B) > 0$, то справедливо равенство

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

Теорема 1.15 (Bayes). *Если* $\mathbb{P}(A) > 0$ *и* $\mathbb{P}(B) > 0$, *то*

$$\mathbb{P}(A \mid B) = \mathbb{P}(A) \cdot \frac{\mathbb{P}(B \mid A)}{\mathbb{P}(B)}.$$