АЛГОРИТМЫ И МОДЕЛИ ВЫЧИСЛЕНИЯ

Вас ждут две (или три) курсовые контрольные. Они будут проходить, соответственно, 20.03 (midterm), 15.05 (final) и 22.05 (утешительная) в Б.Хим. и Акт.зале с 13:55 – 17:05.

Обязанности сторон и правила аттестации (протокол)

1. В нашем курсе поддерживается режим свободного перехода. Обязательным условием перехода является **согласие принимающей стороны**. Для оформления перехода нужно написать заявление, подписать у обоих семинаристов и отнести его на кафедру. Процесс должен завершится в течение трех недель.

2. Протокол получения оценки близок к протоколу, использованному на ТРЯП и состоит из двух пунктов.

(*i*) Студент, получивший положительную оценку по обеим курсовым контрольным и (условно) "сдавший задание"—- это означает, что семинарист положительно оценивает вашу работу в семестре (у каждого семинариста могут быть свои критерии, см. ниже) — имеет право получить зачетную оценку, равную max(полусумма курсовых контрольных, оценка семинариста).

Для этой категории студентов повышение оценки выше протокольной возможно, если это допускает ваш семинарист. Если ваш семинарист считает, что ваша оценка заслуженная, а вы не согласны с этим, то этот вопрос может быть рассмотрен в присутствии всех заинтересованных сторон комиссией.

(*ii*) Студент, получивший хотя бы одну неудовлетворительную оценку на курсовых контрольных или пропустивший одну из них (**даже по уважительной причине**), обязан писать утешительную контрольную, которая может быть дополнена устным опросом.

(*ii*) -а. Если утешительная контрольная написана на положительную оценку, то вопрос о зачете передается семинаристу. И также, если вы настаиваете, что ваша работа недооценена (или переоценена), то этот вопрос может быть рассмотрен в присутствии всех заинтересованных сторон комиссией.

(*ii*)-б. Если утешительная контрольная не написана на положительную оценку, то для получения зачетной оценки нужно следить за графиком пересдач (и процесс продолжается).

3. Идущее ниже задание является **официальным**. В частности, это означает, что при возникновении каких-то коллизий (например, при незачетах и/или стремлении повысить оценку, полученную по протоколу и т.д.) лично я буду исходить из предположения, что студенты представляют себе, как решать задачи из этого задания.

Каждый семинарист имеет право вносить в него необходимые изменения. Кроме того, в конце курса семинаристы будут оценивать работу студентов. Эта оценка может весьма существенно сказаться на зачетной оценке.

ПРОГРАММА КУРСА

Приводим понедельный план наших занятий.

1. **Темы 1-й недели 6.02-12.02. 1 лекция.**

Формальное определение алгоритма. Емкостная и времéнная сложность алгоритма. Различные определения сложности алгоритма. Эффективные (полиномиальные алгоритмы). Примеры алгоритмов и иллюстрация принципов разработки алгоритмов: проверка простоты и факторизация чисел; сортировка слиянием; Линейный алгоритм поиска медианы.

Асимптотические оценки. Нотация: $O(\cdot)$, $o(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, $\Theta(\cdot)$.

2. **Темы 2-й недели 13.02-19.02. 2-я лекция.**

Модели вычислений. Формальное определение алгоритма. Разрешимые и перечислимые языки[1] Теорема о линейном ускорении. Теоремы о времéнной и емкостной иерархии (без доказательства). Определения сложностных классов ($\mathcal{P}, \mathcal{NP}, \mathcal{PSPACE}$).

Примеры алгоритмов и иллюстрация принципов разработки алгоритмов: алгоритм Евклида; индийское возведение в степень; одновременное вычисление максимального и минимального элементов в массиве; поиск ближайшей пары точек; быстрое умножение чисел и матриц: (алгоритмы Карацубы и Штрассена).

Асимптотические оценки. Нотация: $O(\cdot)$, $o(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, $\Theta(\cdot)$. Основная теорема о рекуррентных оценках (нахождение асимптотики рекуррентности вида $T(n) = aT(\frac{n}{b}) + f(n)$). Дерево рекурсии. Линейный вероятностный алгоритм нахождения медианы. Решение линейных рекуррентных уравнений.

4. **Темы 3–4 недель 20.02-5.03.**

Класс $\mathcal{P}$. Примеры языков из $\mathcal{P}$: принадлежность слова регулярному языку; принадлежность слова КС-языку; системы линейных уравнений (полиномиальная реализация метода Гаусса). Классы NP и co-NP. Примеры языков из NP: выполнимость, простые числа; пересечение регулярных языков, заданных конечными автоматами; непланарные графы.

5. **Темы 4–5 недель 27.02-1.03 4–5 лекции.**

Полиномиальная сводимость. Сводимость по Карпу и по Куку (по Тьюрингу). Теорема Кука-Левина. Примеры полиномиально полных языков: выполнимость; протыкающее множество; максимальное 2-сочетание; 3-сочетание; вершинное покрытие; клика; хроматическое число; гамильтонов цикл; рюкзак; разбиение; максимальный разрез; N[ot]A[ll]E[qual]-выполнимость.

6. **Темы 6-й недели 13.03-20.03. 6-я лекция.**

Теория NP-полноты.

Числовые алгоритмы[2]. Обобщенный алгоритм Евклида. Решение линейных диофантовых уравнений. Модульная арифметика. Китайская теорема об остатках. Функция Эйлера. Первообразные корни. Кольца Z_n, в которых существуют первообразные корни. Индексы (дискретные логарифмы). Кодирование с открытым ключом. Квадратичные вычеты. Схемы RSA шифрования и циф-

[1] Повторение того, что вы изучали в курсе ТФС три месяца назад. Исходя из стабильного прошлого опыта, для многих эта тема окажется совсем не повторением, а, если так уместно выразиться, откровением.

[2] Формально повторяются темы второго семестра (за возможным исключением алгоритма RSA).

ровой подписи, дискретное логарифмирование. Протокол Диффи-Хелмана.

В понедельник 20.03 с 13:55–17:05 в Актовом зале и Большой химической прошла первая курсовая контрольная работа по темам 1–6.

Напоминаем, что по протоколу студенты, получившие неудовлетворительную оценку на первой или второй контрольной, не могут быть аттестованы без получения положительной оценки на последующих контрольных (как это происходит на ТРЯП). К этой группе относятся и студенты, пропустившие контрольную даже по уважительной причине.

7. **Темы 7-й недели 20.03-26.03. 7 лекция.**

Структуры данных. Стек и очередь. Двоичная куча (пирамида). Двоичное дерево поиска[3]. Амортизационный анализ. Хеш-таблицы. Разрешение коллизий с помощью цепочек. Хеш-функции (деление с остатком, умножение). Универсальные и k-универсальные хеш-функции.

8. **Темы 8-й недели 27.03-2.04 8-я лекция.**

Дискретное преобразование Фурье (ДПФ); алгоритм быстрого преобразования Фурье (БПФ); перемножение многочленов с помощью БПФ. Поиск подстрок. Использование БПФ для распознавания образцов. Циркулянты. Решение линейных уравнений с циркулянтными матрицами.

9. **Темы 9-й недели 3.04-9.04. 9-я лекция.**

Алгоритмы сортировки: "пузырек, "быстрая сортировка (quicksort); сортировка с помощью кучи (heapsort); сортировка слиянием (mergesort). Анализ трудоемкости алгоритма quicksort по наихудшему случаю и в среднем. Устойчивость алгоритма сортировки. Цифровая сортировка. Порядковые статистики. Нижние оценки в модели разрешающих деревьев. Понятие о более сложных структурах данных (АВЛ- и 2-3 деревья, сбалансированные деревья).

10. **Темы 10-й недели и 11-й недели 10.04-23.04. 10–11-я лекции.**

Потоки и разрезы в сети. Теорема о максимальном потоке и минимальном разрезе. Понятие остаточного графа и увеличивающего пути. Метод Форда-Фалкерсона для вычисления максимального потока и минимального разреза. Обобщения потоковой сети (пропускные способности узлов и пр.). Задача о максимальном паросочетании в двудольном графе. Задача линейного программирования. Основные понятия. Выпуклые многогранники. Теорема двойственности. Задача назначении.

11. **Темы 11-й – 12-й недель 17.04-30.04. 11–12-я лекции.**

Алгоритмы на графах: поиск в ширину; поиск в глубину; определение двусвязных и/или сильносвязных компонент; топологическая сортировка. Остовные деревья: алгоритмы Прима и Краскала. Кратчайшие пути: алгоритмы Дейкстры, Флойда, Беллмана–Форда. Паросочетания. Структура "union-find" для хранения системы непересекающихся множеств.

12. **Тема 13-й недели 1.05-7.05. 13-я лекции.**

Вероятностные алгоритмы. Классы $\mathcal{RP}$, $\mathcal{BPP}$, $\mathcal{ZPP}$. Вероятностные алгоритмы: проверка простоты; вычисление медианы массива; проверка полиномиальных тождеств; поиск паросочетаний в графах; алгоритм Каргера поиска минимального разреза. Приближённые вероятностные алгоритмы поиска максимального разреза. Лемма Шварца–Зиппеля. Дерандомизация.

14. **Тема 14-й недели 8.05-14.05. 14-я лекция.**

Методы решения переборных задач: динамическое программирование, шкалирование, ветви и границы, приближенные алгоритмы для задачи максимального разреза. ε-оптимальная процедура решения задачи о рюкзаке.

Заключительная контрольная пройдет в понедельник 15.05 с 13:55–17:05 в Акт. зале и Б.Хим.

Утешительная контрольная пройдет в понедельник 22.05 с 13:55–17:05 в Акт. зале и Б.Хим .

ЛИТЕРАТУРА

Основная

1. **[АХУ]** *Ахо А., Хопкрофт Д., Ульман Д.* Построение и Анализ Вычислительных Алгоритмов. М.: Мир, 1979.
2. **[ГД]** *Гери М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
3. **[ДПВ]** *Дасгупта С., Пападимитриу Х., Вазирани У.* Алгоритмы. М.: МЦНМО, 2014.
4. **[Кормен 1]** *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: Построение и Анализ. М.: МЦНМО, 2002.
5. **[Кормен 2]** *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: Построение и Анализ. (2-е изд.) М.: Вильямс, 2005.
6. **[Кормен 3]** *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: Построение и Анализ. (3-е изд.) М.: Вильямс, 2013.
7. **[ХМУ]** *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.,: Вильямс, 2002.
8. **[AB]** *Arora S., Barak B.* Computational Complexity: A Modern Approach. theory.cs.princeton.edu/complexity/book.pdf
9. **[GL]** *Gács P., Lovasz L.* Complexity of Algorithms. www.cs.elte.hu/∼lovasz/complexity.pdf

Дополнительная

1. *Верещагин Н., Шень А.* Вычислимые Функции. М.: МЦНМО, 1999. (Электронный вариант: www.mccme.ru/free-books)
2. **[Виноградов]** *Виноградов И.* Основы теории чисел. М.-Л.: Гостехиздат, 1952
3. *Вялый М., Журавлев Ю., Флеров Ю.* Дискретный анализ. Основы высшей алгебры. М.: МЗ Пресс, 2007.
4. **[К-Ш-В]** *Китаев А., Шень А., Вялый М.* Классические и квантовые вычисления. М.: МНЦМО-ЧеРо, 1999.

[3] Формально повторяются уже известные структуры данных, но упор делается на их качественный анализ, например, на сложность выполнения конкретных операций.

5. **[Кнут-1, Кнут-2, Кнут-3, Кнут-4] (цифра отвечает номеру тома** *Кнут Д.* Искусство программирования для ЭВМ. Существует несколько изданий на русском. Первое было выпущено издательством "Мир" в семидесятых. В настоящее время выпускается издательством "Вильямс". Есть многочисленные сетевые варианты.
6. **[К-Ф]** *Кузюрин Н., Фомин С.* Эффективные алгоритмы и сложность вычислений. М.: МФТИ, 2007.
7. **[П-С]** *Пападимитриу Х., Стайглитц К.* Комбинаторная оптимизация. Алгоритмы исложность. М.: Мир, 1985.
8. **[Хинчин]** *Хинчин А.* Цепные дроби. М.: Наука, 1979.
9. **[Ш]** *Шень А.* Программирование. Теоремы и задачи. М.: МЦНМО, 2007. (Электронный вариант: www.mccme.ru/free-books)
10. *Lovasz L.* Computational complexity. www.cs.elte.hu/∼lovasz/complexity.pdf

Задание на недели 7–13

Задание состоит из списка недельных заданий (указаны даты обсуждения на семинарах и/или сдачи соответствующих задач). Каждое недельное задание состоит из четырех-пяти обязательных задач и одной-двух дополнительных задач (дополнительные задачи выделены буквой «Д»). Решение дополнительных задач не обязательно, но может быть полезно студентам, претендующим на более высокую оценку.

Обозначения

Пусть $f(n)$ и $g(n)$ — неотрицательные функции.

- $f(n) = O(g(n))$ означает, что порядок роста g при $n \to \infty$ *не меньше* порядка роста f, т. е. $\exists c > 0|$ при $n > n_0\, f(n) \le cg(n)$;
- $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$ (порядок роста f не меньше порядка роста g);
- $f(n) = (g(n))$, если $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ (f имеет меньший порядок роста, чем g);
- $f(n) = \omega(g(n)) \Leftrightarrow g(n) = o(f(n))$ (g имеет меньший порядок роста, чем f);
- $f(n) = \Theta(g(n))$, если $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$ (порядки роста f и g одинаковы).

По определению, $\log^* M = \min\{k \mid \underbrace{\log\log\ldots\log M}_{k \text{ раз}} \le 1\}$.

$\lfloor a \rfloor$ обозначает наибольшее целое, не превосходящее число a
$\lceil a \rceil$ обозначает наименьшее целое, превосходящее a

Необходимо знать какой-нибудь вывод формулы Стирлинга $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$ и суммы гармонического ряда $\sum_{i=1} n\frac{1}{i} = \ln n + \gamma$, где $gamma = 0.57721\ldots$ — это так называемая константа Эйлера. Кроме того, нужно знать, что такое числа Каталана $c_n = \frac{1}{n+1}\binom{2n}{n}$ (c_n имеет множество комбинаторных интерпретаций, например, равно правильно построенных скобочных выражений или путей Дика длины $2n$) и выражение для их производящей функции $D(t) \stackrel{def}{=} \sum_{n=0}^{\infty} c_n x^n = \frac{1-\sqrt{1-4x}}{2x}$.

Задание на седьмую неделю: 20.03–26.03. [0.2]

Раздел 7 программы

Повторение. Решение линейных диофантовых уравнений и систем линейных сравнений. КТО
Литература: [Виноградов]

Задача 30. (3×0.01) Докажите или опровергните[4] существование полиномиальных алгоритмов для
(i) нахождения вычета, удовлетворяющего К[итайской] Т[еореме об] О[статках], т. е. нахождения единственного вычета, удовлетворяющего системе сравнений по различным простым модулям p_i, $i = 1, \ldots, k$

$$\begin{aligned} x &= a_1 (\mathrm{mod}\ p_1) \\ &\ldots \\ x &= a_k (\mathrm{mod}\ p_k) \\ 0 \le\ x &< p_1 p_2 \ldots p_k. \end{aligned}$$

(ii) проверки совместности системы сравнений

$$\begin{aligned} x &= a_1 (\mathrm{mod}\ m_1) \\ &\ldots \\ x &= a_k (\mathrm{mod}\ m_k) \end{aligned}$$

здесь модули m_i — произвольные натуральные числа;
(iii) решения системы линейных диофантовых уравнений $\sum_{j=1}^{m} a_{ij}x_j = b_i$, $a_{ij}, x_j, b_i \in \mathrm{Z}$, $i = 1, \ldots, n$, $j = 1, \ldots, m$.

Хеширование
Литература: [Кормен 1, Глава 12]
[Кормен 2, Глава 11], [ДПВ, §1.5]

Рекомендуем также посмотреть http://e-maxx/algo/string_hashes

Краткий конспект. Хеш- функции

Хеш- функции описывают специальную дисциплину обращения с большими массивами. Для построения хороших хеш- функций часто используют теоретико- числовые методы.

Мотивация для введения хеш-функций такова. Далее используется удачный пример из **[ДПВ]**.

Допустим, что мы хотим поддерживать динамический массив IP-адресов (скажем, клиентов или друзей в какой-то социальной сети). Напомним, что IP-адрес представляет 32-битный код, обычно разбитый на 8-битные куски $x_1.x_2.x_3.x_4$, например, 193.133.89.10. Понятно, что держать в памяти все 2^{32} возможных IP-адресов накладно, тем более, если, по вашим оценкам, число клиентов или друзей никак не сможет превысит порог, скажем, 250. Однако, если задать IP-адреса списком, то каждое обращение потребует просмотра всего списка, что может быть медленным. Поэтому предлагается дать каждому IP-адресу x "короткое имя" $h(x)$, быстро вычисляемое посредством специальной хеш-функции $h(\cdot)$, и ввести маленький массив из связанных списков, для их размешения. Например, в качестве короткого имени можно выбрать первый байт IP-адреса. При этом мы понимаем, что многие IP-адреса "склеятся", т.е. получат одно и то же короткое имя и будут помещены в связанный список под этим именем. Например, это произойдет, если клиенты (или друзья) обитают в Азии. Также понятно, что как бы мы ни выбирали хеш-функцию, для нее априори всегда существуют совокупность "плохих" IP-адресов. Что же делать?

Вспомним, что неформально хеш-функция должна выдавать на выходе равномерно распределенные "случайные" имена, так чтобы длины связанных списков не слишком отличались. Мы, конечно, не можем разрушить возможную организацию во входах, но кто нам мешает ввести рандомизацию на множестве функций, применяемых для хеширования[5]? Например, можно действовать так.

[4] В предположении $\mathcal{P} \neq \mathcal{NP}$ или трудности задачи факторизации.

[5] Обратите, пожалуйста, внимание на этот прием: мы сознательно разрушаем возможную организацию входных данных, что позволяет нам в дальнейшем считать, что вход уже не содержит возможных "патологий", поскольку механизм разрушения зависимостей **не связан** с исходной организацией. Типичный пример: всем известный QUICKSORT.

Зафиксируем простой модуль $n = 257$ близкий к 250. Выберем случайно четверку чисел $a = \{a_1, a_2, a_3, a_4\}$ по (mod 257) и определим хеш-функцию $h_a(x_1, x_2, x_3, x_4) = \sum_1^4 a_1 x_1 \pmod{257}$. Например, рассмотренная ранее хеш-функция, выделяющая старший байт равна $h_{1,0,0,0}$.

Пусть $x_1.x_2.x_3.x_4$ и $y_1.y_2.y_3.y_4$ — *различные* IP-адреса. Предположим, что коэффициенты a_i, $i = 1, 2, 3, 4$ выбираются случайно и независимо и равновероятно принимают значение из $\{0, 1, \ldots, n-1\}$. Тогда[6]

$$Prob[h_a(x_1, x_2, x_3, x_4) = h_a(y_1, y_2, y_3, y_4)] = \frac{1}{n}. \qquad (1)$$

Иными словами, вероятность совпадения хеш-значений двух различных IP-адресов, вычисленных посредством случайно выбранной хеш-функции семейства, равна вероятности совпадения хеш-значений, если бы они выбирались случайно и независимо. В частности, если мы поместим m IP-адресов в хеш-таблицу, то к каждому имени в среднем будет "привешено" не более $\frac{m}{n}$ IP-адресов, чего мы и добивались.

Описанный принцип выбора хеш-функций называется **универсальным хешированием**.

Формальное определение таково. Семейство $\mathcal{H}$ хеш-функций, отображающих множество возможных ключей U в $\{0, 1, \ldots, m-1\}$, называется *универсальным*, если для двух различных ключей $x, y \in U$ число функций $h \in \mathcal{H}$, для которых $h(x) = h(y)$, меньше или равно[7] $\frac{|\mathcal{H}|}{m}$. Из описанного примера следует, что семейство $\mathcal{H}_1 = \{h_a,\, a \in \{0, \ldots, 256\}^4\}$ является универсальным

Анализ (**[Кормен 1, Теорема 12.3]** или гораздо более подробная **Кормен 2,Теорема 11.3]**) показывает, что универсальные семейства хеш-функций позволяют строить хеш-таблицы, с короткой в среднем длиной связанных списков.

Введем еще одно понятие

Семейство $\mathcal{H}$ хеш-функций, отображающих множество возможных ключей U в $\{0, 1, \ldots, m-1\}$, называется *k-универсальным*, если для любой последовательности k различных ключей $(x_1, \ldots, x_k)$ случайный вектор $\langle h(x_1), \ldots, h(x_k)\rangle$ (здесь h— случайная функция из $\mathcal{H}$) принимает равновероятно все m^k своих возможных значений.

Задача 31. (0.01) Докажите формулу (1).

Задача 32. (3×0.02) В этой задаче p — простое число, а все равенства понимаются по модулю p.
(*i*) Пусть $\vec{x}, \vec{y} \in (\mathrm{Z}/p\mathrm{Z})^n$. Пусть вектор $\vec{v} \in (\mathrm{Z}/p\mathrm{Z})^n$ выбирается случайно (каждый его элемент выбирается из $\mathrm{Z}/p\mathrm{Z}$ равновероятно и независимо от других). Найдите $\mathbb{P}(\langle \vec{v}, \vec{x}\rangle = \langle \vec{v}, \vec{y}\rangle)$.
(*ii*) Пусть $\vec{x}, \vec{y} \in (\mathrm{Z}/p\mathrm{Z})^n$. Пусть матрица $A \in (\mathrm{Z}/p\mathrm{Z})^{n\times m}$ выбирается случайно (каждый её элемент выбирается из $\mathrm{Z}/p\mathrm{Z}$ равновероятно и независимо от других). Найдите $\mathbb{P}(A\vec{x} = A\vec{y})$.
(*iii*) Постройте универсальное семейство $\mathcal{H}$ хеш-функций $h : (\mathrm{Z}/p\mathrm{Z})^n \to (\mathrm{Z}/p\mathrm{Z})^m$. Семейство должно иметь мощность $|\mathcal{H}| = p^{mn}$.

Задача Д–15. (0.02) Для хеширования множества из N элементов (элементы заранее не известны, но принимают значения из множества ключей $\gg N$) используют хеш-таблицу размера N^2 и хеш-функцию, случайно выбранную из универсального семейства. Докажите, что вероятность выбрать хеш-функцию удачно (избежать коллизий) не меньше $\frac{1}{2}$.

[6] Пока можно понимать вероятность "наивно", т. е. как отношение числа благоприятных исходов к общему числу исходов. Полезно заглянуть **[Кормен 1, гл. 12]**, **[Кормен 2, гл. 11]** (лучше посмотреть оба текста, поскольку они сильно отличаются).

[7] В **[Кормен 1]** требуется равенство.

Стандартные структуры данных
Литература: [Кормен 1]
[Кормен 2, Главы 10, 12], [Ш]

В теоминимум входят стандартные простые структуры данных: *массив, список (однонаправленный и двунаправленный), двоичное дерево, стек, очередь. очередь с приоритетами, куча (heap)*[8] и **вопросы, связанные с этими структурами могут быть включены в тест**. Более сложные структуры: *сбалансированное дерево (АВЛ- и/или 2 – 3- и/ или красно-черное деревья)* мы рассмотрим в задании № 9, посвященном сортировке. Аммортизационный анализ будет востребован при изучению алгоритма vманипуляции с деревьями UNION-FIND в задании № 12. Вы должны представлять, как реализовать такие структуры на компьютере, знать, какие операции над ними допускаются и уметь оценивать трудоемкость этих операций. В дальнейшем при обсуждении конкретных алгоритмов (например, поиска-в-глубину или поиска минимального дерева в графе) мы затронем также вопросы сложности этих алгоритмов при их реализации посредством каких-то структур данных, причем вопрос о том, какую структуру следует выбрать является совсем непраздным. Конечно, в отдельных случаях может быть совершенно очевидно, что нужно использовать, но это бывает далеко не всегда.

Краткий конспект: стек, очередь, двоичная куча
автор: И. Козлов

Под структурой данных мы будем понимать некоторый абстрактный тип данных вместе с набором операций, которые составляют ее интерфейс. Операции позволяют добавлять, изменять, искать и удалять данные из структуры.

Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании. Сильной стороной этой методики является именно сокрытие реализации. Если пользователю доступен только интерфейс, то до тех пор, пока структура данных поддерживает этот интерфейс, все программы, работающие с этим абстрактным типом данных, будут функционировать корректно. Разработчики структур данных стараются, не меняя внешнего интерфейса и семантики функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надежности и используемой памяти.

Различие между абстрактными типами данных и структурами данных, которые реализуют абстрактные типы, можно пояснить на примере **стека** и **очереди**. Это простейшие структуры данных, реализующие динамические множества, которые поддерживают только операции вставки (*push* для стека и *enqueue* для очереди) и удаления (*pop* для стека и *dequeue* для очереди). Разница между ними в том, что стек представляет собой список элементов, организованный по принципу L[ast]I[n]F[irst]O[ut] "последним пришёл — первым вышел"), а очередь — F[irst]I[in]F[irst]O[ut]. Оба этих типа данных могут быть реализованы при помощи массива или линейного списка, с использованием различных методов динамического выделения памяти. Подробнее о деталях реализации можно посмотреть в **Кормен**. Однако каждая реализация определяет один и тот же набор функций, который должен работать одинаково (по результату, а не по скорости) для всех реализаций.

Задача 33. (2×0.01) (*i*) Покажите, как реализовать очередь с помощью двух стеков. (*ii*) Покажите, как реализовать стек с помощью двух очередей.

Задача 34. (2×0.01) (*i*) Изобразите последовательность <13,4,8,19,5,11>, хранящуюся в дважды связанном списке, представленном с помощью нескольких массивов.

[8] **Мы исходим из предположения, что этот раздел уже изучался вами на информатике.** В задании проводится качественный анализ этих структур по сложности.

(*ii*) Выполните это же задание для представления с помощью одного массива.

Приведем список типичных операций над динамическим множеством S, элементами которого являются пары ключ-значение. Каждый конкретный интерфейс может реализовывать некоторые операции из этого списка или какие-то иные операции, как правило, связанные с нижеперечисленными.

SEARCH(S, k) Запрос возвращает указатель на элемент x заданного множества S, для которого $key[x] = k$ или NIL, если в множестве S такой элемент отсутствует.

INSERT(S, x) Пополняет заданное множество S элементом x.

DELETE(S, x) Удаляет из заданного множества S элемент x.

MINIMUM(S) Возвращает указатель на элемент множества S с наименьшим ключом. Аналогично формулируется запрос MAXIMUM(S).

SUCCESSOR(S, x) Возвращает указатель на элемент множества S, ключ которого является ближайшим соседом ключа элемента x и превышает его. Если же x — максимальный элемент множества S, то возвращается значение NIL. Аналогично формулируется запрос PREDECESSOR(S, x).

Задача Д–16. (4×0.01) Определите асимптотическое время выполнения в наихужшем случае перечисленных в таблице операций над элементами динамических множеств, если эти операции выполняются со списками перечисленных ниже типов (расшифровка обозначений в таблице: ULL (несортированный однократно связанный список); SLL (сортированный однократно связанный список); UDLL (несортированный дважды связанный список); SDLL (сортированный дважды связанный список).

	ULL	SLL	UDLL	SDLL
SEARCH(S,k)				
INSERT(S,x)				
MAXIMUM(S)				
SUCCESSOR(S,x)				

Очередь с приоритетом (англ. *priority queue*) — абстрактный тип данных в программировании, поддерживающий две обязательные операции— добавить элемент и извлечь максимум. Соответственно, интерфейсы, реализуемые очередью с приоритетом, следующие:

- insert(key, value) — добавляет пару (key, value) в хранилище;
- extract-minimum()— возвращает пару (key, value) с минимальным значением ключа, удаляя её из хранилища.

При этом меньшее значение ключа соответствует более высокому приоритету.

В некоторых случаях более естественен рост ключа вместе с приоритетом. Тогда второй метод можно назвать extract-maximum(). В качестве примера очереди с приоритетом можно рассмотреть список задач исполнителя. Когда он заканчивает одну задачу, он переходит к очередной — самой приоритетной (ключ будет величиной, обратной приоритету) — то есть выполняет операцию извлечения максимума. Начальник добавляет задачи в список, указывая их приоритет, то есть выполняет операцию добавления элемента. Очереди с приоритетом используются в некоторых эффективных алгоритмах на графах, например в алгоритме Дейкстры, а также в алгоритме пирамидальной сортировки.

Очередь с приоритетами может быть реализована на основе различных структур данных. Простейшие (и не очень эффективные) реализации могут использовать неупорядоченный или упорядоченный массив, связный список, подходящие для небольших очередей. При этом вычисления могут быть как "ленивыми" (тяжесть вычислений переносится на извлечение элемента), так и ранними (eager), когда вставка элемента сложнее его извлечения. То есть, одна из операций может быть произведена за время $O(1)$, а другая в худшем случае — за $O(N)$, где N — длина очереди. Более эффективными являются реализации на основе кучи, где обе операции можно производить в худшем случае за время $O(\log N)$. К ним относятся двоичная куча, биномиальная куча, фибоначчиева куча.

Рассмотрим далее двоичную кучу или пирамиду — такое двоичное дерево, для которого выполнены три условия:

- Значение в любой вершине не меньше, чем значения её потомков.
- Глубина листьев (расстояние до корня) отличается не более чем на 1 слой.
- Последний слой заполняется слева направо.

Такая пирамида называется невозрастающей (max-heap). Удобная структура данных для пирамиды — массив A, у которого первый элемент, $A[1]$ — элемент в корне, а потомками элемента $A[i]$ являются $A[2i]$ и $A[2i + 1]$ (при нумерации элементов с первого). При нумерации элементов с нулевого, корневой элемент — $A[0]$, а потомки элемента $A[i]$ — $A[2i+1]$ и $A[2i+2]$. При таком способе хранения последние два условия выполнены автоматически.

Высота пирамиды определяется как высота двоичного дерева. То есть она равна количеству рёбер в самом длинном простом пути, соединяющем корень кучи с одним из её листьев. Высота кучи есть $\Theta(\log N)$. Если в пирамиде изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности. Для восстановления этого свойства служит процедура Heapify. Она восстанавливает свойство кучи в дереве, у которого левое и правое поддеревья удовлетворяют ему. Эта процедура принимает на вход массив элементов A и индекс i. Она восстанавливает свойство упорядоченности во всём поддереве, корнем которого является элемент $A[i]$.

Если i-й элемент больше, чем его сыновья, всё поддерево уже является пирамидой, и делать ничего не надо. В противном случае меняем местами i-й элемент с наибольшим из его сыновей, после чего выполняем Heapify для этого сына.

Процедура выполняется за время $O(\log N)$.

```
Heapify(A, i)
left = 2i
right = 2i+1
heap_size - количество элементов в куче
largest = i
if left <= A.heap_size и A[left] > A[largest]
  then largest = left
if right <= A.heap_size и A[right] > A[largest]
  then largest = right
if largest > i
  then Обменять A[i] , A[largest]
       Heapify(A, largest)
```

Задача 35. $(0.01 + 0.02)$ (*i*) Проиллюстрируйте работу процедуры $Heapify(A, 3)$ с массивом
$A =< 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 >$
(*ii*) Теперь построим пирамиду. Заметим, что если выполнить Heapify для всех элементов массива A, начиная с последнего и кончая первым, он станет пирамидой. В самом деле, легко доказать по индукции, что к моменту выполнения $Heapify(A, i)$ все поддеревья, чьи корни имеют индекс больше i, являются пирамидами, и, следовательно, после выполнения $Heapify(A, i)$ пирамидой будут все поддеревья, чьи корни имеют индекс, не меньший i.

Кроме того, $Heapify(A, i)$ не делает ничего, если $i > N/2$ (при нумерации с первого элемента), где N — количество элементов массива. В самом деле, у таких элементов нет потомков, следовательно, соответствующие поддеревья уже являются пирамидами, так как содержат всего один элемент.

Таким образом, достаточно вызвать Heapify для всех элементов массива A, начиная (при нумерации с первого элемента) с $[N/2]$-го и кончая первым. **(Упражнение. Вставьте, пожалуйста, нужные значки вместо вопросов в листинг ниже.)**

```
Build-Max-Heap(A)
  A.heap_size ? A.length
  for i ? [A.length/2] downto 1
    do Heapify(A, i)
```

Докажите, что хотя здесь происходит $n/2$ вызовов функции Heapify со сложностью $O(\log N)$, но время работы равно процедуры $Build - Max - Heap(A)$ равно $O(N)$.

Задача Д–17. (3×0.01) (*i*) Покажите, что пирамида реализует очередь с приоритетами. Для этого напишите функции $Heap - Extract - Max(A)$ и $Max - Heap - Insert(A, key)$. Докажите, что обе функции работают за $O(\log N)$.
(*ii*) Пирамиду можно построить с помощью многократного вызова процедуры Max-Heap-Insert для вставки элементов в пирамиду. Рассмотрим следующий вариант процедуры Build-Max-Heap:

```
Build-Max-Heap-By-Inserts(A)
  A.heap_size = 1
  for i ? 2 to A.length
    do Max-Heap-Insert(A,A[i])
```

Всегда ли процедуры Build-Max-Heap и Build-Max-Heap-By-Inserts для одного и того же входного массива создают одну и ту же пирамиду? Докажите что это так или приведите контрпример.
(*iii*) Покажите, что в наихудшем случае для создания N-элементной пирамиды процедуре Build-Max-Heap-By-Inserts потребуется время $O(N \log N)$.

Амортизационный анализ
Литература: [Кормен 1, Главы 18, 22] [Кормен 2, Главы 17, 21], [АХУ, §§4.7–4.8]

Помимо измерений сложности конкретной процедуры вне контекста можно интересоваться её сложностью в длинной серии исполнений над постоянно меняющимися данными. В этой ситуации вполне может оказаться, что среднее время исполнения значительно меньше худшего времени исполнения. Поэтому имеет смысл рассмотреть так называемую *амортизированную* (или *учётную*) сложность алгоритма. Следующая стадартная задача поясняет и мотивирует эту постановку. Более сложные примеры мы разберем с следующем задании.

Задача 36. (0.01) Рассматривается тривиальная структура данных, которая хранит натуральное число n в двоичном представлении ($\lceil \log_2 n \rceil$ битов). При инициализации $n = 1$. Единственная поддерживаемая операция *Add* – увеличить хранимое число на 1. Ясно, что если число заканчивается k нулями, то эта операция потребует выполнения $(k+1)$ элементарных действий. Поэтому оценка времени работы *Add* по худшему случаю – $\Theta(\log n)$. Представим теперь, что программист длительное время работает с этой структурой и совершает большое число N запросов к процедуре *Add*. Докажите, что суммарное время работы всех выполнений процедуры оценивается как $\Theta(N)$, а не $\Theta(N \log N)$. В таких ситуациях говорят, что *амортизированное* время работы операции *Add* есть $\Theta(1)$.

Задание на 8-ю неделю: 27.03–2.04. [0.12]
Умножение многочленов и алгоритм БПФ
Раздел 8 программы
Литература: [Кормен 1, гл.6 и §32] [Кормен 2, гл 6. и §30], [ДПВ] [Виноградов]

Нужно обязательно прочитать статью (она выложена на нашем сайте): P. Clifford, R. Clifford. Simple deterministic wildcard matching. Information Processing Letters 101 (2007) 53–54. и посмотреть сайт

https://en.wikipedia.org/wiki/Circulant_matrix

КЛЮЧЕВЫЕ СЛОВА (минимальный необходимый объем понятий и навыков по этому разделу): Дискретное преобразование Фурье (ДПФ); быстрое преобразование Фурье (БПФ); схемы БПФ, перемножение многочленов с помощью БПФ. Поиск подстрок посредством БПФ. Циркулянты. Решение линейных уравнений с циркулянтными матрицами с помощью БПФ. Системы линейных уравнений с теплицевыми и ганкелевыми матрицами.

Краткий конспект

Вы уже знакомы с парой процедур, которые до какой-то степени определяют лицо нашего предмета (и не только нашего!). Это, конечно, алгоритм Евклида и метод Гаусса решения систем линейных уравнений. В этом задании мы разберем третий великий алгоритм: быстрое преобразование Фурье (БПФ). Несмотря на то, что этот метод был предложен значительно позже двух своих знаменитых собратьев[9], но в настоящее время он вряд ли уступает им по распространенности. Можно сказать, что в обыденной жизни вы встречаетесь с ним даже чаще, поскольку БПФ "вшито" чуть ли ни в каждый мобильник.

По определению, *дискретное преобразование Фурье (ДПФ)* — это отображение, переводящее последовательность коэффициентов (вообще говоря, комплексных) многочлена $f(x) = a_0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1} \in \mathrm{C}[x]$ в последовательность его значений в корнях n-й степени из единицы, т.е. $(a_0, a_1, \ldots, a_n) \mapsto (f(\omega_n^0), f(\omega_n^1), \ldots, f(\omega_n^{n-1}))$, где $\omega_n = \exp \frac{2\pi i}{n}$. На первый взгляд, кажется, что вычисление требует $\Omega(n^2)$ элементарных арифметических операций, ведь нужно подставить каждый корень из единицы в многочлен $f(\cdot)$ и повторить операцию n раз.

Обратите внимание, что мы пользуемся не нашей стандартной битовой моделью сложности, а используем "наивную" 'ньюмерическую' модель вычислений с плавающей запятой (float)". Иначе говоря, считаем, что у нас есть возможность достаточно точно производить ПРИБЛИЖЕННЫЕ операции с плавающей точкой так, что они существенно не повлияют на ответ. Сложностью называем полное число арифметических операций. Такая модель широко распространена на практике, но требует определенной квалификации вычислителя, причем, если степени полиномов и/или порядок используемых чисел достаточно велики, то уже обычным рецептом — повторением вычислений с двойной точностью — не отделаешься. Мы еще вернемся к этому вопросу.

Быстрое преобразование Фурье (БПФ) — это алгоритм, позволяющий вычислить ДПФ, используя $O(n \log n)$ операций.

Какое все это имеет отношение к мобильным телефонам? Приведу небольшой комментарий, хотя уверен, что вам все это известно на несколько порядков лучше, чем мне. Как известно, на заре эпохи вычислительных машин цифровые и аналоговые устройства серьезно конкурировали, но в настоящее время остались лишь островки аналоговых вычислений[10]. Иначе говоря, любой непрерывный сигнал $\phi(t)$, будь то электромагнитный импульс, посылаемый вашим мобильником, давление в аэродинамичесой трубе, распределение цветов на экране вашего монитора и т.д. должен быть отцифрован (это, как все понимают, может быть очень даже нетривиальной задачей), т.е. представлен массивом коэффициентов его значений в достаточно большом количестве точек $f(t) \mapsto (a_0, a_1, \ldots, a_{n-1})$. Здесь t может быть временем, пространственной координатой и т.д. Предположим теперь, что моделируемое устройство, на которое поступает сигнал (модем, телефон,

[9] Алгоритм опубликовали в 1965 г. Д.Кули (James Cooley) и Д.Тьюки (John Tukey) — последний, между прочим, был консультантом по научным вопросам Джона Кеннеди,— но известно, что он применялся ранее и другими авторами. Как и следовало ожидать, его использовал "король математиков" К.-Ф. Гаусс в начале 19 века (записи были расшифрованы недавно).

[10] Хотя, с другой стороны, столь популярную в последнее время и даже проникшую в таблоиды модель квантовых вычислений, до которых мы, возможно, доберемся, можно рассматривать как явный реванш аналоговых вычислительных устройств. Если их удастся реализовать, то можно будет эффективно выполнять некоторые процедуры, весьма трудоемкие для обычных компьютеров, например факторизовать числа.

экран, система управления ракетой и т.д.) линейное[11] (грубо говоря, при изменении интенсивности вдвое воздействие также изменяется в два раза и выполняется принцип суперпозициии: реакция от суммы сигналов равна сумме реакций), тогда его поведение может быть описано как импульсная или весовая функция в технике, фундаментальное решение или функция Грина в математике и физике и пр. По определению, импульсная функция — это реакция системы на единичный импульс в нулевой момент $t = 0$, т.е. на δ-функцию. В нашей дискретной модели реакция тоже должна задаваться массивом $(b_0, b_1, \ldots, b_{n-1})$ (считаем, что после i тактов на выходе b_i, а через n тактов реакция затухает).

Итак, пусть в момент времени $T = 0$ на вход поступает сигнал с интенсивностью a_0, который преобразуется системой в $b_0 \cdot a_0$, т.е. отклик системы в момент $T = 0$ равен $a_0 b_0$. Далее, в момент $T = 1$ на вход поступает сигнал с интенсивностью a_1, вызывающий реакцию $b_0 a_1$, которую согласно принципу суперпозиции нужно просуммировать с *остаточным* воздействием сигнала, пришедшего в момент $T = 0$, т.е. $a_0 b_1$, и отклик системы равен $a_1 b_0 + a_0 b_1$ и т.д. Таким образом, при наших предположениях о поведении системы получаем, что если $T < n$ (т.е. сигнал еще поступает), то отклик равен $c_T = \sum_{i=0}^{T} a_i b_{T-i}$, а если $T \geq n$ (т.е. входной сигнал уже затух), то $c_T = \sum_{i=T-n+1}^{n-1} a_i b_{T-i}$. В момент $T = 2n-2$ отклик равен $c_{2n-2} = a_{n-1} b_{n-1}$, а далее отклик равен нулю. Таким образом, отклик системы $c_0, c_1, \ldots, c_{2n-2}$ в точности совпадает с последовательнстью коэффициенов многочлена- произведения:

$$\sum_{i=0}^{2n-2} c_i x^i \stackrel{def}{=} \sum_{i=0}^{n-1} a_i x^i \sum_{i=0}^{n-1} b_i x^i.$$

Итак, если мы хотим эффективно обрабатывать сигналы, то нужно научиться быстро умножать полиномы. Обычный способ умножения "столбиком" требует порядка n^2 операций, а алгоритм БПФ позволяет умножать многочлены почти на порядок быстрее. Но сначала разберемся, как, собственно, реализовать само это быстрое преобразование. Для этого мы используем симметрии корней из единицы. Заметим[12], что если n четное, то квадраты всех корней из единицы степени n образуют $\frac{n}{2}$ корней из единицы степени $\frac{n}{2}$ (можете ли вы сказать, сколько корней степени n перейдет в один корень степени $\frac{n}{2}$?). Теперь все готово для алгоритма. Попробуем применить технику "разделяй-и-властвуй". Итак, надо вычислить $f(\omega_n^k)$, $k = 0, \ldots, n-1$. Перепишем выражение по четным и нечетным индексам (считаем n четным) $f(\omega_n^k) = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_n^{2ik} + \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \omega_n^{2ik+k} = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} (\omega_n^2)^{ik} + \omega_n^k \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} (\omega_n^2)^{ik}$, и задача разбивается на две аналогичные задачи для многочленов степени $\frac{n}{2} - 1$, поскольку ω_n^2 — это корень из единицы степени $\frac{n}{2}$. Получаем следующую рекуррентность для трудоемкости алгоритма $T(n) = 2T(\frac{n}{2}) + O(n)$, откуда $T(n) = O(n \log n)$

Остается применить изложенную технику для умножения многочленов. Для этого нужно вспомнить, что произвольный многочлен степени $n-1$ может быть однозначно восстановлен по его значениям в произвольных n точках. Поэтому вместо того, чтобы пересчитывать коэффициенты многочлена $h(x) = f(x) \cdot g(x) = (\sum_{i=0}^{n-1} a_i x^i)(\sum_{i=0}^{n-1} b_i x^i) = \sum_{i=0}^{2n-2} c_i x^i = \sum_{i=0}^{2n-2} (\sum_{j=0}^{i} a_j \cdot b_{i-j}) x^i$ (прямое вычисление коэффициентов требует $O(n^2)$ операций) мы сначала вычислим значения многочленов $f(x)$ и $g(x)$ в некоторых $2n-1$ точках интерполяции, получив массивы $F = \{f_i\}$ и $G = \{g_i\}$. Покомпонентно перемножая F и G, получаем массив $H = \{f_i \cdot g_i\}$, который, по построению, отвечает значениям многочлена-произведения $h(\cdot)$ в тех же точках интерполяции, и $h(\cdot)$ можно однозначно восстановить и получить ответ.

Итак мы получили массив значений многочлена-произведения в корнях из единицы, и теперь нам нужно восстановить его коэффициенты. Конечно, если выбирать точки интерполяции совершенно произвольно, то не понятно, почему такой метод мог бы быть эффективнее прямого вычисления коэффициентов h. Однако, выбор точек интеполяции в корнях из единицы позволяет построить $O(n \log n)$-алгоритм перемножения многочленов! Точнее говоря, трудоемкость процедуры будет $O(n \log n \cdot M)$, где параметр M отвечает за трудоемкость отдельных арифметических операций. Конечно, если считать, что точность фиксирована, то M — константа, связанная с конкретной реализацией арифметических операций с плавающей точкой, и такая запись не очень осмыслена. Значит нужно попробовать реализовать точные вычисления, считая, как обычно, коэффициенты многочленов целыми. (Такая постановка реализуется во многих системах символьных вычислений или, например, в алгоритмах быстрого перемножения больших чисел, которые, как мы помним, как раз и используют точное БПФ в качестве подпрограмы.) Мы не будем сейчас останавливаться на этом подробно (в качестве частичных рецептов посмотрите упр. 32.2-6 и задачу 32.5 из Кормена (1-е издание, — во втором издании номер параграфа 30), а также задачу **Д-2** ниже.

А сейчас ответим на исходный вопрос, почему можно быстро за $O(n \log n)$ операций восстановить коэффициенты многочлена по его ДПФ. По определению, ДПФ задается умножением матрицы типа Вандермонда $\tilde{f} = \|(\omega_n)^{ij}\|_{i,j=0,1,\ldots,n-1}$ на вектор-столбец $(a_0, a_1, \ldots, a_{n-1})$. Действительно, для корней из единицы справедливо тождество: для любого k выполнено

$$\sum_{i=0}^{n-1} (\omega_n^k)^i = \delta_{0k \bmod n} \cdot n \tag{2}$$

(δ_{ij} — это символ Кронекера — дискретная δ-функция), поэтому обратную матрицу можно задать так:

$$\tilde{f}^{-1} = \frac{1}{n} \|(\omega_n^{-1})^{ij}\|_{i,j=0,1,\ldots,n-1} \tag{3}$$

и по аналогичной причине и обратное ДПФ (умножение матрицы $\tilde{f}^{-1}$ на массив коэффициентов) можно вычислить, используя $O(n \log n)$ операций.

Еще один момент, который я хотел бы затронуть в связи с БПФ, это его возможное нерекурсивное исполнение. Оказывается, что БПФ можно задать явной схемой, которая к тому же допускает параллельное выполнение (см. рис. 1).

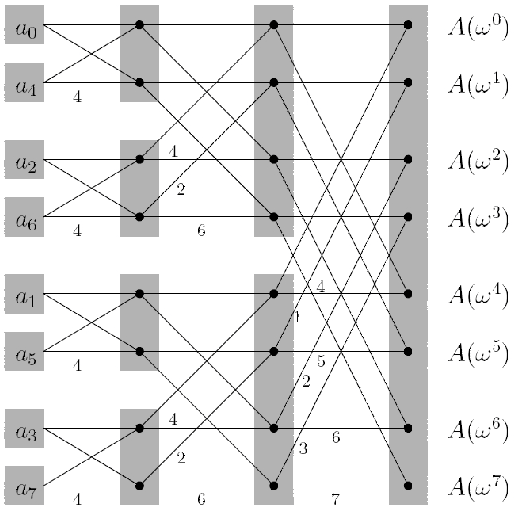


Рис. 1: Схема БПФ

На схеме ребра-провода "несут" информацию слева направо, т. е. каждому ребру приписано (комплексное) число. Метка i на ребре означает, что число, приписанное проводу, нужно умножить на ω^i. В вершинах, обозначенных черными кружками, нужно просуммировать числа, приписанные проводам, приходящим *слева*. По схеме видно, что ее основу составляет знаменитая "бабочка", отвечающая используемой в БПФ группировке степеней на четные и нечетные и переходу к половинной размерности:

$$s_i = t_i + \omega^i u_i, \qquad s_{i+n/2} = t_i - \omega^i u_i.$$

Зачем нужно писать явные схемы, когда уже построен рекурсивный алгоритм? Во-первых, рекурсия может быть неэффективной при реализации, а явная схема проще. Во-вторых, схема показывает, как запустить алгоритм параллельно. В третьих, используя

[11] Это почти общее предположение. Если же система нелинейная, то рассматривают ее линеаризацию.

[12] Хорошо бы действительно понять эти простые, но фундаментальные факты. Все они, конечно, доказываются в Кормене, но лучше попробовать получить их самому — это буквально одна строчка выкладок

структуру схемы можно решать полезные смежные задачи, например, искать некоторые ошибки, см., задачу **№ 44**.

Мы разобрались, как быстро перемножать многочлены с комплексными коэффициентами. А можно ли придумать похожий трюк для умножения многочленов с коэффициентами в кольце вычетов Z_n? Если бы это удалось, то при умножении многочленов с целыми коэффициентами можно было бы обойти упомянутый выше вопрос о точности вычислений в ДПФ, рассматривая достаточно большие модули и проводя операции в Z_n. Частичный ответ на этот вопрос будет дан в задаче **№ Д-13**.

Быстрое вычисление свертки посредством БПФ используется во многих областях, например, в такой важной области, как анализ изображений. Изображение можно считать числовой матрицей [большого] размера, а типичной задачей может быть выделение [сравнительно небольшого] блока с заданными свойствами (под блоком понимается не минор, а подматрица исходной матрицы, в которой столбцы и строки идут подряд). Интересно, что содержательна и одномерная задача — поиск подстрок, с которой успешно справляется произвольный текстовый редактор. Казалось бы, эту задачу мы успешно решили в курсе ТРЯП, построив **оптимальный** линейный КМП-алгоритм. Но оказывается, что примерно в то же время (в 1974 г.), когда были предложены и КМП, и Boyer-Moore алгоритмы, был предложен быстрый алгоритм, основанный на БПФ. Его современная версия изложена в задаче задания **№ 42**. Трудоемкость БПФ-алгоритма поиска подстрок отличается от оптимально логарифмическим фактором.

Идея БПФ-алгоритма поиска подстрок следующая. Нужно найти подстроку (образец) $p_0, \ldots, p_{m-1}$ в строке (тексте) $t_0, \ldots, t_{n-1}$, здесь p_i, t_j — это символы некоторого алфавита. Говорят, что подстрока входит с i-й позиции, если $p_j = t_{i+j}$, $j = 0, \ldots, m-1$. Если считать буквы алфавита различными целыми числами, то вхождение подстроки с i-й позиции эквивалентно обнулению суммы квадратов: $B_i = \sum_{j=0}^{m-1}(p_j - t_{i+j})^2 = \sum_{j=0}^{m-1}(p_j^2 - 2p_j t_{i+j} + t_{i+j}^2) = 0$, а вычисление массива чисел $\{B_i, i = 0, \ldots, n-m\}$ позволяет определить все места вхождения подстроки в текст.

"Наивный" алгоритм — прямой перебор требует $O(mn)$ операций. Но в курса ТРЯП изучался более быстрый, линейный, алгоритм. С помощью БПФ можно построить $O(n \log m)$-процедуру. Сумма $S = \sum\limits_{j=0}^{m-1} p_j^2$ присутствует как слагаемое в каждом B_i, и вычисляется за $O(n)$ шагов.

Рассмотрим два полинома степени не выше n: $T(x) = t_{n-1}x^{n-1} + \cdots + t_1 x + t_0$, $P(x) = p_0 x^{n-1} + \cdots + p_{m-1}$ Их произведение можно вычислить с помощью БПФ за $O(n \log n)$. Посмотрим на коэффициент их произведения $C(x)$ при x^{m-1+i} $(0 \le i \le n-m)$:

$$c_{m-1+i} = p_0 t_i + p_1 t_{i+1} + p_2 t_{i+2} + \cdots + p_{m-2} t_{m-2+i} + p_{m-1} t_{m-1+i} = \sum_{j=0}^{m-1} p_j t_{j+i}$$

Как видим, это одно из слагаемых для B_i. Таким образом, алгоритм для подсчета всех B_i таков. Сначала вычисляем за $O(n \log n)$ шагов коэффициенты произведения $C(x)$ указаных многочленов. Далее за максимум $O(n)$ шагов вычисляем сумму квадратов S и за $O(n)$ шагов считаем сумму первых m квадратов t_i, т. е. $H = \sum\limits_{j=0}^{m-1} t_j^2$. Далее вычисляем $B_0 = S - 2 \cdot c_{m-1} + H$ и $B_1 = S - 2 \cdot c_m + \sum\limits_{j=0}^{m-1} t_{j+1}^2 = B_1 + 2 \cdot c_{m-1} - 2 \cdot c_m - t_1^2 + t_m^2$.

Для этого нам потребуется $O(1)$ операций. Аналогично получим $B_i = B_{i-1} + 2(c_{m-2+i} - c_{m-1+i}) - t_{i-1}^2 + t_{m-1+i}^2$, т. е. для получения каждого следующего члена требуется $O(1)$ операций. Таким образом, для вычисления всех членов потребуется еще $O(n)$ операций. Таким образом, весь алгоритм асимптотически требует $O(n \log n)$ операций.

Ту же идею для проверки вхождения подстроки можно использовать, если разрешается использовать символ джокера (в курсе ТРЯП он обозначался знаком «?»). Тогда в тех же обозначениях нужно вычислить массив $\{A_i, i = 0, \ldots, n-m\}$, где $A_i = \sum_{j=0}^{m-1}(p_j^3 t_{i+j} - 2p_j^2 t_{i+j}^2 + p_j t_{i+j}^3)$, см. ниже задачу 2.

Наконец, еще одно важное приложение ДПФ связано с эффективным решением специальных систем линейных уравнений, в которых матрицы коэффициентов имеют специальные симметрии, а именно, постоянны вдоль каких-то "диагоналей" и тогда умножение на такую матрицу можно представить сверткой. В качестве примера таких матриц обычно рассматриваются циркулянты (строки получены циклическим сдвигом фиксированного вектора), теплицевы или ганкелевы матрицы, у которых на всех диагоналях, параллельных главной (соответственно, параллельных побочной), стоят равные элементы.

Вы должны знать, как, используя БПФ, решать системы линейных уравнений порядка $n \times n$ с циркулянтной матрицей, используя $o(n \log n)$ операций, и представлять себе, как выполнить эту задачу для случая теплицевых или ганкелевых матриц, используя $O(n^2)$ операций.

Задача 37. $(0.03 + 0.02)$. (i) Найдите произведение многочленов $A(x) = 3x + 2$ и $B(x) = x^2 + 1$, используя рекурсивный $O(n \log n)$-алгоритм БПФ.

Для этого нужно выбрать n, Найти БПФ обоих полиномов, затем вычислить ДПФ многочлена- произведения и на последнем шаге вычислить обратное ДПФ, т. е. умножить вектор-столбец, полученный в предыдущем пункте, на обратную матрицу ДПФ. Используя тождество (3), и при этом вычисление можно тоже использовать БПФ (каким образом?).

(ii) Вычислите **обратное ДПФ** массива (I — мнимая единица) $A = [10, 3I\sqrt{2} + 2 + 2I, 0, 3I\sqrt{2} + 2 - 2I, -2, -3I\sqrt{2} + 2 + 2I, 0, -3I\sqrt{2} + 2 - 2I]$, используя схему БПФ для $n = 8$ на рис. 1. Возможно, вычисление будет удобнее проводить символически для $\omega = \exp\left(\frac{\pi I}{4}\right)$.

Задача 38. $(0.01 + 0.02)$. (i) Постройте $O(n \log n)$-БПФ-алгоритм для поиска подстрок в тексте с "джокерами". (ii) Покажите, как понизить трудоемкость вашей процедуры до $O(n \log m)$.

Задача 39. (0.01). Используя ДПФ, найдите решение системы линейных уравнений $Cx = b$. где C — это циркулянтная матрица, порожденная вектором столбцом $(1, 2, 4, 8)^t$, а $b^t = (16, 8, 4, 2)$.

Задача Д–18. (0.03). Дан множество различных чисел $A \subseteq \{1, \ldots, m\}$. Рассмотрим множество $A+A$, образованное суммами элементов A. Докажите или опровергните существование процедур построения $A+A$, имеющих субквадратичную трудоемкость $o(m^2)$.

Задача 40. (0.02). (**[Кормен 1, упр. 32.3-4] или [Кормен 2, упр. 30.3-4]**. Предположим, что в схеме БПФ, изображенной на рис. 1, вышел из строя ровно один сумматор, причем он выдает число 0 независимо от входа. Сколько и каких последовательностей нужно подать на вход, чтобы идентифицировать дефектный сумматор?

ДПФ и БПФ в кольце вычетов Z_n

Сначала мы рассмотрим простой случай, когда кольцо — это простое поле GF_p и попробуем просто осуществить в нем ДПФ. Понятно, что роль корня из единицы должен играть первообразный корень, поскольку мы знаем, что определитель Вандермонда, составленный из степеней произвольного первоообразного корня, обладает всеми нужными нам свойствами, причем в арифметике Z_p, потому что речь идет опять о сумме конечной геометрической прогрессии, период которой делится на порядок данного элемента. Итак для того, чтобы перемножить многочлены можно взять

достаточно большое (насколько большое?) простое число p, перемножить многочлены $\pmod p$, используя ДПФ в конечном поле, а затем восстановить коэффициенты.

Задача 41. (0.03). Выберите подходящее простое число p и перемножьте многочлены $A(x)$ и $B(x)$ над конечным полем, а затем восстановите многочлен-произведение над Z.. Обоснуйте выбор p. Скажем, можно ли взять $p = 5$? Или следует выбрать $p = 7$ или больше?

В предыдущей задаче мы обосновали ДПФ над конечным полем, но можно ли провести трюк с БПФ? Об этом следующие задачи. Напоминаем, что, на самом деле, мы пытаемся уточнить предыдущую модель, когда предполагалось, что арифметические операции проводятся точно.

Предположим, что в Z_n есть примитивный корень ξ степени[13] 2^k. Неформально модуль n будет означать максимальную границу модулей коэффициентов, которые могут встретиться в сомножителях и в произведении, а степень произведения не должна превышать $l = 2^k - 1$. Степень двойки требуется для того, чтобы прошла рекурсия в алгоритме БПФ

Задача Д–19. $(0.01 + 0.01 + 3 \times 0.02)$.

(i) Покажите, что 3 является примитивным корнем 8-й степени в простом поле Z_{41}.

Рассмотрим многочлен $h \in \mathrm{Z}_n[x]$ степени не выше $l = 2^k - 1$. Матрицу $\Xi = \|\xi^{ij}\|_{i,j=0,1,\ldots,l}$ назовем матрицей ДПФ. По определению, ДПФ многочлена h равно произведению матрицы Ξ на вектор-столбец коэффициентов h

(ii) Покажите, что для любого $j = 0, 1, \ldots, k-1$ элемент ξ^{2^j} — есть примитивный корень степени 2^{k-j} в Z_n.

Из последнего упражнения вытекает, что можно быстро вычислять ДПФ произвольного многочлена h степени не выше l в Z_n ровно таким же рекурсивным алгоритмом, который мы использовали в C.

(iii) Считая, что многочлены A и B (из задания) заданы над Z_{41} и $\xi = 3$ вычислите рекурсивным алгоритмом их ДПФ.

Перемножая покомпонентно вычисленные в предыдущем пункте ДПФ многочленов A и B, мы получаем ДПФ произведения C. Теперь нам осталось проверить, что можно эффективно восстановить коэффициенты C по его ДПФ. Для этого нужно проверить выполнения равенства, аналогичного (3).

(iv) Докажите, что $\Xi^{-1} = (l+1)^{-1}\|(\xi^{-1})^{ij}\|_{i,j=0,1,\ldots,l}$. Формула, конечно, справедлива для произвольной степени примитивного элемента ξ, а не только для степени 2^k, как в нашей спецификации.

(v) Используя результаты (iii) и (iv), вычислите ДПФ многочлена C и восстановите его коэффициенты рекурсивным алгоритмом БПФ [умножения обратной матрицы Ξ^{-1} на вектор столбец ДПФ].

Конечно, выбранный нами размер поля может оказаться недостаточным для того, чтобы произвести вычисления, поэтому дополнительно приведите обоснование корректности (или некорректности) выбора размера поля (само вычисление должно быть выполнено в любом случае).

Задание на 9-ю неделю: 10.04–16.04. [0.12] Сортировка. Раздел 9 программы Литература: [Кормен 1], [Кормен 2, гл 2], [ДПВ]

Разрешающие деревья и нижние оценки сортировки.

[13] Это, по определению значит, что $\xi^i \neq 1$, $i = 0, 1, \ldots 2^k - 1$, а $\xi^{2^k} = 1$.

Обсудим вопрос о минимальном числе T_{min} попарных сравнений, необходимых для нахождения минимального из n чисел. Для этой задачи алгоритм очевиден: нужно последовательно сравнивать числа, оставляя при каждом сравнении минимальное. Возникает правдоподобная гипотеза, что $T_{min} = n - 1$. Заметим, что даже в столь простой задаче ответ не очевиден, в частности, не проходит традиционный аргумент "по размеру входа", поскольку в $\frac{n}{2}$ сравнениях могут участвовать все числа, и речь фактически идет о том, какую часть информации о числах можно при сравнении передать. Рассмотрим два подхода к получению нижних оценок подобного рода.

Первый подход связан с понятием разрешающего дерева для алгоритмов сортировки **[Кормен 1 §9.1], [Кормен 2 §8.1]**. Напомним, что произвольный алгоритм A сортировки массива из n чисел $\{a_1, \ldots, a_n\}$ посредством попарных сравнений можно следующим образом изобразить в виде корневого двоичного дерева D_A. Каждая внутренняя вершина v дерева помечена некоторым сравнением $a_i ? a_j$, а в паре выходящих из v ребер одно ребро имеет пометку $\leq$, а другое $\geq$. Листья D_A помечены соответствующими перестановками $\{\pi_1, \ldots, \pi_n\}$, которые упорядочивают массив. Каждому конкретному входу $\{a_1, \ldots, a_n\}$ отвечает его **реализация** — путь от корня к листу в D_A.

Совершенно аналогично дается определение разрешающего дерева для задачи поиска минимального элемента, поиска медианы и т. д. (все сводится к изменению пометок листьев). Мы сохраним для этих «специализированных» деревьев обозначение D_A. На языке разрешающих деревьев утверждение о том, что $T_{min} = n - 1$, эквивалентно следующему: в любом корректном алгоритме поиска минимального элемента в массиве из n чисел, использующем только попарные сравнения, каждый реализуемый путь от корня к листу имеет не менее $(n-1)$-го ребра.

Назовем это **утверждением** $\mathcal{A}$.

Произвольному корректному алгоритму A нахождения минимума попарными сравнениями и произвольному реализуемому пути P в разрешающем дереве D_A отвечает (неориентированный) граф $G_A^P = (V, E)$ на n вершинах, в котором есть ребро $(v_i, v_j) \in E$, если и только если в пути P какая-то вершина имеет пометку $a_i ? a_j$.

Утверждение $\mathcal{B}$. Для корректности алгоритма A нахождения минимума необходимо, чтобы граф G_A^P был связен.

Задача 42. $(2 \times 0.01 + 0.02)$ (i) Докажите импликацию: $\mathcal{B} \Rightarrow \mathcal{A}$.

(ii) Докажите утверждение $\mathcal{B}$.

Теперь дадим другое доказательство этой нижней оценки. Отметим, что само доказательство будет иллюстрацией методов **аммортизационного анализа** для получения нижних оценок.

Для этого запишем шаги алгоритма в формате конфигураций (a, b, c, d), где a элементов пока не сравнивались, b элементов были больше во всех сравнениях, c элементов были меньше во всех сравнениях, d были и больше, и меньше в сравнениях, т. е. начальная конфигурация такова: $Init = (n, 0, 0, 0)$. Введем "потенциальную функцию", определенную на конфигурациях: $f[(a, b, c, d)] = a + c$. Мы оценим трудоемкость алгоритма, просто поделив "разность потенциалов" между начальной и конечной конфигурациями на максимальное изменения потенциала за один шаг алгоритма.

(iii) Покажите, что при любом сравнении потенциал $f(\cdot)$ может уменьшиться не больше, чем на единицу, и что отсюда вытекает, что число шагов любого такого алгоритма не меньше $n - 1$.

Покажем, что любой алгоритм нахождения медианы массива из n элементов посредством попарных сравнений имеет сложность $T(n) = \frac{3n}{2} - O(\log n)$.

Задача 43. $(0.02 + 0.03)$ (i) Покажите, что любое разрешающее дерево поиска медианы позволяет также восстановить индексы всех элементов, бóльших медианы, и всех элементов, мéньших медианы.

Из этой задачи вытекает, что нахождение медианы эквивалентно с виду более сложной задаче: найти медиану и массив L элементов, бóльших ее $(\frac{n}{2}-1)$ элементов.

(ii) Покажите, что любое разрешающее дерево для медианы содержит путь от корня к листу длины $\frac{3n}{2}-O(\log n)$.

Комментарий. Можно использовать два соображения. Во-первых, если из дерева T для медианы выкинуть все сравнения, в которых участвуют элементы L, то получится дерево T_L поиска максимума (в нем максимум — это медиана). А из предыдущей задачи следует, что T_L должно иметь $\geq 2^{\frac{n}{2}-1}$ листьев. Во-вторых, массив L может быть произвольным, а отсюда можно получить оценку снизу на число листьев (и на высоту) T.

Наилучшие известные современные оценки: $(2+\varepsilon)n \leq T(n) \leq 2.95n$.

Задача Д–20. (0.03) **[Кормен 1, задача 10-1-2]** Рассмотрим стандартную рекурсивную процедуру одновременного поиска максимума и минимума **[Кормен 1, §10.1]**. Покажите, используя подходящую потенциальную функцию, что этот алгоритм является оптимальным по числу использованных сравнений.

Комментарий. Здесь начальная и конечная конфигурации таковы: $Init = (n, 0, 0, 0)$, $Final = (0, 1, 1, n-2)$. Нужно показать, что необходимо не менее $k = \lceil \frac{3n}{2} \rceil - 2$ сравнений. В тексте можно использовать только аргументы, относящиеся к потенциальной функции. Нельзя апеллировать к авторскому представлению о том, что какие-то действия "неоптимальны". Формат ответа, как и для рассмотренного выше выбора минимального элемента, должен быть таков.

1. Потенциальную функцию $f(\cdot) = f(a, b, c, d)$ следует указать явно.
2. При любом сравнении значения $f(\cdot)$ не могут уменьшиться больше, чем на некоторую величину δ (скажем, единицу).
3. $f(Init) - k\delta = f(Final)$.
4. На самом деле, легко проверить, что в классе линейных функций такая потенциальная функция не существует, поэтому **нужно либо усложнить вид функции, либо считать, что функция определена только на части входов**.

Задача Д–21. $(0.03 + 0.03)$ (i) Дано n ключей и n замков. Все ключи и все замки различны между собой, а каждый ключ подходит к единственному замку. Ключи (и замочные скважины) упорядочены по величине, но визуально отличия неразличимы. На каждом шаге можно попытаться вставить конкретный ключ в конкретный замок и заключить, что он подходит или больше, или меньше искомого. Постройте вероятностный алгоритм подбора ключей, требующий в среднем $o(n^2)$ шагов.

Комментарий. Очевидно, что прямой перебор подходящих пар ключей и замков требует квадратичного числа шагов. Удивительным кажется то, что для этой задачи построен детерминированный $o(n^2)$-алгоритм. Он очень хитрый.

(ii) Покажите, что любая детерминированная процедура подбора ключей требует $\Omega(n \log n)$ шагов.

Структуры данных для динамической сортировки Манипуляции с деревьями

Литература: **[Кормен 1, гл. 18–19] или [Кормен 2, гл. 17–18]**, **[АХУ]**, **[Ш]**.

Изучим несколько структур данных древесного типа, которые позволяют эффективно динамически обращаться с упорядоченными массивами (т. е. нужно динамически поддерживать упорядоченный массив, считая, что элементы могут, например, добавляться и/или удаляться и пр.). Впервые подобную структуру данных, позволяющую выполнять удаления/вставки в n элементном массиве за $O(\log n)$ операций, предложили в 1963 году Г. Адельсон-Вельский и Е. Ландис[14]. Мы начнем изучение этой темы и рассмотрим несколько структур данных, позволяющих эффективно поддерживать динамические операции. При анализе трудоемкости опять очень полезным оказывается метод т.н. *амортизационного анализа*, т. е. вычисление трудоемкости не отдельной, а сразу целой серии операций.

Пусть x — двоичное дерева поиска[15]; обозначим $size[x]$ число ключей в поддереве с вершиной x. Выделим для $size[\cdot]$ поле в каждой вершине дерева. Пусть α — число и $1/2 \leq \alpha < 1$. Будем говорить, что вершина x дерева, не являющаяся листом, *α-сбалансирована*, если $size[[left[x]] \leq \alpha size[x]$ и $size[[right[x]] \leq \alpha size[x]$ ($left$ и $right$ — это указатели на левого и, соответственно, правого потомка x в двоичном дереве). Дерево называется *α-сбалансированным*, если все его внутренние вершины α-сбалансированы.

Задача 44. $(0.02 + 0.02)$ (i) Покажите из определения, что для *любой* вершины x 1/2-сбалансированного дерева выполнено:

$$size[[left[x]] - size[[right[x]] \in \{-1, 0, +1\}$$

.

(ii) **[Кормен 1, задача 18.3 (б)]или [Кормен 2, задача17.3 (б)]** Покажите, что поиск элемента в α-сбалансированном двоичном дереве с n вершинами выполняется за $O(\log n)$.

В следующей задаче будет показано, что α-сбалансированные деревья можно эффективно динамически балансировать, т. е. осуществлять вставку и удалению можно за *учетное время* $O(\log n)$. О том, что такое учетное время тоже немного говорилось и ранее, но сейчас необходимо обязательно прочитать в **Кормене** начало главы 18 (I) (или 17 (II)).

Мы снова используем *метод потенциалов*.

Задача Д–22. $(0.02 + 0.01 + 0.03 + 0.03)$. **[Кормен 1, задача 18.3 (б)]или [Кормен 2, задача17.3 (б)]**

(i) Пусть x — вершина двоичного дерева поиска. Постройте алгоритм, использующий время $\Theta(size[x])$ и дополнительную память $O(size[x])$, для преобразования поддерева с корнем x в 1/2-сбалансированное дерево.

Далее считаем, что $\alpha > 1/2$, и после выполненных стандартным способом[16] операций удаления или вставки, следующим образом производится балансировка: выбирается самая высокая вершина результирующего дерева, которая перестала быть α-сбалансированной и все ее корневое поддерево перестраивается в 1/2-сбалансированное посредством алгоритма из пункта (i).

Используем метод потенциалов с потенциальной функцией:

$$\Phi(T) = c \sum_{x \in T\,:\,|\Delta(x)| \geq 2} |\Delta(x)|,$$

здесь T — двоичное дерево поиска, а $c > 0$ — достаточно большая константа, зависящая от α.

(ii) Покажите, что потенциал 1/2-сбалансированного дерева равен нулю.

[14] Отсюда и название самой структуры по первым буквам фамилий аторов — АВЛ-дерево.

[15] Что это такое?

[16] Контрольный вопрос: каким это таким стандартным способом?

(*iii*) Считаем, что реальная стоимость [в единицах потенциала] описанного в пункте (*i*) преобразования не α-сбалансированного поддерева T с m вершинами в 1/2-сбалансированное дерево равна m. Какой нужно выбрать константу c в зависимости от α, чтобы учетная стоимость такого преобразования T равнялась $O(1)$?

(*iv*) Покажите, что учетная стоимость удаления или вставки элемента в α-сбалансированное дерево с n вершинами равна $O(\log n)$.

Из предыдущей задачи вытекает, что для α-сбалансированных деревьев удаление или вставка выполняются эффективно в смысле учетной стоимости. Но можно построить другие более сложные древесные структуры данных, в которых эти операции теоретически выполняются быстрее. Например, в т.н. *фибоначчиевых* кучах **[Кормен 1, гл. 21] или [Кормен 2, гл 20]** удалить или вставить элемент удается с учетной стоимостью $O(1)$. Но, как и всегда, платой является более сложная организация программы. В ранее упомянутых АВЛ-деревьях[17] добавление требует $O(1)$, а удаление — $O(\log n)$ операций, но **в наихудшем случае**. В определенном смысле и фибоначчиевы кучи, и АВЛ-деревья входят в теормининум по крайней мере для физтехов, претендующих на высокую оценку. Амортизационный анализ фибоначчиевых деревьев требует достаточно тонких аргументов, и его полезно посмотреть для закрепления изложенного материала.

Задание на 10-ю неделю: 17.04–23.04. [0.15] Потоки и разрезы. Раздел 10 программы Литература: [Кормен 1], [Кормен 2, гл 26], [ДПВ]

Повторение: анализ сложности алгоритма Quicksort

Рассмотрим алгоритм быстрой сортировки с каким-нибудь детерминированным выбором "барьерного элемента". Обозначим через $\mathrm{t}(A_n)$ время работы алгоритма на массиве A_n длины n. По определению, средним временем работы алгоритма называется величина

$$\mathbb{E}\mathrm{t}(\mathrm{n}) = \frac{1}{n!}\sum_{A_n} \mathrm{t}(A_n). \qquad (4)$$

На эту формулу можно взглянуть и немного по-другому. *Если считать равновероятными все $n!$ возможных способов упорядочения входного массива длины n*, то среднее время работы алгоритма, заданное формулой выше,— это, опять по определению, *математическое ожидание* времени работы алгоритма. Определим рекурсивную процедуру сортировки типа Quicksort, которую мы назовем Permsort следующим образом. *Сначала алгоритм случайным образом переставляет элементы текущего массива, причем таким образом, чтобы все возможные перестановки были равновероятны*[18], *а потом сортирует массив, выбирая барьерный элемент каким-то стандартным способом, например, используя самый правый.*

Задача 45. (2×0.02) (*i*) Покажите, что для среднего времени работы алгоритма Permsort справедливо рекуррентное соотношение.

$$\mathbb{E}\mathrm{t}(n) = \Theta(n) + \frac{1}{n}(\mathbb{E}\mathrm{t}(0)+\mathbb{E}\mathrm{t}(n-1)) + \frac{1}{n}(\mathbb{E}\mathrm{t}(1)+\mathbb{E}\mathrm{t}(n-2)) + \ldots$$
$$+ \frac{1}{n}(\mathbb{E}\mathrm{t}(n-2)+\mathbb{E}\mathrm{t}(1)) + \frac{1}{n}(\mathbb{E}\mathrm{t}(n-1)+\mathbb{E}\mathrm{t}(0)) =$$
$$= \Theta(n) + \frac{2}{n}(\mathbb{E}\mathrm{t}(1)+\mathbb{E}\mathrm{t}(1)+\ldots+\mathbb{E}\mathrm{t}(n-1)). \qquad (5)$$

Комментарий. (Возможный вариант решения.) Считаем, что время работы алгоритма на конкретном входе пропорционально числу проведенных сравнений[19]. *Нужно понять, что, по определению, в* (4) *записана сумма средних значений, т. е. сумма сумм. Теперь формулу* (5) *можно получить, изменяя порядок суммирования.*

(*ii*) Используя эту рекурренту, покажите, что $\mathbb{E}\mathrm{t}(n) = \Theta(n \log n)$.

В стандартном алгоритме Quicksort с рандомизацией при каждом (рекурсивном) обращении барьерный элемент выбирается случайным образом, причем равновероятно. Во всех изданиях **[Кормен]** проводится подробный анализ среднего времени работы алгоритма, но в **[Кормен 1]** описан специальный прием, который и будет проанализирован в следующей задаче.

Переопределим выбор случайного барьерного элемента следующим образом Будем считать, что изначально каждому элементу приписывается **ранг** — целое число от 1 до n. А выбор барьерного элемента будем проводить, выбирая каждый раз из текущего массива элемент "с минимальным рангом"[20].

Приведем цитату из **[Кормен 1]**: "Можно проверить, что это равносильно независимым выборам элементов на каждом шаге: *на первом шаге каждый из элементов может быть выбран с равной вероятностью, после такого выбора в каждой из групп все элементы также равновероятны и т.д.*"

Задача 46. (0.02) Обоснуйте эквивалентность обоих способов выбора, т. е. проверьте корректность утверждения выделенного курсивом в цитате выше.

Используя "ранги", легко вывести, что в алгоритме Quicksort элементы, которые после упорядочения попадают на i-е и j-е место, соответственно ($i \neq j$), сравниваются в $p_{ij} = \frac{2}{|i-j|+1}$-й доле случаев, что сразу дает оценку для трудоемкости в среднем: $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} p_{ij} = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = O(n\log n)$.

Задача 47. (0.01) Покажите, что если в любой модификации алгоритме Quicksort в качестве барьерного элемента использовать медиану текущего массива, причем искать ее посредством стандартного линейного алгоритма, то его сложность по наихудшему случаю станет $O(n \log n)$.

Потоки

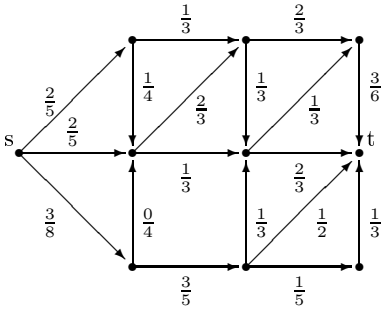
Задача 48. ($3 \times 0.01 + 0.02 + 0.01$) На рисунке изображен потоковый граф (метка $\frac{f}{u}$ на ребре означает поток и пропускную способность, соответственно).

[17] Как обычно, рекомендую посмотреть книгу **[Ш]**. Она имеется в открытом доступе на сайте www.mccme.ru, и, хотя автор декларирует, что она написана для школьников, но объем, ясность и глубина изложенного в ней материала значительно превосходит стандарты изучения информатики в вузах.

[18] О том, что под этим выражением понимается формально, и как реализовать такую процедуру за линейное время, написано в **[Кормен 2, §5.3]**.

[19] В изданиях **[Кормен]** выше второго аналогичный факт явно формулируется и доказывается для стандартного алгоритма Quicksort.

[20] Трюк этот совсем нетривиальный: вместо того, чтобы каждый раз вызывать подпрограмму RANDOM, вы генерируете случайную перестановку и в дальнейшем используете только ее.



(*i*) Чему равен поток f?

(*ii*) Изобразите остаточный граф, соответствующий потоку f.

(*iii*) Максимален ли поток f?

В следующих двух пунктах нужно по шагам применить метод[21] Форда-Фалкерсона, доведя его до алгоритма. **При отсутствии алгоритма (например, если увеличивающие пути находятся методом "внимательного рассмотрения" потокового графа или если разрез просто отгадывается) задача не оценивается. *Это требование будет тем более актуально при написании тестов.***

Метод остаточных графов из книг **[Кормен 1]** или **[Кормен 2]** аналогичен оригинальному методу пометок Форда- Фалкерсона, изложенному в их книге

(*iv*) С помощью алгоритма Форда-Фалкерсона по шагам найдите максимальный поток. На каждом шаге должен быть построен остаточный граф и указан увеличивающий путь.

(*v*) Укажите модификацию алгоритма Форда-Фалкерсона для нахождения минимального разреза. По шагам постройте минимальный разрез между s и t. Найдите его пропускную способность.

Задача 49. (0.02 + 0.01) В больнице каждому из 169 пациентов нужно перелить по *одной дозе* крови. В наличии имеется 170 доз. Распределение по группам таково.

Группа	I	II	III	IV
В наличии	45	32	38	55
Запрос	42	39	38	50

При этом пациенты, имеющие кровь группы I, могут получать только кровь группы I. Пациенты, имеющие кровь группы II (группы III), могут получать только кровь групп I и II (групп I и III, соответственно). Наконец, пациенты с IV группой могут получать кровь любой группы.

(*i*) Распределите дозы, чтобы обслужить максимальное число пациентов с помощью *решения подходящей задачи о максимальном потоке*. Решение нужно аккуратно оформить: должна быть нарисована потоковая сеть и показаны все шаги алгоритма ФФ, начиная с нулевого потока, т. е. должны быть построены остаточные графы и показаны увеличивающие пути.

(*ii*) Если всех пациентов обслужить нельзя, то приведите *простое* объяснение этому, *доступное администрации* больницы.

Задача 50. (0.01) Покажите на примере конкретной сети, что алгоритм Форда-Фалкерсона не является полиномиальным.

[21] Отметим, что выражение "метод" употребляется не случайно (некоторые этапы описаны неявно или подразумеваются). **Вы должны самостоятельно придумать, как дополнить процедуру до алгоритма.**

Рассмотрим следующую задачу Сеть. Дан ориентированный граф $G = (V, E)$, дугам которого приписаны неотрицательные числа $l_i \le u_i, i \in E$. Нужно проверить, можно ли приписать ребрам числа $F = \{f_i, i \in E\}$, чтобы в любой вершине v была нулевая дивергенция $div\, F = \sum_{\text{по входящим в v ребрам}} f_i - \sum_{\text{по выходящим из v ребрам}} f_j = 0$ и выполнялись неравенства $l_i \le f_i \le u_i, i \in E$.

Задача 51. (0.01 + 0.02) Покажите, как можно решить задачу Сеть с помощью решения подходящей задачи о максимальном потоке в сети и наоборот.

Задача 52. (0.01) Рассмотрим следующую задачу. В потоковой сети нет ограничений пропускной способности на дугах, но есть ограничения пропускной способности вершин. Формально, для каждой вершины v, отличной от истока и стока, задано целое неотрицательное число $c(v)$, и для потока в сети должно выполняться

$$\sum_u f(u, v) = \sum_u f(v, u) \le c(v).$$

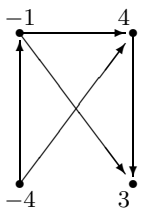
Опишите алгоритм нахождения максимального потока в такой сети.

Задача Д–23. (0.02) Задан двудольный неориентированный граф, в котором обе доли имеют n вершин, а степени (количество инцидентных рёбер) всех вершин равны d, **т.е. однородный двудольный граф степени** d. Приведите полиномиальный алгоритм, который раскрасит рёбра в d цветов так, чтобы из каждой вершины исходили рёбра разных цветов. Оцените сложность предложенного алгоритма.

Задача Д–24. (0.02) На вход задачи подаётся ориентированный граф $G = \langle V, E\rangle$ без контуров (ориентированных циклов). Необходимо покрыть его наименьшим числом простых путей, т. е. найти наименьшее количество не пересекающихся по вершинам простых путей, чтобы каждая вершина принадлежала одному из них. Допускаются пути нулевой длины (состоящие из одной вершины). Предложите полиномиальный алгоритм.

Задание на 11-ю неделю: 24.04–30.04. [0.15] Потоки и разрезы. Алгоритмы на графах Разделы 10–11 программы Литература: [Кормен 1], [Кормен 2, раздел 6], [ДПВ]

Задача 53. (0.02 + 0.01 + 0.02) Последовательность выполнения проектов задана ациклическим орграфом $G = (V, E)$ (если в орграфе есть ребро (u, v), то проект v не может начаться, пока не будет выполнен проект u). Выполнение проекта v приносит прибыль $p(v)$ (она может быть и отрицательна). Требуется выбрать подмножество проектов, приносящих максимальную суммарную прибыль, т.е. найти такое подмножество проектов $M \subseteq V$, что $M = argmax_{S \subseteq V}\{p(S) \stackrel{def}{=} \sum_{v \in S} p(v)\}$.



Оказывается, что эту задачу можно свести к задаче о минимальном разрезе. **Конструкция.** *Дополняем граф G источником s и стоком t, и задаем* **бесконечные** *пропускные способности на ребрах G. Далее, для всех вершин $v \in V$, если $p(v) < 0$, то задаем ребро (s, v) с пропускной способностью $-p(v)$, а если $p(v) > 0$, то задаем ребро (v, t) с пропускной способностью $p(v)$.*

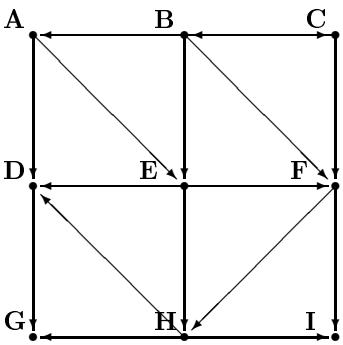
(*i*) С помощью алгоритма Форда-Фалкерсона найдите максимальный поток в полученной сети. Начальный поток нулевой. Приведите подробное описание: на каждом шаге процедуры на вспомогательных чертежах изобразите остаточные графы и укажите увеличивающие пути.
(*ii*) Затем, используя алгоритм Форда-Фалкерсона, найдите минимальный разрез.
(*iii*) Обоснуйте конструкцию в общем случае.

Поиск в глубину и поиск в ширину

Комментарий (особенно он предназначен для студентов, которые знают — или считают, что знают, — что такое поиск в ширину или в глубину). *Здесь мы изучаем качественные свойства указанных процедур, чтобы понять какую информацию о графе мы дополнительно получаем, когда запускаем один из этих естественных и с виду совершенно бесхитростных обходов вершин. В качестве простейшего контрольного вопроса можно спросить, почему обе процедуры* **линейные по входу**, *т. е. линейные по длине естественной кодировки графа: числу вершин и ребер. Вы должны быть аккуратны с ответом, например, уже потому, что обе процедуры предусматривают возвраты, так что каждая вершина или ребро может просматриваться не один раз.*

Если специально не оговорено, то рассматриваются графы без петель и кратных ребер (простые).

Задача 54. (0.02) Проведите поиск в глубину в графе на рисунке.



Используйте алфавитный порядок вершин. Укажите типы всех дуг графа и вычислите для каждой вершины значение функций $d(\cdot)$ и $f(\cdot)$.

Согласно Теореме 23.4 из книги **[Кормен 1]**, процедура *поиска в ширину*, начиная с данной вершины s, в графе[22] присваивает просматриваемым вершинам отметки, равные *кратчайшему пути (длина = число ребер)* от s.

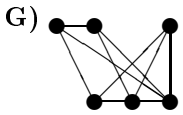
Задача 55. (0.02 + 0.01) (*i*) Докажите или опровергните, что следующее условие дает критерий, когда остовное дерево $F \subseteq G$ является *деревом некоторого поиска в ширину* связного неориентированного графа G.

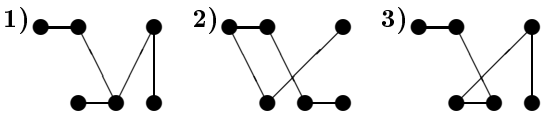
Остовное дерево $T \subseteq G$ является **деревом некоторого поиска в ширину** *связного неориентированного графа G, если и только если в нем можно выбрать одну из вершин s за корень так, чтобы T было деревом кратчайших путей из s в графе G. Иными словами, путь по дереву из s в произвольную вершину t содержит не больше ребер, чем кратчайший путь между s и t в G.*

Если в настоящем виде критерий неверен, то модифицируйте его до корректного.
(*ii*) В соответствии с полученным в предыдущем пункте критерием установите, какие из нарисованных деревьев являются деревьями поиска в ширину.

[22] Или в ориентированном графе.

Формат ответа. Пусть, скажем, критерий верен, тогда при положительном ответе нужно указать корень дерева кратчайших путей, а при отрицательном — для каждого возможного выбора корня нужно указать вершину, расстояние которой до корня в графе меньше, чем соответствующее расстояние по дереву.





Связность

Связностью или вершинной связностью $\kappa(G)$ неориентированного графа G называется наименьшее число вершин, удаление которых превращает граф в несвязный или тривиальный. **Реберной связностью** $\lambda(G)$ графаG называется наименьшее число ребер удаление которых превращает граф в несвязный или тривиальный.

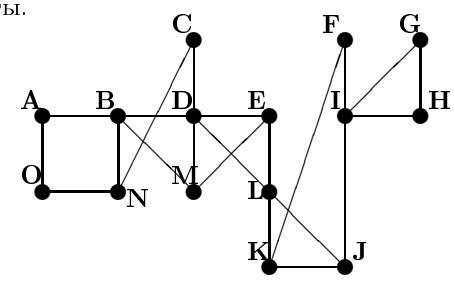
Максимальный по включению k-(реберно) связный подграф графа G называется его k- компонентой (соответственно, k-реберной компонентой). Обычно предполагается, что k-компонента имеет не менее $k+1$ вершин.

Задача 56. (0.01 балла) Покажите, что для любого G $\kappa(G) \leq \lambda(G) \leq \delta(G)$ ($\delta(G)$ — это минимальная степень вершин G).

Задача 57. (2×0.02 баллов) Постройте полиномиальный алгоритм или покажите NP-полноту проверки (*i*) k-связности и проверки (*ii*) k-реберной связности графа (k — двоичное число).

Точка раздела связного неориентированного графа G — это вершина, удаление которой делает граф несвязным. **Мост** — это ребро с аналогичным свойством. **Двусвязная компонента** связного графа содержит ≥ 3 вершин (или ≥ 2 ребер) и состоит из максимального набора ребер, в котором каждая пара ребер принадлежит общему простому (несамопересекающемуся) циклу.

Задача 58. (0.01) Для графа, изображенного на рисунке, укажите точки раздела, мосты и двусвязные компоненты.



Сильная связность

Задача Д–25. (2×0.02) Дана выполнимая 2-КНФ φ, каждый дизъюнкт которой содержит ровно два различных литерала (литерал и его отрицание считаются различными). Будем говорить, что φ 1-*минимальна*, если к

ней можно добавить один дизъюнт, содержащий два различных литерала так, чтобы она стала невыполнимой.
(*i*) Докажите или опровергните, что следующеее условие является критерием 1-минимальности.

Рассмотрим ориентированный граф G_φ, в котором литералы и их отрицания являются вершинами, а каждый дизъюнкт порождает пару ребер вида: $x \vee y \Rightarrow [e_1 = (\neg x, y), e_2 = (\neg y, x)]$.

φ является 1-минимальной тогда и только тогда, когда в G_φ есть путь P, соединяющий противоположные литеральные вершины, $x \rightsquigarrow y$, $x = \neg y$ и имеется ребро, ведущее из вершины y в вершину $z \notin P$.

Если в указанном виде критерий не верен, то дополните его до корректного.
(*ii*) Постройте для задачи проверки 1-минимальности как можно более быстрый полиномиальный алгоритм.
Подсказка. Полезно вспомнить, полиномиальные алгоритмы проверки выпоолнимости 2-КНФ.

Задача Д–26. (0.03) Постройте линейный по входу алгоритм, который, имея на входе граф G и некоторое его остовное дерево T, определяют, является ли T деревом поиска-в-ширину при старте с некоторой вершины G.

Задача Д–27. ($2\times0.01+0.02+0.01+2\times0.02$) **Линейный алгоритм разбиения графа на двухсвязные компоненты**
(*i*) Покажите, что множества вершин, принадлежащие двум разным двусвязным компонентам, либо не пересекаются, либо имеют единственную общую вершину — точку раздела.

Построим по G новый граф G_b, в котором имеются вершины двух типов: v_a, отвечающие точкам раздела G, и v_b, отвечающие двусвязным компонентам G. Ребра G_b соединяют каждую вершину v_b со всеми вершинами v_a, попадающими в двусвязную компоненту, отвечающую v_b.
(*ii*) Покажите, что G_b — дерево, и постройте соответствующее дерево для G из задачи **№ 62**.

Оказывается, что точки раздела можно находить по дереву поиска в глубину. Затем, опять используя поиск в глубину, можно определить все двусвязные компоненты, т. е. двусвязные компоненты можно находить за линейное время. Мы ограничимся только алгоритмом выделения точек раздела графа.
(*iii*) Докажите, что корень дерева поиска в глубину является точкой раздела тогда и только тогда, когда у него больше одного потомка.
(*iv*) Постройте контрпример к следующему утверждению из книги **[Кормен 1, задача № 23-2 (б)]**: отличная от корня вершина v дерева поиска в глубину является точкой раздела, если и только если в дереве поиска в глубину не существует обратного ребра от потомка v (включая саму v) до собственного предка v (т. е. отличного от самой v).
(*v*) **[Кормен 1, упр. 23-2(в)]**.

Определим функцию $low(v) = \min[d(v), d(w)$, *если для некоторого потомка и вершины v в G есть обратное ребро* $(u, w)]$.

Покажите, как вычислить $low(\cdot)$ за время $O(|E|)$ [например, модифицируя поиск в глубину].

(*vi*) Покажите, как в линейное время вычислить все двусвязные компоненты графа[23]

[23] **Подсказка.** Сначала, используя решение предыдущих задач, покажите, как с помощью поиска в глубину идентифицировать все точки раздела за линейное время. Этот факт и свойства функции $low(\cdot)$ позволяют находить двусвязные компоненты при поиске в глубину, например, используя дополнительный стек. Можно также находить двусвязные компоненты другим способом, выделяя мосты графа (см. **[Кормен I, упр. 23-2(д)–(з)]**).

Задание на 12-ю неделю: 1.05–7.05. [0.15]
Алгоритмы на графах II
Разделы 10–11 программы
Литература: [Кормен 1], [Кормен 2], [ДПВ], [АХУ]
Алгоритмы на графах II

k-связность графов
Краткий конспект

В этом конспекте сформулированы утверждения, которые в принципе можно доказать, используя потоки в сетях.

Сначала мы будем рассматривать только неориентированные графы.

Вершинной (соответственно, реберной) связностью $\kappa(G)$ (соответственно, реберной $\lambda(G)$) называется наименьшее число вершин (ребер), удаление которых приводит к несвязному или тривиальному графу. .

В предыдущем задании мы установили неравенство $\kappa(G) \leq \lambda(G) \leq \delta(G)$ ($\delta(G)$ — максимальная степень вершин графа G).

Граф G называется вершинно n-связным или просто n-связным (соответственно, реберноно n-связным), если $\kappa(G) \geq n$ ($\lambda(G) \geq n$). Нетривиальный граф 1-связен, тогда и только тогда, когда он связен, и 2-связен, если и только если в нем более одного ребра и он не имеет точек сочленения. Например, полный граф K_2 не является 2-связным.

Попробуйте в качестве упражнения доказать, что граф двусвязен, тогда и только тогда, когда в нем любые две вершины принадлежат простому циклу.

Теоремы Менгера

Пусть u и v — две различные вершины связного графа G. Две простые цепи, соединяющие u и v, называются вершинно-непересекающимися, если у них нет общих вершин, отличных от u и v и реберно-непересекающимися, если у них нет общих ребер. Множество S вершин, ребер или вершин и ребер разделяет u и v, если u и v принадлежат различным различным компонентам графа $G \setminus S$.

Теорема 1 (Карл Менгер (1927)) *Наименьшее число вершин, разделяющих вершины u и v, равно наибольшеиу числу непересекающихся простых u–v цепей.*

Теорема 2 (Форд–Фалкерсон, Элайес–Файнштейн–Шеннон) *Для любых двух вершин графа наибольшее число реберно-непересекающихся цепей, соединяющтх их, равно наименьшему числу ребер, разделяющих эти вершины.*

Теорема 3 (Хасслер Уитни) *Граф n-связен тогда и только тогда, когда любая пара его вершин соединена не менее, чем n вершинно-непесекающимися путями.*

Теорема 4 *Граф реберно n-связен тогда и только тогда, когда любая пара его вершин соединена не менее, чем n реберно-непесекающимися путями.*

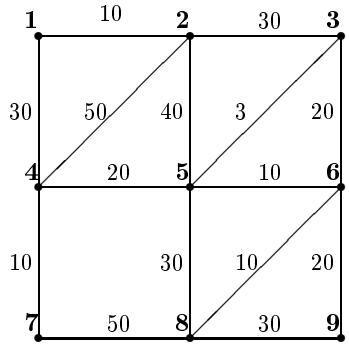
Теорема 5 *Наибольшее число непересекающихся цепей, соединяющих два непустых непересекающихся вершин V_1 и V_2, равно наименьшему числу вершин, разделяющих V_1 и V_2.*

Назовем линией матрицы любую ее строку или столбец. Пусть M — $\{0,1\}$-матрица. Набор единичных элементов матрицы называется независимым, если никакая пара не лежит в общей линии.

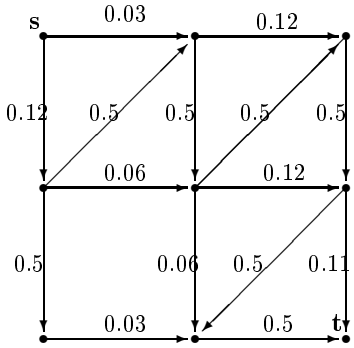
Теорема 6 *В любой бинарной матрице наибольшее число независимых единичных элементов равно наименьшему числу линий, поекрывающих все единицы.*

Кратчайшие пути и связывающие сети

Задача 59. (2×0.01) Найдите минимальное остовное дерево взвешенного графа G, изображенного на рисунке, с помощью алгоритмов Прима и Краскала. (Изобразите графы, полученные на трех последовательных итерациях алгоритмов.)



Задача 60. ($2 \times 0.02 + 0.03$) Коммуникационная сеть является ориентированным графом, причем каждому ребру (каналу связи) (u, v) приписано число $r(u, v)$— "надежность соединения", где $0 \le r(u, v) \le 1$, так что $1 - r(u, v)$ можно рассматривать как вероятность разрыва соединения при передаче. В первом приближении считаем, что "вероятности" $r(u, v)$ независимые, и таким образом, надежность передачи сообщения по пути $v_1, \ldots, v_k$ равна $\prod_{i=1}^{k-1} r(v_i, v_{i+1})$.



(i) Постройте эффективный алгоритм нахождения наименее надежного пути в сети между вершинами s и t и укажите класс сетей, в которых алгоритм будет эффективным.
(ii) Проведите вычисления по вашему алгоритму для сети, изображенной на рисунке.
(iii) Постройте наименее надежную сеть, позволяющую передавать сообщения из вершины s в любую другую вершину графа, содержащую **минимальное** число дуг. Под *надежностью* сети понимается произведение надежностей всех входящих в нее дуг.

Вершины ориентированного **грид-графа** расположены в целых точках плоскости: $V(G) = \{(i, j), i = 0, \ldots, m, j = 0, \ldots, n\}$, а дуги соединяют соседние точки: $E(G) = \{[(i, j) \to (i + 1, j)], i = 0, \ldots, m - 1, j = 0, \ldots, n$ или $[(i, j) \to (i, j + 1)], i = 0, \ldots, m, j = 0, \ldots, n - 1\}$, причем дугам G приписаны целочисленные веса. Рассмотрим задачу поиска экстремального (самого "тяжелого" или самого "легкого") пути между вершинами $(0, 0)$ и (m, n) (по определению, вес пути равен сумме весов входящих в него ребер). В таком виде — это типичная задача так называемого "динамического программирования", описанная во многих источниках. Для нее легко придумать оптимальный $O(mn)$-алгоритм

Задача 61. (2×0.01) (i) Постройте $O(mn)$-алгоритм поиска экстремального пути.
(ii) Покажите, что ваш алгоритм оптимальный по сложности, поскольку любой алгоритм, решающий задачу, обязан прочитать вход (таблицу весов, размер которой равен mn). Иначе говоря, нужно показать, что ответ существенно зависит от каждого входного параметра.

Задача Д–28. (0.3) (Это задача № Д13 из методички.) Пусть теперь веса всех горизонтальных дуг $[(i, j) \to (i + 1, j)]$ зависят только от j, а веса всех вертикальных дуг $[(i, j) \to (i, j + 1)]$ зависят только от i. Таким образом, веса полностью заданы, если указаны n "горизонтальных" весов u_j и m "вертикальных" весов v_i, и длина входа равна в этом случае $O(m + n)$.

Постройте оптимальный линейный $O(m+n)$-алгоритм вычисления экстремальных путей для этого класса грид-графов.

Структуры UNION-FIND
Амортизационный анализ

Для эффективной реализации алгоритма Краскала нужно научиться эффективно обрабатывать непересекающиеся множества. Для этого был придуман специальный алгоритм, который при опросе на одной референтном профессиональном сайте с большим отрывом был признан наиболее выдающимся (это, видимо, достаточный повод, чтобы с ним ознакомиться).

Мы хотим реализовать следующие операции над множествами.

- $MAKESET(x)$ — создать множество с единственным элементом x;
- $UNION(x, y)$ — заменить множества с именами x и y их объединением
- $FIND(x)$ — возвратить имя множества, содержащего элемент x;
- $LINK(x, y)$ — (x и y — корни) перевесить указатель корня x на корень y; в этих обозначениях $UNION(x, y) = LINK(FIND(x), (FIND(y))$.

Можно действовать совсем бесхитростно и считать, что множества — это, например, связные списки, имеющие специальный указатель на *имя множества*. Тогда, грубо говоря, поиск, объединение и пр. пропорционален (суммарному) размеру участвующих в операции множеств. Понятно, что это может быть весьма медленным. Попробуем действовать чуть похитрее (и замечу, вполне практически разумно. Для этого введем специальную характеристику множества: его размер и будем при объединении присваивать *мéньшему* множеству имя *бóльшего*.

Внимание! При решении следующей задачи вы должны уточнить детали и описать конкретную структуру данных и алгоритм, реализующий это предложение! Предполагается, что все множества являются подмножествами $\{1, \ldots, n\}$ и над ними проводятся только операции UNION-FIND, **причем объединяются только непересекающиеся множества.**

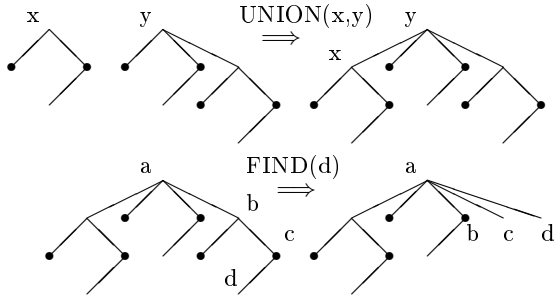
Задача 62. (0.03) Постройте алгоритм, который выполняет ($k \le n - 1$) операций UNION и m операций FIND и имеет трудоемкость $O(\max(m, n \log n))$.

Если $m = O(n \log n)$, то построенный в предыдущей задаче алгоритм является оптимальным по трудоемкости, но, если $m = O(n)$, то процедуру можно ускорить и предложить алгоритм, выполняющую $O(n)$ операций UNION-FIND за почти линейное время. Для

этого вместо связного списка нужно представлять множества лесом корневых (ориентированными) деревьев, ребра которых (указатели в вершинах) направлены к предку. Корень отождествляется с именем множества и его указатель замкнут на себя. Будем считать, что трудоемкость операции FIND равна числу просмотренных в дереве элементов, т. е. не превышает высоты дерева, а сложность UNION равна сложности двух операций FIND. Если теперь использовать изложенный выше прием и при объединении перемещать указатель корня в дереве ме́ньшей высоты на корень дерева бо́льшей высоты, то можно получить следующий результат (опять в силе остается требование формального описания структуры данных, алгоритма и оценки его трудоемкости)

Задача Д–29. (0.03) Постройте алгоритм, который выполняет $O(n)$ операций UNION и FIND и имеет трудоемкость $O(n \log n)$.

Важно понять, что предыдущий результат был получен потому, что нам удалось поддерживать структуру данных, в которой представляющие множества деревья имеют маленькую высоту. Сам прием объединения деревьв по высоте, называтся алгоритмом ОБЪЕДИНЕНИЯ ПО РАНГУ, и, как мы видим, он и практичен, и достаточно эффективен. Но оказывается, что если дополнить его еще одной процедурой, называемой СЖАТИЕ ПУТЕЙ, то мы получим для той же задачи еще более быстрый, почти линейный алгоритм, который и возглавляет список "Алгоритмов из КНИГИ". На самом деле, такая высокая оценка, видимо, дается не только за элегантность процедуры, но и за очень интересный метод оценки трудоемкости. Неформально, первая процедура "привешивает" более низкие деревья с меньшей высотой (=рангом) к более высоким, а вторая,— при поискее любого элемента дерева сразу перевешивает его указатель на корень, как показано на рисунке.



Формальное описание следующее.

```
procedure MAKESET(x)
p(x):=x
rank(x):=0
end

function FIND(x)
if x не равно p(x) then
p(x):=FIND(p(x))
Выполняем сжатие путей, т. е. направляем на
корень указатели всех элементов в пути от корня до x
return(p(x))
end

function LINK(x,y)
if rank(x))>rank(y) then  обменять x и y местами
if rank(x)=rank(y) then rank(y):=rank(y)+1
p(x):=y
return(y)
end

procedure UNION(x,y)
LINK(FIND(x),FIND(y))
end
```

Заметим, что каждая операция FIND или UNION для множеств из n элементов может выполняться за время $O(\log n)$, Но ниже, **используя, как в свое время было обещано, амортизационный анализ**, мы покажем, что выполнение m операций FIND или UNION требует не более $O((m+n)\log^* n)$ операций. А на самом деле, и еще меньше: вместо $\log^* n$ можно поставить обратную к функции Аккермана, рост которой совершенно ничтожен даже по сравнению с $\log^* n$.

Изучим свойства ранга.

- по определению, если $v \neq p(v)$, то $rank(p(v)) > rank(v)$;
- по определению, если $p(v)$ изменяется, то $rank(p(v))$ увеличивается.

Задача 63. (3×0.01) (i) Докажите, что число элементов ранга k не превышает $\frac{n}{2^k}$.
(ii) Докажите. что число элементов ранга $\geq k$ не превышает $\frac{n}{2^{k-1}}$.
(iii) Докажите, что ранг произвольного элемента не превышает $\log n$.

Теперь используем амортизационный анализ, и будем оценивать трудоемкость не одной, а сразу m операций FIND или UNION. Поскольку, как мы уже говорили, последняя выражается через LINK и две операции FIND, то достаточно оценить сложность $2m$ операций FIND (LINK, требует $O(1)$ операций).

Но сначала нужно понять, в чем основная трудность. Дело в том. что FIND *— рекурсивная процедура. Кроме того, и это выглядит не только угрожающим, но и совершенно безнадежным, нам нужно оценить трудоемкость процедуры, в которой текущий лес, представляющий систему непересекающихся множеств, изменяется с каждой операцией (и тем самым изменяются трудоемкость конкретных поисков и обьединений).*

Ниже будет приведен набросок оценки. Его нужно продумать, ибо он нетривиальный и содержит тонкие места. Грубо говоря, мы применим хитрый бухгалтерский трюк: мы разделим алгоритм на этапы, через которые должна проходить любая операция, и будем оценивать трудоемкость каждого этапа сразу для всех m операций. Поскольку речь идет о том, что нужно оценить сложность последовательности операций FIND, т. е. длину путей в соответствующих деревьях, то этапы заключаются в разделении всего возможного диапазона высот деревьев на части и подсчет операций в каждой части.

Формальная конструкция. Ранг элемента не меняется, как только он перестает быть корневым. Разделим все некорневые элементы на группы по величине их ранга r, причем отнесем в группу i все элементы, для которых выполнено равенство $\log^* r = i$, т. е. в i-ю группу попадут элементы с рангами из полуинтервала $(2^{i-1}, 2^{2^{i-1}}]$. Ниже будем использовать сокращение $k = 2^{i-1}$.

Задача Д–30. (2×0.01) (i) Покажите. что число различных групп не превышает $\log^* n$.
(ii) Число элементов в i-й группе не превышает $\frac{n}{F(i)}$.

Теперь используем следующие обозначения, пусть $\Sigma = \{\sigma_i,\ i = 1\ldots, m\}$, где σ_i — это i-й оператор FIND последовательности Σ длины m. σ_i индуцирует путь L_i в соотвествующем текущем дереве от найденного элемента до корня. Формула ниже обслуживает сразу великое множество допустимых последовательностей FIND, и, что выглядит совершенно неправдополобным,— ее удается проанализировать. Итак, подсчитаем для любой последовательности Σ следующую величину (знак "$\rightarrow$"над суммой, говорит о том, что величины зависят от порядка операторов, которые изменяют деревья в процессе поиска).

$$\vec{\sum}_{\sigma_i\in\Sigma} [\text{ число таких } (u \in L_i) \wedge (u, p(u)) \text{ из одной группы }] +$$
$$\vec{\sum}_{\sigma_i\in\Sigma} [\text{ число таких } (u \in L_i) \wedge (u, p(u)) \text{ из разных групп или u — корень}].$$

Пояснение. Понимать это выражение можно следующим образом. Давайте мысленно покрасим каждый оператор FIND и все рекурсивные вызовы (путь по дереву к корню), которые он порождает, своим цветом, так что можно говорить о траектории. Конечно, достаточно, скажем, изменить первый оператор в последовательности и вся картинка (и траектории), возможно, изменится. Тем не менее, каждая траектория пересекает границы групп (число таких ребер-пойнтеров подсчитывается во второй сумме) и, кроме того, поскольку, по построению, при сжатии путей ранги родителей монотонно возрастают по крайней мерее на единицу, то любой элемент из i-й группы может быть подвергнут процедуре сжатия путей не более $F(i)-F(i-1)$ раз, прежде чем он получит родителя из следующей группы (и тогда трудоемкость просмотра элементов будет учитываться во второй сумме).

Вторая сумма оценивается как $O(m\log^* n)$ **просто потому. что число групп (уровней) по построению** $O(\log^* n)$**, а мы подсчитываем,"события", когда ребра-пойнтеры пересекают границы групп**. А первую сумму можно оценить следующим образом. Число операций, которым может подвергаться отдельный элемент внутри группы по порядку равен его рангу

$$\sum_{\text{по всем группам}} [\text{число элементов в группе}]\times$$

$$[\text{максимальный ранг элементов группы}] \le \sum_{k=1}^{\log^* n} \frac{n}{F(k)}F(k) \le n\log^* n,$$

откуда получаем требуемую трудоемкость $O((m+n)\log^* n)$.

Задание на 13-ю неделю: 8.05–14.05. [0.1] Вероятностный алгоритмы Разделы 12 программы Литература: [Кормен 2, §5 и дополнение C] [Кормен 1, §6], [GL], [ДПВ], [К-Ф], [К-Ш-В]

Краткий конспект

В этом задании мы рассмотрим процедуры, использующие *рандомизацию*. Но мы не будем делать *никаких априорных предположений о входном распределении*,

Мы уже обсуждали как минимум три подобные процедуры (быструю сортировку, поиск медианы и проверку простоты), использующие бросание монетки (вероятностный алгоритм) и перейдем теперь к их формальному описанию. К сожалению, из-за недостатка времени мы успеем только ознакомиться с определениями и решить несколько задач. Но можно без преувеличения сказать, что вероятностные подходы к алгоритмам являются стержнем многих современных исследований[24]. Представить без них мир алгоритмов совершенно невозможно, так что всем заинтересованым лицам будет нелишне продолжить изучение этого подхода самостоятельно.

И основной лозунг, под которым оформлено это задание звучит так.

К возможности использования случайных битов нужно относиться как к дополнительному вычислительному ресурсу, позволяющему иногда существенно понизить трудоемкость и концептуально упростить процедуру.

Основным источником таких возможностей является (гипотетическая) возможность алгоритмического порождения "случайных объектов" в конкретных "универсумах", так сказать, "псевдослучайные генераторы". Мы будем использовать простейшие варианты: "орлянку", выбор случайного натурального числа на отрезке $[1,N]$, выбор случайного ребра в графе, выбор случайной плоскости, проходящей через начало координат и т.д. Практически это рутинные программистские операции, используемые часто рефлекторно. Хотя даже на практическом уровне серьезно обсуждаются вопросы о качестве этих генераторов.

В соответствие с идеологией нашего курса, в идеале, в каждом случае, когда нам нужно породить "случайный объект", мы должны указывать конкретную процедуру порождения, уметь проверять (доказывать), что она действительно порождает нужные объекты, и уметь оценивать ее трудоемкость. Хочу отметить, что эти вопросы достаточно тонкие и, более того, многие из них пока не имеют удовлетворительных ответов.

Вероятностная Машина Тьюринга (ВМТ) представляет из себя обычную МТ, которой в некоторых состояния разрешено совершать переходы в зависимости от бросания монеты. В отличии от недетерминированной МТ легко представить себе практическую реализацию такой конструкции. Более того, как утверждают некоторые апологеты теории вероятностей, иных устройств в природе просто не существует. Будем считать, что используются стандартные монетки для игры в орлянку[25], так что каждое вычисление ВМТ на входе x полностью определено, если считать (по аналогии с определением класса $\mathcal{NP}$), что одновременно с x на вход детерминированной МТ подается (вообще говоря, бесконечное) $\{0,1\}$-слово.

Как определить, что слово или язык принимается ВМТ?

В соответствии с определением ВМТ любое вычисление имеет некоторую вероятность.

Будем говорить, что язык $L\subset\Sigma^*$ принимается ВМТ M в *СЛАБОМ смысле [по стандарту МОНТЕ-КАРЛО]*, если для любого слова $x\in\Sigma^*$ вероятность получения ошибочного ответа на вопрос: $(x\in L?)$ не превосходит $\frac{1}{3}$. Иначе говоря, если $x\in L$, то M с вероятностью, не меньшей $\frac{2}{3}$ ПРИНИМАЕТ x; а если $x\notin L$, то M с вероятностью, не меньшей $\frac{2}{3}$ ОТВЕРГАЕТ x.

Будем говорить, что язык $L\subset\Sigma^*$ принимается ВМТ M в *СИЛЬНОМ смысле [по стандарту ЛАС-ВЕГАС]*, если она дает *с вероятностью* 1 правильный ответ для любого слова $x\in\Sigma^*$. Это, в частности, означает, что M не может за конечное число шагов принять какое-нибудь слово $x\notin L$.

Языки, принимаемые ВМТ в СЛАБОМ смысле за *полиномиальное в среднем число шагов*, образуют класс $\mathcal{BPP}$.

Языки, принимаемые ВМТ в СИЛЬНОМ смысле за *полиномиальное в среднем число шагов*, образуют класс $\mathcal{ZPP}$.

Наконец, языки, для которых удается построить ВМТ, которая за полиномиальное в среднем число шагов принимает каждое слово из языка с вероятностью $\ge\frac{1}{2}$ и отвергает любое слово, не входящее в язык, образуют промежуточный класс $\mathcal{RP}$.

По построению, классы $\mathcal{BPP}$ и $\mathcal{ZPP}$ замкнуты относительно дополнения и $\mathcal{BPP}\supseteq\mathcal{RP}\supseteq\mathcal{ZPP}\supseteq\mathcal{P}$. В основном, мы будем изучать $\mathcal{BPP}$. Рекомендую почитать книгу Кузюрина и Фомина (гл. 4–5, §6.2) или книгу "Классические и квантовые вычисления". В последней, в разделах 1.3-1.4 дано определение класса $\mathcal{BPP}$, приведен вероятностный тест простоты Миллера-Рабина, показано, что $\mathcal{BPP}\subset\Sigma_1^p\cap\Pi_1^p$ (т.е. принадежит второму этажу т.н. полиномиальной иерархии). В разделе 12.2 построена вероятностная полиномиальная сводимость (что бы это значило?) вычисления дискретного логарифма (а это, что такое?) к задаче разложения числа на множители (задаче факторизации). Кроме того, сами квантовые вычисления являются аналогом вероятностных вычислений со специально определенным правилом вычисления вероятности. При этом, однако, оказывается, что квантовые компьютеры позволяют решать, например, задачу факторизации за полиномиальное время. Как известно, никто пока не знает, есть ли полиномиальный (детерминированный или вероятностный) классический алгоритм для этой задачи. Кроме того, никто пока не умеет решать на квантовом компьютере какую-нибудь NP-полную задачу за полиномиальное время.

Задача 64. $(0.01+0.01)$ Покажите, что класс $\mathcal{BPP}$ не изменится, если

(i) константу стандарта Монте-Карло $\frac{1}{3}$ заменить на любое число, строго меньшее $\frac{1}{2}$, а

(ii) **полиномиальное в среднем число шагов** заменить на **полиномиальное число шагов**.

[24] Достаточно посмотреть на изменения, внесенных во 2-е издание Кормена.

[25] Монетки у которых, скажем, вероятность выпадения герба является каким-нибудь *невычислимым* числом, в принципе можно использовать, для распознавания невычислимых языков.

Последняя задача позволяет дать определение класса $\mathcal{BPP}$ по аналогии с классом $\mathcal{NP}$.

Предикат L принадлежит классу $\mathcal{BPP}$, если существуют такие полином $q(\cdot)$ и предикат $R(\cdot,\cdot) \in \mathcal{P}$, что

$L(x)=1 \implies$ *доля слов r длины $q(|x|)$, для которых выполнено $R(x,r)$, больше $2/3$;*

$L(x)=0 \implies$ *доля слов r длины $q(|x|)$, для которых выполнено $R(x,r)$, меньше $1/3$.*

Иначе говоря, на вход недетерминированной МТ подается слово-вход x и слово-подсказка r, и x принимается, если и только если при некоторой (не слишком длинной) подсказке принимается пара (x,r). Соответственно, ВМТ читает слово-вход x, а роль слова-подсказки r выполняют результаты бросания монетки в процессе вычисления, причем слово x принадлежит языку, если пары (x,r) принимаются для фиксированной доли C_1 подсказок (бросаний монеты) и отвергается, если число допустимых пар (x,r) меньше некоторой фиксированой доли C_2. По определению, константы C_1 и C_2 должны иметь "зазор" $C_1 - C_2 > \varepsilon$. Если последнее требование опустить (т. е. положить $C_1 = C_2 = \frac{1}{2}$), то вычислительные возможности ВМТ неизмеримо возрастают, а класс распознаваемых на таких ВМТ языков называется $\mathcal{PP}$. В частности, $\mathcal{NP} \subseteq \mathcal{PP}$.

Проверка (полиномиальных) тождеств является одним из наглядных и убедительных примеров нетривиального использования вероятностных алгоритмов. В основе подхода лежит т. н. **Лемма Шварца- Зиппеля**[26].

Пусть $f(x_1,\ldots,x_n)$, не равный тождественно нулю полином степени не выше k по каждой переменной[27], **и пусть принимающие целые значения случайные величины $\xi_1,\ldots,\xi_n$ независимо и равномерно распределены на отрезке $[0, N-1]$.**

Тогда $Prob\{f(\xi_1,\ldots,\xi_n)=0\} \le \frac{kn}{N}$.

Это утверждение мы обсудим на лекции, и на эту тему будут задачи как в ближайших контрольных, так и в финальном тесте.

Доказательство леммы *проводится индукцией по числу переменных n. Утверждение верно при $n=1$, поскольку нетривиальный полином степени k имеет не более k корней. При $n>1$ разложим f по переменной x_1: $f = f_0 + f_1 x_1 + \ldots + f_t x_1^t$, где полиномы $f_0,\ldots,f_t$ не зависят от x_1, а f_t не равен нулю тождественно. Тогда по формуле полной вероятности $Prob\{f=0\} = Prob\{f=0 \mid f_t=0\}Prob\{f_t=0\} + Prob\{f=0 \mid f_t\neq 0\}Prob\{f_t\neq 0\} \le Prob\{f_t=0\} + Prob\{f=0 \mid f_t\neq 0\}$. Первый член оценивается по индуктивному предположению, а второй — не больше, чем $\frac{k}{N}$, поскольку на каждом отрезке $[(0,\xi_2,\ldots,\xi_n),(N-1,\xi_2,\ldots,\xi_n)]$ полином $f = f_0 + f_1 x_1 + \ldots + f_t x_1^t$ с ненулевым старшим коэффициентом f_t может иметь не более $t \le k$ корней, так что $Prob\{f=0\} \le \frac{k(n-1)}{N} + \frac{k}{N}$.*

Задача 65. (4×0.01) Проверьте матричное равенство $C = AB$, где A, B, C– $n\times n$ матрицы, имеющие целочисленные элементы, не превышающие по абсолютной величине h, используя рандомизацию.

Пусть x— случайный n-мерный вектор, компоненты которого независимые целые числа, равномерно выбранные из интервала $[0,1,\ldots,N-1]$. Проверка равенства состоит в вычислении $A(Bx) = Cx$: если это равенство справедливо, то вы предполагаете, что исходное равенство верное, иначе вы сигнализируете об ошибке. Заметим, что каждую такую проверку можно выполнить за $O(n^2)$ операций над $O(\log(nh^2))$-разрядными числами, а любой сигнал об ошибке говорит о том, что исходное равенство неверное. С другой стороны, если проверка прошла успешно, то возможно, что исходное равенство неверное, но мы неудачно подобрали тестовый вектор x.

(i) Каким нужно выбрать N, чтобы вероятность ошибки вашей процедуры была меньше заданной вероятности p?

(ii) Тот же вопрос, если разрешается проводить несколько независимых проверок, а минимизировать нужно общую *битовую сложность* вычислений.

(iii) Сравните битовую сложность вероятностных процедур с стандартным детерминированным алгоритмом перемножения матриц для $n = 10000; h = 2^{15}; p = 0.001$

(iv) Для дальнейшей экономии вы решили использовать проверку $(A(Bx)x) = (Cx)x$ или проверку $(A(Bx)y) = (Cx)y$, где n-вектор y выбирается независимо от x и имеет те же характеристики. Как изменится для этих случаев N?

Язык 2-ВЫПОЛНИМОСТЬ состоит из выполнимых КНФ, в которых каждый дизъюнкт содержит не более двух литералов. Вы знаете, что задачу можно решать за линейное время, используя линейный алгоритм выделения сильно связных компонент в орграфах (об этом мы говорили на последней лекции). В этой задаче мы построим быстрый вероятностный алгоритм для проверки выполнимости 2-КНФ. Пусть 2-КНФ имеет n литералов и m дизъюнктов.

Алгоритм случайного поиска для языка 2-КНФ. Сначала всем переменным присваивается значение TRUE. На каждой итерации, пока формула невыполнима, берется произвольный невыполненный дизъюнкт, в нем равновероятно выбирается произвольный литерал и его логическое значение обращается. Этот процесс напоминает случайное блуждание и в принципе может продолжаться бесконечно (например, если взять невыполнимую КНФ). Однако для выполнимой 2-КНФ можно получить полиномиальные оценки среднего числа итераций. Дело в том, что *число отличий* между текущим набором логических значений переменных и их значениями в некотором произвольном (но фиксированном) выполняющем наборе (существующем по предположению) изменяется на каждой итерации с вероятностью $\frac{1}{2}$ на 1. Таким образом, наш алгоритм можно интерпретировать как случайное блуждание на отрезке [0,1,...,n].

Задача 66. $(0.04 + 0.01)$ (i) Покажите, что для выполнимой 2-КНФ среднее число итераций алгоритма (математическое ожидание числа итераций) равно $O(n^2)$.

(ii) Пусть пункт (i) справедлив (а больше ничего о языке 2-КНФ неизвестно). В какой из вероятностных классов, определенных выше, попадает тогда язык 2-КНФ?

Задача о минимальном разрезе в неориентированном графе $G = (V,E)$ заключается в том, чтобы разбить вершины графа на два дизъюнктных подмножества $(S,\bar{S})$, $S \neq V$, $S \neq \varnothing$ так, чтобы минимизировать число ребер с концами в разных долях. Конечно, для ее решения можно применить потоковый алгоритм, но мы рассмотрим простую вероятностную процедуру. При этом основной будет операция стягивания ребра (кратные ребра остаются, а петли удаляются). Граф, полученный стягиванием ребра $(x,y) \in E$, обозначим $G/(x,y)$. Первоначальная идея заключается в следующем: при стягивании ребер величина минимального разреза не убывает (пока в графе остается не менее двух вершин), так что если стянуть все ребра $\{e_1,\ldots,e_p\}$, не входящие в минимальный разрез, то останется пара вершин, соединенная k ребрами, где k — величина минимального разреза в G. Остается понять, как часто реализуется подобная ситуация, если ребра стягиваются случайно.

[26] Смысл леммы в том, что если полином не равен нулю тождественно, то он не может слишком часто обращаться в нуль, например, в точках целочисленной решетки. Это утверждение известно как Schwartz-Zippel Lemma. На самом деле, Шварцу, видимо, принадлежит вероятностная интерпретация, поскольку сам факт давно известен (см., например, главу "Сравнения" в книге Боревича и Шафаревича "Теория чисел"), но ему не придавали вероятностной интерпретации.

[27] Лемма остается верной, если считать, что k — это суммарная степень по совокупности переменных.

Задача 67. $(2 \times 0.02 + 0.01)$ (i) Покажите, что вероятность того, что случайно выбранное ребро в графе входит в минимальный разрез не превышает $\frac{2}{|V|}$.

Из предыдущей задачи вытекает следующий вероятностный алгоритм определения минимального разреза:

MINCUT $[G(V,E), |V| = n]$

$G_0 \leftarrow G$; $i \leftarrow 0$;

while $|V(G_i)| > 2$ **do**

В G_i выбираем случайное ребро e_i (с равномерным распределением на ребрах G_i).

$G_{i+1} \leftarrow G_{/e_i}$; $i \leftarrow i+1$;

end while

Комментарий. На выходе из цикла получаем (мульти)граф $\tilde{G}$, имеющий две вершины, соединенные ребрами, иногда отвечающими разрезу в исходном графе.

return Разрез в исходном графе G, отвечающий разрезу в $\tilde{G}$.

Времы работы алгоритма $O(n^2)$.

(ii) Покажите что MINCUT выдает минимальный разрез с вероятностью $\geq \frac{2}{n(n-1)}$.

(iii) Покажите, что если независимо повторить процедуру MINCUT n^2 раз, то минимальный разрез будет найден с вероятностью > 0.85.

В следующей задаче мы покажем, что если привлечь дополнительные соображения, то можно понизить трудоемкость до $O(n^2 \log^{O(1)} n)$. При этом алгоритм столь же прост и допускает параллелизацию.

Описанная выше процедура поиска минимального разреза последовательно выбирает случайные ребра и стягивает их концы до тех пор, пока в графе не останутся две вершины, соединенные (кратными) ребрами. Если при выборе случайных ребер мы ни разу не выбирали ребра разреза, то мы получаем ответ. Выше мы показали, что вероятность на i-м шаге выбрать ребро, входящее в разрез, равна $p_i = \frac{2}{n-i}$, отсюда вероятность того, что за i шагов не будет выбрано ни одно ребро, входящее в минимальный разрез (мы будем говорить, что выбранные *ребра не задевают разрез*), равна $P_i = (1-p_1)(1-p_2)\dots(1-p_i)$. Мы хотим ускорить алгоритм. Заметим, что чем больше ребер мы стягиваем, тем больше вероятность, что следующее выбранное ребро заденет минимальный разрез. Поэтому новая идея, которую мы хотим исследовать, заключается в том, чтобы стягивать ребра до какого-то порога, пока вероятность попадания в разрез еще достаточно мала, а дальше использовать рекурсию. Из формулы для P_i видно, что если стянуть $n/2$ случайных ребер, то они с вероятностью $\geq \frac{1}{4}$ не заденут минимальный разрез.

Блок-схема нового алгоритма приведена ниже. Процедура использует подпрограмму СТЯГИВАНИЕ (G,k), которая стягивает ребра до тех пор пока число вершин не уменьшится ниже порога k (при стягивании произвольного ребра число вершин уменьшается на единицу).

СТЯГИВАНИЕ (G,k)
for $i := n$ **downto** k
В G выбираем случайное ребро e
(с равномерным распределением
на ребрах).
$G \leftarrow G/e$
endfor
return G

МИН-РАЗРЕЗ (G)
if в G больше восьми вершин **then**
Повторить 4 раза процедуру
$X_1 \leftarrow$ МИН-РАЗРЕЗ [СТЯГИВАНИЕ $(G, \frac{n}{2})$];
$X_2 \leftarrow$ МИН-РАЗРЕЗ [СТЯГИВАНИЕ $(G, \frac{n}{2})$];
$X_3 \leftarrow$ МИН-РАЗРЕЗ [СТЯГИВАНИЕ $(G, \frac{n}{2})$];
$X_4 \leftarrow$ МИН-РАЗРЕЗ [СТЯГИВАНИЕ $(G, \frac{n}{2})$];
return $\min\{X_1, X_2, X_3, X_4\}$
else находим минимальный разрез вручную.

Задача Д–31. $(0.01 + 0.01 + 0.04 + 0.01)$

(i) Запишите рекуррентную оценку сложности $T(n)$ вычисления функции МИН-РАЗРЕЗ (G) для графа с $|V| = n$ вершинами.

(ii) Найдите Θ-асимптотику $T(n)$ (для этого нужно сначала оценить трудоемкость процедуры СТЯГИВАНИЕ (G,k)).

Оценим теперь с какой вероятностью $\mathbb{P}(n)$ алгоритм МИН-РАЗРЕЗ(G) выдает минимальный разрез для графа G с n вершинами. Вероятность успеха равна вероятности того, что хотя бы один рекурсивный вызов дал корректный ответ. Таким образом, получаем: $\mathbb{P}(n) = 1 - \left(1 - \frac{1}{4}\mathbb{P}(\frac{n}{2})\right)^4$, откуда следует, что $\mathbb{P}(n) \geq \mathbb{P}(\frac{n}{2}) - \frac{3}{8}\mathbb{P}(\frac{n}{2})^2$. Считая, что n является степенью двойки, обозначим $p_k = \mathbb{P}(2^k)$. По определению, p_k равно вероятности успеха, если потребовалось k рекурсивных вызовов процедуры МИН-РАЗРЕЗ. В частности, $p_0 = 1$.

Получаем рекурсию $p_{k+1} = 1 - (1 - \frac{1}{4}p_k)^4$. Откуда, если раскрыть скобки, следует: $p_{k+1} \geq p_k - \frac{3}{8}(\mathcal{P}_k)^2$.

(iii) Приведите как можно более точную оценку снизу рекурсии: $p_0 = 1; p_{k+1} = p_k - \frac{3}{8}(p_k)^2$ вида $p_k = \Omega(f(k))$.

Подсказка. Предположите, что $p_{k+1} - p_k \approx \frac{dp}{dk}$, и оцените порядок роста функции $p(k)$, а потом обоснуйте вашу гипотезу.

(iv) Получите оценку числа итераций модифицированно вероятностного алгоритма поиска минимального разреза, для получения заданной точности ε и приведите оценку общего числа операций.

Задача Д–32. $(0.01 + 0.03 + 0.01)$ **[Вероятностный алгоритм для языка ВЫПОЛНИМОСТЬ и дерандомизация].** Предположим, КНФ содержит m дизъюнктов и в каждый дизъюнкт входит ровно k литералов. Пусть X — случайная величина, равная числу выполненных дизъюнктов, если независимо и равновероятно приписать каждому литералу значения 0 или 1. Поскольку каждый дизъюнкт ложен лишь при одном значении литералов и матожидания суммируются, то математическое ожидание числа выполненных дизъюнктов равно: $E(X) = m(1 - 2^{-k})$.

Это значит, что существует логический набор, на котором выполнено не менее $E(X)$ дизъюнктов. Далее, как и в случае с задачами из $\mathcal{NP}$, возникает та же проблема. Мы знаем, что такой набор существует, но не знаем, как его найти, не используя полный перебор. В нашем случае, для нахождения искомого набора можно, во-первых, построить эффективный вероятностный алгоритм. И более того, можно провести дерандомизацию, т. е. конвертировать вероятностную процедуру в детерминированную (не сильно увеличивая сложность). Последнее возможно далеко не всегда, и одним из центральных вопросов теории сложности является соотношение между классами $\mathcal{P}$ и $\mathcal{BPP}$, т. е. верно ли, что всякую процедуру из $\mathcal{BPP}$ можно дерандомизировать?

Алгоритм нахождения набора заключается в следующем. Оценим вероятность события $p = Prob[X \geq m(1 - 2^{-k})]$ (т. е. что в КНФ *выполнено не менее* $m(1-2^{-k})$ *дизъюнктов*). Для этого распишем математическое ожидание: $E(X) = \sum_{i=1}^{m} i * Prob[X = i] \leq [m(1 - 2^{-k}) - 1](1 - p) + mp$. Отсюда $p \geq (m2^{-k} + 1)^{-1}$.

Такая оценка позволяет построить следующий вероятностный алгоритм для определения набора, выполняющего не менее $m(1 - 2^{-k})$ дизъюнктов: достаточно независимо повторить процедуру $\geq m2^{-k} + 1$ раз и тогда с вероятностью $> \frac{1}{2}$ одна из попыток даст искомый набор.

(i) Будет ли предложенный алгоритм 1) Лас-Вегас алгоритмом; 2) Монте-Карло алгоритмом; 3) ни тем, ни другим?

Поскольку порождение случайного набора требует $O(n)$ операций, а проверка числа выполненных дизъюнктов требует $O(mk)$ операций, то одна итерация алгоритма занимает $O(m2^{-k}(mk+n))$

Теперь попробуем дерандомизировать процедуру.

Свяжем с каждым литералом x_i случайную величину Y_i, принимающую равновероятно значения 0 или 1, и будем считать $\{Y_i\}$, $i = 1, \dots, n$ независимыми. Значение величины Y_i будем обозначать ма-

ленькой буквой y_i. Мы используем т.н. метод *условных вероятностей*, который заключается в последовательном приписывании логических значений литералам так, чтобы на каждом шаге выполнялось неравенство: $E(X|y_1,\ldots,y_j) \leq E(X|y_1,\ldots,y_{j+1})$. Осуществить такой подход удается не всегда. В нашем случае ключевым является тождество[28]: $E(X|y_1,\ldots,y_j) = \frac{1}{2}[E(X|y_1,\ldots,y_j,Y_{j+1} = 1)+E(X|y_1,\ldots,y_j,Y_{j+1}=0)]$. Таким образом, нужно научиться *детерминированно* приписывать литералу x_{j+1} значение, имеющее *большее условное математическое ожидание*. Для этого нужно разбить множество дизъюнктов на 4 непересекающиеся подмножества [1) уже выполненных; 2) не зависящих от x_{j+1}; 3) выполняющихся при $x_{j+1}=1$; 4) выполняющихся при $x_{j+1}=0$|, вычислить условные мат. ожидания и присвоить литералу x_{j+1} значение, отвечающее большему мат. ожиданию.

(*ii*) Закончите вычисления, т. е. явно укажите, какие истинностные значения следует присваивать литералам. При вычислении условных вероятностей не забывайте о вкладе переменных, значения которым **еще не присвоены**!

(*iii*) Проведите вычисления для 2-КНФ[29] $(\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_6) \wedge (x_2 \vee \bar{x}_4) \wedge (\vee\bar{x}_5 \vee x_7) \wedge (\bar{x}_7 \vee x_8\vee) \wedge (x_1 \vee \bar{x}_7)$

В заключение отметим, что по ходу рассуждений мы показали следующее полезное утверждение (которое полезно доказать каким-то иным способом): *всякая k-КНФ, имеющая меньше* 2^k *дизъюнктов, выполнима.*

Кроме того, *для случая 3-КНФ, каждый дизъюнкт которой содержит ровно 3 литерала, мы построили* $\frac{7}{8}$*-приближенный (детерминированный и вероятностный) полиномиальный алгоритмы для NP-трудной задачи MAX-3-SAT, в которой требуется выполнить максимальное число дизъюнктов*[30]. И в этом бы не было ничего удивительного, если бы J.Håstad не показал (это довольно тяжело), что при $\mathcal{P} \neq \mathcal{NP}$ **никакая** эффективная процедура для задачи MAX-ВЫПОЛНИМОСТЬ не может давать большую точность.

Заключительная контрольная пройдет в понедельник 15.05 с 13:55–17:05 в Акт. зале и Б.Хим.

Утешительная контрольная пройдет в в понедельник 22.05 с 13:55–17:05 в Акт. зале и Б.Хим .

[28] На самом деле, достаточно потребовать *квазивогнутости*: $E(X|\ldots) \leq \max[E(X|\ldots,Y_{j+1}=1), E(X|\ldots,Y_{j+1}=0)]$.

[29] Контрольный вопрос: является ли указанная процедура полиномиальным алгоритмом для 2-КНФ?

[30] Это, значит, что алгоритм находит набор, на котором выполняется не менее $\frac{7}{8}$ от максимально возможного числа дизъюнктов, которые можно одновременно выполнить.