ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

СЛОЖНОСТЬ АЛГОРИТМОВ

АБРАМОВ
СЕРГЕЙ АЛЕКСАНДРОВИЧ

ВМК МГУ

КОНСПЕКТ ПОДГОТОВЛЕН СТУДЕНТАМИ, НЕ ПРОХОДИЛ ПРОФ. РЕДАКТУРУ И МОЖЕТ СОДЕРЖАТЬ ОШИБКИ. СЛЕДИТЕ ЗА ОБНОВЛЕНИЯМИ НА VK.COM/TEACHINMSU.

ЕСЛИ ВЫ ОБНАРУЖИЛИ ОШИБКИ ИЛИ ОПЕЧАТКИ, ТО СООБЩИТЕ ОБ ЭТОМ, НАПИСАВ СООБЩЕСТВУ VK.COM/TEACHINMSU.

Содержание

Лекция 1

Зачем нужнен анализ сложности алгоритмов? Разумеется, слово «сложность» здесь является математическим понятием, а не бытовым. Анализ сложности алгоритмов позволяет из нескольких алгоритмов выбрать оптимальный. Теория сложности необходима, чтобы обосновать преимущество нового алгоритма перед старыми.

Понятие сложности алгоритмов

Чтобы говорить о сложности алгоритма, нужно сначала определить помимо алгоритма ещё два понятия: во-первых размер входа, во-вторых в чём будут измеряться затраты алгоритма. Будем рассматривать алгоритм A. Обозначим x входные данные алгоритма (или просто «вход»), а y результат работы алгоритма (или же «выход»). Затраты, связанные с применением алгоритма A ко входу x, будем обозначать

$$C_A^T(x). \tag{1.1}$$

Здесь верхний индекс T указывает на временные затраты. Пространственные затраты, или затраты по памяти, связанные с применением алгоритма A ко входу x, будем обозначать

$$C_A^S(x). \tag{1.2}$$

Следует отметить, что $C_A^S(x)$ не учитывает затраты на хранение снепосредственно входа x. Речь идёт только о дополнительных затратах.

Помимо этого нам потребуется понятие размера входа. Размер входа x обозначим как $||x||$. Как правило $||x||$ — неотрицательное целое число. Вообще говоря, можно считать $||x||$ вещественным, но это сопряжено с дополнительными трудностями.

Пусть $a \in \mathbb{R}$. Округление a до целого числа в меньшую сторону обозначим $\lfloor a \rfloor$. Иногда $\lfloor a \rfloor$ называют полом. Пример:

$$\begin{aligned} &\lfloor 3.14 \rfloor = 3; \\ &\lfloor -3.14 \rfloor = -4. \end{aligned} \tag{1.3}$$

В свою очередь, потолком называется округление до целого числа в большую сторону. Пример:

$$\begin{aligned} &\lceil 3.14 \rceil = 4; \\ &\lceil -3.14 \rceil = -3. \end{aligned} \tag{1.4}$$

Если же a целое, то пол и потолок совпадают:

$$\lfloor a \rfloor = \lceil a \rceil = a. \tag{1.5}$$

Приведём несколько простых примеров алгоритмов, на которых мы в дальнейшем будем рассматривать понятие сложности.

1) **Перемножение матриц.** Исходя из определения произведения матриц, для умножения двух квадратных матриц размера $n \times n$ необходимо совершить

n^2 операций умножения чисел. Часто в задачах затраты измеряют именно количеством мультипликативных операций, считая, что они значительно превышают затраты аддитивных операций. Алгоритм перемножения матриц в дальнейшем будем обозначать MM.

2) **Проверка числа на простоту.** Один из алгоритмов проверки простоты данного целого числа $n \geqslant 2$ состоит в поиске делителя числа n среди целых чисел от 2 до $\lfloor\sqrt{n}\rfloor$. Если хотя бы один делитель найден, то n, очевидно, простым числом не является. Если же среди чисел от 2 до $\lfloor\sqrt{n}\rfloor$ нет делителей n, то n не является простым и поиски делителя можно прекратить, поскольку если бы существовал делить больший чем $\sqrt{n}$, то существовал бы и делитель меньший чем $\sqrt{n}$. Такой алгоритм называется алгоритмом пробных делений и обозначается TD.

   Данный алгоритм имеет существенное отличие от предыдущего. В случае перемножения матриц было достоверно известно, что потребуется именно n^2 мультипликативных операций, здесь же сразу количество мультипликативных операций сказать нельзя. Может потребоваться от 0 до $\lfloor\sqrt{n}\rfloor - 1$ мультипликативных операций.

3) **Алгоритм сортировки простыми вставками.** В дальнейшем для краткости мы будем говорить не «алгоритм сортировки», а просто «сортировка», например «сортировка простыми вставками» вместо «алгоритм сортировки простыми вставками». Будем считать, что элементы массива, который мы сортируем, попарно различны. От этого предположения можно отказаться, но отказ от него усложнит рассмотрение. Будем считать, что элементы массива — числа. Речь всегда будет идти о сортировке по возрастанию.

   Напомним кратко, в чём заключается алгоритм сортировки простыми вставками. Пусть дан массив чисел

$$a_1, \ldots, a_i, a_{i+1}, \ldots, a_n, \tag{1.6}$$

   где участок $a_1, \ldots, a_i$ уже упорядочен. В этом упорядоченном участке необходимо найти место для элемента a_{i+1}. Действуем следующим образом: сравниваем a_{i+1} с каждым элементом из $a_1, \ldots, a_i$, причем элементы $a_1, \ldots, a_i$ перебираем с конца. На каждом шаге происходит перестановка, пока a_{i+1} не окажется на своём месте. Это один из вариантов сортировки простыми вставками, далее мы коснёмся и другого варианта.

   В алгоритме сортировки простыми вставками существует два вида операций: сравнение и перемещение. Для простоты термин «премещение» используется как общий: в некоторых алгоритмах это означает присваивание $:=$, а в некоторых - обмены $\leftrightarrow$ (то есть два элемента меняются местами). Постренные на операциях обмена алгоритмы называются обменными. Так, обменным является описанный выше вариант алгоритма сортировки простыми вставками. Итак, поскольку в алгоритмах сортировки выделяют два типа операций, то вводится также и два вида временных затрат: затраты, основанные на числе перемещений, и затраты, основанные на числе сравнений. Иногда эти два вида временных затрат не различают и обсуждают некие общие затраты.

Сделаем оценки для двух выделенных видов сложности. Число элементов в массиве будем называть его длиной. Пусть n — длина массива. Как и в случае алгоритма пробных делений, точную формулу для количества сравнений написать не получится, поскольку заранее не известно, сколько сравнений потребуется. Однако можно написать некоторые пределы для количества сравнений:

$$n - 1 \leqslant \text{ число сравнений } \leqslant \frac{n(n-1)}{2}. \tag{1.7}$$

Аналогичную формулу можно записать лля числа перемещений:

$$0 \leqslant \text{ число перемещений } \leqslant \frac{n(n-1)}{2}. \tag{1.8}$$

Дадим теперь определение понятию сложности алгоритма.

Определение 1. Пусть определен размер входа $||x||$ — неотрицательная целая функция. Пусть также определены целочисленные неотрицательные функции $C_A^T(x)$ и $C_A^S(x)$. **Временной сложностью** будем называть

$$T_A(n) = \max_{||x||=n} C_A^T(x). \tag{1.9}$$

Пространственной сложностью будем называть

$$S_A(n) = \max_{||x||=n} C_A^S(x). \tag{1.10}$$

Замечание. Отметим, что функции $T_A(n)$ и $S_A(n)$ являются функциями числового аргумента. Это действительно важно, поскольку позволяет исследовать эти функции апппаратом матанализа. Функции $C_A^T(x)$ и $C_A^S(x)$ такой возможности не дают, поскольку их аргументом может быть всё что угодно: массив, таблица, любой файл. В таком случае говорить, например, о росте функции не представляется возможным.

Замечание. В большинстве случаев под затратами $C_A^T(x)$ подразумевается число операций (заранее оговаривается, каких именно операций). Что касается $C_A^S(x)$, то обычно подразумевается число объектов определённого типа.

Замечание. Более полно определенные выше сложности называются сложностями в худшем случае. Существует понятие сложности в среднем, его мы будем обсуждать через несколько лекций. Вообще говоря, вместо «сложность в худшем случае» было бы правильнее сказать «сложность как затраты в худшем случае».

Вычислим сложность для приведённых выше элементарных примеров. Для умножения матриц число мультипликативных операций однозначно определено, поэтому сложность попросту совпадает с числом мультипликативных операций.

Обсудим сложность алгоритма простых вставок: и для затрат по числу сравнений, и для затрат по числу обменов максимум равен $n(n-1)/2$. Можно показать, что для каждого n существует x такой что $||x|| = n$, причём для сортировки x потребуется $n(n-1)/2$ обменов.

Таблица 1.1. Сложность алгоритма пробных делений

n	111	112	113	114	115	116
$T_{TD}(n)$	2	1	9	1	4	1

Перейдём к алгоритму пробных делений. В таблице 1.1 представлены значения функции $T_{TD}(n)$ на некотором диапазоне n. Как видно, поведение очень «неровное». О функции сложности $T_{TD}(n)$ в данном случае можно сказать только то, что для любых n значение функции $T_{TD}(n)$ не превосходит $\lfloor n \rfloor - 1$. Найдутся сколь угодно большие n, для которых сложность равна $\lfloor n \rfloor - 1$, поскольку множество простых чисел бесконечно.

Сформулируем задания для самостоятельного решения:

1) Доказать, что $\forall n \in \mathbb{Z}$ верно $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$.

2) Доказать, что $\forall a \in \mathbb{R}$ верно $\lfloor a \rfloor = -\lceil -a \rceil$.

3) Доказать, что $\forall k \in \mathbb{N}$ и $\forall n \in \mathbb{N}$, причём $k > 1$, верно $\lfloor \log_k n \rfloor + 1 = -\lceil \log_k (n + 1) \rceil$.

4) Пусть есть железнодорожный сортировочный узел с тремя сортировочными опеарциями (см. рис. 1.1): «мимо», «в тупик» и «из тупика». На входе черные и белые вагоны: вагонов каждого цвета n штук. Размером входа будем называть число n. Задача состоит в том, чтобы составить вагоны в чередующемся порядке, то есть за белым вагоном всегда долен следовать чёрный вагон, и, наоборот, за чёрным всегда должен следовать белый. Алгоритм должен иметь сложность $3n - 1$.

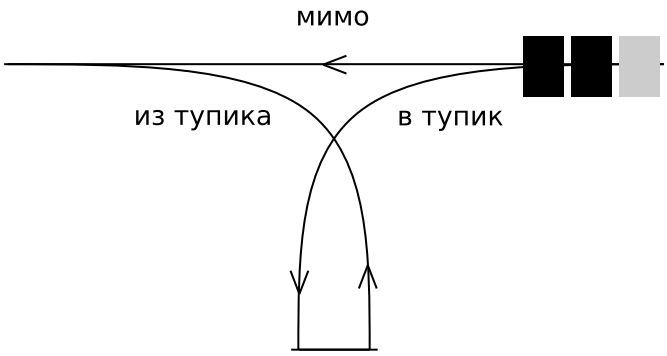

Рис. 1.1. Иллюстрация к задаче.

Мы обсудили временную сложность некоторых алгоритмов, обсудим теперь их пространственную сложность. Пространственная сложность — это количество необходимых дополнительных ячеек. Сортировка простыми вставками делается на том же месте, поэтому дополнительный массив не требуется. Что касается матричного

умножения, то здесь пространственная сложность равна n^2, поскольку создаётся новый двумерный массив, в который записывается результат.

Разговаривая в таком ключе о сложности, мы не учитываем некоторые накладные расходы, например, связанные с переменными циклов, используемых в программе. Также могут не учитываться операции, которые мы считаем «неважными», то есть значительно более простыми, чем те, которые мы учитываем. Показателен пример с матрицами: здесь мы пренебрегаем аддитивными операциями. В этом вопросе следует проявлять аккуратность. «Неважных» операций может оказаться очень много, в таком случае они будут оказывать существенное влияние на сложность алгоритма.

Также следует помнить, что при подсчёте операций не учитывается тот факт, что ресурсозатратность отдельной операции тоже зависит от размера входа. Например, умножать короткие числа проще, чем длинные. Однако мы принимаем соглашения: операции равнозатратны. Такую сложность называют алгебраической (делается упор на число операций) или однородной.

Конечно, размер входа можно определять по-разному. В результате полученная сложность тоже окажется разной.

Обсудим теперь ситуацию, когда при рассмотрении алгоритма операциям разного типа приписываются отдельные сложности, как, например, в алгоритме сортировки простыми вставками. Пусть хотим имея две сложности получить одну, которая бы учитывала все операции. Возникает вопрос: можно ли просто сложить две имеющиеся сложности? Оказывается, что нет. Причина в том, что максимумы двух функций затрат могут достигаться на разных входах, поэтому складывать сложности некорректно. Ясно, что суммарная сложность окажется меньше или равна сумме сложностей по двум опеарциям.

Вернёмся к примеру алгоритма сортировки простыми вставками. Первый вариант сортировки простыми вставками обозначим I_1. Предложим другой вариант: пусть упорядоченный массив просматривается с начала, а не с конца как в I_1, ищется место для нового элемента, а после того как место найдено, часть элементов сдвигается и вставляется новый элемент. Обозначим такой алгоритм I_2. Заметим, что в алгоритме I_1 место для новорого элемента уже было заведомо свободно, потому перемещения элементов происходили после каждого сравнения, то есть в процессе поиска месте для нового элемента, а не уже после того, как место найдено. Зададимся вопросом: как соотносятся алгоритмы I_1 и I_2 в смысле сложности? Теперь имеем три сложности: по числу сравнений, по числу перемещений, общая сложность.

В первом варианте алгоритма максимум числа сравнений достигается, когда массив имеет обратную упорядоченность. На этом же входе достигается и максимум числа обменов. Справедливо

$$\widetilde{T}_{I_1}(n) = T'_{I_1}(n) + T''_{I_1}(n) = n(n-1). \tag{1.11}$$

Здесь $T'_{I_1}(n)$ – сложность по числу сравнений, $T''_{I_1}(n)$ – сложность по числу перемещений, $\widetilde{T}_{I_1}(n)$ – общая сложнсоть. Во втором варианте алгоритма максимум по числу сравнений достигается, когда массив уже упорядочен так как требуется. Максимум по числу сравнений и перемещений достигается, когда массив имеет

обратную упорядоченность, причём

$$\widetilde{T}_{I_2}(n) = \frac{(n+3)(n+1)}{2}. \tag{1.12}$$

Если рассмотреть отношение сложности алгоритма I_1 к сложности алгоритма I_2, а затем устремить n к бесконечности, то получим в результате 2. Выходит, разница между алгоритмами I_1 и I_2 есть. Второй вариант имеет общую сложность в два раза меньшую при больших n.

Замечание. Кроме вычислительной сложности существует сложность описательная или изобразительная. Описательной сложностью называют число символов, использованных в записи алгоритма. Опсиательная сложность в данном курсе обсуждаться не будет.

Обсудим те предположения об алгоритмах, которые мы неявно используем. Мы считаем, что наше вычислительное устройство обладает бесконечным числом ячеек, причём каждая ячейка имеет неограниченный размер. Другое настолько же абстрактное предположение заключается в том, что попользовавшись одной ячейкой, можно перейти к другой, и этот переход ничего не стоит. Такая модель вычислений называется РАМ, то есть «равнодоступная адресная машина» или машина с равнодоступными адресными ячейками. На английском языке это RAM: random access machine. Другая модель вычислений — машина Тьюринга. Машина Тьюринга представляет собой ленту. Операция — это такт машины. Чтрбы добраться от одной ячейки до другой, может понадобить много тактов. При оценке сложности для машины Тьюринга это приходится учитывать.

Асимптотические оценки сложности

Итак, мы сформулировали определение сложности алгоритма. Однако не всегда легко вычислить такую сложнсоть. Более того, явный вид сложности может быть не интересен. Интерес представляет рост сложности при возрастании размера входа, поскольку на входах малого размера как правило проблем не возникает. Когда размер входа возрастает, возникают проблемы. Вот почему использование асимптотических оценок более чем оправдано.

Будем использовать символ $\mathcal{O}$. Запишем с использованием $\mathcal{O}$ асимптотические оценки для явно вычисленных сложностей двух вариантов алгоритма сортировки (1.11) и (1.12):

$$\begin{aligned} \widetilde{T}_{I_1}(n) &= n^2 + \mathcal{O}(n); \\ \widetilde{T}_{I_2}(n) &= \tfrac{1}{2}n^2 + \mathcal{O}(n). \end{aligned} \tag{1.13}$$

Для сортировки простыми вставками получена асимптотическая оценка сверху. Такая оценка означает, что функция сложности растёт не быстрее, чем n^2. Но под такую оценку сверху попадает и, например, функция n, поскольку такая функция тоже растёт не быстрее, чем n^2. По этой причине только оценки сверху для нас недостаточно. Говорят: «сортировка простыми вставками имеет квадратичную сложность», и пишут символ

$$\Theta(n^2). \tag{1.14}$$

Это означает, что функция сложности является величиной порядка n^2. Сформулируем более аккуратно.

Определение 2. Запись $f(n) = \Theta(g(n))$ означает, что для любого целого N найдутся константы c_1 и c_2 такие, что верно

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \tag{1.15}$$

для всех $n > N$.

Замечание. В математическом анализе вместо записи $f(n) = \Theta(g(n))$ используют другую запись:

$$f(n) \asymp g(n). \tag{1.16}$$

Нетрудно видеть, что это отношение является отношением эквивалентности. Для отношения эквивалентности запись (1.16) более естественна. При использовании символа θ эта эстетика пропадает. Однако пользоваться мы будем именно записью $f(n) = \Theta(g(n))$. Дело в том, что функция g обычно имеет значительно более простой вид, чем фунцкия f, поэтому символ θ прижился.

Также можно дать определение асимптотической оценки снизу.

Определение 3. Пишут $f(n) = \Omega(g(n))$, если для любого целого N найдётся константа c_1 такая, что верно

$$c_1|g(n)| \leq |f(n)| \tag{1.17}$$

для всех $n > N$.

Нетрудно видеть, что

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \& f(n) = \Omega(g(n)). \tag{1.18}$$

Замечание. Из $f(n) = O(g(n))$ следует $g(n) = \Omega(f(n))$.

Наконец, обсудим понятие точной оценки на примере алгоритма проверки делением числа на простоту. Сложность алгоритма проверки делением составляет $O(\sqrt{n})$:

$$T_{TD}(n) = O(\sqrt{n}) \tag{1.19}$$

В то же время, справедливы будут оценки

$$\begin{aligned} T_{TD}(n) &= O(n); \\ T_{TD}(n) &= O(n^2), \end{aligned} \tag{1.20}$$

но такие оценки будут более грубыми. Оценку (1.19) называют точной оценкой.

Определение 4. Оценка $f(n) = O(g(n))$ называется **точной**, если оценка $f(n) = o(g(n))$ не верна.

Домашнее задание. *Рассмотрим множество натуральных чисел. Каждому натуральному числу сопоставим количество простых множителей с учётом кратности (обозначим эту функцию ϕ). Например, поскольку $4 = 2 \cdot 2$, то $\phi(4) = 2$. Доказать, что для функции ϕ справедлива оценка $\mathcal{O}(\log n)$ и эта оценка является точной.*

Обратим внимание на то, что в асимптотических оценках не принято писать основание логарифма. Это связано с тем, что переход между различными основаниями можно осуществить с помощью константного множителя, а константу в асимптотических оценках можно игнорировать.

Лекция 2

Оценка алгоритмов O-символикой, асимптотические оценки

Обсудим подробнее понятие точной оценки. Точные оценки могут быть использованы для сравнения алгоритмов. Вспомним алгоритм пробных делений для проверки простоты числа. Здесь размер входа — само число n, простоту которого мы проверяем, а затраты измеряются в количестве проведённых делений. На прошлой лекции был получен результат

$$T_{TD}(n) = \mathcal{O}(\sqrt{n}). \tag{2.1}$$

Здесь оценка является точной, поскольку оценка $T_{TD}(n) = o(\sqrt{n})$ не верна. Если мы найдём алгоритм проверки простоты со сложностью, например, $O(\log n)$, то этот алгоритм будет работать гарантированно быстрее. Именно поэтому полезно проверять полученные оценки: являются ли они точными?

Асимптотические оценки:

$$T_A(n) \sim g(n). \tag{2.2}$$

Утверждение (2.2) более сильное, чем утверждение

$$T_A(n) = \Theta(g(n)). \tag{2.3}$$

Из отношения (2.2) следует отношение (2.3), но не наоборот.

Домашнее задание. *Путешественник столкнулся со стеной. Ему известно, что в стене есть калитка. Ему неизвестно, справа или слева от него находится калитка. Предложить алгоритм поиска калитки, сложность которого составляет $\mathcal{O}(n)$, где n — расстояние от путника до калитки. Число n путешественнику неизвестно. Затраты измеряюся в расстоянии, которое пройдёт путешественник до выхода через калитки.*

Зачем нам всё это нужно? Приведём два примера без мелких деталей. Кстати, асимптотические оценки позволяют избегать мелких деталей. Примеры будут из вычислительной геометрии. Одна из центральных задач — задача построения выпуклой оболочки множества в евклидовом пространстве. Рассматриваются и многомерные задачи. Мы будем рассматривать двумерный случай. Пусть есть конечное множество M на евклидовой плоскости: $M \subset \mathbb{R}^2$. Множество M конечно и задано координатами входящих в него точек. Всего точек n штук. Размером входа будем называть число точек n. Требуется построить выпуклую оболочку. Выпуклой оболочкой называется наименьший выпуклый многоугольник, который содержит все точки множества M. Искомую выпуклую оболочку обозначим H (см. рис.2.1). Как построить H? Построение выпуклой оболочки означает задание последовательности координат вершин многоугольника H. Начинать можно с любой вершины, устраивать обход принято против часовой стрелки.

Ясно, что задача разрешима. Можно путём перебора всех подмножеств M отыскать вершины выпуклой оболочки H. Однако сложность такого алгоритма составит 2^n с точностью до умножения на константу.

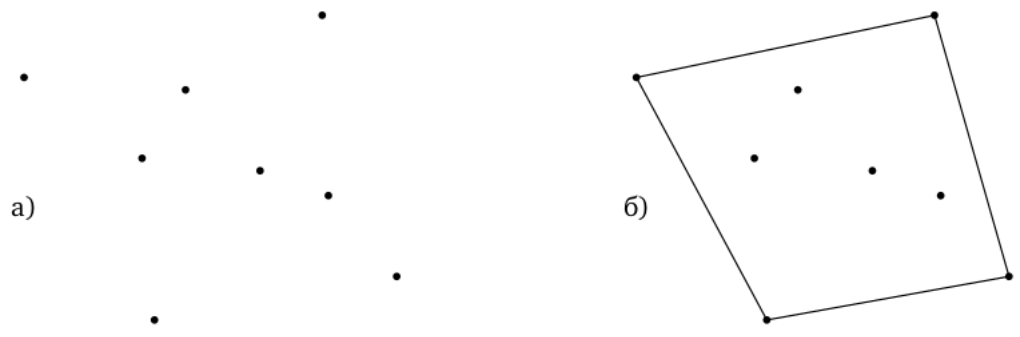

Рис. 2.1. а) Конечное множество M точек плоскости; б) Выпуклая оболочка множества M.

Задачу можно решить значительно менее сложными алгоритмами, например, алгоритмом Грэхема. Прежде всего, из заданных точек множества M можно выбрать точку, которая будет заведомо принадлежать выпуклой оболочке H. Эта точка — точка с наименьшей ординатой. Если же наименьшее значение ординаты достигается на нескольких точках множества M, то можно выбрать из этих точек ту, которая, например, имеет наименьшую абсциссу. Обозначим данную точку как P_1. Ещё одна точка, которую полезно немедленно определить — точка, которая заведомо не совпадает с выпуклой оболочкой H. Чтобы её получить, можно соединить точку P_1 с любой другой и найти середину полученного отрезка. Эту точку обозначим как O. Соединим точку O со всеми точками множества M. Каждому возникшему отрезку припишем угол, отсчитываемый от оси OP. Отсортируем углы каким-либо алгоритмом сортировки. В результате получим пронумерованный набор точек $P_1, \ldots, P_N$. После этого соединим точки отрезками в том порядке, в котором они пронумерованы. Построенная ломанная линия является несамопересекающейся. Однако полученная оболочка не является выпуклой, некоторые вершины являются «вдавленными». Полученную оболочку необходимо «спрямить». После исключения вдавленных вершин может оказаться, что те вершины, которые ранее «вдавленными» не были, окажутся «вдавленными» после этой процедуры. У этой проблемы существует два решения. Одно из них заключает в круговом обходе фигуры и исключении «вдавленных» вершин до тех пор, пока не удастся сделать полный круг и не наткнуться на «вдавленную» вершину. Другое решение, которое было предложено Грэхемом, является менее сложным и заключается в том, что после каждого удаления «вдавленной» вершины необходимо возвращаться на шаг назад и проверять, не стала ли «вдавленной» соседняя точка. Путь назад прекращается, когда обнаруживается не «вдавленная» точка.

Попробуем оценить сложность данного алгоритма. В алгоритм входит этап сортировки. Можно использовать любой алгоритм сортировки. Сложность этого этапа составит

$$c \cdot r(n), \tag{2.4}$$

где c — некоторая константа. Предварительные этапы (выбор точки P_1 и выбор

точки O) дадут вклад в общую сложность, равную

$$c_0 n. \tag{2.5}$$

Рассматрим вершину P_i. Пусть с ней связано с проверкой вдавленность v_i вершин. При этом будет удалено $v_i - 1$ вершин. Справедливо

$$\sum_{i=2}^{n}(v_i - 1) < n, \tag{2.6}$$

поскольку невозможно удалить более чем n вершин, а также нельзя удалить точку P_1. Из этого неравенства следует неравенство

$$\sum_{i=2}^{n} v_i < 2n. \tag{2.7}$$

Итак, сложность алгоритма Грэхема не превосходит следующей величины:

$$c' \cdot r(n) + c''n. \tag{2.8}$$

Для сложности алгоритма Грэхема справедливо

$$T_G(n) = c' \cdot r(n) + c''n = r(n)(c' + c''\frac{n}{r(n)}). \tag{2.9}$$

Заметим, что выбор алгоритма сортировки здесь не уточняется. Однако для любого алгоритма сортировки верно

$$r(n) \geqslant \frac{n}{2}, \tag{2.10}$$

поэтому

$$T_G(n) \leqslant r(n)(c' + 2c''). \tag{2.11}$$

Константы c' и c'' можно заменить на одну константу:

$$T_G(n) = \mathcal{O}(r(n)). \tag{2.12}$$

Что касается сложности сортировки по числу перемещений, то тут необходимо сказать следующее: точки множества M не переставляются, поэтому можно считать, что сложность сортировки по числу сравнений равна нулю. Обозначим общую сложность сортировки $s(n)$, тогда сложность алгоритма Грэхема составит $\mathcal{O}(s(n))$.

Итак, зная лишь эскизное описание алгоритма и опуская детали, мы всё равно смогли написать асимптотическую оценку для сложности этого алгоритма.

Предварительные асимптотические оценки также позволяют определить наиболее удачную структуру данных. Для конкретного алгоритма какое-то определённое представление данных может оказаться более удобным, чем другое. Обсудим этот вопрос на примере графа. Граф представляет собой множество вершин и рёбер $G = (V, E)$. Количество вершин и количество рёбер составляют размер входа. Пусть граф связный, то есть из любой вершины можно попасть в любую другую вершину. Пусть граф ориентированный или направленный. Пусть зафиксирована некоторая

вершина. Вояжем называется такой путь, при котором никакое из рёбер не проходится дважды, а у конечной вершины нет не пройденных рёбер (то есть нельзя двигаться дальше, не нарушая предыдущего правила). Как представлять граф для решения данной задачи? Спрямить путь можно двумя вариантами. Первый вариант хорошо, если у графа нет кратных рёбер: использовать матрицу смежности. Матрица смежности состоит из нулей и единиц. Элемент M_{ij} равен нулю, если i-ая и j-ая вершины не соединены ребром, и единице, если соединены. Второй вариант: использовать массив списков смежных вершин. Если гарф содержит n вершин, то создаётся массив $a_1, \ldots, a_n$, где a_i — список номеров вершин, смежных с i-ой вершиной. Второй вариант более удобен в задаче вояжа, чем матрица смежности. При использовании матрицы смежности сложность алгоритма составляет $\mathcal{O}(|E|^2)$, где E — число рёбер. Поиск по матрице смежности смежной вершины может быть затруднительным. С массивом списков смежных вершин сложность уже составит $\mathcal{O}(|E|)$.

Домашнее задание. *Пусть заданы два выпуклых многоугольника. Пересечение двух выпуклых многоугольниках также является выпуклым многоугольников. Как построить наименее сложный алгоритм для построения многоугольника-пересечения?*

Решение задачи называется алгоритмом Шеймоса. Обсудим идею алгоритма без деталей. Сначала абсциссы точек сортируются по возрастанию. После этого рассматриваются «полосы». Внутри полоски лишних вершин не будет. Отбираются полосы, которые содержат пересечение многоугольников. Внутри плос находятся трапеции, и задача сводится к нахождению пересечения трапеций. Если размером входом считать сумму числа вершин $n_1 + n_2$, то сложность алгоритма будет равняться $\Theta(n_1 + n_2)$. Требуется самостоятельно доказать это.

Лекция 3

Гипотеза Шольца-Брауна, вероятностные пространства

Сложность алгоритма определяется из двух вещей: размер входа и затраты. Мы уже обсудили измерение затрат. Перейдём к обсуждению измерения размера входа.

Рассмотрим уже знакомый нам пример: проверку простоты числа с помощью алгоритма пробных делений. Затраты мы измеряли числом пробных делений, остальные затраты игнорировались. Была получена оценка $\mathcal{O}(\sqrt{n})$. Было показано, что данная оценка является точной. Размер входа совпадал с исходным числом. Пойдём по несколько другому пути: в качестве размера входа возьмём не само число, а его битовую длину. Например, если исходное число 7, то размер входа составит 3. Обозначим такой размер входа как $\lambda(n)$:

$$\lambda(n) = \begin{cases} 1 \text{ при } n = 0; \\ \lfloor \log_2 n + 1 \rfloor \text{ при } n > 0. \end{cases} \tag{3.1}$$

Можно записать и по-другому:

$$\lambda(n) = \lceil \log_2(n+1) \rceil. \tag{3.2}$$

Теперь сложность будет функцией аргумента $\lambda(n) = m$. Оказывается, что эта функция ведёт себя «плавно», в отличие от сложности как функции n. Введём обозначение

$$T^*_{TD}(m) \tag{3.3}$$

для «новой» сложности. В таблице 3.1 представлены некоторые значения функции $T^*_{TD}(m)$. Используется обозначение $\widehat{n}$ для входа, на котором достигается максимум затрат. Асимптотика для функции $T^*_{TD}(m)$ следующая:

Таблица 3.1. Таблица значений функции $T_{TD} * (m)$

m	2	3	4	5	6	7
n	2-3	4-7	8-15	16-31	32-63	64-127
$\widehat{n}$	3	7	13	31	61	127
$T^*_{TD}(m)$	1	1	2	4	6	10

$$T^*_{TD}(m) = \Theta(2^{\frac{m}{2}}). \tag{3.4}$$

Несмотря на то, что функция сложности имеет экспоненциальный рост, она обладает преимуществом перед функцией $T_{TD}(n)$: в отличие от $T_{TD}(n)$, где удавалось дать только точную $\mathcal{O}$-оценку, в случае $T^*_{TD}(m)$ получилось дать Θ-оценку. Остаётся только доказать оценку (3.4). Для этого заметим, что в каждом интервале,

задаваемом числом m, находилось простое число, которое мы и выбирали в качестве $\widehat{n}$. Предположим, что для любой заданной битовой длины m найдётся по крайней мере одно простое число. Вообще говоря, это следует из постулата Бертрана. Постулат Бертрана гласит: для любого $N > 1$ найдётся простое число, лежащее в диапазоне $(N, 2N)$. Постулат Бертрана является даже более сильным утверждением, чем нам необходимо. Покажем это: число битовой длины m — это число, лежащее в диапазоне

$$2^{m-1} \leqslant n \leqslant 2^m. \tag{3.5}$$

Естественно, постулат Бертрана гарантирует нам существование простого числа в таком диапазоне. Само название «постулат» связано с тем, что данное утверждение некоторое время оставалось без доказательства. Бертран высказал предположение в 1845 году. Чебышев доказал его в 1852 году. Доказательство Чебышева достаточно нетривиальное, приводится в данном курсе не будет.

Итак, имеем два различных способа определять размер входа для задач подобного типа. Заметим, что сложность $T^*_{TD}(m)$ будет подпоследовательностью сложности $T_{TD}(n)$, то есть $T^*_{TD}(m)$ менее информативна.

Рассмотрим следующий пример: нахождение a^N, где a — фиксированная величина, N — неотрицательное целое. Самый простой способ: n сомножителей, $n-1$ умножений. Затраты будут измеряться числом умножений. Более удачный способ: бинарный алгоритм или алгоритм повторного возведения в квадрат. Обозначим этот алгоритм как RS (от английского repeating squaring). В алгоритме RS число n записывается в двоичной системе:

$$n = \beta_k \ldots \beta_1 \beta_0. \tag{3.6}$$

Здесь β_i — двоичные символы. Тогда вычислим величины q_i:

$$q_i = a^{(2^i)}. \tag{3.7}$$

После этого перемножим те q_i, для которых соответствующие $b_i = 1$. Приведём пример: пусть ищем a^{11}. Поскольку $(11)_{10} = (1011)_2$, то перемножение будет производиться следующим образом:

$$a^1 1 = a^8 \cdot a^2 \cdot a. \tag{3.8}$$

Оценим количество умножений: имеем три сомножителя, сомножитель a^2 предполагает одно умножение, сомножитель a^8 предполагает ещё два дополнительных умножения. Итого имеем $3 + 1 + 2 = 6$ умножений.

Введём величину $\lambda^*(n)$ — число единиц в двоичной записи числа n. Утверждается, что число умножений составит

$$\lambda(n) + \lambda^*(n) - 2. \tag{3.9}$$

Если же в качестве размера входа брать m, то получим

$$2m - 2. \tag{3.10}$$

Выпишем асимптотику:

$$\begin{aligned} T^*_{RS}(m) &= \Theta(m); \\ T_{RS}(n) &= \Theta(\log n). \end{aligned} \tag{3.11}$$

В «бесхитростном» алгоритме сложность составит $\Theta(n)$. То есть алгоритм RS имеет преимущество. Алгоритм RS применим и при возведении в степень матриц.

Часто говорят об аддитивных цепочках. Для каждого n можно строить последовательность $n_1, \ldots, n_k$. Число n_1 всегда полагается равным единице, последнее число n_k полагается равным n. Последовательность неубывающая (даже возрастающая). Каждый элемент последовательности равен сумме каких-то двух предыдущих чисел, то есть для любого i найдутся такие j и k, что $n_i = n_j + n_k$, причём $j < i$ и $k < i$. Допускается ситуация $j = k$. Сложность алгоритма, соответствующего той или иной адддитивной цепочке длиной k, равна $k - 1$. Таким образом, чем короче будет выстроенная нами аддитивная цепочка, тем меньше будет произведено умножений. Стандартное обозначение $l(n)$ применяется для наиболее короткой аддитивной цепочки. Верно следующее:

$$l(2^n - 1) \leqslant n - 1 + l(n). \tag{3.12}$$

Это так называемая гипотеза Шольца-Брауна. Она всё ещё не доказана.

Домашнее задание. *Построение производится с помощью циркуля и линейки. Затраты измеряются числом проведения прямых и окружностей. Пусть есть отрезок и натуральное число n. Требуется построить отрезок, длина которого составляет $1/n$ часть от длины исходного отрезка. Сложность алгоритма должна составлять $\mathcal{O}(\log n)$.*

Домашнее задание. *Данная задача напрямую касается аддитивных цепочек и бинарного алгоритма возведения в степень. Бинарному алгоритму возведения в степень соответсвует аддитивная цепочка, которую будем называть бинарной цепочкой. Оказывается, для случая $n = 15$ длина бинарной цепочки не совпадает с наименьшей длиной цепочки $l(n)$. Доказать это.*

Далее речь будет идти о сложности в среднем. До сих пор у нас была сложность в худшем случае. Чтобы говорить о сложности в среднем, необходимо определить вероятностное пространство. Берётся множество входов одного размера: $||x|| = s$. Обозначим множество таких x как X_s. Каждое из таких множеств должно быть превращено в вероятностное пространство. При фиксированном s все $x \in X_s$ считаются равновероятными. Число N_s входов заданного размера s предполагается конечным. В таком случае отдельному входу будет соответствовать вероятность $1/N_s$. Рассмотрим две функции:

$$\overline{T}_A(s) = \sum_{x \in X_s} C^T_A(x) P_s(x), \tag{3.13}$$

$$\overline{S}_A(s) = \sum_{x \in X_s} C^S_A(x) P_s(x). \tag{3.14}$$

Здесь $P_s(x)$ — вероятность входа x, $C_A^T(x)$ — временные затраты, $C_A^S(x)$ — пространственные затраты. Функция $\overline{T}_A(s)$ называется временной сложностью в среднем, а функция $\overline{T}_A(s)$ — пространственной сложностью в среднем.

Функции затрат при таком подходе являются случайными величинами. Что такое случайная величина? Функция на вероятностном пространстве. Написанные формулы — это математические ожидания. Сложностями в среднем называются математические ожидания сложностей.

Докажем некоторые простые факты.

$$\begin{aligned} \overline{T}_A(s) &\leqslant T_A(s); \\ \overline{S}_A(s) &\leqslant S_A(s). \end{aligned} \tag{3.15}$$

Докажем это:

$$\overline{T}_A(s) = \sum_{x \in X_s} C_A^T(x) P_s(x) \leqslant \sum_{x \in X_s} (\max_{x \in X_s} C_A^T(x)) P_s(x) = \sum_{x \in X_s} T_A(s) P_s(x) = T_A(s) \sum_{x \in X_s} P_s(x) = T_A(s) \cdot 1 = \tag{3.16}$$

Рассмотрим уже знакомый пример: возведение числа в степень n. В качестве размера входа возьмём битовую длину числа n. Ранее мы уже оценили сложность в худшем случае:

$$T_{RS}(m) = 2m - 2. \tag{3.17}$$

Получим сложность в среднем $\overline{T}_A(s)$.

$$\frac{1}{2^{m-1}} \sum_{n=2^{m-1}}^{2^m-1} (\lambda(n) + \lambda^*(n) - 2) \tag{3.18}$$

Суммирование проводится по диапазону, в котором располагаются числа битовой длины m. Таких чисел $1/2^{m-1}$ штук. Как уже обсуждалось ранее, $\lambda^*(n)$ означает количество единиц в двоичной записи числа n, а $\lambda(n) = m$ — общее число цифр в двоичной записи числа n.

$$= (m-2) \frac{1}{2^{m-1}} \sum_{n=2^{m-1}}^{2^m-1} \lambda^*(n) \tag{3.19}$$

Введём новое обозначение: количество всех единиц в двоичной записи числа n без учёта «старшей» обозначим как k. Тогда количество чисел, имеющих k единиц в двоичной записи помимо «старшей», будет равно числу сочетаний $\binom{m-1}{k}$. Искомая сложность может быть переписана в виде

$$m - 2 + \frac{1}{2^{m-1}} (2^{m-1} + \sum_{k=1}^{m-1} k \binom{m-1}{k} = m - 1 + \frac{1}{2^{m-1}} \sum_{k=1}^{m-1} k \binom{m-1}{k} \tag{3.20}$$

Можно самостоятельно убедиться в том, что

$$k \binom{m-1}{k} = (m-1) \binom{m-2}{k-1}. \tag{3.21}$$

В таком случае

$$= m - 1 + \frac{1}{2^{m-1}}(m-1)\sum_{k=1}^{m-1}\binom{m-2}{k-1} \tag{3.22}$$

Имеем сумму биномиальных коэффицентов. Сделаем замену индекса суммирования:

$$= m - 1 + \frac{1}{2^{m-1}}(m-1)\sum_{l=0}^{m}\binom{m-2}{l} = \frac{3}{2}(m-1). \tag{3.23}$$

Действительно, сложность в среднем оказалась меньше, чем сложность в худшем случае, как и ожидалось.

Обсудим сложность в среднем алгоритма пробных делений.

Асимптотический закон распределения простых чисел. Удобно ввести в рассмотрение функцию $\pi(n)$ — количество простых чисел, меньших числа n. Тогда асимптотический закон формулируется так:

$$\pi(n) \sim \frac{n}{\ln n}. \tag{3.24}$$

Благодаря этому закону можно доказать

$$2^{m-1} \leqslant n \leqslant 2^m \tag{3.25}$$

Лекция 4

Алгоритмы и математическое ожидание

В качестве размера входа рассматривали само число n. Позже в качестве размера входа мы использовали битовую длину числа n. Теперь же мы изучаем сложность в среднем. В качестве размера входа берём битовую длину. Затраты измеряются числом делений, которые нужно произвести. Чтобы изучать сложность в среднем, нужно превратить каждое из множеств входов фиксированного размера в вероятностное пространство. Множество входов фиксированного размера конечно. Иллюзия состоит в том, что, возможно, в среднем алгоритм не так уж и плох, ведь по крайней мере чётные числа проверяются очень быстро. С другой стороны, простых чисел много, а на каждое простое число тратится очень много делений. Сейчас установим, что сложность в среднем не будет полиномиально ограниченной.

На прошлой лекции мы ввели функцию $\pi(n)$. Оказалось, что

$$\pi(n) \sim \frac{n}{\ln n}. \tag{4.1}$$

Этот факт мы оставили без доказательства. Напомним, что

$$2^{m-1} \leqslant n \leqslant 2^m. \tag{4.2}$$

Оказывается,

$$\pi(2^m) \sim \frac{2^m}{(\ln 2)^m} \tag{4.3}$$

$$V(m) \sim \frac{1}{(2\ln 2)m} \cdot 2^m \tag{4.4}$$

Каждое простое число превышает 2^{m-1}. На каждое простое число p алгоритм пробных делений тратит примерно $\sqrt{p}$ делений. Докажем, что асимптотическая нижняя граница для алгоритма пробных делений

$$\Omega(\frac{1}{m}2^{\frac{m}{2}}) \tag{4.5}$$

Тогда не будет справедливо оценка

$$\mathcal{O}(m^d) \tag{4.6}$$

Существует алгоритм, имеющий полиномиальную битовую сложность в худшем случае.

Обсудим сортировки в контексте сложности в среднем. В качестве размера входа возьмём длину массива n. Элементы подразумеваются попарно различными. Для простоты рассматриваем числовые массивы. Сортировка означает упорядочение по возрастанию. Пусть нас интересует сложность по числу сравнений. Зафиксируем размер входа n. Множество входов фиксированного размера n бесконечно. Вск возможные входы разобьем на классы. Множество классов должно быть конечным. Классы должны быть такими, что для любого входа из этого класса

алгоритмом будут проделаны одни и те же действия. Так мы справимся с проблемой бесконечности множества входов фиксированного размера. Ясно, как в случае сортировок объединять входы в классы: если элементы расположены друг относительно друга одинаковым образом, то они принадлежат одному классу. Итак, при рассмотрении сортировки мы можем расматривать не настоящие массивы, а перестановки чисел $1, 2, \ldots, n$. Каждому классу сопоставляется какая-то перестановка. Перестановок существует $n!$ штук. Множество всех перестановок длины n будем обозначать Π_n. Точно так же будем обозначать вероятностные пространства. Вероятность каждого элемента составляет $1/n!$. Рассмотрим событие B_n^v. Здесь нижний индекс n говорит о том, что речь идёт о вероятностном пространстве Π_n. Верхний индекс — число $1 \leqslant v \leqslant n$. Событие состоит в том, что реализована такая перестановка $(a_1, a_2, \ldots, a_n)$, что $a_v = v$. Ясно, что вероятность такого события равна $(n-1)!/n! = 1/n$.

У каждой перестановке есть неподвижные точки. Их может быть даже n штук — для единичной перестановки. Можно сопоставить каждой перестановке число неподвижных точек. Зададимся вопросом: какого математическое ожидание числа неподвижных точек? Оказывается, это число равно единице. Введём случайную величину ζ_n^v:

$$\zeta_n^v = \begin{cases} 1, & \text{если } a_v = v; \\ 0, & \text{если } a_v \neq v. \end{cases} \tag{4.7}$$

Вероятность $P_n(\zeta_n^v) = P_n(B_n^v) = 1/n$. Отсюда следует и результат для математического ожидания

$$E\zeta_n^v = \frac{1}{n}. \tag{4.8}$$

Обозначим ξ_n случайную величину, равную числу неподвижных точек перестановки. Верно следующее:

$$E\xi_n = E(\zeta_n^1 + \ldots + \zeta_n^n) = E\zeta_n^1 + \ldots + \zeta_n^n = 1. \tag{4.9}$$

Здесь мы пользовались известным свойством математического ожидания.

Можно обсуждать полные перестановки, то есть такие, у которых нет неподвижных точек. Например, можно изучить вопрос о количестве таких престановок. Существует задача о шляпах. Посетители оставили шляпы в гардеробе, а когда пришли их забирать, оказалось, что лампочка в гардеробе перегорела. Шляпы пришлось рзбирать наугад. Какова вероятность того, что все придут домой в чужих шляпах? Ответ на задачу о шляпах:

$$\frac{1}{2!} - \frac{1}{3!} + \ldots + \frac{(-1)^n}{n!}. \tag{4.10}$$

Докажите это самостоятельно. Видно, что если n велико, то это $\exp^{-1}$. То есть вероятность довольно большая. Можно интересоваться вопрсоом сколько человек придут в своих шляпах домой? Ответ: в среднем один человек.

Домашнее задание. *Алгоритм поиска минимального элемента в массиве. Псевдокод:*

```
m := x_1;
for i=2 to n do
if x_i < m then m := x_i fi
```

Будем говорить о числе присваиваний. Сколько раз меняется «лидер»? Требуется оценить сложность в среднем по числу присваиваний. Ответ: число имеет порядок $\ln n$.

Домашнее задание. *В книге это 29ая задача.*

Домашнее задание. *Уже знакомая нам задача о сортировке вагонов. Разработанный алгоритм оценить на сложность в среднем. Ответ:* $5n/2 + 1/4 + \mathcal{O}$.

Обсудим теперь сложность в среднем для сортировки простыми вставками. Причём обсуждать будем традиционный вариант. Упорядоченную часть массива будем называть сегментом. Сложность в худшем случае нами уже обсуждалась, она квадратична. Для исследования сложности в среднем введём случайные величины $\xi_n^1, \ldots, \xi_n^{n-1}$, где ξ_n^i — число сравнений на i-м шаге. Сложность в среднем обозначим $\overline{T}_{I_1}(n) = E\xi_n^1 + \ldots + E\xi_n^{n-1}$. Вспомним формулу полного математического ожидания. Пусть есть вероятностное пространство W, распределние вероятностей на ней P, случайная величина ξ. Пусть есть так называемая полная группа событий $W_1, \ldots, W_l$. Эти события попарно несовместны и имеет место равенство $W = W_1 \cup W_2 \cup \ldots \cup W_l$. Это равенство будем называть разложением пространства W полной группе событий. Формулой полного математического ожидания называется формула

$$E\xi = \sum_{k=1}^{l} E(\xi|W_k)P(W_k). \tag{4.11}$$

Здесь $E(\xi|W_k)$ — условное математическое ожидание.

Что взять в качестве полной группы событий в случае алгоритма сортировки? Обозначим $H_n^{u,v}$ событие, состоящее в том, что в перестановке $(a_1, \ldots, a_n)$ одержится ровно u штук элементов a_i таких, что $a_i < a_v$. Эти события дадут нам полную группу событий. Заметим, что

$$P(H_n^{u,v}) = \frac{1}{v}. \tag{4.12}$$

Это равенство следует получить самостоятельно. Рассмотрим события $H^{k,i+1}$ при $k = 0, 1, \ldots, i$ образуют полную группу событий пространства Π_n. Тогда

$$E\xi_n^i = \frac{1}{i+1} \sum_{k=0}^{i} E(\xi_n^i | H_n^{k,i+1}). \tag{4.13}$$

Кроме этого,

$$\xi_n^i = \begin{cases} i - k + 1 \text{ если } k > 0; i \text{ если } k = 0. \end{cases} \tag{4.14}$$

Верно

$$E(\xi_n^i | H_n^{k,i+1}) = \begin{cases} i - k + 1 \text{ если } k > 0; i \text{ если } k = 0. \end{cases} \tag{4.15}$$

Теперь можем применить формулу полного математического ожидания.

$$E\xi_n^i = \frac{1}{i+1}(i + \sum_{k=1}^{i}(i-k+1)) = \frac{i}{2} + 1 - \frac{1}{i+1} \tag{4.16}$$

Получаем выражение для сложности:

$$n - 1 + \sum_{i=1}^{n}(\frac{i}{2} + \frac{1}{i+1}) \tag{4.17}$$

Воспользуемся формулой

$$\sum_{i=1}^{n}\frac{1}{i} = \ln n + \mathcal{O}(1). \tag{4.18}$$

Итак,

$$\overline{T}_{I_1}(n) = \frac{(n+4)(n-1)}{4} - \ln n. \tag{4.19}$$

Исследуемая сложность оказывается квадратичной:

$$\overline{T}_{I_1}(n) = \frac{n^2}{4} + \mathcal{O}(n). \tag{4.20}$$

Не следует ожидать, что сложность в среднем будет средним арифметическим между сложностью в худшем и в лучшем случаях. Сложность в среднем является более сложной конструкцией.

А что же со сложностью в среднем по числу перемещений? Алгоритм устроен так, что число перемещений практически совпадает с числом сравнений, поэтому сложность будет точно такой же.

Сложность в среднем обладает замечательным свойством. Пусть есть несколько различных операций и сложности в среднем для каждой из них. Оказывается, в отличие от сложности в худшем случае, сложности в среднем можно складывать. Это следует из аналогичного свойства математического ожидания. Это выгодное отличие сложности в среднем от сложности в худшем случае. Таким образом, для случая сортировок получаем общую сложность в среднем по обеим операциям $n^2/2$.

Лекция 5

Элементарные сортировки

Что касается пространственной сложности алгоритмов сортировки, то это всегда будет $\mathcal{O}(1)$, поскольку новых ячеек памяти алгоритм не занимает.

Обсудим сортировку пузырьком. Там всё дело в числе проходов по массиву. Происходит поиск разупорядоченной пары. Пара упорядочивается, далее проход начинается с начала. Сколько проходов потребуется? Снова предполагется, что все варианты взаимного расположения элементов в массиве равновероятны, а элементы попарно различны. Вместо настоящих массивов будем рассматривать перестановки длины n. Можно доказать, что для числа проходов верно

$$s(n) = n - \sum_{k=0}^{n} \frac{k!k^{n-k}}{n!}. \tag{5.1}$$

Отсюда следует, что $s(n) < n$ и

$$s(n) \geqslant \frac{n-1}{2}. \tag{5.2}$$

Этот факт легко получить самостоятельно. Отсюда следует, что

$$s(n) = \Theta(n). \tag{5.3}$$

Далее идёт рассуждение про то, что от n^2 деваться некуда (уточни этот момент по книге).

Рассмотрим пример, когда сложность в среднем растёт существенно медленнее, чем сложность в худшем случае. Речь пойдёт о так называемой быстрой сортировке (quick sort). Этот алгоритм принято обозначать QS. Сложность в худшем случае алгоритма быстрой сортировки:

$$T_{QS}(n) = \Theta(n^2). \tag{5.4}$$

Сейчас же нас интересует сложность в среднем $\overline{T}_{QS}(n)$. Приведём стандартное доказательство. Но перед этим приведём некоторые факты, использование которых потребуется при проведении доказательства:

$$\overline{T}_{QS}(n) = n + 1 + \frac{1}{n}\sum_{i=1}^{n}(\overline{T}_{QS}(i-1) + \overline{T}_{QS}(n-1)) \tag{5.5}$$

$$\overline{T}_{QS}(0) = \overline{T}_{QS}(1) = 0 \tag{5.6}$$

Пусть в качестве разбивающего элемента выбран первый элемент массива. Разбивающий элемент сравнивается с каждым из оставшихся $n-1$ элементов массива. С вероятностью $1/n$ разбивающий элемент может занять одно из возможных мест. Корректна ли эта формула? Неплохо бы ещё доказать, что после этапа разбиения перестановки слева и справа от разбивающего элемента соответствующей длины

равновероятны. Откуда это следует? Это можно доказать, но мы не будем. Почему? Потому что более употребительным является вариант, когда разбивающий элемент выбирается случайно. Там будем аккуратно всё обсуждать, а тут позволим себе опустить доказательство.

Итак, имеем реккурентное сооотношение. Перепишем его в следующем виде:

$$\overline{T}_{QS}(n) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}_{QS}(k). \tag{5.7}$$

Преобразуем равенство таким образом, чтобы избавиться от знаменателя:

$$n\overline{T}_{QS}(n) = n(n-1) + 2\sum_{k=0}^{n-1}\overline{T}_{QS}(k). \tag{5.8}$$

Аналогичное равенство можно написать для $\overline{T}_{QS}(n-1)$ простой заменой аргумента n на $n-1$:

$$(n-1)\overline{T}_{QS}(n-1) = (n-1)(n-2) + 2\sum_{k=0}^{n-2}\overline{T}_{QS}(k). \tag{5.9}$$

Вычтем из равенства (5.8) равенство (5.9):

$$n\overline{T}_{QS}(n) - (n-1)\overline{T}_{QS}(n-1) = n(n-1) + 2\sum_{k=0}^{n-1}\overline{T}_{QS}(k) - (n-1)(n-2) - 2\sum_{k=0}^{n-2}\overline{T}_{QS}(k). \tag{5.10}$$

Приведём подобные:

$$n\overline{T}_{QS}(n) - (n+1)\overline{T}_{QS}(n-1) = 2n - 2. \tag{5.11}$$

Разделим равенство на $n(n+1)$:

$$\frac{1}{n+1}\overline{T}_{QS}(n) - \frac{1}{n}\overline{T}_{QS}(n-1) = \frac{2(n-1)}{n(n+1)}. \tag{5.12}$$

Введём обозначение

$$t(n) = \frac{1}{n+1}\overline{T}_{QS}(n). \tag{5.13}$$

Тогда равенство перепишется следующим образом:

$$t(n) - t(n-1) = \frac{2(n-1)}{n(n+1)}. \tag{5.14}$$

Ясно, что $t(0) = 0$. Получили простейшее конечно-разностное уравнение. Такое уравнение решается с помощью суммирования. После суммирования получим

$$t(n) = 2\sum_{k=1}^{n}\frac{(k-1)}{k(k+1)} = 2\sum_{k=1}^{n}\frac{1}{k+1} - 2\sum_{k=1}^{n}\frac{1}{k(k+1)} \tag{5.15}$$

$$t(n) = 2\ln n + \mathcal{O}(1). \tag{5.16}$$

Вернёмся от $t(n)$ к $\overline{T}_{QS}(n)$:

$$\overline{T}_{QS}(n) = 2n\ln n + \mathcal{O}(n). \tag{5.17}$$

Итак, мы вывели формулу для сложности в среднем.

Домашнее задание. *Доказать для сложности в среднем $\overline{T}_{QS}(n)$ формулу*

$$\overline{T}_{QS}(n) = 2(n+1)H_n - 4n. \tag{5.18}$$

Здесь $H_n = \sum_{k=1}^{n} 1/k$.

Домашнее задание. *Рассмотрим навес из костяшек домино. Принцип такой: на одну костяшку с некоторым смещением можем положить другую, а на неё третью. Можно соорудить навес любой наперёд заданной длины l. Если дано l, то сколько костяшек потребуется?*

Пространственная сложность составит $\mathcal{O}(1)$. Однако это рекурсивная сортировка, а значит при её выполнении будет использован стэк отложенных заданий. Пусть первый этап разбиения прошёл и разбивающий элемент встал на своё место. Та же самая процедура должна быть применена к левой и к правой части. Однако они имеют различную длину. С какой части лучше начинать — с более короткой или с более длинной? Выбор более короткой или более длиннйо части влияет на объем стэка. С точки зрения объема стэка более выгодно браться за короткую часть. Можно доказать, что при таком подходе длина стэка отложенных заданий не будет превосходить $\log_2 n$.

Рандомизированные алгоритмы

Ранее упоминалось, что более употребительным вариантом алгоритма быстрой сортировки является алгоритм, где разбивающий элемент выбирается случайно. Мы приходим к идее рандомизированных алгоритмов: таких, в которых присутствуем элемент случайности. Рандомизированные алгоритмы бывают двух типов: типа Монте-Карло и типа Лас-Вегас. В таких алгоритмах затраты, связанные с получением ответа, не определяются однозначно входом. С подобным мы сталкиваемся впервые. Чтобы говорить о сложности таких алгоритмов, необходимо вводить вероятностные пространства. Над каждым входом растёт пространство сценариев — мы заведомо не знаем, какой из них реализуется. Таким образом, с каждым входом связано вероятностное пространство. На этих сценариях определяется случайная величина, равная затратам. Усреднёнными затратами мы называем математическое ожидание данной величины. Усреднение происходит по сценариям. Далее действуем точно так же как действовали раньше.

Рассмотрим пример. Пусть есть массив. Массивом, сожержащим большинство, называется такой массив, что больше половины его элементов имеют одно и то же значение. Пусть наш массив является массивом, содержащим большинство. Задача ставится следующим образом: определить индекс какого-либо элемента, который

входит в большинство. Рандомизированный алгоритм состоит в том, чтобы взять случаный элемент и посмотреть, принадлежит ли он большинству путем прохода через все элементы массива. Сложность такого алгоритма по числу сравнений будет менее чем $2n$, где n — число элементов в массиве. Сценариями в данном случае являются объекты $(i_1, \ldots, i_{m-1}, i_m)$, характеризующиеся тем, что элементы с индексами $i_1, \ldots, i_{m-1}$ не принадлежат большинству, а элемент с индексом i_m принадлежит большинству. Обозначим усреднённые затраты как a. Вероятность того, что выбранный элемент попадает в большинство, превышает $1/2$: $p > 1/2$. Поэтому верно

$$a = p(n-1) + (1-p)(a+n-1) \tag{5.19}$$

С вероятностью p выбран элемент из большинства и потребуется $n-1$ сравнение. С вероятностью $1-p$ это не элемент из большинства, поэтому потребуется провести $n-1$ сравнений, чтобы в этом убедиться, а потом придётся начать процедуру с начала.

Рандомизированные алгоритмы могут рассматриваться как множество детерминированных алгоритмов, на котором определена вероятность. Вообще говоря, этот подход очень близок к подходу сценариев.

Рандомизация сортировки состоит в том, что первый разбивающий элемент выбирается случайно. Мы считаем, что в нашем распоряжении имеется функция $\mathrm{random}(N)$, выдающая случайное число от 1 до N. Пусть есть полоска клетчатой бумаги длины n. Задача состоит в том, чтобы разрезать полоску на отдельные клетки. Выбирается случайное число i и вырезается i-ая клетка. За эту операцию разрезальщику платят $n-1$ рубль.

Лекция 6

Кстати, сложность в среднем не должна превышать сложность в худшем случае, однако может с ней совпадать. Например, так происходит при сортировке выбором. При сортировке выбором мы ищем наименьший элемент и меняем его местами с первым. Повторяем процедуру для оставшегося куска массива.

Продолжим разбирать рандомизирвоанный вариант быстрой сортировки. Какое отношение имеет сложность рандомизирванного алгоритма к сложности в среднем? Вероятностных пространств рассматривается столько, сколько значений размера возможны. А в рандомизирвоанном алгоритме с каждым входом связывается своё вероятностное пространство — прсотранство всех вычислительных сценариев. Вход один, но заранее мы не знаем, по какому сценарию пойдёт вычисление. Когда мы значем, каковы будут затраты по каждому сценарию, то можно рассматривать математическое ожидание этих затрат, которе мы называем усредненными затратами. Таким образом, для каждого входа имеем функцию затрат. После это действуем так, как действовали со сложностью в худшем случае. Можно рассматривать и сложность в среднем. Однако мы будем обсуждать именно сложность в худшем случае.

Что касается рандомизирвоанной сортировки, оказывается, что усредненные затраты для каждого входа одинаковы.

На прошлой лекции была сформулирована задача о разрезании полоски клетчатой бумаги. Вопрсо формулировался так: сколько в среднем придётся заплатить за разрезание всей полоски на отдельные клетки? Как связана эта задача с рандомизированной быстрой сортировкой? Случайный выбор клетки, которая будет вырезана, соответствует случайному выбору разбивающего элемента. Стоимость разрезания $n-1$ соответсвует $n-1$ сравнению.

Изучим возможные сценарии, которые могут возникнуть в алгоритме разрезания бумажной полоски длиной в три клетки. Один вариант: слева может не оказаться ни одного элемента, а справа два, а вторым разрезанием полоска длины два разрезается так, что слева не остаётся ни одной клетки, а справа остаётся одна клетка. Или же при втором разрезании слева может оказаться одна клетка, а справа ни одной. Также возможен вариант, когда при первом вырезании вырезается центральная клетка и полоска тут же распадается на две отдельные клетки. Кроме того, возможны ещё два сценария, симметричные описанным первым двум. Естественно написать следующую реккурентную форумулу:

$$P_n(s) = \frac{1}{n} P_{i-1}(s') P_{n-i}(s''). \tag{6.1}$$

Если вернуться к нашему примеру и воспользоваться этой формулой, получим вероятности $1/3$ и $1/6$.

$$S_n = S_n^1 \cup S_n^2 \cup \ldots \cup S_n^n. \tag{6.2}$$

Верхний индекс означает номер клетки, которая вырезается при первом шаге. Ясно, что это полная группа событий и разложение допустимо. Очевидно,

$$P_n(S_n^1) = \ldots = P_n(S_n^n) = \frac{1}{n}. \tag{6.3}$$

Обозначим случайную величину для затрат
chi_n на пространстве сценариев S_n. Справедливо

$$\chi_n = n - 1 + \chi'_n(s) + \chi''_n(s). \tag{6.4}$$

Здесь $\chi'_n(s)$ — затраты по левому подсценарию, а $\chi''_n(s)$ — затраты по правому подсценарию. Теперь мы можем воспользоваться формулой полного математчиеского ожидания.

$$E\chi_n = \sum_{i=1}^{n} E(\chi_n|S_n^i)\frac{1}{n} = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(E(\chi'_n|S_n^i) + E(\chi''_n|S_n^i)). \tag{6.5}$$

Избавимся от фунцкий со штрихами:

$$= n - 1 + \frac{1}{n}\sum_{i=1}^{n}(E\chi_{i-1} + E\chi_{n-i}) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} E\chi_k \tag{6.6}$$

Перпишем то что мы получили:

$$E\chi_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} E\chi_k. \tag{6.7}$$

Мы уже сталкивались с таким соотношением, когда имели дело с быстрой сортировкой без рандомизации. Будем опираться на то, что уже было сделано, и выпишем ответ:

$$E\chi_n = 2n\ln n + \mathcal{O}(n). \tag{6.8}$$

Выпишем результат для сложности:

$$\widetilde{T}_{QS} = 2n\ln n + \mathcal{O}(n). \tag{6.9}$$

Волна над $\widetilde{T}$ отсылает к рандомизированности алгоритма. Сложность по числу перемещений элементов здесь мало отличается от сложности по числу сравнений.

Домашнее задание. *Пусть исходно есть некоторая перестановка длины n: $(a_1, a_2, \ldots, a_n)$. К этой перестановке мы применяем следующий алгоритм:*

```
down to 2 do
j := 1 + random(i-1);
a_i <-> a_j
```

Требуется доказать, что описан генератор случайных перестановок, то есть может возникнуть любая перестановка длины n с одинаковой вероятностью.

На этом мы заканчиваем раздел о рандомизированных алгоритмах.

Алгоритм Евклида

Обратим внимание на то, что часто алгоритм представляет собой серию однотипных шагов. Шаги более-менее равнозатратны. Тогда всё дело упирается в оценку числа шагов. Расмотрим алгоритм Евклида.

$$\begin{aligned}
&a_0 \geqslant a_1;\\
&a_0 = q_1 a_1 + a_2;\\
&a_1 = q_2 a_2 + a_3;\\
&\dots\\
&a_{n-3} = q_{n-2} a_{n-2} + a_{n-1};\\
&a_{n-2} = q_{n-1} a_{n-1} + a_n;\\
&a_{n-1} = q_n a_n.
\end{aligned} \tag{6.10}$$

Остаток всегда меньше делителя, поэтому последовательность убывающая. Как оценить число делений? Ясно, что число делений с остатком $C_E(a_0, a_1)$ не превышает a_1:

$$C_E(a_0, a_1) \leqslant a_1. \tag{6.11}$$

Можно ли получить более сильную оценку? Для того чтобы это проделать, необходимо определить неотрицательную целочисленную убывающую функцию на парах чисел. Чило шагов не превзойдёт значение функции на первой паре. **Предложение:** пусть k и l — натуральные числа, $k > l$, r — таток от деления k на l. Пусть $\lambda(k)$ — битовая длина числа k. Тогда

1) $\lambda(k) > \lambda(r)$;

2) $L(k, l) = \lambda(k) + \lambda(l) - 2$ верно $L(k, l) < L(l, r)$.

Функцию $L(k, l)$ можно брать в качестве искомой функции. Видно, что она удовлетворяет всем предъявляемым требованиям. **Доказательство.** Запишем исходное равенство

$$k = ql + r. \tag{6.12}$$

Поскольку $k \geqslant l + r$, а $l > r$, то $k \geqslant 2r$. Итак, первый пункт доказан.

$$\lambda(k) + \lambda(l) - 2 > \lambda(l) + \lambda(r) - 2. \tag{6.13}$$

Теперь можно записать

$$C_E(a_0, a_1) \leqslant \lambda(a_0) + \lambda(a_1) - 2. \tag{6.14}$$

Отсюда следует

$$C_E(a_0, a_1) \leqslant 2\lambda(a_1) - 1. \tag{6.15}$$

В свою очередь, эта оценка даёт следующее:

$$T_E(a_1) = \max_{a_0 \geqslant a_1} C_E(a_0, a_1) \leqslant 2\lambda(a_1) - 1 = 2\lfloor \log_2 a_1 \rfloor + 2 - 1 \leqslant 2\log_2 a_1 + 1. \tag{6.16}$$

Для алгоритма Евклида оказывается верна логарифмическая оценка.

Пояснение почему выбрали в качестве обозначения букву L: функция Ляпунова — функция, которая убывает вдоль решения дифференциального уравнения. Идея, использованная нами, очень похожа, поэтому мы испольщовали ту же самую букву.

Алгоритм Евклида имеет большое число обобщений. В частности, используя эту последовательность шагов и дополнив её, можно получить числа s и t такие, что

$$sa_0 + ta_1 = \text{ наибольший общий делитель } (a_0, a_1). \tag{6.17}$$

Здесь имеется ввиду $s = s_n$, $t = t_n$.

$$\begin{aligned} &s_0a_0 + t_0a_1 = a_0; \\ &s_1a_0 + t_1a_1 = a_1; \\ &\dots \\ &s_ia_0 + t_ia_1 = a_i. \end{aligned} \tag{6.18}$$

Изначально мы знаем s_0, t_0, s_1, t_1. Как двигаться дальше? Необходимо после каждого деления с остатком

$$a_{i-1} = q_ia_i + a_{i+1}. \tag{6.19}$$

$$a_{i+1} = a_{i-1} - q_ia_i. \tag{6.20}$$

Тогда

$$(s_{i-1} - q_is_i)a_0 + (t_{i-1} - q_it_i)a_1 = a_{i+1}. \tag{6.21}$$

$$\begin{aligned} s_{i+1} &= s_{i-1} - q_is_i; \\ t_{i+1} &= t_{i-1} - q_it_i. \end{aligned} \tag{6.22}$$

Это дополнение к алгоритму евклида усложняет каждый шаг на две мультипликативные и две аддитивные операции. Это не повлияет на асимптотику сложности — всё равно будет логарифм. Другое замечательное обстоятельство заключается в том, что можно вычислять только s, игнорируя t. Зная, что s и t удовлетворяют равенству (11.4), можно легко установить t, зная s. Расширенный алгоритм Евклида будем обозначать EE.

Лекция 7

Полиномиальные языки

Введём необходимые обозначения. Расматриваем машину Тьюринга со следующими параметрами. Буквой C будем обозначать набор символов

$$C = \{c_0, \ldots, c_m\}, \tag{7.1}$$

где $c_0 = \Lambda$ — алфавит ленты машины Тьюринга. Будем рассматривать множество Q:

$$Q = \{q_0, \ldots, q_l\} \tag{7.2}$$

— алфавит машины Тьюринга. Будем рассматривать программу Π, то есть множество команд вида

$$\Pi = \{(c_i, q_j) \to (c_k, q_r, T)\}. \tag{7.3}$$

Входной символ и состояние маштны сопоставляются тройке, где T — символ движения R, L, S (вправо, влево, на месте). Предполагается бесконечная лента, ячейки пронумерованы от 1 и до бесконечности, влево ячейки 0, -1, и так далее. В каждый момент машина обозревает одну из ячеек.

Машина Тьюринга M называется детерминированной, если в её программе для каждой пары (c_i, q_j) существует не более одной команды с соответсвующим префиксом. Если же машина является недетрминированной, то для каждой пары (c_i, q_j) существует несколько команд.

Существует два возможных подхода. Мы будем пользоваться подходом сертификата.

Функционирование машины Тьюринга M. Рассмотрим входной алфавит $C_0 = C \setminus \{a_0\}$, рассмотрим $A \subseteq C_0$. Машина M реализует отображение

$$\varphi_M : A^* \to {}^*_0 \cup \omega \tag{7.4}$$

Стандартное обозначение A^* используется для множества всех конечных слов в алфавите A. Символ ω — специальный пустой символ, отсутствие остановки. Если же машина Тьюринга остановилась, то

$$\varphi_M(\overline{a}) = \overline{b}. \tag{7.5}$$

Здесь $\overline{a} \in A$, а $\overline{b}$ — слово на ленте машины Тьюринга после её остановки, причём мы требуем, чтобы это слово не содержало пустых символов и из подряд идущих символов алфавита C_0*. Машина всегда запускается с первой ячейки и работает в дискретные моменты времени $t = 1, 2, \ldots$. Машина запускается над словом $\overline{a} \in A$ далее возможны два варианта: машина либо не останавливается, либо останавливается.

У функции может быть и два аргумента. Можно рассматрвиать функции из любого конечного числа аргументов. Тогда слово $\overline{a} \in A$ будет записано как несколько слов, разделённых пробелами.

Пусть $D \in C_0$ — выходной алфавит, а φ — отображение $\varphi : A^* \to D^*$. Отображение φ — алгоритм, преобразование. Говорят, что φ — полиномиальный алгоритм,

если сущестует машина Тьюринга M такая, что $\varphi_M = \varphi$ и существует такая полиномиальная фунцкия $q(n)$, $n = 1, 2, \ldots$, такая что время работы машины M над входом $\overline{a}$ длины n не превышает $q(n)$. Здесь не рассматривается вариант, когда машина Тьюринга не останавливается.

Как оценивать сложность алгоритмов? Для простоты будем считать, что $D = \{0, 1\}$, φ — алгоритм распознавания, выходное слово всегда имеет единичную длину $\lambda(\varphi(\overline{a})) = 1$, то есть выходное слово — либо ноль, либо единица. Тогда вся сложнсоть задачи будет сосредоточена в анализе входного слова. Такие отображения будем называть языками. Выделим множество $L = \{\overline{a} \in A : \varphi(\overline{a}) = 1\ \}$ и назовём его языком, распознаваемым с помощью отображения φ.

Введём класс P — класс полиномиальных языков, то есть языков, распознаваемых за полиномиальное время. Будем говорить, что язык $L_1 \in A^*$ полиномиально сводится к языку $L_2 \in D^*$, если существует полиномиальное преобразование $\varphi : A^* \to D^*$, что $\overline{a} \in L_1 \Leftrightarrow \varphi(\overline{a}) \in L_2$. Здесь $\overline{a} \in A^*$, а $\varphi(\overline{a}) \in D^*$.

Докажем утверждение о полиномиальной сводимости языков. Утверждение: Пусть $L_1 \in A^*$, $L_2 \in D^*$, $L_2 \in P$. Тогда $L_1 \in P$. Доказательство. Пусть машина $M_{1,2}$ — машина Тьюринга, полиномиально сводящая язык L_1 к языку L_2. Пусть машина M_2 — машина Тьюринга, полиномиально распознающая язык L_2. Тогда легко понять, как построить машину Тьюринга M_1, полиномиально распознающая язык L_1. Запустим над словом $\overline{a}$ машину $M_{1,2}$, а над результатом запустим машину M_2:

$$\overline{a} \longrightarrow \overline{b} = \varphi(\overline{a}) \longrightarrow \begin{cases} 1 \text{ если } \overline{b} \in L_2 \\ 0 \text{ если } \overline{b} \notin L_2 \end{cases} \tag{7.6}$$

Итак, мы провели распознавание суперпозицией двух машин Тьюринга.

Чтобы наши рассуждения не казались совсем уж какой-то абстракцией, рассмотрим как примеры два конкретных языка — язык выполнимости и язык Клика.

Пример №1. Язык выполнимости.

Пусть S — алфавит языка ВЫП(КНФ)=$\{x, 0, 1, \vee, \wedge, \mathrm{NOT}, (,)\}$. В этом алфавите, по сути, мы можем записать любую формул с поднятыми отрицаниями. Мы договорились, что переменную с номером x_i будем записывать следующим образом:

$$x_i \longrightarrow x\{\text{двоичная запись числа } i\}. \tag{7.7}$$

Пусть $\bar{a} \in S^*$ — запись некоторой КНФ. В этом случае:

$$\bar{a} \in \text{Вып} \Longleftrightarrow \exists \alpha = (x_1 \ldots x_n) : \ L(\alpha) = 1. \tag{7.8}$$

То есть это будет множество тех коньюнктивных нормальных форм, которые обращаются в единицу хотябы на одном наборе.

Пример №2. Язык клика. Это уже пример языка, связанного с графами. $\hat{S} = \{0, 1, ; \}$. Смысл этого языка следующий. Дается неориентированный граф G с множеством вершин V и с множеством рёбер E без параллельных рёбер, и некоторое число k. Сопостовляем этому графу его матрицу смежности. После этого будем построчно читать эту матрицу смежности и записывать:

$$\{(\text{Первая строка матрицы}); \ldots ; (\text{Последняя строка матрицы}); k\}, \tag{7.9}$$

причём число k записываем в двоичной системе. Языком клика называется множество таких слов, которые отличаются тем, что в графе G есть клика мощности k (полный подграф на k вершинах).

Теперь довайте попробуем доказать следующее нетривиальное утверждение.

Утверждение. Язык выполнимости полиномиально сводится к языку клика.

Доказательство. Нам нужно научиться полиномиально переводить КНФ в (G, k) таким образом, что выполнимость КНФ означает, что в нашем графе есть клика мощности k. Конструировать преобразование мы будем следующим образом. Для начала каждой букве элементарной дизъюнкции сопоставим множество вершин некоторого графа, и так для каждой элементарной дизъюнкции. Множество вершин, которые сопоставляются каждой элементарной дизъюнкции, будем воспринимать как некие компоненты связности в том смысле, что не будем проводить между ними рёбра. То есть рёбра проводятся только между вершинами, относящимися к разным элементарным дизъюнкциям. Причем рёбра мы будем проводить между буквами, не являющимися противоположными (их конъюнкция равна единице). Другими словами, между двумя буквами t и t' проводится ребро, если $t \cdot t' \not\equiv 0$. Граф, который мы получили таким образом, и есть наш граф G. Оказывается, что наше КНФ выполнимо, если в построенном графе есть клика мощности S. Также понятно, что все наши построения полиномиальны, и никаких экспоненциальных сложностей не возникает. $\square$

Определение. Язык L, лежащий в алфавите A^*, входит в класс $\mathbb{NP}$, если существует алфавит D, полином натурального аргумента q и предикат $Q(x, y)$ на $A^* \times D^*$ такие, что $Q \in \mathbb{P}$ и

$$\forall \bar{a} \in A^* \text{ и } \bar{a} \in L \iff \exists \bar{d} \in D^* : \ \lambda(d) \leq g(\lambda(a)), \ Q(\bar{a}, \bar{d}) = 1. \tag{7.10}$$

Слово $\bar{d}$ называется сертификатом.

Глядя на это определение не очень понятно, как оно работает, поэтому разберёмся, что здесь есть что на примере языка клика. Установим соответствие:

- $\bar{a}$ есть (G, k).
- Сертификат $\bar{d}$ есть список номеров тех k вершин графа G, которые образуют клику.

Предикат проверки легко вычисляется за полиномиальное время. Тем самым мы показали, что язык клика является языком из класса $\mathbb{NP}$.

Утверждение. Класс $\mathbb{P}$ по крайней мере вложен в класс $\mathbb{NP}$. Данное утверждение приводим без доказательства. Справедливость ли обратного включения науке неизвестно.

Определение.
Говорят, что язык L является $\mathbb{NP}$ — трудным, если любой язык $L' \in \mathbb{NP}$ полиномиально сводится к языку L.
Говорят, что язык L является $\mathbb{NP}$ — полным, если он является $\mathbb{NP}$ — трудным и сам входит в класс $\mathbb{NP}$.

Утверждение. Пусть язык L_1 является $\mathbb{NP}$ — трудным и полиномиально сводится к L_2, то язык L_2 тоже является $\mathbb{NP}$ — трудным.

Теорема Кука. Язык выполнимости является $\mathbb{NP}$ — полным. Идея состоит в том, что мы доказали, что язык выполнимости полиномиально сводится к языку клика, который лежит в классе $\mathbb{NP}$, а это значит, что и сам язык выполнимость лежит в классе $\mathbb{NP}$. Для того, чтобы закончить доказательство этой теоремы, надо доказать, что любой язык из класса $\mathbb{NP}$ полиномиально сводится к языку выполнимости.

Лекция 8

Интегральные схемы

Современные интегральные схемы создаются на куске кремния с помощью специальной процедуры, которая называется «*фотолитография*». Это очень сложный процесс, в наших реалиях позволяющий создавать на масштабах порядка десятков нанометров схемы из активных элементов, соединённых проводниками.

На заре этой технологии всё делалось вручную. Если нужно было на поверхности кремния создать некую геометрическую структуру, то под это руками вырезались специальные шаблоны. Сейчас же вырезать руками что-то размером 10 нанометров не представляется возможным, поэтому используются более продвинутые технологии.

Сам процесс фотолитографии представляет собой примерно следующее. Глобальная задача состоит в том, чтобы на поверхности кремния создавать простые фигуры типа прямоугольников. На поверхность кремния наносится специальный слой, который называется фоторезистром (некоторая химическая структура, которая может менять свои свойства под действием ультрафиолета). Далее нужные части пластины путём наложения специальной плёнки засвечиваются ультрафиолетом, за счёт чего происходят структурные изменения вещества в некоторой области пластины. После этого фоторезист смывается, и получается некоторый фотошаблон. В дальнейшем, используя эту маску, можно напылять металл, ионизировать и проделывать другие манипуляции с интересующей нас геометрической структурой на поверхности кремния. Это, по сути, процесс, который помогает точно и аккуратно рисовать на пластинке. Это основы работы этого метода, хотя сейчас он выглядит гораздо сложнее. Теперь приходится каким-то образом создавать не только двухмерные, но и трёхмерные структуры, и так далее.

Многие слышали про так называемый «*закон Мура*». В вольной интерпретации он утверждает, что число транзисторов в инетегральной схеме должно удваиваться каждые два года. Однако сейчас этот закон (или, точнее говоря, закономерность) работает плохо, так как современная промышленность дошла до состояния, когда увеличивать число транзисторов на единицу площади невозможно ввиду чисто физических ограничений (делать транзисторы на масштабах меньше 10 нанометров фактически не имеет смысла).

Про современные интегральные схемы нужно понимать следующие вещи:

- Современные интегральные схемы невозможно спроектировать вручную.
- Нужны специальные программы для автоматизации различных этапов проектирования интегральных схем.
- При работе с интегральными схемами требуется понимание не только возникающих при этом математических задач, но и особенностей технологий производства.

Так как современные схемы содержат в себе невообразимое число транзисторов, то думать о схеме в терминах этих транзисторов становится уже невозможно. Поэтому используют следующие *уровни абстракции:*

1) Уровень топологии. На этом уровне из геометрических примитивов составляются принципиальные транзисторные схемы. Происходит моделирование топологии (структуры и геометрии всех слоёв) проектируемого устройства.

2) Транзисторный уровень. На данном уровне происходит моделирование структуры основных логических элементов интегральной схемы, а также определение и оценка основных физических характеристик логических элементов (размер, задержка, энергопотребление и так далее).

3) Логический (схемный) уровень. На данном уровне происходит моделирование структуры и основных элементов блока, реализующего данный процесс/сигнал.

4) Автоматный (поведенческий) уровень. На данном уровне рассматривается поведение какого-то отдельного блока с заданными входами и выходами. Примером такого блока может служить сумматор. То есть происходит моделирование поведения/функционирования процесса/сигнала.

5) Системный уровень. На этом уровене выполняется моделирование системы взаимодействующих процессов/сигналов.

Основные тратегии проектирования интегральных схем следующие:

- Заказное проектирование. При этом все основные элементы интегральной схемы проектируются индивидуально и вручную. Высокая стоимость.

- Полу-заказное проектирование. При этом используются заранее спроектированные элементы (IP-блоки, библиотечные элементы), средства автоматизации.

- Программируемые интегральные схемы. При этом все основные элементы интегральной схемы заранее спроектированы, и возможно настраивать устройство во время работы и/или на этапе проектирования.

Упрощённый маршрут проектирования:

$$\begin{aligned}&\text{Спецификация системы} \Longrightarrow\\ \Longrightarrow&\text{Проектирование архитектуры} \Longrightarrow\\ \Longrightarrow&\text{Функциональное проектирование} \Longrightarrow\\ \Longrightarrow&\text{Логическое проектирование} \Longrightarrow\\ \Longrightarrow&\text{Физическое проектирование} \Longrightarrow\\ \Longrightarrow&\text{Верификация топологии} \Longrightarrow\\ \Longrightarrow&\text{Изготовление} \Longrightarrow\\ \Longrightarrow&\text{Корпусирование и финальное тестирование.}\end{aligned}$$

Теперь немного о ДНФ. Основные подходы к минимизации:

- Эквивалентные преобразования. Сложно применять при большом числе переменных. Неясно, как оценить качество полученного результата.

- Метод карт Карно. Применим только при малом числе элементов.
- Метод Квайна. Экспонециальная сложность в худшем случае.

Также посмотрим на применения каталогов оптимальных схем:

- Основа для построения логических элементов.
- Системы логического синтеза на основе эквивалентных преобразований.
- Исследование структуры и других свойств оптимальных схем.

Лекция 9

Схемы и сложность оценки

Пусть U — полный класс схем, Ψ — функционал сложности схем из класса U. Он обладает следующими свойствами:

1) Функционал неотрицателен:

$$\forall \Sigma \in U \Longrightarrow \Psi(\Sigma) \geq 0. \tag{9.1}$$

2) Функционал монотонен.

Примеры классов схем:

- Класс дизъюнктивных нормальных форм.
- Класс формул в стандартном базисе.
- Схемы из функциональных элементов.
- Формулы в произвольном базисе.
- Схемы из функциональных элементов в произвольном базисе.

Это схемы функционального типа. Есть ещё так называемый базис контактных схем. Примеры функционалов:

- Длинна и ранг ДНФ.
- Ранг формулы и ранг схемы.
- Сложность формулы и сложность схемы.
- Глубина формулы и глубина схемы из функциональных элементов.

Теперь рассмотрим задачу синтеза. Она заключается в том, чтобы по данной системе функций в данном классе схем построить оптимальную реализацию этих функций, то есть

$$\forall F = (f_1, \ldots, f_m) \in P_2^m(n)$$

построить реализующую её схему $\Sigma \in U$ такую, что

$$\Psi(F) = \Psi(\Sigma) = \min_{\Sigma' \in U} \Psi(\Sigma'), \tag{9.2}$$

где Σ' реализует F. Решение этой задачи, на самом деле, можно найти следующим перебором: сначала нужно просмотреть схемы сложности 1, потом схемы сложности 2, и так далее. Однако, если оценить число схем, то можно понять, что перебором эта задача будет решаться невероятно долго.

Также введём функцию Шеннона:

$$\Psi(n) = \max_{f \in P_2(x)} \Psi(f). \tag{9.3}$$

Заметим, что нет смысла рассматривать отдельно функцию Шеннона для глубины для формул и для схем, так как они совпадают. Второе, что мы заметим, это то, для любой функции f её формульная сложность больше или равна её схемной сложности в каком-то базисе. Поймём также следующее. Рассмотрим $L^A(F)$, где $F = (f_1, \ldots, f_m)$. Утверждается, что эта сложность не больше, чем сумма сложностей составляющих её функций. А с другой стороны

$$\max_{1 \le i \le m} L^A(f_i) \le L^A(F). \tag{9.4}$$

В классе формул будет реализовываться равенство:

$$L^{\Phi}_{\text{Б}}(F) = \sum_{i=1}^{m} L^{\Phi}_{\text{Б}}(f_i). \tag{9.5}$$

Это же будет выполняться и в случае π-схем.

Утверждение. Для любой функции $f \not\equiv 0$ из $P_2(n)$ существует реализующая её формула F_f и π-схема Σ_f такие, что

$$L(F_f) \le 2n|N_f| - 1, \tag{9.6}$$

а также

$$L(\Sigma_f) \le n|N_f|. \tag{9.7}$$

Оказывается, что это утверждение можно уточнить более точными оценками.

Справедливы следующие неравенства:

Следствие 1.

$$L^e \le L^{\Phi} \le n2^{n+1} - 1 \tag{9.8}$$

Следствие 2.

$$D(n) \le n + \lfloor \log n \rfloor + 2. \tag{9.9}$$

Утверждение. На самом деле справедливы более точные оценки:

$$L(\Sigma_f) \le 2^{n+1} + |N_f| - 4, \tag{9.10}$$

$$L(F_f) \le 2^n + |N_f| - 2. \tag{9.11}$$

Следствие. Справедливы оценки для следующих функций Шеннона:

$$L^{\pi}(n) \le 2^{n+1} - 2, \tag{9.12}$$

$$L^{\Phi}(n) \le 3 \cdot 2^n - 4. \tag{9.13}$$

Строго приведённые схемы

Пусть Σ есть схема из функциональных элементов в каком-то базисе Б. Пусть v и w принадлежат множеству вершин схемы Σ, причем в этой схеме из вершины w в вершину v попасть нельзя. Получим схему Σ' по схеме Σ наложением (присоединением) вершины v на вершину w. Эта операция выражается в том, что мы все дуги,

исходящие из вершины v, отцепляем и присоединяем к вершине w. Таким образом вершина v остаётся висячей вершиной в схеме Σ'. Теперь удаляем из этой схемы все висячие вершины и приходим к приведённой схеме. Полученная таким образом схема Σ', вообще говоря, не является эквивалентной схеме Σ.

Две вершины будем называть эквивалентными, если функция, реализуемая в вершине v, совпадает с функцией, реализуемой в вершине w. Оказывается, что если две вершины v и w изначально были эквивалентны, то и схемы Σ и Σ' также эквивалентны.

Таким образом ясно, что находя в схеме две эквивалентные вершины, мы можем одну вершину всегда присоединить к другой. Мы будем говорить о строго приведённой СФЭ, если в ней нет висячих и эквивалентных вершин. От любой схемы всегда можно перейти к строго определённой схеме, которая заведомо будет иметь меньшую сложность.

Пусть Q лежит в $P_2(n)$, тогда $\overrightarrow{Q}$ есть система всех различных функций множества Q, упорядоченных в соответствии с номерами их столбцов значений.

Утверждение. Для любого n существует схема из функциональных элементов Σ_n в произвольном базисе Б, реализующая систему всех функций от n переменных со сложностью

$$L(\Sigma_n) = 2^{2^n} - n. \tag{9.14}$$

Следствие.

$$L^e_{\text{Б}}\left(\overrightarrow{P}_2(n)\right) \leq 2^{2^n} - n. \tag{9.15}$$

Утверждение. Схема Σ_n минимальна, то есть сложность реализации всех функций будет в точности равняться

$$2^{2^n} - n.$$

Это можно показать с помощью следующего утверждения и оценок.

Утверждение. Для любой системы функций F, состоящей из m функций, таких, что

$$f_i \neq f_j, \tag{9.16}$$

причём

1) $f_i \neq x_l$,

2) $f_i \neq 0, 1$,

будет справедлива следующая оценка

1) для первого случая

$$L^e_{\text{Б}}(F) \geq m,$$

2) для второго случая

$$L^k \geq m. \tag{9.17}$$

Применяя первую оценку для системы функций $\overrightarrow{P}_2(n)$, мы получаем, что в этой системе функций именно

$$2^{2^n} - n \tag{9.18}$$

функций, отличных от переменных. Это значит, что

$$L_{\text{Б}}^{c}\left(\overrightarrow{P}_2(n)\right) \geq 2^{2^n} - n, \tag{9.19}$$

что и доказывает минимальность схемы Σ_n.

Лекция 10

Машина Тьюринга

Утверждение. Пусть L есть язык в алфавите A^*, который принадлежит к классу $\mathbb{NP}$. Из этого следует, что L полиномиально сводится к языку выполнимости. Пусть S — алфавит языка выполнимость. Можно доказать, что существует полиномиальное преобразование

$$\varphi : A^* \longrightarrow S^*, \tag{10.20}$$

обладающее тем свойством, что

$$\forall \bar{a} \in L \tag{10.21}$$

тогда и только тогда, когда

$$\varphi(\bar{a}) = F_{\bar{a}} \tag{10.22}$$

есть выполнимое КНФ.

Напомним определение класса $\mathbb{NP}$. По определению,

$$L \in \mathbb{NP}$$

тогда и только тогда, когда существует полином $q(n)$, алфавит D, где D есть какое-то подмножество алфавита C_0 нашей машины Тьюринга, предикат $Q(x, y)$ такой, что

$$Q(x, y) : A^* \times D^* \longrightarrow \{0, 1\}, \tag{10.23}$$

$Q \in \mathbb{P}$ и обладает тем свойством, что

$$\forall \bar{a} \in A^* \ \bar{a} \in L \tag{10.24}$$

тогда и только тогда, когда существует $\bar{d}$ из D^* такое, что

$$\lambda(\bar{d}) \leq q\big(\lambda(\bar{a})\big) \tag{10.25}$$

и

$$Q(\bar{a}, \bar{d}) = 1, \tag{10.26}$$

то есть этот предикат выполним.

Что означает, что $Q \in \mathbb{P}$? Это означает, что существует такая машина Тьюринга M_1, вычисляющая предикат Q за время

$$p_1(\lambda(x) + \lambda(y) + 1), \tag{10.27}$$

то есть заканчивает работу, и на ленте остаётся $\sigma \in \{0, 1\}$ — значение предиката $Q(\bar{x}, \bar{y})$. Можно считать, что p_1 есть просто полином $p(n)$, где n есть длина слова x.

Теперь из нашей машины Тьюринга сделаем машину M, которая зацикливается в одном из состояний q_0^σ.

Пусть A есть подмножество алфавита C_0, который есть алфавит нашей машины Тьюринга, а предикат $\tilde{Q}$ есть множество состояний нашей машины. Пусть

$$\Pi = \{c_i q_j \to c_k q_r T\}$$

есть программа машины Тьюринга, где c_i есть символ на ленте, а q_j есть состояние. И пусть также

$$D = \{c_{r_1}, \ldots, c_{r_m}\},$$
$$\bar{a} \in A^*,\ \bar{d} \in D^*,\ Q \in \mathbb{P}. \tag{10.28}$$

Теперь мы хотим с помощью КНФ представить факт того, что $\bar{a} \in A^*$ принадлежит L тогда и только тогда, когда

$$\exists \bar{d} \in D^* : \lambda(\bar{d}) \leq q\big(\lambda(\bar{a})\big) \tag{10.29}$$

и машина Тьюринга M с начальной конфигурацией $\bar{a}\Lambda\bar{d}$ в момент времени $q(n)$ находится в состоянии q_0.

Пусть i есть номер ячейки, который обозревает наша машина Тьюринга, j есть номер символа из C_0, который машина видит на ленте, k есть состояние нашей машины. Эти три параметра будут определять состояние нашей машины Тьюринга в разные моменты времени. Введём булевские переменные $x^t_{i,j}$ следующим образом. Потребуем, чтобы $x^t_{i,j} = 1$ выполнялось тогда и только тогда, когда в конфигурации K_t машины Тьюринга в ячейке памяти с номером i находится символ c_j. Введём еще одну переменную y^t_i, которая равна единице тогда и только тогда, когда в конфигурации K_t машина Тьюринга обозревает ячейку с номером i. И последние переменные z^t_k, которые равны единице тогда и только тогда, когда когда в конфигурации K_t машина Тьюринга находится в состоянии q_k.

Можно предьявить три КНФ, перемножение которых даёт КНФ, которое решает нашу задачу:

1)
$$F_1 = \&\Big(\&H_m(x^t_{i,0}, \ldots, x^t_{i,m})\&H_{2p(n)+1}(y^t_{-p(n),0}, \ldots, y^t_{p(n),0})\&H_{l+1}(z^t_0, \ldots, z^t_l)\Big),$$

   где H_{m+1} есть симметрическая функция с рабочим числом m, первая конъюнкция берётся по $t = 0 \ldots p(n)$, а вторая по $i = -p(n) \ldots p(n)$. Эта КНФ описывает нам тот факт, что в любой момент времени существует допустимая конфигурация.

2) Эта КНФ будет выражать факт того, что в нулевой момент времени записано стартовое $\bar{a}\Lambda\bar{d}$. Такое КНФ есть

$$F_2 = y^0_0 x^0_{0,j_1} \ldots x^0_{n-1,j_n} x^0_{n,0} \& \Big((x^0_{i,r_1} \vee \ldots \vee x^0_{i,r_m} \vee x^0_{i,0}) \cdot (x^0_{i,0} \to x^0_{i+1,0})\Big),$$

   где конъюнкция берётся от $i = n+1$ до $p(n)$.

3) Следующая КНФ будет выражать тот факт, что наша машина в момент времени $p(n)$ находится в состоянии q_0:

$$F_3 = z_0^{p(n)}.$$

4) И последний блок будет выражать правильность перехода от предыдущей конфигурации к последующей. Такую КНФ можно тоже предъявить, однако выражение получается слишком громоздкое.

Утверждение. Пусть τ есть конечная полная система тождеств для эквивалентных преобразований формул в базисе Б, а П и П$'$ есть тожества перехода от базиса Б к базису Б$'$ и от базиса Б$'$ к базису Б соответственно. Тогда $\{\Pi'(\tau), \Pi'(\Pi)\}$ есть конечная полная система тождеств для эквивалентных преобразований формул в базисе Б$'$.

Лекция 11

Алгоритм Евклида

Вернёмся к теме «установление числа шагов итерационных алгоритмов». Мы говорим о сложности, напрмиер, в худшем случае. В качестве первого примера был рассмотрен алгоритм Евклида. Мы обсудили, что естественно принять за размер входа a_1. Мы установили, что верхней границей для числа делений с остатков является $\log_2 a_1$ с каким-либо числовым множителем.

Наряду с обычным алгоритмом Евклида рассматривается расширенный алгоритм Евклида, позволяющий представить наибольший общий делитель двух чисел. К алгоритму Евклида добавляются дополнительные действия. Общее число операций увеличивается не больше чем втрое, поэтому асимптотика сохраняется.

Напомним, как выглядит обычный алгоритм Евклида:

$$\begin{aligned}
&a_0 \geqslant a_1;\\
&a_0 = q_1 a_1 + a_2;\\
&a_1 = q_2 a_2 + a_3;\\
&\dots\\
&a_{n-3} = q_{n-2} a_{n-2} + a_{n-1};\\
&a_{n-2} = q_{n-1} a_{n-1} + a_n;\\
&a_{n-1} = q_n a_n.
\end{aligned} \tag{11.1}$$

Вспомним основные полученные результаты. Установили, что затраты по числу делений с остатком

$$C_E(a_0, a_1) \leqslant \lambda(a_0) + \lambda(a_1) - 2. \tag{11.2}$$

Для сложности же получено

$$T_E(a_1) \leqslant \log_2 a_1 + 1. \tag{11.3}$$

Что касается общего делителя, то для его нахождения мы находили последовательности s_i и t_i, что

$$sa_0 + ta_1 = \text{ наибольший общий делитель } (a_0, a_1). \tag{11.4}$$

Здесь имеется ввиду $s = s_n$, $t = t_n$. Приняв такие обозначения, были получены формулы

$$\begin{aligned}
s_{i+1} &= s_{i-1} - q_i s_i;\\
t_{i+1} &= t_{i-1} - q_i t_i,
\end{aligned} \tag{11.5}$$

причём $s_0 = 1$, $t_0 = 0$, $s_1 = 0$, $t_1 = 0$. Замечательно то, что последовательности s_i и t_i вычисляются независимо друг от друга.

Выпишем теперь несколько замечательных свойств чисел s и n, доказать которые следует самостоятельно.

1) $|s_2| \leqslant |s_3|$

2) $|t_2| < |t_3|$

3) $|s_3| < |s_4| < \ldots < |s_{n+1}|$

4) $|t_3| < |t_4| < \ldots < |t_{n+1}|$

Бывает, что построение рядов s и t удобно продлить на один шаг. Тогда получим равенство

$$s_{n+1}a_0 + t_{n+1}a_1 = 0. \tag{11.6}$$

Этот дополнительный шаг позволяет вычислить наименьшее общее кратное.

Каждая пара s_i и t_i является парой взаимно простых чисел. Часто используют обозначение $s_i \perp t_i$. Этот факт также можно доказать самостоятельно.

Имеет место матричное равенство

$$\begin{pmatrix} -q_1 & 1 \\ 1 & 0 \end{pmatrix} \ldots \begin{pmatrix} -q_{i-1} & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} t_i & t_{i-1} \\ s_i & s_{i-1} \end{pmatrix}. \tag{11.7}$$

Доказать его можно, например, по индукции. Если доказать это равенство, то легко оказывается и доказать взаимную простоту s_i и t_i. Для этого необходимо посчитать определители левой и парвой частей равенства (11.7).

Ещё одно свойство:

$$s_{n+1} = \frac{a_1}{\text{НОД}(a_0, a_1)} \tag{11.8}$$

$$|t_{n+1}| = \frac{a_0}{\text{НОД}(a_0, a_1)} \tag{11.9}$$

Однако, больший интерес представляют неравенства

$$|s_n| \leqslant a_1 \tag{11.10}$$

$$|t_n| \leqslant a_0 \tag{11.11}$$

Алгоритм Евклида используется в модулярной арифметике. Расширенный алгоритм Евклида может применяться для обращения целого числа по модулю p.

Обсудим ещё один алгоритм. Пусть есть числовой массив $x_1 < x_2 < \ldots < x_n$ и число y. Алгоритм поиска места элемента y в массиве $x_1 < x_2 < \ldots < x_n$, он же бинарный алгоритм:

```
p := 1; q := n+1;
while p < q do
r :=  floor[ ( p + q )/2 ] ; if x_r < y then
p := r + 1
else q := r
fi
```

Существует $n+1$ возможный исход: возможен вариант $y < x_1$, возможен вариант $x_1 < y < x_2$, ..., возможен вариант $y > x_n$. Алгоритм основан на том, что на каждом шаге уменьшается длина упорядоченного массива, в котором необходимо найти место для нового элемента. Число шагов алгоритма не может превосходить бинарной длины числа n. С другой стороны, если изначально $y < x_1$, то алгоритму

придётся совершить $\lambda(n)$ шагов. В других случаях число шагов окажется меньшим. Итак, для сложности имеет место равенство

$$T_{BS}(n) = \lambda(n). \tag{11.12}$$

Бинарный поиск имеет широкое применение, например, при поиске информации в таблицах. Однако бинарный поиск используется не только в таких задачах. Так, он находит применение в геометрии. пусть есть выпуклый многоугольник. Многоугольник задан последовательностью координат вершин. Пусть есть точка P, заданная координатами. Необходимо определить, принадлежит ли точка многоугольнику. Можно рассуждать следующим образом: найдём сначала точку, которая заведомо лежит внутри многоугольника. Обозначим её как O. Точка P принадлежит многоугольнику тогда и только тогда, когда относительно каждой прямой, которая является продолжением какой-либо стороны многоульника, точки O и P лежат по одну сторону. Проверка того, по одну сторону лежат точки или нет, проводится за конечное число шагов с помощью аналитической геометрии. Если заданный многоугольник имеет n сторон, то сложность алгоритма составит $\Theta(n)$.

Как применить бинарный поиск в данной задаче? Разобьём плоскость на клинья: из найденной точки O проведём лучи через вершины многоугольника. Необходимо понять, какому из клиньев принадлежит точка P. Если точки O и P лежат по разные стороны от соответствующей стороны многоугольника, то точка P не принадлежит прямоугольнику. Таким образом, оба алгоритма сводят задачу к задаче определения, по одну сторону от заданной прямой лежат две точки или по разные. Однако второй вариант позволяет использовать бинарный поиск для определения того, какому именно из клиньев принадлежит точка P. На опредление нужного клина потребуется порядка $\log_2 n$ операций. Но всё-таки на проведение лучей придётся потратить $\Theta(n)$ операций. С третьей стороны, для вычислений сами эти лучи не нужны. Мы знаем углы, можем сравнивать их по величине, фактическое проведение прямых не нужно, чтобы строить бинарный поиск. Итак, имеем алгоритм, для которого верна оценка $\Theta(\log n)$.

Домашнее задание. *Если точка лежит вне выпуклого многоугольника, алгоритм позволяет найти сторону, которая «видна» из этой точки. Требуется построить опорные прямые к многоугольнику из данной точки. Прямая является опорной, если проходит через вершину многоугольника и многоугольник лежит по одну сторону от этой прямой. Алгоритм должен иметь сложность* $\Theta(\log n)$.

Затронем сортировку фон Неймана. Пусть есть массив, который нужно упорядочить. Создаётся ещё один массив. Упорядочивать будем поэтапно. На самом первом этапекаждый отдельный элемент считается упорядоченным элементом длины один. Далее сливаем элементы в массивы длины два, каждый упорядочиваем. На следующем этапе сливаются соседние массивы длины два и получаются упорядоченные массивы длины четыре. Такая сортировка обозначается vN. Интересоваться будем сложностью этой сортировки. Сколько этапов потребуется для того, чтобы отсортировать массив длиной n. Ответ на этот вопрос — в худшем случае число этапов составит $\lceil \log_2 n \rceil$. В худшем случае число перемещений составит $n\lceil \log_2 n \rceil$. Сложность по числу сравнений будет меньше, чем $n\lceil \log_2 n \rceil$.

Мы уже пользовались специальным приёмом для определния числа шагов или доказательства того, что процесс когда-либо завершится: определяется функция (в последнем примере это число уже упорядоченных сегментов) и показывается, что эта функция с каждым шагом убывает. Получаем, что число шагов не превосходит значение этой функции на исходных данных задачах. Иногда эту функцию сравнивают с какой-либо выделенной последовательностью, чтобы показать, что функция убывает достаточно хорошо. Этим же путем устанавливаются некоторые оценки и в численных методах. Например, доказывают, что функция убывает не хуже, чем геометрическая прогрессия. Для экскурса в численные методы рассмотрим алгоритм деления пополам. Ошибка не должн превышать ε. Тогда число итераций, которое потребуется, имеет вид $\log_2 1/\varepsilon + \mathcal{O}(1)$. Если же мы используем метод касательных, то потребуется $\mathcal{O}(\log\log 1/\varepsilon)$ итераций.

Домашнее задание. *Пусть есть некоторое количество монет. Одна из них фальшивая, она отличается по весу. Есть чашечные весы без гирь. Если число монет 3^n, то достаточно n взвешиваний, чтобы определить фальшивую монету. Пусть имеем n монет. Выделим две группы из $\lceil n/3 \rceil$ и будем взвешивать их. Необходимо показать, что в случае $\lfloor n/3 \rfloor$ сложность будет больше.*

Домашнее задание. *Задача на тему алгоритма бинарного поиска. В лучшем случае затраты составят $\lfloor \log_2(n+1) \rfloor$. Как следствие, для значений n вида $2^k - 1$ и только для них число сравнений однозначно определено.*

Лекция 12

Точность алгоритмов

Пусть для некоторого алгоритма получена асимптотическая оценка $\mathcal{O}$, то есть асимптотическая оценка сверху. Возможно, эту оценку можно улучшить. Обсудим отдельно качество оценок на примере алгоритма Евклида.

На первой лекции обсуждалось, что в некоторых случаях оценку $\mathcal{O}(f(n))$ мы можем называть точной. Оценка $\mathcal{O}(f(n))$ называется точной, если оценка $o(f(n))$ не верна. Является ли оценка $\mathcal{O}(\log a_1)$ для алгоритма Евклида точной? Чтобы показать, что является, нужно найти такую последовательность входов для алгоритма Евклида, чтобы размер возрастал от входа к входу (то есть a_1 должен расти) и чтобы на этих парах алгоритм Евклида работал достаточно медленно. В качестве такой последовательности возьмём последовательность чисел Фибоначчи:

$$\begin{aligned} &F_0 = 0; \\ &F_1 = 1; \\ &F_2 = F_1 + F_0 = 1; \\ &\dots \\ &F_i = F_{i-1} + F_{i-2}. \end{aligned} \tag{12.1}$$

В качестве входа будем брать пары (F_n, F_{n-1}). На таких парах алгоритм Евклида работает медленно. По построению последовательности Фибоначчи все частные будут единице, а остаток будет равен F_{n-2}. Мы будем пользоваться формулой Бине для чисел Фибоначчи:

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - (-\phi)^{-n}), \tag{12.2}$$

где $\phi = (1+\sqrt{5})/2 = 1.61803...$ Формула Бине может быть доказана по индукции. С помощью этой функции мы будем делать необходимые оценки для сложности. Для начала запишем формулу

$$C_E(F_n, F_{n-1}) = n. \tag{12.3}$$

Это следует из тех же самых соображений о построении ряда Фибоначчи. Поскольку верно

$$\phi^n = \sqrt{5}F_n(1 + o(1)), \tag{12.4}$$

то теперь, зная формулу Бине, продолжим равенство:

$$C_E(F_n, F_{n-1}) = n = log_\phi F_n + \mathcal{O}(1) = \Omega(\log F_n). \tag{12.5}$$

Как обычно, Ω — асимптотическая оценка снизу. В таком случае, наша оценка действительно является точной. Заменить $\mathcal{O}(\log a_1)$ на $o(\log a_1)$ нельзя, поскольку мы предъявили пары чисел, для которых справедливо $\Omega(\log F_n)$.

Вообще говоря, справедливо и более сильное утверждение:

$$T_E(a_1) = \Theta(\log a_1). \tag{12.6}$$

Чтобы получить это результат, нужно доказать, что для любого a_1 можно подобрать a_0 такое, что алгоритм будет работать достаточно медленно. У нас же a_1 были не любые, а специальные. Как доказать это для любого a_1? Ранее мы пользовались

$$F_n \approx \phi F_{n-1}. \tag{12.7}$$

Теперь действовать будем так: для a_1 положим $a_0 := \lceil \phi a_1 \rceil$. Тогда по аналогии с рядом Фибоначчи ситуация будет развиваться похожим образом. Доказательство остаётся для самостоятельного размышления.

Рассмотрим сортировку бинарными вставками. Пусть первая часть массива вплоть до x_i уже упорядочена. Чтобы найти место x_{i+1}, применяется бинарный поиск. Сложность алгоритма бинарного посика нам уже известна, поэтому можем получить оценку для сложности сортировки бинарными вставками. К алгоритму бинарного поиска придётся обратиться $n-1$ раз. Итак, сложность $T_B(n)$ удовлетворяет неравенству

$$T_B(n) \leqslant (n-1)\lceil \log_2 n \rceil. \tag{12.8}$$

Эта оценка довольно грубая. Попробуем её улучшить. На разных шагах требуется разное число сравнений. Более точное значение для числа сравнений:

$$\sum_{l=0}^{n-1} \lceil \log_2 l \rceil. \tag{12.9}$$

Тогда

$$T_B(n) = \sum_{l=0}^{n-1} \lceil \log_2 (l+1) \rceil = \sum_{k=1}^{n} \lceil \log_2 k \rceil. \tag{12.10}$$

Домашнее задание. *Задача на бинарный алгоритм Евклида. Пусть два числа a_0 и a_1 чётные, тогда можно вместо вычисления НОД(a_0, a_1) вычислять 2НОД$(a_0/2, a_1/2)$. Если чётно a_0, но нечётно a_1, то вычислять будем НОД$(a_0/2, a_1)$. Если чётно a_1, но нечётно a_0, то вычислять будем НОД$(a_0, a_1/2)$. Если обы числа нечётны, то вычислим Если чётно a_0, но нечётно a_1, то вычислять будем НОД$(a_0 - a_1, a_1)$ и повторяем описанные выше рассуждения. Процесс идёт до тех пор, пока одно из чисел не окажется равным нулю. Второе число тогда даст общий делитель. Доказать, что число шагов есть $\mathcal{O}(\log a_1)$.*

Домашнее задание. *Рассмотрим уравнения вида*

$$ax + by = c, \tag{12.11}$$

где числа a, b, c — целые. Требуется найти целочисленные решения. Для этого может быть использован расширенный алгоритм Евклида. Составить алгоритм, оценить его сложность.

Домашнее задание. *Пусть есть два выпуклых многоугольника. Можно устроить их объединение и интересоваться оболочкой объединения. Требуется продумать алгоритм, который основан на построении опорных прямых, сложность которого составит $\mathcal{O}(n_1 + n_2)$, где n_1 и n_2 — число вершин в первом и во втором многоугольниках соответственно.*

Избавимся от потолка.

$$T_B(n) = \sum_{k=1}^{n} \log_2 k + \mathcal{O}(n) = \log_2 n! + \mathcal{O}(n). \tag{12.12}$$

Для того, чтобы двигаться дальше, нам потребуется формула Стирлинга:

$$n! = n^n e^{-n} \sqrt{2\pi n}(1 + o(n)). \tag{12.13}$$

Прологарифмируем формулу Стирлинга:

$$\log_2 n! = n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 n + \frac{1}{2} \log_2 \pi + o(1). \tag{12.14}$$

Если учесть, что $1 < log_2 e < 2$, то можно получить

$$n \log_2 n - 2n < \log_2 n! < n \log_2 n - n \tag{12.15}$$

для достаточно больших n. Отсюда получаем

$$\log_2 n! = n \log_2 n + \mathcal{O}(n). \tag{12.16}$$

Эту формулу тоже иногда называют формулой Стирлинга.

Итак,

$$T_B(n) = n \log_2 n + \mathcal{O}(n). \tag{12.17}$$

Запишем асимптотику:

$$T_B(n) \sim n \log_2 n. \tag{12.18}$$

Поговорим теперь о завершимости работы алгоритмов. Если работа алгоритма не завершается, то сложность равна бесконечности. Рассмотрим пример.

```
while n > 3 do
if 2|n then n := n/2 + 1 else n := n + 1 fi
od
```

Подразумевается, что n — натуральное число. Запись $2|n$ означает, что n — чётное число. Можно доказать, что выполнение этого алгоритма завершается. Для этого нужно рассмотреть функцию

$$n - (-1)^n \tag{12.19}$$

Достаточно доказать, что функция убывает по ходу выполнения алгоритма. Поскольку функция является заведомо неотрицательной, то бесконечно убывать она не может. Для этого полезно эту функцию следующим образом:

$$n - (-1)^n = \begin{cases} n - 1 \text{ если } n \text{ чётно;} \\ n + 1 \text{ если } n \text{ нечётно.} \end{cases} \tag{12.20}$$

Самостоятельно доказать, что число шагов алгоритма может быть оценено как $\mathcal{O}(\log n)$.

Существует известная нерешённая задача. Доказать, что выполнение алгоритма

```
while n > 1 do
if 2|n then n := n/2  else n := 3n + 1 fi
od
```

завершается для любого натурального n. Этот алгоритм называется 3x+1.

До сих пор мы рассматривали натуральные числа. Будем теперь говорить о кортежах натуральных чисел произвольной длины.

Домашнее задание. *Бизнесмен и чёрт играют в игру: каждый день бизнесмен даёт чёрту монету, а взамен получает любой набор монет по своему выбору, но меньшего достоинства. Видов монет конечное число. Менять или получать деньги в другом месте бизнесмен по условию не может. Если у бизнесмена не остаётся монет достоинством выше минимального, он проигрывает. На таких условиях бизнесмен рано ли поздно терпит поражение, каким бы ни был его первоначальный запас монет.*

Домашнее задание. *Имеется конечная последовательность нулей и единиц. Разрешается найти в ней группу соседствующих цифр* 01 *и заменить её на* 1 *и сколько угодно нулей. Доказать, что это преобразование не может быть применено бесконечно много раз.*

Обсуждаем завершимость. Следующий сюжет, который нужно обсудить в связи с завершимостью, это рандомизированные алгоритмы. В рандомизированных алгоритмах завершимость может пониматься двумя способами. Первый: алгоритм завершается с вероятностью 1. Второй: математическое ожидание времени, которое потребуется для завершения, конечно. Второе утверждения является более сильным, чем первое. Увидим это на примере. Пусть есть прямая и движущаяся частица на ней. Частица через равные промежутки времени совершает перемещения в соседнюю точку (целую) — либо направо, либо налево. Выбор направления производится случайным образом. С вероятностью 1/2 частица перемещается налево, с вероятностью 1/2 направо. Доказывается в курсе теории вероятности, что если на прямой зафиксировать точку m, то с вероятностью 1 по прошествию какого-то времени частица окажется в точке m. Но каково время ожидания? Оказывается, время ожидания равно бесконечности, даже когда мы берём одну из соседних точек.

Пусть время, за которое частица окажется в точке 1, равно a. Тогда по формуле полного математического ожидания можно представить это время следуюим образом:

$$a = 1 \cdot \frac{1}{2} + 2a \cdot \frac{1}{2}. \tag{12.21}$$

С вероятностью 1/2 мы сразу окажемся в нужной точке, с вероятностью 1/2 мы окажемся на расстоянии, в два раза большем первоначального расстояния от нужной точки. Если a конечно, то приходим к равенству $0 = 1/2$. Остаётся только положить $a = \infty$.

Вспомним задачу про путника, который ищет калитку в стене. Пусть путник действует как эта частица: бросает монетку и решает, куда идти. Таким образом он

дверь найдёт, но за бесконечное время. Здесь нет парадокса. Пусть p_k — вероятность того, что путник найдёт дверь через k шагов. Тогда верно

$$\sum_{k=1}^{\infty} p_k = 1. \tag{12.22}$$

С другой стороны,

$$\sum_{k=1}^{\infty} k \cdot p_k = \infty. \tag{12.23}$$

Между этими двумя равенствами нет противоречия.

В следующей задаче речь пойдёт об автоматизации обучения. Пусть для запоминания какой-либо информации используется система карточек. Обучаемый случаным образом вытаскивает карточку из колоды с вопросом. Если он отвечает верно, то карточка удаляется из колоды. Если он отвечает неверно, то карточка возвращается в колоду, но кроме того в колоду добавляется дублирующая её карточка. Пусть обучаемый всё время отвечает неправильно. Настанет ли момент, когда он увидит все вопросы? Оказывается, что с вероятностью 1 он увидит все вопросы. Что касается времени, за которое он увидит все вопросы, то здесь требуется более сложное рассуждение.

Лекция 13

Сложность алгоритмов сортировок

Мы прододжаем обсуждать завершимость рандомизированных алгоритмов.

Задачу о карточках с вопросами можно переформулировать как задачу о вытягивании шаров из сосуда. Пусть есть n шаров разного цвета. Вытягиваем шар, смотрим на цвет, возвращаем шар и добавляем шар такого же цвета. Наступит ли момент, когда мы увидим все возможные цвета шаров? Эти задачи идентичны. Оказывается, время, которое требуется для того, что увидить все цвета шаров или все возможные вопросы, бесконечно. Однако время, которое требуется для того, чтобы увидить все цвета шаров кроме может быть одного или все вопросы кроме может быть одного, оказывается конечным. Доказательство этого факта нехитрое, приводить его не будем.

Домашнее задание. *Пусть есть квадратная матрица размера n. Разрешено менять умножать на -1 строку или столбец. Доказать, что такими действиями можно добиться, чтобы сумма элементов была неотрицательной. Показать, что за конечное число шагов мы придём к тому, что эти операции применять невозможно.*

Нижние границы сложности классов алгоритмов.

Начнём с примера. Задача выбора из массива наименьшего элемента. Выбор должен быть сделан с помощью сравнений. На каком основании можно утверждать, что элемент не является наименьшим? Только если он участвовал в сравнении и оказался больше какого-то элемента. Если в массиве n элементов, то сравнений потребуется не менее $n-1$. Эта функция от n является нижней границей сложности класса алгоритмов выбора наименьшего элемента из массивов длины n попарно различных числовых элементов.

Ещё один пример: сортировки. Можно доказать, что нижняя граница сложности составляет $\log_2 n!$. Аналогично можно рассмотреть поиск места элемента в упорядоченном массиве. Там ответ $\log_2(n+1)$.

Ещё один пример: вычисление a^N. Ранее обсуждался алгоритм повторного квадрирования. Какие нижние границы можно указать? Нижняя граница показывает некоторый предел, до которого можно улучшать алгоритм. Изначально у нас есть только a^1. На каждом шаге у нас уже есть предварительно вычисленные a^1, a^{l_1}, a^{l_2}, $\ldots$, a^{l_k}. Такой набор можно характеризовать максимумом из всех известных степеней: $\max\{l_1, l_2, \ldots, l_k\}$. Заметим, что один шаг алгоритма может увеличить этот максимум не больше чем вдвое. Расмотрим такую функцию $f(n) = \log_2 n - \log_2 l$. Эта функция изменяется с каждым шагом не более чем на единицу. Функция эта не может оставаться положительной, на последнем шаге она может обратиться в ноль или вообще стать отрицательной. Итак, нижняя граница $-\log_2 n$. Таким образом, если имеем алгоритм сложностью $\log_2 n$, то лучшего алгоритма уже не составить.

Нижняя граница сложности не определяется однозначно. Когда мы предъявляем некоторую нижнюю границу сложности, эта информация тем нетривиральнее, чем эта граница больше. Сигналом к тому, что повысить нижнюю границу сложности

нельзя, является предъявление алгоритма, чья сложность совпадает с указанной нижней границей сложности. Такой алгоритм называется оптимальным.

Пример оптимального алгоритма: алгоритм бинарного поиска. Нижней границей является $\lceil \log_2 n + 1 \rceil$, поскольку для элемента существует $n+1$ возможное место. В то же время мы знаем алгоритм с такой сложностью — это алгоритм бинарного поиска. Поэтому границу улучшить нельзя, а кроме того известен оптимальный алгоритм.

Следующий алгоритм: пусть есть функция

$$f(n) = \lceil \frac{3n}{2} \rceil - 2. \tag{13.1}$$

Эта функция является нижней границей сложности для алгоритмов одновременного поиска наибольшего и наименьшего элементов в массиве. Можно установить, что эта функция явялется нижней границей сложности и указать оптимальный алгоритм. На каждой стадии алгоритма происходит сравнение. Каждый этап характеризуется четвёркой множеств: в A попадают элементы, которые ещё ни разу не сравнивались. В B попадают элементы, которые участвовали в некоторых сравнений и всегда оказывались большими. В C попадают элементы, которые участвоали в некоторых сравнениях и всегда оказывались меньшими. В D попадают элементы, которые участвоали в некоторых сравнениях и оказывались то большими, то меньшими.

Все сравнения, которые мы проводим, тоже можно расклассифицировать в соответсвие с этими множествами: сравнения типа AA, AB, AC, AD, BB, BC, CC, CD, DD. Маленькими буквами будем обозначать число элементов в том или ином множестве: a — число элементов в множестве A, b — число элементов в множестве B, c — число элементов в множестве C, d — число элементов в множестве D. Выпишем, как сравнения разных типов могут менять a, b, c, d:

$$\begin{array}{ll}
AA: & (a-2, b, c, d) \\
AB: & (a-1, b, c+1, d) | (a-1, b, c, d+1) \\
AC: & (a-1, b+1, c, d) | (a-1, b, c, d+1) \\
AD: & (a-1, b+1, c, d) | (a-1, b, c+1, d) \\
BB: & (a, b-1, c, d+1) \\
BC: & (a, b, c, d) | (a, b-1, c-1, d+2) \\
BD: & (a, b, c, d) | (a, b, c, d+1) \\
CC: & (a, b, c, d+1) \\
CD: & (a, b, c, d) | (a, b, c, d+1) \\
DD: & (a, b, c, d)
\end{array} \tag{13.2}$$

Рассмотрим теперь функцию

$$L(a, b, c, d) = \frac{3}{2}a + b + c - 2. \tag{13.3}$$

Как может измениться эта функция под действием одного из сравнений?

$$\begin{array}{ll} AA, BB, CC: & -1 \\ BD, CD: & -1|0 \\ AB, AC: & -3/2|-1/2 \\ BC: & -2|0 \\ AD: & -1/2 \\ DD: & 0 \end{array} \tag{13.4}$$

Заметим, что некоторые сравнения могут приводить к тому, что значение функции уменьшится больше, чем на единицу. В худшем же случае функция изменяется не больше чем на единицу. Отсюда следует, что быстрее эту задачу решать нельзя.

Приведём оптимальный алгоритм для этой задачи. Самый примитивный вариант предполагает два сравнения на каждый элемент: сравнение с уже известными наибольшим и наименьшим. Такой вариант нам не подходит. Более удачен алгоритм, где элементы сравниваются попарно, выбирается наименьший и наибольший, далее пары объединяются в четвёрки и из уже имеющихся вариантов наибольших и наименьших выбираются общие для четверки наибольший и наименьший, и так до момента пока не будут объединены все элементы массива.

Вообще говоря, оптимальный алгоритм может и не существовать. В классе может быть, например, два алгоритма, один из которых быстро работает при чётных n, а дргуой при нечётных n. Тогда оптимального алгоритма не существует.

Оптимальных алгоритмов известно не очень много. Даже с сортировками. Долгое время считалось, что сортировка бинарными вставками является оптимальной. На самом деле это неверно. Было показано, что существует алгоритм, который для некоторых n работает быстрее, чем бинарные вставки. Основа этого алгоритма в том, что пять чисел можно отсортировать за семь сравнений. Над этим стоит подумать самостоятельно. Алгоритм сортировки бинарными вставками в худшем случае потребует восемь сравнений. Упомянутый нами алгоритм обобщается на любое n. Он, опять же, не оптимальный. Можно показать, что для некоторых n можно сделать более быстрый алгоритм.

Вводится ещё одно понятие: оптимальность по порядку сложности. Вводится понятие асимптотической нижней границы сложности. Функция $f(n)$ является асимптотической нижней границей, если для любого алгоритма A из рассматриваемого класса $\mathcal{A}$ для сложности верна оценка

$$T_A(n) = \Omega(f(n)). \tag{13.5}$$

Предполагаем, что в классе $\mathcal{A}$ существует такой алгоритм A, что его сложность совпадает с асимптотической нижней границей. Тогда этот алгоритм является оптимальным по порядку сложности.

Может оказаться, что два алгоритма A и B оба оптимальны по порядку сложности. Это не означает, что у них одинаковая сложность. Можно только утверждать, что

$$T_A(n) = \Theta(T_B(n)). \tag{13.6}$$

Это верно, поскольку из определения следует, что

$$\begin{array}{l} T_A(n) = \Omega(T_B(n)); \\ T_B(n) = \Omega(T_A(n)). \end{array} \tag{13.7}$$

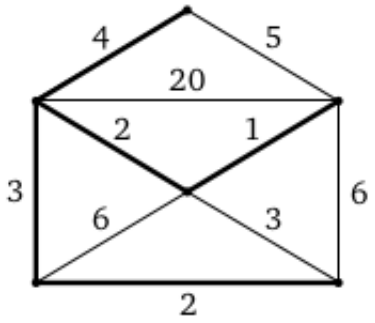

Рис. 13.1. Граф с приписанными ребрам весами и его остовное дерево.

Вспомним, что знак Ω можно перекидывать с одной части равенства на другую с заменой на $\mathcal{O}$. Перекинем его справа налево во втором равенстве:

$$T_A(n) = \mathcal{O}(T_B(n)). \tag{13.8}$$

Сравнивая результат с первым равенством, делаем вывод о том, что верно

$$T_A(n) = \Theta(T_B(n)). \tag{13.9}$$

Рассмотрим ещё один пример. Рассмотрим асимптотическую нижнюю границу вычисления a^N:

$$f(n) = \log_2 n. \tag{13.10}$$

Оптимальный алгоритм нам неизвестен. Асимптотически оптимальным является бинарный алгоритм.

Рассмотрим ещё один пример. Этот пример будет касаться объекта, изучаемого в теории графов: остовного дерева. Пусть есть граф без кратных рёбер, и каждому ребру приписан некоторый вес. Остовным деревом называется дерево с тем же набором вершин, а по рёбрам этого дерева можно из любой вершины добраться в любую, а кроме того суммарная «стоимость» (сумма всех весов, приписанная каждому ребру) дерева минимальна. Пример изображён на рисунке (13.1). Для построения остовных деревьев известно много алгоритмов. Для алгоритма Прима сложность составляет

$$\mathcal{O}(|E| + |V| \log |V|). \tag{13.11}$$

Здесь E — множество рёбер, а V — множество вершин, $|E|$ — количество рёбер, а $|V|$ — количество вершин. Интересно, что если размером входа считать только число вершин $|V|$, то алгоритм Прима является оптимальным.

Лекция 14

Оптимальность и сложность

Продолжим изучать сложность алгоритма Прима. Докажем его оптимальность в случае, если размером входа считать число вершин $|V|$. Мы выведем, что для этого алгоритма из приведённой оценки можно вывести оценку $\mathcal{O}(|V|^2)$, а из самой задачи получим оценку $\Omega(|V|^2)$.

Поскольку мы изучаем граф без кратных рёбер, то число рёбер не будет превосходить

$$\frac{|V|(|V|-1)}{2}. \tag{14.1}$$

Тогда мы избавляемся от переменной $|E|$ и оценка $\mathcal{O}(|E|+|V|\log|V|)$ превратится в оценку $\Omega(|V|^2)$.

Продолжим разговор об оптимальности алгоритмов. Мы вели речь о бинарном алгоритме возведения в степень и установили, что он оптимальным не является, но является оптимальным по порядку сложности.

Домашнее задание. *Пусть в качестве размера входа мы выбираем двоичный размер входа. По порядку сложности алгоритм остаётся оптимальным. Будет ли алгоритм бинарного возведения в степень оптимальным?*

Если некоторая сортировка допускает оценку $\mathcal{O}(\log n!)$, то он автоматически будет допускать оценку $\Theta(\log n!)$ и будем являться оптимальным. Ранее мы пользовались формулой Стирлинга и установили, что функции $\log_2 n!$ и $n\log_2 n$ довольно близки и можно заменять одно на другое. Так можно показать оптимальность по порядку сложности некоторых алгоритмов сортировок.

Сортировка фон Неймана и сортировка бинарными вставками при разных n дают различные затраты. Например, для $n=5$ в худшем случае фон Нейман затратит девять сравнений, бинарные вставки восемь сравнений, а на самом деле минимум сравнений равен семи.

Итак, мы обсуждали оптимальные алгоритмы и оптимальные по порядку сложности. Речь везде шла о сложности в худшем случае. Можно ли строить похожие конструкции для сложности в среднем? Будем говорить о сортировках. Каждой сортировки для фиксированного n можно сопоставить дерево сортировки. Листьями таких деревьев являются некотоыре перестановки. Лемма. В любом двоичном дереве с m листьями сумма высот всех листье не меньше, чем $m\log_2 n$. От корня к листу проводим некоторый путь, подсчитываем количество ребёр и считаем это число высотой листа. Обозначим множество двоичных деревьев, для которых это утверждение не верно как M. Предполагаем, что M не пусто. Выделим дерево с наименьшей суммой высот всех листье. Рассуждаем так: это дерево не может состоять из одного корня, потому что для такого дерева утверждение леммы выполнено. Можно утверждать, что одно ребро из корня исходить не может, потмоу что если выкинуть этот корень и это ребро, останется тоже дерево, для которого утверждение леммы тоже не выполнено, а сумма высот его листьев меньше. Поэтому из корня исходит два ребра. Из вершин тоже растут конечные деревья. Для

этих деревьев утверждение леммы выполнено, иначе мы бы опять вступиои в противоречие с предположением о минимальности суммы высот для рассматриваемого дерева. Обозначим сумму высот как $H(U)$:

$$H(U) = H(U_1) + H(U_2) + m. \tag{14.2}$$

Здесь U — исходное дерево, U_1, U_2 — поддеревья. Поскольку для U_1 и U_2 верно утверждение леммы, то запишем

$$H(U) \geqslant m_1 \log_2 m_1 + m_2 \log_2 m_2 + m. \tag{14.3}$$

Перепишем это неравенство как

$$H(U) \geqslant m_1 \log_2 m_1 + (m - m_1) \log_2 m_2 + m. \tag{14.4}$$

Ясно, что $1 \leqslant m_1 \leqslant m - 1$. Рассмотрим вспомогательную функцию

$$f(x) = x \log_2 x + (m - x) \log_2 (m - x). \tag{14.5}$$

Можно показать, что функция достигает минимума а отрезке $[1, m-1]$ при $x = m/2$. По этой причине

$$H(U) \geqslant \frac{m}{2} \log_2 \frac{m}{2} + \frac{m}{2} \log_2 \frac{m}{2} + m = m \log_2 m. \tag{14.6}$$

Итак, мы доказали то что хотели. Для сложности в среднем имеем ту же самую оценку.

Обсудим сложность в среднем для алгоритма выбора наименьшего и наибольшего элементов в массиве. Нижняя граница здесь такая:

$$f(n) = \begin{cases} \frac{3}{2}n - 2, \text{ если } n\ text \\ \frac{3}{2}n = 2 + \frac{1}{2n}, \text{ если } n\ text \end{cases} \tag{14.7}$$

Эту формулу можно вывести самостоятельно.

Домашнее задание. *Вспомним задачу с сортировкой вагонов. Задача ставилась так: предъявить алгоритм, сложность которого в худшем случае составляет* $3n - 1$. *Теперь же требуется показать, что* $3n - 1$ *является нижней границей.*

Домашнее задание. *Вспомним задачу про путника, который ищет калитку в стене. Требовалось найти алгоритм поиска двери со сложностью* $\mathcal{O}(n)$. *Показать,* $\mathcal{O}(n)$ *— асимптотическая нижняя граница.*

Домашнее задание. *Схема Горнера. Этот алгоритм является оптимальным, если речь идёт о худшем случае. Существуют полиному, значение которых вычисляются значительно быстрее.*

Домашнее задание. *Найти* k*-ый по величине элемент в массиве. Другой вариант этой задачи: найти* k *первых элементов в массиве.*

Домашнее задание. *Деление отрезка на n разных частей с помощью циркуля и линейки. Сложность алгоритма измеряется числом линий, которые нужно провести. Доказать, что нижняя граница равна $(n-1)/2$.*

Домашнее задание. *Поиск места элемента в массиве.*

Домашнее задание. *Плитка шоколада k x l клеток. Разломы проводятся по прямым. Предложить оптимальный алгоритм. Затраты измеряются числом разломов.*

Мы говорили о границах в худшем случае, в среднем, асимптотические. А что делать с рандомизированными алгоритмами? Например, быстрая сортировка, где разбивающий элемент выбирается случайно. Не ломается ли нижняя граница $\log_2 n!$? Чтобы доказать, что этого не происходит, используется некоторый факт из теории игр.

Имеется ввида матричная игра двух игроков. Имеется квадратная матрица $M = (m_{ij})$ порядка k. Два игрока: игрок I и J. Игра состоит в том, что игрок I выбирает i, а игрок J независимо от I выбирает j. Числа i и j должны лежать в диапазоне $[1, k]$. Далее они ищут элемент матрицы m_{ij}, и если это число положительное, то игрок I выплачивает сумму, равную m_{ij}, игроку J. Если же число m_{ij} отрицательное, то игрок J выплачивает сумму, равную $|m_{ij}|$, игроку I. Рассматривается такое поведения игроков, что игрок I выбирает некоторые вероятности $(p_1, \ldots, p_k)$, игрок J выбирает некоторые вероятности $(q_1, \ldots, q_k)$. Игрок I выбирает значение i вероятностью p_i, а игрок J выбирает значение j вероятностью q_i. Игра идёт несколько раундов. Вычислим математическое ожидание выигрыша игрока I:

$$M(p, q) = \sum_i \sum_j m_{ij} p_i q_j. \tag{14.8}$$

Есть так называемые чистые стратегии: такие, что одно из $p_i = 1$, а остальные равны нулю. Существуют оптимальные стратегии p^* и q^* такие, что величина $M(p^*, q^*)$ ялвяется одновременно значением следующих величин:

$$\max_p \min_q M(p, q) = \min_q \max_p M(p, q) = M(p^*, q^*). \tag{14.9}$$

Заметим, что если стратегия p фиксировано, то

$$\min_q M(p, q) = \min\{\sum_i m_{i1} p_i, \sum_i m_{i2} p_i, \ldots, \sum_i m_{ik} p_i\} = \min_j \sum_j m_{ij} p_i. \tag{14.10}$$

Рассуждая аналогично о случае, когда фиксировано q, мы получим равенство

$$\max_p \min_q \sum_i m_{ij} p_i = \min_q \max_i \sum_j m_{ij} q_j. \tag{14.11}$$

Отсюда для любых стратегий p и q получаем то, что требуется:

$$\min_j \sum_i m_{ij} p_i \leqslant max_i \sum_j m_{ij} q_j. \tag{14.12}$$

Какое это отношение имеет к рандомизированным алгоритмам? Мы можем рассматривать рандомизированные алгоритмы как множество детерминированных с каким-то распределнием вероятностей. То есть задачу мы решаем с помощью детерминированного алгоритма, но какого именно мы заранее не знаем. Рассмотрим семейство $\mathcal{A}$ детерменированных алгоритмов. Рассмотрим множество X возможных входов. Введём вероятности на множестве алгоритмов и на множестве входов. Составим матрицу из затрат. Задача становится более похожа на описанную выше матричную игру. Верно следующее:

$$\min_{A \in \mathcal{A}} E_X C_A(x) \leqslant \max_{x \in X} E_{\mathcal{A}} C_A(x). \tag{14.13}$$

Здесь E_X — математическое ожидание, $C_A(x)$ — затраты.

Лекция 15

Итак, мы рассматриваем рандомизированные алгоритмы как множество детерминированных. Пусть нижняя граница для сложности в среднем составляет $f(n)$. Тогда из неравенства (14.13) следует, что и для рандомизированных алгоритмов нижняя граница $f(n)$ будет иметь место.

Битовая сложность

Перейдём к новой теме. Будем обсуждать битовую сложность. До сих пор была сложность, которую называют алгебраической или однородной. Однородность понимается в том смысле, что затраты измеряем числом оперций над входами. Битовая сложность делает шаг в сторону более точного изучения затрат. Каждое число записывается набором бит, все операции, которые производятся над числами или входами другой природы, производядтся над набором бит. Часто они реализуются с помощью булевых операций. Некоторые исследователи называют битовую сложность булевой.

Мы будем в основном заниматься простыми арифметическими задачами.

Наивное сложение и наивное деление — сложение и деление в столбик. Будем считать, что числа записаны в двоичной системе. Для сложения чисел в столбик: пусть складываются два числа битовой длины m_1 и m_2. В качестве размера входа возьмём $m = \max\{m_1, m_2\}$. Нетрудно получить равенство

$$T^*_{Add}(m) = \Theta(m). \tag{15.1}$$

Что касается наивного умножения, то при таком же выборе размера входа верно

$$T^*_{NM}(m) = \Theta(m^2). \tag{15.2}$$

Если же для сложения брать две компоненты размера входа, то можно записать

$$T^{**}_{Add}(m_1, m_2) = \Theta(\max\{m_1, m_2\}). \tag{15.3}$$

Две компоненты для размера входа можно вводить и в случае умножения:

$$T^{**}_{NM}(m_1, m_2) = \Theta(m_1 \cdot m_2). \tag{15.4}$$

Мы давно не обсуждали пространственную сложность. Напишем некоторые равенства для пространственной сложности:

$$S^*_{NM}(m) = 2m + \mathcal{O}(1). \tag{15.5}$$

$$S^{**}_{NM}(m_1, m_2) = m_1 + m_2 + \mathcal{O}(1). \tag{15.6}$$

Что касается наивного деления с остатком, то для него верно

$$T^*_{ND}(m) = \Theta(m^2). \tag{15.7}$$

$$T^{**}_{ND}(m_1, m_2) = \Theta((m_1 - m_2 + 1)m_2). \tag{15.8}$$

Сомножитель $m_1 - m_2 + 1$ отвечает числу вычитаний.

Рассмотрим пример. Пусть для умножения двух целых положительных чисел a и b мы используем следующую схему:

$$a + a, (a + a) + a, \ldots, \underbrace{(a + \ldots + a)}_{b-1} + a. \tag{15.9}$$

Какова сложность? Возьмём в качестве размера входа $m = \max\{m_1, m_2\}$. На каждое сложение уйдёт не менее m битовых операций. Получаем для битовой сложности оценку

$$\Omega(2^m \cdot m). \tag{15.10}$$

С другой стороны, $ab < 2^{2m}$. Это даёт оценку

$$\mathcal{O}(2^m \cdot m). \tag{15.11}$$

Отсюда получаем для битовой сложности наивного умножения

$$\Theta(2^m \cdot m). \tag{15.12}$$

Ещё один пример: вычисление факториала. Используется следующая схема:

$$2 \cdot 3, (2 \cdot 3) \cdot 4, \ldots, (2 \cdot 3 \cdot \ldots \cdot (n-1)) \ldots n. \tag{15.13}$$

Тогда с учётом того, что было получено для наивного умножения, число битовых операций не превзойдёт $c \cdot f(n)$, где

$$f(n) = \log_2 \cdot \log_2 + \log_2(2 \cdot 3) \cdot \log_2 4 + \ldots + \log_2(2 \cdot 3 \cdot \ldots \cdot (n-1)) \cdot \log_2 n. \tag{15.14}$$

Как оценить функцию $f(n)$? Вынесем за скобку $\log_2 n!$. Верно следующее:

$$f(n) \leqslant \log_2 n!(\log_2 3 + \log_2 4 + \ldots + \log_2 n) = \log_2^2 n!. \tag{15.15}$$

Ещё один пример. Когда много входом, что принимать за размер? Когда говорим о битовом размере, можно найти для каждой компоненты входа найти битовую длину и взять максимум. Другой путь — суммарная битовая длина. Следующий пример показывает, что иногда при тком подходе тоже можно получить компактный и информативный результат. Пусть есть числа $a_1, a_2, \ldots, a_n$. Требуется вычислить их произведение. Обозначим их суммарную битовую длину как M. Будем пользоваться наивным умножением. Оценка сложности будет следующей: $\mathcal{O}(M^2)$. Получить её достаточно легко.

Домашнее задание. *Пусть нужно вычислить сумму чисел $1, 2, \ldots, n$. Размером входа будем считать битовую длину числа n. Оценить битовую сложность.*

Домашнее задание. *Пусть F_n — число Фибоначчи. Оценить битовую сложность вычисления F_n.*

Домашнее задание. *Речь идёт об n-разрядном двоичном счётчике. Изначально его содержимое равно 0, потом начинают прибавлять по единице, пока содержимое не станет равным $2^n - 1$. Сколько переносов из разряда в разряд будет произведено суммарно? Ответ известен: $2^n - n - 1$. Доказать это.*

Ещё один пример: деление с остатком. Есть положительные целые числа n, k, причём $k \geqslant 2$. Числа заданы в двоичной системе счисления. Требуется перевести n в k-ичную систему счисления. Перевод в другую систему счисления основан на делении с остатком. Возьмём в качестве размера входа число n. Тогда сложность составит $\mathcal{O}(\log^2 n)$. Получить эту оценку самостоятельно.

Модулярная арифметика

Обсудим теперь сложность алгоритма Евклида. Кроме того, мы будем рассматривать алгоритмы модулярной арифметики. Алгоритм Евклида играет важную роль в таких вычислениях. Например, расширенный алгоритм Евклида нужен для обращения чисел по модулю.

Ранее мы интересовались алгебраической сложностью алгоритма Евклида. В качестве размера входа рассматривалось меньшее число, то есть a_1. Даже если a_0 очень велико, первое же деление с остатком даст число, меньшее чем a_1. В битовом случае уже нельзя так поступить. Здесь нужно брать битовую длину числа a_0 в качестве размера входа.

Напомним, в чём заключается алгоритм Евклида:

$$\begin{aligned}
&a_0 \geqslant a_1;\\
&a_0 = q_1 a_1 + a_2;\\
&a_1 = q_2 a_2 + a_3;\\
&\dots\\
&a_{n-3} = q_{n-2} a_{n-2} + a_{n-1};\\
&a_{n-2} = q_{n-1} a_{n-1} + a_n;\\
&a_{n-1} = q_n a_n.
\end{aligned} \tag{15.16}$$

Сформулируем утверждение:

$$a_0 \geqslant q_1 \cdot q_2 \cdot \ldots \cdot q_n. \tag{15.17}$$

Лекция 16

В прошлой лекции мы вернулись к алгоритму Евклида. Докажем сформулированное в конце прошлой лекции утверждение (15.17). Верно следующее:

$$a_0 > q_1 a_1. \tag{16.1}$$

В свою очередь,

$$a_1 > q_2 a_2. \tag{16.2}$$

Продолжая эту цепочку, получим

$$a_0 \geqslant q_1 \cdot q_2 \cdot \ldots \cdot q_n a_n, \tag{16.3}$$

но поскольку $a_n \geqslant 1$, то выполнено

$$a_0 \geqslant q_1 \cdot q_2 \cdot \ldots \cdot q_n. \tag{16.4}$$

Итак, утверждение (15.17) доказано.

Чтобы перейти к битовой сложности, заметим, что при получении a_{i+1} получается путём деления a_{i-1} на a_i с остатком. Для этого потребуется не более чем

$$c(\lfloor \log_2 q_i \rfloor + 1)(\lfloor \log_2 a_i \rfloor + 2) \tag{16.5}$$

битовых операций, где c — некоторая ненулевая константа. Это следует из того, что наивное деление с остатком состит из ряда последовательных вычитаний. Оценка состоит из количества вычитаний $\lfloor \log_2 q_i \rfloor + 1$ и длины вычитаемого $\lfloor \log_2 a_i \rfloor + 2$. Общее число битовых операций не превосходит

$$c \sum_{i=1}^{n} (\lfloor \log_2 q_i \rfloor + 1)(\lfloor \log_2 a_i \rfloor + 2). \tag{16.6}$$

Эта сумма не превосходит

$$c(\lfloor \log_2 a_1 \rfloor + 2) \sum_{i=1}^{n} (\lfloor \log_2 q_i \rfloor + 1). \tag{16.7}$$

Упростим это выражение. Для этого используем слудующие факты: $a_1 > 1$, отсюда $\log_2 a_1 \geqslant 1$, тогда выполнено

$$\lfloor \log_2 a_1 \rfloor + 2 \leqslant 3 \log_2 a_1. \tag{16.8}$$

Используя полученное неравенство, перепишем сумму (16.7):

$$\begin{aligned} c(\lfloor \log_2 a_1 \rfloor + 2) \sum_{i=1}^{n} (\lfloor \log_2 q_i \rfloor + 1) \leqslant 3c \log_2 a_1 (n + \sum_{i=1}^{n} \log_2 q_i) = \\ = 3c \log_2 a_1 (n + \log_2 (q_1 \cdot \ldots \cdot q_n)) \leqslant 3c \log_2 a_1 (n + \log_2 a_0) \end{aligned} \tag{16.9}$$

Для n имеем логарифмическую оценку сверху. Напомним эту оценку: $n \leqslant 2\log_2 a_1 + 1$. Используя это, получаем оценку сверху для битовой сложности:

$$3c\log_2 a_1(3\log_2 a_1 + \log_2 a_0) \tag{16.10}$$

Итак, имеем оценку $\mathcal{O}(\log^2 a_0)$. Если же в качестве размера входа выбрать битовую длину m числа a_0, то получаем оценку $\mathcal{O}(m^2)$.

Число делений с остатком в алгоритме Евклида есть $\mathcal{O}(\log a_0)$. На каждое деление с остатком приходится $\mathcal{O}(\log^2 a_1)$. Отсюда сразу же получаем оценку для битовой сложности алгоритма Евклида $\mathcal{O}(\log^3 a_1)$. Если в качестве размера входа взять битовую длину m первого числа, то получим оценку $\mathcal{O}(m^3)$. Однако можно показать, что здесь имеет место оценка $\mathcal{O}(m^2)$. В следующих лекциях, посвященных модулярной арифметике, будет использоваться именно оценка $\mathcal{O}(m^2)$ для алгоритма Евклида.

Алгоритм Евклида не диктует, каким образом делить с остатком. Когда мы получали оценку, мы предполагали, что используется наивный алгоритм деления с остатком. Есть и более быстрые алгоритмы деления. Алгоритм Карацубы — один из быстрых алгоритмов умножения. Можно доказать, что быстрые алгоритмы умножения позволяют получить быстрые алгоритмы деления.

Для числа делений устанавливалась оценка сверху, а из неё следовало, что число делений с остатком есть $\mathcal{O}(\log a_1)$. Нельзя ли улучшить эту оценку? Для каждого a_1 можно подобрать a_0, что события будут развиваться очень медленно. Для этого можно взять данное a_1, умножить его на φ и полученное произведение взять в качестве a_0. Мы уже использовали ранее числа Фибоначчи таким образом. Можно показать что если брать такие числа a_0 a_1, то число делений будет как раз порядка лограрифма.

Особенно полезен расширенный алгоритм Евклида. С его помощью можно, например, найти наибольший общий делитель. Асимптотические оценки для расширенного алгоритма Евклида сохраняются. Ранее мы устанавливали неравенства

$$\begin{aligned} |s_n| &\leqslant a_1; \\ |t_n| &< a_0. \end{aligned} \tag{16.11}$$

Эти неравенства окажутся необходимы, чтобы написать оценки.

Модулярные алгоритмы

Модулярные алгоритмы описывают вычисления по некоторому модулю. Важный момент в этой теме — это само определение сравнимости. Пусть задано число k — модуль, целое положительное число. Целые числа называются сравнимыми по модулю k (пишут $a \equiv b(\mod k)$), если $a - b$ делится на k (пишут $k|a-b$). Имеем отношение эквивалетности.

Полезное свойство: если

$$a_1 = b_1(\mod k);\ a_2 = b_2(\mod k), \tag{16.12}$$

то $a_1 \pm a_2 \equiv b_1 \pm a_2(\mod k)$.

Другое полезное свойство: каждое целое число сравнимо по модулю k в точности с одним числом из множества $\{0, 1, \ldots, k-1\}$.

Описанные свойства означают, что операции сложения и умножения можно перенести на классы эквивалентности. То есть, например, сами классы эквивалетности можно складывать. Выбор представителя класса не имеет значения.

Классы можно изобрать числами $\{0, 1, \ldots, k-1\}$. Имеем кольцо $\mathbb{Z}_k$. Это кольцо называется кольцом вычетов по модулю k.

Если k простое, то кольцо превращается в поле. Если же число $k = k_1 \cdot k_2$ составное, то возьмём два класса: первый, содержащий k_1, второй, содержащий k_2. Их произведение равно нулю. В поле не может быть делителей нуля по причине того, что каждый ненулевой элемент имеет обратный. Рассмотрим простой пример. Пусть $a \neq 0,\ b \neq 0$, и

$$a \cdot b = 0. \tag{16.13}$$

Домножим левую и правую часть на a^{-1}:

$$\underbrace{a^{-1} \cdot a}_{=1} \cdot b = 0 \tag{16.14}$$

Получаем, что, $b = 0$, хотя это не так. Значит в поле делителей нуля дествительно не существует.

Как можно искать этот обратный элемент? Рассмотрим $\mathbb{Z}_p$, a и p взаимно простые. Найдутся такие s и t, что будет выполнено

$$sp + ta = 1. \tag{16.15}$$

Утверждается, что t и будет обратным элементом.

Кстати говоря находить обратные можно не только с помощью алгоритма Евклида. Есть метод, например, основанный на малой теореме Ферма.

Теперь зададимся вопросом — можно ли проверить, что данное число простое, за полиномиальное время? Такой алгоритм очень долго искали, и в итоге нашли, сейчас он имеет название AKS. В основе этого алгоритма лежит достаточно простой факт. Пусть a есть целое число, n есть число натуральное, причём a взаимно просто с n. Оказывается, что n будет простым числом тогда и только тогда, когда выполняется

$$(x - a)^n = x^n - a(\mod n). \tag{16.16}$$

Однако нужно использовать некоторую модификацию этого факта и использовать два сравнения по модулю для достижения полиномиальной сложности. Если использовать этот факт в таком виде, то алгоритм тоже можно построить, однако его сложность не будет полиномиальной.

Домашнее задание. *Пусть a, b есть некоторые целые числа, p — простое число, s — натуральное. Доказать формулу*

$$(a + b)^{p^s} = a^{p^s} + b^{p^s}(\mod p). \tag{16.17}$$

Домашнее задание. *Доказать, что аддитивная группа кольца вычетов по модулю есть циклическая группа.*

Домашнее задание. *Доказать, что если p — простое, то мультипликативная группа поля $\mathbb{Z}_p$ есть циклическая группа.*

Домашнее задание. *Пусть p — простое, $0 \leq k < p$, то число сочетаний C_p^k делится на p.*

Лекция 17

Булева арифметика, алгоритм Уолшера, алгоритм Флойда

На этой лекции будет рассматриваться битовая сложность в контексте булевой арифметики. Булева алгебра, конечно, интересна сама по себе, но мы будем рассматривать приложения к графам. Задача в контексте теории графов, которая будет нам полезна для дел, связанных со сложностью, будет задача о построении матрицы соединений (эта матрица показывает, из какой в какую вершину в данном графе можно попасть). Также нас будет интересовать задача перемножения булевых матриц.

Говоря о матрице соединений, заметим следующее. Рассмотрим граф, который изначально у нас есть, без кратных рёбер и конечным числом вершин. Этот граф можно рассматривать как задающий некоторое бинарное отношение на конечном множестве, которое есть множество вершин. И построение матрицы соединений есть ни что иное, как построение транзитивно-рефлексивного замыкания заданного бинарного отношения. Поэтому матрицу соединений часто называют замыканием исходной матрицы смежности. И если исходная матрица смежности обозначается как C, то матрица замыкания обозначается как C^*.

Если у нас есть исходная матрица смежности C, то если мы возьмём её квадрат (в булевом смысле), то мы получем матрицу, которая оказывает, из какой вершины в какую можно добраться ровно за два шага. C^3, соответсвенно, будет показывать, из какой вершины в какую можно добраться ровно за три шага, и так далее. Можно рассматривать какие угодно степени, хотя вряд-ли нам будет интересно рассматривать более чем n шагов в графе из n вершин, так как мы не хотим несколько раз проходить одну и туже вершину. Поэтому имеет смысл рассматривать вот такую, неформально говоря, сумму:

$$I \vee C \vee C^2 \vee C^3 \vee \ldots \vee C^n = C^*. \tag{17.1}$$

Есть соображение, которые позволяют использовать быстрое возведение в степень. Оно заключается в следующем:

$$I \vee C \vee C^2 \vee C^3 \vee \ldots \vee C^n = (I \vee C)^n. \tag{17.2}$$

Это соотношение, например, можно доказать по индукции. Теперь всё, что нам нужно, это вычислить эту одну степень. Здесь никто не мешает использовать алгоритм, который требует не более двух логарифмов двоичных n умножений.

Можно и дальше раскручивать соображения в эту сторону. Заметим, что увеличивая n мы не будем портить результат, поэтому никто не запрещает увеличить n до степени двойки в большую сторону. Тогда это уменьшит количество действий, необходимых для возведения в степень, так как на каждом шаге мы будем каждый раз возводить в квадрат. Если идти таким путём, то задача построения замыкания булевой матрицы (построения рефлексивно-транзитивного замыкания бинарного отношения) будет иметь сложность $\Theta(n^3 \log_2 n)$.

Оценка нам, конечно, нравится. Но всё равно возникает вопрос — а можно ли её ещё как-нибудь улучшить? Но в нашем случае вряд-ли это можно сделать.

За счет чего? Множитель n^3 возникает из тривиального алгоритма умножения матриц, с этим бороться совсем сложно. С логарифмом тоже непонятно, как можно его уменьшать.

Алгоритм Уолшера

Посмотрим ещё на другой алгоритм, который называется алгоритмом Уоршела. Рассмотрим ту же ситуацию, в которой нужно пробегать от вершины к вершине в графе и искать матрицу замыкания. Идея Уоршела похожая — шаг за шагом продвигаться к решению задачи, правда шаги будут уже другие. На шаге с номером k рассматриваются все пути из вершины i в вершину j, где i и j пробегают все номера от 1 до n, которые в качестве промежуточных путей используют вершины, принадлежащие множеству $\{1, 2, \ldots, k\}$. k будет меняться от 1 до n. На последнем шаге мы разрешаем промежуточным вершинам быть любыми, тем самым получая на последнем шаге ту матрицу, которую изначально и хотели получить. Будем строить матрицу

$$D^{(k)} = \left(d_{ij}^{(k)}\right), \tag{17.3}$$

в которой элементы удовлеворяют следующему соотношению:

$$d_{ij}^{(k)} = d_{ij}^{(k-1)} \vee \left(d_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)}\right). \tag{17.4}$$

Это соотношение позволяет идти от матрицы с меньшим значком к матрице со значком на единицу больше. Так и будем ходить, с $k = 1$ до $k = n$. В итоге мы должны получить $C^* = D^{(n)}$. Также понятно, что так как нас интересует только матрица с последним номером, то сохранять полученные в процессе счёта матрицы с меньшими номерами не нужно, достаточно обновлять входную матрицу, используя рекурентное соотношение. Пространственная сложность этого алгоритма будет ограниченной величиной, то есть $\mathcal{O}(1)$.

Давайте теперь обозначать входную матрицу как C, а её элементы c_{ij}. Напишем некоторую программу на этот алгоритм:

```
for l=1 to n do c_{ll}:=1 od;
for k=1 to n do
    for i=1 to n do
        for j=1 to n do
            c_{ij}:=c_{ij} OR (c_{ik} AND c_{kj})
        od
    od
od
```

Этот алгоритм требует булевых операций в количестве $2n^3 + n$.

Рекуррентные соотношения как средства анализа сложности алгоритма

Рассмотрим n — разрядный двоичный счётчик. Зададимся вопросом, сколько переносов из разряда в разряд потребуется для прогона счётчика от всех нулей до

всех единиц. Ответ для этой задачи есть $2^n - n - 1$. *Для того, чтоб получить этот ответ, придется решить следующее рекурентное уравнение:*

$$s(n) = 2s(n-1) + n - 1,\ \ s(0) = 0. \tag{17.5}$$

Такие соотношения решаются практически как дифференциальные уравнения — нужно найти общее решение однородного уравнения, решив характеристическое уравнение, и сложить его с каким-нибудь частным решением неоднородного уравнения. В нашем случае характеристическое уравнение имеет очень простой вид:

$$\lambda - 2 = 0, \tag{17.6}$$

следовательно общее решение может быть записано в виде $C \cdot 2^n$. *А частное решение в нашей ситуации имеет вид* $(-n-1)$. *Собирая всё вместе, получаем:*

$$s(n) = C \cdot 2^n - n - 1, \tag{17.7}$$

где C *есть некоторая постоянная. Эту постоянную можно легко определить из условия* $s(0) = 0$. *Получим* $C = 1$. *В итоге мы и получили желаемый ответ:*

$$s(n) = 2^n - n - 1. \tag{17.8}$$

Рассмотрим ещё какой-нибудь пример, где возникают рекуррентные соотношения с постоянными коэффициентами. Пусть имеем рекурсию, организованную следующим образом. Пусть Y_n *— нечто вычисляемое, причём может быть вычислена как*

$$Y_n = U(Y_{n-1}, Y_{n-2}), \tag{17.9}$$

Y_0 *и* Y_1 *заранее известны. Вычисляя значения* Y_n, *можно задаться вопросом, а сколько раз будет вычислена функция* U? *На этот счёт можно написать рекурентное уравнение:*

$$y(n) = y(n-1) + y(n-2) + 1, \tag{17.10}$$

где функция y *показыват, сколько раз будет вычислена функция* U. *Решая аналогичным способом, что и предыдущее, получаем:*

$$y(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1} - 1. \tag{17.11}$$

Хотя если написать некоторые нехитрые асимптотические оценки, то можно удивиться, почему получилось так много — показательная степень вместо очевидной полиномиальной. Всё дело в том, что рекурсия устроена так, что некоторые величины будут вычисляться дважды, что повлечёт за собой прирост сложности.

Можно пойти гораздо дальше и рассмотреть следующую рекурсию:

$$Y_n = U(Y_{n-1}, Y_{n-k}), \tag{17.12}$$

где $k \geq 2$. А вообще наибольший интерес представляет ситуация, когда k — довольно большое число. Насколько хуже предыдущей будет эта ситуация? Напишем рекурентное соотношение для этой задачи:

$$y(n) - y(n-1) - \ldots - y(n-k) = 1, \tag{17.13}$$

причём $y(0), y(1), \ldots, y(k-1) = 0$. Частное решение такого уравнения есть $\frac{-1}{k-1}$, проверяется подстановкой. Анализируя расположения корней характеристического уравнения на комплексной плоскости можно установить, что сложность будет $\Theta(\alpha_k^n)$, где α_k — так называемый главный корень:

$$2 - \frac{1}{k} \leq \alpha_k < 2. \tag{17.14}$$

Домашнее задание. *В алгоритме Уоршела можно ли первую строчку написанного нами кода поместить в конец?*

Домашнее задание. *Преобразовать алгоритм Уоршела в алгоритм Флойда.*

Домашнее задание. *Можно считать, что алгоритм Флойда построен на операциях $a \odot b = a + b$ и $a \oplus b = \min(a, b)$. Показать, что для полноты нужно считать, что 0 является нейтральным элементом по отношению к такому умножению, а ∞ нулевым элементом по отношению к такому сложению.*

Лекция 18

Асимптотические оценки

Рассмотрим алгоритм сортировки, который называется рекурсивной сортировкой слияния. Исходный массив делится на две примерно равные части, после чего они сортируются рекурсивным обращением к той-же сортировке. Получаются два упорядоченных массива, и далее прибегаем к процедуре слияния. MS будем записывать как аббревиатуру этого алгоритма. Можно записать, что сложность по числу сравнений будет

$$T_{MS}(n) \leq \begin{cases} 0, \text{ если } n = 1, \\ T_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1, \text{ если } n > 1. \end{cases} \tag{18.1}$$

Первые два записаны в таком виде из-за того, что наш массив может быть нечётным, поэтому нужно округление. Знак неравенства берётся из-за того, что мы рассматриваем число сравнений в худшем случае. На самом деле можно доказать, что неравенство можно заменить на равенство, и всё останется верным (хотя это уже придётся доказывать). Рекуррентные неравенства, схожие с этим неравенством, типичны для алгоритмов, построенных на стратегии разделяй и властвуй. Эта стратегия заключается в том, что исходная задача заменяется на некоторое количество задач того же типа, но меньшего размера, которые решаются независимо, и в конце решения этих более мелких задач собираются некоторым образом в решение исходной задачи.

Всё это было мотивацией к тому, чтобы научиться работать с подобного вида неравенствами.

Предложение.

1) *Пусть вещественная функция натурального аргумента $f(n)$ удовлетворяет неравенству:*

$$f(n) \leq \begin{cases} u, \text{ если } n = 1, \\ v \cdot f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + w \cdot f\left(\left\lceil \frac{n}{2} \right\rceil\right) + \varphi(n), \text{ если } n > 1, \end{cases} \tag{18.2}$$

где u, v, w — неотрицательные вещественные числа, причём $v + w \geq 1$, а функция φ неотрицательная и неубывающая. Тогда $f(n) \leq t(\lceil \log_2 n \rceil)$. Здесь функция t имеет иследующий вид:

$$t(k) = \begin{cases} u, \text{ если } k = 0, \\ (v + w)t(k - 1) + \varphi(2^k), \text{ если } k > 0, \end{cases} \tag{18.3}$$

Существенно то, что функция t определеяется уже в виде равентсва. Интересно то, что определение функции t можно прочитать как некоторое рекуррентное уравнение с постоянными коэффициентами. Будем говорить о нём как о уравнении, ассоциированным с исходным неравенством.

2) *Пусть теперь*

$$f(n) \geq \begin{cases} u, \text{ если } n = 1, \\ v \cdot f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + w \cdot f\left(\left\lceil \frac{n}{2} \right\rceil\right) + \varphi(n), \text{ если } n > 1. \end{cases} \tag{18.4}$$

Тогда $f(n) \leq t(\lfloor \log_2 n \rfloor)$.

Доказательство. *Можно доказать по индукции. В доказательстве рассматриваются следующие функцию:*

$$F(n) = \begin{cases} u, \text{ если } n = 1, \\ v \cdot F\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + w \cdot F\left(\left\lceil \frac{n}{2} \right\rceil\right) + \varphi(n), \text{ если } n > 1. \end{cases} \tag{18.5}$$

При аргументе функции, большем единицы, эта функция является неубывающей. Это доказывается по индукции, а затем, используя это свойство, и доказывается само утверждение. Полного доказательства приводить не будем, так как оно не содержит идейно интересных моментов. □

Давайте вернёмся к нашему алгоритму сортировки. Для него ассоциированное уравнение будет иметь следующий вид:

$$t(k) = \begin{cases} 0, \text{ если } k = 0, \\ 2t(k-1) + 2^k - 1, \text{ если } k > 0. \end{cases} \tag{18.6}$$

Решить это уравнение несложно:

$$t(k) = (k-1)2^k + 1. \tag{18.7}$$

Вспоминая сформулированное предложение получаем, что

$$(\lfloor \log_2 n \rfloor - 1)2^{\lfloor \log_2 n \rfloor} + 1 \le T_{MS}(n) \le (\lceil \log_2 n \rceil - 1)2^{\lceil \log_2 n \rceil - 1} + 1. \tag{18.8}$$

Так как эти две оценки отличаются лишь округлением, то это хорошие оценки.

Если подумать для полноты картины над пространственной оценкой, то можно получить что-то в духе $n + \mathcal{O}(1)$.

То есть, если у нас есть алгоритм, основанный на принципе разделяй и властвуй, то можно так и работать — писать ассоциированные уравнения. Но зададимся вопросом, можно ли это как-то обойти? Чтобы ответить на этот вопрос, сформулируем некоторую теорему.

Теорема о рекурсивных неравенствах

Master theorem. *Пусть вещественная функция натурального аргумента* $f(n)$ *удовлетворяет неравенству:*

$$f(n) \le \begin{cases} u, \text{ если } n = 1, \\ v \cdot f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + w \cdot f\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn^d, \text{ если } n > 1, \end{cases} \tag{18.9}$$

где u, v, w, d *— неотрицательные вещественные постоянные,* $v + w \ge 1$, $c > 0$. *Тогда*

$$f(n) = \begin{cases} \mathcal{O}(n^d \log_2 n), \ d = \log_2(v+w), \\ \mathcal{O}(n^d), \ d > \log_2(v+w), \\ \mathcal{O}(n^{\log_2(v+w)}), \ d < \log_2(v+w). \end{cases} \tag{18.10}$$

Эта теорема позволяет не держать в голове никаких ассоциированных уравнений, а лишь искать соответствующую альтернативу в рамках утверждения

этой теоремы. Теорема доказывается с помощью взгляда на ассимптотики ассоциированного уравнения. Альтернатива как раз и появляется из установления того факта, какое слагаемое из решения ассоциированного уравнения растёт быстрее.

Можно рассмотреть также случай, когда

$$f(n) \geq \begin{cases} u, \text{ если } n = 1, \\ v \cdot f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + w \cdot f\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn^d, \text{ если } n > 1. \end{cases} \tag{18.11}$$

Тогда утверждается, что

$$f(n) = \begin{cases} \Omega(n^d \log_2 n), \ d = \log_2(v+w), \\ \Omega(n^d), \ d > \log_2(v+w), \\ \Omega(n^d), \ d < \log_2(v+w). \end{cases} \tag{18.12}$$

Доказательство этого факта также опустим.

Венёмся к примеру о сортировке со слияниями и рассмотрим сложность по числу перемещений элементов. Можно написать, что

$$\tilde{T}_{MS}(n) = \begin{cases} 0, \text{ если } n = 1, \\ \tilde{T}_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \tilde{T}_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + n, \text{ если } n > 1. \end{cases} \tag{18.13}$$

В этом случае $w + v = 2$. Из сформулированных утверждений вытекают следующие факты:

$$\tilde{T}_{MS}(n) = \mathcal{O}(n \log n), \tag{18.14}$$

$$\tilde{T}_{MS}(n) = \Omega(n \log n). \tag{18.15}$$

Задача о построении выпуклой оболочки

Как можно применить стратегию разделяй и властвуй применительно к задаче о построении выпуклой оболочки? Можно разбить множество точек на две равные или примерно равные (если их число нечётно) части, построить выпуклые оболочки этих частей, а затем построить выпуклую оболочку объединения этих двух получившихся многоугольников. Для этих построений имеется алгоритм, имеющий сложность $\mathcal{O}(n)$, где n есть исходное число точек. В этом случае получим следующее соотношение:

$$T(n) = \begin{cases} 0, \text{ если } n = 1, \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n), \text{ если } n > 1. \end{cases} \tag{18.16}$$

Таких ситуаций в наших теоремах не предполагалось, однако очень легко выйти из такого положения. Достаточно заметить, что

$$T(n) \leq \begin{cases} 0, \text{ если } n = 1, \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + C \cdot n, \text{ если } n > 1, \end{cases} \tag{18.17}$$

где C — некоторая постоянная. И тогда, в соответствии с нашими теоремами:

$$T(n) = \mathcal{O}(n \log n). \tag{18.18}$$

Домашнее задание. *Разработать оптимальный алгоритм перемещений и найти его сложность по числе перекладываний дисков в игре Ханойские Башни.*

Домашнее задание. *Решить предыдущую задачу при условиях, что цена перекладывания i-ого диска есть i^2 и 2^i.*

Домашнее задание. *Доказать, что*

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1, \tag{18.19}$$

где F_n есть n-ое число Фибоначи.

Лекция 19

Алгоритмы умножения, сводимость алгоритмов

Когда мы имеем какие-то арифметические задачи, удобно, чтобы длина чисел была степенью двойки, чтобы оно всё время делилось пополам, сколько его не дели. Это иногда бывает очень удобно. Чтобы так получалось, можно в начало числа добавлять какое-то количество нулей, ведь это ничего не меняет. Хотя это добавление увеличивает количество операций. Однако иногда это приводит к алгоритмам, имеющим меньшую сложность, чем тривиальный алгоритм. Как пример можно привести умножение Карацубы.

Алгоритм Карацубы

Смысл его в следующем. Пусть мы хотим перемножить целые числа a и b, имеющие битовую длину 2^k. Представим эти числа в следующем виде:

$$a = e2^l + f, \tag{19.1}$$

$$b = g2^l + h, \tag{19.2}$$

где $l = 2^{k-1}$, e, g, f, h — целые числа битовой длины l. Справедливо следующее соотношение.

$$ab = eg2^l + \big((e+f)(g+h) - eg - fh\big)2^l + fh. \tag{19.3}$$

Наблюдение Карацубы состоит в том, что в этой формуле фигурируют всего 3 нетривиальных умножения, тогда как если перемножать в лоб числа a и b, раскрывая скобки, то получится 4 операции умножения. Тут, конечно, есть и другие операции, но они явно менее затратны (умножение на степень двойки есть дописывание некоторого количества нулей, сложение гораздо проще умножения). Будем этот алгоритм обозначать KM. Для этого алгоритма можно написать

$$T_{KM}(m) \le \begin{cases} 1, \text{ если } m = 1, \\ 3T_{KM}\left(\frac{m}{2}\right) + cm, \end{cases} \tag{19.4}$$

где c есть некоторая константа. Из ассоциированного уравнения для этого неравенства моментально следует, что сложность алгоритма будет $\mathcal{O}(m^{\log_2 3}) = \mathcal{O}(m^{1.58\ldots})$, в то время как для тривиального алгоритма была бы справедлива оценка $\mathcal{O}(m^2)$, так что выигрыш явно есть. Также можно доказать, что будет справедлива аналогичная оценка $\Omega(m^{\log_2 3})$.

Алгоритм Штрассена

Пусть у нас есть матрицы A и B, размер которых есть чётное число. Тогда их можно поделить на блоки следующим образом:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{19.5}$$

Тривиальный алгоритм перемножения таких матриц содержит в себе восемь умножений, а алгоритм Штрассена показывает, что можно ограничиться семью умножениями. В этом алгоритме вычисляются семь матриц:

$$\begin{aligned}
X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}),\\
X_2 &= (A_{21} + A_{22})B_{11},\\
X_3 &= A_{11}(B_{12} - B_{22}),\\
X_4 &= A_{22}(B_{21} - B_{11}),\\
X_5 &= (A_{11} + A_{12})B_{22},\\
X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}),\\
X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),
\end{aligned}$$

вычисление которых, очевидно, содержит семь умножений. Обозначим результат произведения

$$A \cdot B = C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. \tag{19.6}$$

Используя вычисленные нами матрицы X_i возможно записать результат произведения следующим образом:

$$C_{11} = X_1 + X_4 - X_5 + X_7, \tag{19.7}$$
$$C_{21} = X_2 + X_4, \tag{19.8}$$
$$C_{12} = X_3 + X_5, \tag{19.9}$$
$$C_{22} = X_1 + X_3 - X_2 + X_6. \tag{19.10}$$

Это утверждение проверяется непосредственно из перемножения матриц и последующего упрощения получившегося выражения. Про сложность этого алгоритма можно написать следующее:

$$T_{St}(n) \leq \begin{cases} 1, \text{ если } n = 1, \\ 7T_{St}\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2, \text{ если } n > 1. \end{cases} \tag{19.11}$$

Также для сложности можно получить оценку $\mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.81\ldots})$, что меньше сложности тривиального алгоритма $\mathcal{O}(n^3)$.

Теперь зададимся следующим вопросом. Ранее на лекциях мы расматривали задачу о нахождении матрицы замыкания по матрице смежности. Можно ли для этой задачи, не задумываясь о булевости, применить алгоритм Штрассена? На самом деле нет, так как матрицы X_i в своём определении содержат операцию вычитания. А для полей и колец алгоритм Штрассена прекрасно работает. Но на самом деле и для булевых матриц есть выход. Оказывается, что если все вычисления проводить не в кольце $\mathbb{Z}$, а в кольце $\mathbb{Z}_{n+1}$, то всё будет работать. Останется в получившейся окончательной матрице все ненулевые элементы заменить на единицы. Для этого алгоритма будет справедлива оценка $\mathcal{O}(\log^2 n \cdot n^{\log_2 7})$.

Алгоритм Тоома

Идея Тоома является обобщением идеи Карацубы. Тоом предложил рассматривать числа не вида степеней двойки, а вида степеней некоторого целого числа s, и показал, что перемножение таких чисел будет иметь $2s-1$ нетривиальных умножений чисел длины в s раз меньше исходных. Сложность умножения Тоома подчиняется неравенству

$$T_{TM}^{(s)}(m) \leq \begin{cases} 1, \text{ если } m = 1, \\ (2s-1)T_{KM}^{(s)}\left(\frac{m}{2}\right) + cm. \end{cases} \tag{19.12}$$

Для этого алгоритма будет справедлива следующая оценка:

$$T_{TM}^{(s)}(m) = \mathcal{O}(m^{\log_s(2s-1)}). \tag{19.13}$$

Из этого следует, что для любого ε существует такой алгоритм умножения Тоома с некоторым достаточно большим числом s, что его сложность будет $\mathcal{O}(m^{1+\varepsilon})$. Но можно пойти ещё дальше. Оказывается, что существует алгоритм, сложность которого будет $\mathcal{O}(m \log m \log\log m)$ (алгоритм Шенхаге — Штрассена).

Линейная сводимость

Шутка. *Если $\mathbb{P} = 0$ или $\mathbb{N} = 1$, то $\mathbb{P} = \mathbb{NP}$.*

Неформально говоря, сводимость позволяет установить, что если для какой-то задачи нет быстрого алгоритма, то и у сводящейся задачи его тоже нет, или что наличие такого алгоритма у одной задачи влечёт наличие такого-же алгоритма у другой, может быть даже непохожей, задачи. Будем предполагать, что сопоставляя между собой какие-то задачи, то ответом к этим задачам будет некоторый алгоритм, который и решает задачи, сложность алгоритмов для первой и для второй задачи есть сложность в худшем случае по числу некоторых операций, и операции эти в одной и в другой задаче одинаковы. Также предполагаем, что размер входа для первой и второй задачи выбран как-то единообразно, чтобы можно было как-то осмысленно сравнивать сложности.

Немного формализуем наши рассуждения.

Определение. *Пусть есть некоторые задачи P и Q. Говорят, что задача P линейно сводидится к задаче Q, если для любого алгоритма A_P существует алгоритм A_Q такой, что*

$$T_{A_P} = \mathcal{O}(T_{A_Q}). \tag{19.14}$$

Иногда это записывают как $P \leq Q$ и говорят, что задача P не сложнее задачи Q.

Примеры. *Будем рассматривать задачи возведение в квадрат целого числа (обозначим как задача S) и умножение двух целых чисел (обозначим как задача M).*

В задаче об умножении чисел на вход подаются два числа n_1 и n_2, размером входа считаем максимальную из битовых длин этих чисел, обозначаем как m. В задаче об возведении в квадрад на вход подаётся некоторое целое число, размер

входа есть его битовая сложность m. Так как имеет место очевидное равенство $n^2 = n \cdot n$, то задача S не сложнее задачи M (или, другими словами, $S \leq M$). Если наложить некоторые ограничения на задачи квадрирования (отсекая нерациональные алгоритмы), то с помощью формулы

$$n_1 \cdot n_2 = \frac{(n_1 + n_2)^2 - n_1^2 - n_2^2}{2} \tag{19.15}$$

можно доказать, что задача M не сложнее задачи S. Без дополнительных ограничений сделать это не удастся, так как сумма n_1 и n_2 может иметь большую битовую сложность, чем входная. Дополнительные ограничения, которые отсекают нерациональные алгоритмы, могут быть, например, следующими:

1) $T(m) \geq m$,
2) $T(m)$ *не убывает при возрастании* m,
3) $T(2m) \leq 4T(m)$.

Без этих ограничений доказательство того, что $M \leq S$ было бы невозможным.

Домашнее задание. *Доказать, что*

$$\forall \varepsilon > 0 \exists N > 0 : \ 3T_{KM}(m) < T_{KM}(2m) < (3 + \varepsilon)T_{KM}(m) \ \forall m \geq N,$$

где m — битовая длина.

Домашнее задание.

$$\forall \varepsilon > 0 \exists N > 0 : \ 7T_{St}(n) < T_{St}(2n) < (7 + \varepsilon)T_{St}(n) \ \forall n \geq N.$$

Домашнее задание. *Можно ли вычислить произведение двух комплексных чисел $a + ib$ и $c + id$, затратив всего три произведения вещественных чисел a, b, c, d?*

Лекция 20

Сводимость

В качестве справки к предыдущей лекции ответим на вопрос, развилась ли куда-то дальше история с алгоритмами умножения? Вспомним, что мы остановились на алгоритме Шенхаге — Штрассена, сложность которого оценивается как $\mathcal{O}(m \log m \log\log m)$. Да, действительно развилась. Математик по фамилии Фюрер доказал, что существует алгоритм, сложность которого оценивается как $\mathcal{O}\big(m \log m \cdot f(m)\big)$, где функция $f(m)$ растёт медленнее, чем двойной (и, более того, кратный) логарифм. Продвинулись люди и с алгоритмом перемножения матриц. Напомним, что в на прошлой лекции мы узнали, что существует алгоритм перемножения, сложность которого оценивается как $\mathcal{O}(n^{2.81\ldots})$. Оказывается, что существует так называемый алгоритм Василевска — Вильямс, предложенный в 2011 году, который имеет сложность $\mathcal{O}(n^{2.23728642})$. Дальше, вроде бы, дело не продвинулось. Но существует открытая гипотеза, что и для матриц $\forall \varepsilon > 0$ существует алгоритм, сложность которого оценивается как $\mathcal{O}(n^{2+\varepsilon})$.

Теперь продолжим разговор о сводимости. Рассмотим более содержательный, чем на прошлой лекции, пример сопоставления умножения булевых матриц и задачи построения рефлексивно-транзитивного замыкания графа. Вспомним, что если n — число вершин в графе, то задача построения матрицы замыкания C^* по данной матрице смежности строится следующим образом:

$$C^* = (I + C)^n. \tag{20.1}$$

Однако это не есть сведение задачи о построения замыкания к задаче умножения матриц, так как тут перемножаются не две, а n матриц.

Вспомним, что у нас был алгоритм Уоршела, который давал сложность $\backslash^{\ni}$. Но если мы докажем, что имеет место неравенство

$$\text{Построение замыкания} \leq \text{умножение булевых матриц}, \tag{20.2}$$

то тем самым мы докажем, что существует алгоритм, который работает лучше, чем алгоритм Уоршела, так как обобщение алгоритма Штрассена на случай перемножения булевых матриц работает лучше, чем $\mathcal{O}(n^3)$.

Для начала будем предполагать, что в графе чётное число вершин, то есть пусть $n = 2l$, где l — целое. Обозначим множество вершин как $V = \{v_1, v_2, \ldots, v_{2l}\}$. Разобьём это множество на две части:

$$V_1 = \{v_1, v_2, \ldots v_l\},$$
$$V_2 = \{v_{l+1}, v_{l+2}, \ldots, v_{2l}\}.$$

Множество рёбер, которые мы обозначим E, разобьётся тем самым на четыре множества, которые будем обозначать $E_{11}, E_{12}, E_{21}, E_{22}$. Разобьём нашу матрицу C следующим образом:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. \tag{20.3}$$

Тогда можно сказать, что блок C_{pq} несёт полную инфорацию о множестве рёбер E_{pq}.

Рассмотрим пути, связанные с множеством V_1, то есть начинающийся в вершине из V_1 и заканчивающийся в вершине V_1. Допускается, что промежуточные вершины могут принадлежать множеству V_2. Такие пути могут быть реализованы разными вариантами. Это может быть путь, который проходит только по рёбрам из E_{11}. А может быть, что путь проходит по рёбрам E_{12}, путешествует по рёбрам E_{22} и заканчивает рёбрами E_{21}. Будем, для краткости, такие пути обозначать как $E_{12} \cdot E_{22} \cdot E_{21}$. Такую последовательность путей будем называть ходом. Матрица, характеризующая ходы из точки в точку, есть

$$(C_{11} \vee (C_{12} \wedge C_{22}^* \wedge C_{21}))^* := F_{11}. \tag{20.4}$$

По аналогии можно определеть матрицы F и с другими индексами:

$$\begin{aligned} F_{12} &= F_{11} \wedge C_{12} \wedge C_{22}^*, \\ F_{21} &= C_{22}^* \wedge C_{21} \wedge F_{11}, \\ F_{22} &= C_{22}^* \vee (C_{22}^* \wedge C_{21} \wedge F_{11} \wedge C_{12} \wedge C_{22}^*). \end{aligned}$$

Тогда наша матрица C^* может быть представлена как

$$C^* = \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}. \tag{20.5}$$

Пусть сложность алгоритма перемножения булевых матриц есть B, а T — сложность алгоритма построения замыкания, то можно написать следующее неравенство:

$$T(2^k) \leq \begin{cases} 0, \text{ если } k = 0, \\ 2T(2^{k-1}) + 6B(2^{k-1}) + 2 \cdot 2^{2(k-1)}, \ k > 0. \end{cases} \tag{20.6}$$

На булево умножение налагаются следующие условия:

1) $B(1) = 1$,

2) $B(n)$ *не убывает при возрастании* n,

3) $4B(n) \leq B(2n) \leq 8B(n)$.

Неравенство можно переписать в несколько ином виде:

$$T(2^k) \leq \begin{cases} 0, \text{ если } k = 0, \\ 2T(2^{k-1}) + 8B(2^{k-1}), \ k > 0. \end{cases} \tag{20.7}$$

По индукции можно доказать, что

$$T(2^k) \leq 4B(2^k). \tag{20.8}$$

Мы, таким образом, доказали, что

$$\text{Построение замыкания} \leq \text{умножение булевых матриц.} \tag{20.9}$$

Можно возразить, что мы доказали это только для специфического размера матриц, а конкретно для размера 2^k. Но это не проблема, так как можно увеличить размер матрицы до нужного безболезненым путём например вот так:

$$C' = \begin{pmatrix} C & 0 \\ 0 & I \end{pmatrix}. \tag{20.10}$$

Теперь зададимся следующим вопросом, а верно ли обратное неравенство:

$$\text{Построение замыкания} \geq \text{умножение булевых матриц?} \tag{20.11}$$

Оказывается, что это тоже можно доказать при некоторых предположениях относительно алгоритмов.

Сводимость и нижние границы

Вспомним про алгоритмы построения выпуклой оболочки. Эти алгоритмы используют арифметические операции и операции сравнения над точками на плоскости. Можно доказать, что сортировка сводится (в нашем смысле) к построению выпуклой оболочки многоугольника на плоскости. Но мы знаем, что слишком уж быстро сортировку не выполнишь. Из этого следует, что выпуклую оболочку тоже не слишком уж быстро можно построить. Но заметим, что выпуклая оболочка строится не только с помощью операций сравнения, там тоже есть арифметика. На самом деле можно доказать, что используя алгоритмы сортировки со сравнениями и арифметическими операциями, то нижняя граница сложности для числа сравнений в виде двоичного логарифма от факториала сохраняется. Докажем это на следующей лекции.

А сейчас займёмся вопросом сводимости сортировки к построению выпуклой оболочки. Пусть у нас есть некоторые числа — $x_1, x_2, \ldots, x_n$. Рассмотрим эти числа на оси обсцисс вещественной плоскости. На этой плоскости отметим точки (x_i, x_i^2), $i = \overline{1, n}$ и построим выпуклую оболочку получившегося множества. Можно ли из этого множества извлечь порядок нашего исходного массива? Оказывается можно, если вспомнить, что при построении выпуклой оболочки указывается обход. Поэтому достаточно обойти наше множество и найти точку снаименьшей абсциссой. Это можно сделать, и для этого понадобится $\mathcal{O}(n)$ операций. Потом проходим по ребру от найденой вершины к соседней в правильном направлении по ребру и находим вторую по величине точку и т.д. Так мы и свели оду задачу к другой. Мы, конечно, совершили лишнюю работу при нахождении точки с наименьшей абсциссой, но это не проблема, так как известно, что не существует алгоритма построения выпуклой оболочки, который затрачивал бы меньше, чем n операций.

Домашнее задание. *Доказать, что умножение матриц сводится к умножению симметричных матриц.*

Домашнее задание. *Доказать, что умножение матриц сводится к умножению верхних треугальных матриц.*

Домашнее задание. *Доказать, что умножение матриц сводится к обращению невырожденной матрицы.*

Всё это можно доказывать в предположении, что $T(kn) = \mathcal{O}\big(T(n)\big),\ k = 2, 3.$

Лекция 21

Рациональные функции

Пусть у нас есть две задачи, P и Q, и выполнено $P \leq Q$. Утверждается, что если у нас есть асимптотическая нижняя граница для алгоритмов решения задачи P, то эта же функция и будет асимптотической нижней границой для алгоритмов решения задачи Q. Это вытекает из:

$$T_{A_P}(n) = \mathcal{O}\big(T_{A_Q}(n)\big) \Longrightarrow T_{A_Q}(n) = \Omega\big(T_{A_P}(n)\big), \tag{21.1}$$

и, по условию,

$$T_{A_P}(n) = \Omega\big(f(n)\big). \tag{21.2}$$

Из этих двух формул вытекает

$$T_{A_Q}(n) = \Omega\big(f(n)\big). \tag{21.3}$$

Правда, это всё работает, если никаких ограничений на алгоритмы задачи Q не наложено.

Казалось бы, можно взять как задачи Q и P задачи о сортировке и построении выпуклой оболочки. Однако нам мешают арифметические операции. Когда у нас примешиваются арифметические операции, мы вовлекаем в наши сравнения не только элементы массива, но и значения рациональных функций(отношение двух полиномов) от эих элементов. Нам будут полезны некоторые свойства:

- *Пусть у нас есть два множества. Первое множество есть множество, в котором рациональная функция обращается в ноль, а второе множество есть такое множество, где рациональная функция не определена. Оба этих множества замкнуты.*
- *Если полином многих переменных тождественно равен нулю на некотором открытом множестве, то он тождественно равен 0.*

Нам это нужно в контексте того, что каждой перестановке некоторого массива соответствует сектор в $\mathbb{R}^n$. Это множество таких точек, координаты которых попарно различны, и их взаимный порядок по велечине соответствует заданной перестановке.

Вспомним, что сортировке мы сопоставляли дерево. Также заметим, что сравнение двух рациональных функций эквивалентно сравнению одной с нулём (в вершине дерева). А в листах дерева будут записаны рациональные функцыы, вычисляя значение которых можно получить исходный порядок:

$$x_{l_k} = G_{lk}(x_1, \ldots, x_n), \;\; k = \overline{1, n}. \tag{21.4}$$

Можно показать, что листьев должно быть не меньше, чем $n!$, иначе в один и тот-же лист можно было бы придти с двух разных точек. Из этого всего можно вывести, что для алгоритмов построения выпуклой оболочки двоичный логарифм факториала есть асимптотическая нижняя граница по числу операций сравнения и арифметики.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ