SPb HSE, 1 курс, весна 2023/24
Конспект лекций по алгоритмам

Собрано 10 июня 2024 г. в 17:03

Содержание

Лекция #0: Введение в теорию сложности

22 и 29 января 2024

0.1. Decision/search/разрешимость

• Decision/search problem

Если в задаче ответ – `true`/`false`, то это *decision problem (задача распознавания)*.
Иначе это *search problem (задача поиска)*. **Примеры:**

1. Decision. Проверить, есть ли x в массиве a.
2. Search. Найти позицию x в массиве a.
3. Decision. Проверить, есть ли путь из a в b в графе G.
4. Search. Найти сам путь.
5. Decision. Проверить, есть ли в графе подклика размера хотя бы k.
6. Search. Найти максимальный размер подклики.
7. Search. Найти подклику максимального размера.

Decision problem f можно задавать, как язык (множество входов) $L = \{x\colon f(x) = \texttt{true}\}$.

• Почти все decision-задачи алгоритмически не разрешимы!

Любое множество $L \subseteq \mathbb{N}$ это decision-задача $\Rightarrow$ всего задач $|2^{\mathbb{N}}|$ (несчётно).
Алгоритм — строка на питоне, строк счётное число ($|\mathbb{N}|$), ведь их можно занумеровать.

$$|2^{\mathbb{N}}| > |\mathbb{N}| \Rightarrow \text{ почти все задачи неразрешимы}$$

Тут можно сослаться на общую теорему Кантора ($\left|2^A\right| > |A|$),
либо вспомнить «диагонализацию Кантора» [wiki] для $|\mathbb{R}| = |2^{\mathbb{N}}|$, $|\mathbb{R}| > |\mathbb{N}|$.

Посмотрим на примеры неразрешимых задач. Примеров много[1], much enough[2], see also[3].
Можно выделить класс задач «предсказание будущего сложного процесса».
Например, «в игре жизнь [wiki] достижимо ли из состояния A состояние B?». Или не в игре...

Канонический пример — «проблема останова» (halting problem):
Дана программа, остановится ли она когда-нибудь на данном входе?

Теорема 0.1.1. *Halting problem алгоритмически не разрешима.*

Доказательство. От противного. Пусть есть алгоритм `terminates(code,x)`, всегда останавливающийся, и возвращающий `true` iff `code` останавливается на входе `x`. Рассмотрим программу:

```
1 def invert(code):
2   if terminates(code,code): while (true)
```

Если `invert(invert)` останавливается, то должен зависнуть, и наоборот. Противоречие. ■

Теорема Успенского-Райса [ifmo]: любое нетривиальное свойство программ неразрешимо.
«Нетривиальное» = хотя бы одна программа ему удовлетворяет, но не все программы.
Примеры: программа возвращает только простые числа; решает задачу о рюкзаке.

0.2. Основные классы

• **DTime, P, EXP (классы для decision задач)**

Def 0.2.1. $\mathrm{DTime}(f(n))$ – *множество задач распознавания, для которых* $\exists C > 0$ *и детерминированный алгоритм, работающий на всех входах не более чем* $C \cdot f(n)$*, где* n *– длина входа.*

Def 0.2.2. $\mathrm{P} = \bigcup\limits_{k>0} \mathrm{DTime}(n^k)$. *Т.е. задачи, имеющие полиномиальное решение.*

Def 0.2.3. $\mathrm{EXP} = \bigcup\limits_{k>0} \mathrm{DTime}(2^{n^k})$. *Т.е. задачи, имеющие экспоненциальное решение.*

Теорема 0.2.4. *Об иерархии по времени*
$\mathrm{DTime}(f(n)) \subsetneq \mathrm{DTime}(f(n)\log^2 f(n))$ *для конструктивных по времени* f *(см. ниже).*

Доказательство. Задача, которую нельзя решить за $f(n)$: завершится ли данная программа за $f(n)\log f(n)$ шагов. Подробнее на практике и в wiki, там же дана более сильная формулировка теоремы, требующая большей аккуратности при доказательстве. ■

Следствие 0.2.5. $\mathrm{P} \neq \mathrm{EXP}$ т.к. $\mathrm{P} \subseteq \mathrm{DTime}(2^n) \subsetneq \mathrm{DTime}(2^{2n}) \subseteq \mathrm{EXP}$

0.3. NP (non-deterministic polynomial)

Def 0.3.1. $\mathrm{NP} = \{L\colon \exists M$*, работающий за полином от* $|x|$, $\forall x \left((\exists y\, M(x,y) = 1) \Leftrightarrow (x \in L)\right)\}$

Неформально: NP – класс языков L: $\forall\, x \in L$, $\exists$ такая подсказка $y(x)$, что мы за полином сможем проверить, что $x \in L$. Напомним, «язык» $\equiv$ «decision задача».
Ещё более неформально: «NP – класс задач, ответ к которым можно проверить за полином». Подсказку y так же называют свидетелем того, что x лежит в L.

• **Примеры NP-задач**

1. HAMPATH $= \{G \,|\, G$ – неорграф, в котором есть гамильтонов путь$\}$. Подсказка y – путь. M получает вход $x = G$, подсказку y, проверяет, что y прост, $|y| = n$ и $\forall (e \in y)\ e \in G$.
2. k-CLIQUE – проверить наличие в графе клики размером k. Подсказка y – клика.
3. IS-SORTED – отсортирован ли массив? Она даже лежит в P.

Def 0.3.2. $\mathrm{coNP} = \{L\colon\ \overline{L} \in \mathrm{NP}\}$

Def 0.3.3. $\mathrm{coNP} = \{L\colon \exists M$*, работающий за полином от* $|x|$, $\forall x \left((\forall y\, M(x,y) = 0) \Leftrightarrow (x \in L)\right)\}$

Def 0.3.4. $\mathrm{coNP} = \{L\colon \exists M$*, работающий за полином от* $|x|$, $\forall x \left((\exists y\, M(x,y) = 1) \Leftrightarrow (x \notin L)\right)\}$

Неформально: дополнение языка лежит в NP или «$\exists$ свидетель того, что $x \notin L$».

Пример coNP-задачи: PRIME – является ли число простым. Подсказкой является делитель. На самом деле PRIME $\in$ P, но этого мы пока не умеем понимать.

Замечание 0.3.5. $\mathrm{P} \subseteq \mathrm{NP}$ (можно взять пустую подсказку, M просто решит задачу).

Замечание 0.3.6. Вопрос $\mathrm{P} = \mathrm{NP}$ или $\mathrm{P} \neq \mathrm{NP}$ остаётся открытым (wiki). Предполагают, что $\neq$.

0.4. NP-hard, NP-complete

Def 0.4.1. $\exists$ *полиномиальное сведение (по Карпу) задачи* A *к задаче* B*:* $(A \leqslant_P B) \Leftrightarrow \exists$ *алгоритм* f*, работающий за полином,* $(x \in A) \Leftrightarrow (f(x) \in B)$

Замечание 0.4.2. f работает за полином $\Rightarrow |f(x)|$ полиномиально ограничена $|x|$

Def 0.4.3. $\exists$ *сведение по Куку задачи* A *к задаче* B*:* $(A \leqslant_C B) \Leftrightarrow \exists M$*, решающий* A*, работающий за полином, которому разрешено обращаться за* $\mathcal{O}(1)$ *к решению* B*.*

Ещё говорят «задача A сводится к задаче B».

В обоих сведениях мы решаем задачу A, используя уже готовое решение задачи B.
Т.е. доказываем, что «A не сложнее B». Различие: в первом случае решением B можно воспользоваться один раз (и инвертировать ответ нельзя), во втором случае полином раз.

Def 0.4.4. NP-hard = NPh = $\{L\colon \forall A \in \text{NP} \quad A \leqslant_P L\}$

NP-трудные задачи – класс задач, которые не проще любой задачи из класса NP.

Def 0.4.5. NP-complete = NPc = NPh $\cap$ NP

NP-полные задачи – самые сложные задачи в классе NP.
Если мы решим хотя бы одну из NPc за полином, то решим все из NP за полином.
Хорошая новость: все NP-полные по определению сводятся друг к другу за полином.

Замечание 0.4.6. Когда хотите выразить мысль, что задача трудная в смысле решения за полином (например, поиск гамильтонова пути), **неверно** говорить «это NP задача» (любая из P тоже в NP) и странно говорить «задача NP-полна» (в этом случае вы имеете в виду сразу, что и трудная, и в NP). Логично сказать «задача NP-трудна».

Lm 0.4.7. $A \leqslant_P B, B \in \text{P} \Rightarrow A \in P$

Доказательство. Сведение f работает за n^s, B решается за $n^t \Rightarrow A$ решается за n^{st}. ■

Lm 0.4.8. $A \leqslant_P B, A \in \text{NPh} \Rightarrow B \in \text{NPh}$

Доказательство. $\forall L \in \text{NP}$ ($\exists f\colon L$ сводится к A функцией $f(x)$) $\wedge$ ($A \leqslant_P B$ функцией $g(x)$) $\Rightarrow$ L сводится к B функцией $g(f(x))$ (за полином). ■

0.5. NH, NP-полные задачи существуют!

Def 0.5.1. BH = BOUNDED-HALTING*: вход* $x = \langle \underbrace{11\ldots1}_{k}, M, x\rangle$*, проверить,* $\exists$ *ли такой* y*:* $M(x, y)$ *остановится за* k *шагов и вернёт* `true`*.*

BH $\in$ NP. Подсказка – такой y. Алгоритм – моделирование k шагов M за $\mathcal{O}(\text{poly}(k))$.

Важно, что если бы число k было записано, используя $\log_2 k$ бит, моделирование работало бы за экспоненту от длины входа, и нельзя было бы сказать «задача лежит в NP».

Докажем, что BH $\in$ NPc. На экзамене доказательство можно сформулировать в одно предложение, здесь же оно для понимания расписано максимально подробно.

Теорема 0.5.2. BH = BOUNDED-HALTING ∈ NPc

Доказательство. NPc = NP ∩ NPh. Мы уже показали Def 0.5.1, что BH лежит в NP.
Теперь покажем, что BH ∈ NPh. Для этого нужно взять $L \in$ NP и свести его к BH.
Пусть $L \in$ NP $\Rightarrow \exists$ полиномиальный M, проверяющий подсказки для L.
Полиномиальный $\Leftrightarrow \exists P(n)$, ограничивающий время работы M. $(x \in L) \Leftrightarrow \exists y\ M(x, y) = 1$.
Программа M всегда отрабатывает за $P(|x|)$, если запустить её с будильником $P(|x|)$, она не поменяется. Рассмотрим $f(x) = \langle \underbrace{11\ldots1}_{P(|x|)}, M, x \rangle$.

Получили полиномиальное сведение: $(x \in L) \Leftrightarrow (\exists y\ M(x, y) = 1) \Leftrightarrow (f(x) \in$ BH$)$.
Заметьте, зная L, мы не умеем предъявить ни M, ни f, мы лишь знаем, что $\exists M, f$. ■

0.6. Сведения, новые NP-полные задачи

Началось всё с того, что в 1972-м Карп опубликовал список из 21 полной задачи, и дерево сведений. Кстати, в его работе [pdf] все сведения крайне лаконичны. Итак, приступим:

Чтобы доказать, что $B \in$ NPh, нужно взять любую $A \in$ NPh и свести A к B полиномиально. Пока такая задача A у нас одна – BH. На самом деле их очень много.

Чтобы доказать, что $B \in$ NPc, нужно ещё не забыть проверить, что $B \in$ NP.
Во всех теоремах ниже эта проверка очевидна, мы проведём её только в доказательстве первой.

• BH → CIRCUIT-SAT → SAT → 3-SAT → k-INDEPENDENT → k-CLIQUE

Def 0.6.1. CIRCUIT-SAT. *Дана схема, состоящая из входов, выхода, гейтов* `AND`, `OR`, `NOT`. *Проверить, существует ли набор значений на входах, дающий* `true` *на выходе.*

Теорема 0.6.2. CIRCUIT-SAT ∈ NPc

Доказательство. Подсказка – набор значений на входах ⇒ CIRCUIT-SAT ∈ NP.
Сводим BH к CIRCUIT-SAT ⇒ нам даны программа M, время выполнения t, вход x.
За время t программа обратится не более чем к t ячейкам памяти.
Обозначим за $s_{i,j}$ состояние `true`/`false` j-й ячейки памяти в момент времени i.
$s_{0,j}$ – вход, $s_{t,output}$ – выход, $\forall i \in [1, t]$ $s_{i,j}$ зависит от $\mathcal{O}(1)$ переменных $(i-1)$-го слоя.
Сейчас значение s_{ij} – произвольная булева формула f_{ij} от $\mathcal{O}(1)$ переменных из слоя s_{i-1}.
Перепишем f_{ij} в КНФ-форме, чтобы получить гейты вида `AND`, `OR`, `NOT`. Получили $\mathcal{O}(t^2)$ булевых гейтов ⇒ по (M, t, x) за полином построили вход к CIRCUIT-SAT. ■

Теорема 0.6.3. SAT ∈ NPc

Доказательство. В разборе практики смотрите сведение из CIRCUIT-SAT. ■

Теорема 0.6.4. 3-SAT ∈ NPc

Доказательство. Сводим SAT к 3-SAT. Пусть есть клоз $(x_1 \vee x_2 \vee \cdots \vee x_n), n \geqslant 4$.
Введём новую переменную w и заменим его на $(x_1 \vee x_2 \vee w) \wedge (x_3 \vee \cdots \vee x_n \vee \overline{w})$. ■

Def 0.6.5. k-INDEPENDENT*: есть ли в графе независимое подмножество размера k?*

Теорема 0.6.6. k-INDEPENDENT $\in$ NPc

Доказательство. Наша формула – m клозов $(l_{i1} \vee l_{i2} \vee l_{i3})$, где l_{ij} – литералы.
Построим граф из ровно $3m$ вершин – l_{ij}. $\forall i$ добавим треугольник (l_{i1}, l_{i2}, l_{i3}) (итого $3m$ рёбер).
В любое независимое множество входит максимум одна вершина из каждого треугольника.
$\forall k = 1..n$ соединим все вершины $l_{ij} = x_k$ со всеми вершинами $l_{ij} = \overline{x_k}$.
Теперь $\forall k = 1..n$ в независимое множество нельзя одновременно включить x_k и $\overline{x_k}$.
Итог: $\exists$ независимое размера $m \Leftrightarrow$ у 3-SAT было решение. ■

Def 0.6.7. k-CLIQUE *задача: есть ли в графе полное подмножество вершин размера k?*

Теорема 0.6.8. k-CLIQUE $\in$ NPc

Доказательство. Есть простое двустороннее сведение k-CLIQUE $\leftrightarrow$ k-INDEPENDENT.
c_{ij} – есть ли ребро между i и j вершинами. Создадим новый граф: $c'_{ij} = \overline{c_{ij}} \wedge (i \neq j)$. ■

0.7. Задачи поиска

Def 0.7.1. $\overline{\text{NP}}$, $\overline{\text{NPc}}$, $\overline{\text{NPh}}$ – *аналогичные классы для задач поиска подсказки.*

• Сведение задач минимизации, максимизации к decision задачам

Пусть мы умеем проверять, есть ли в графе клика размера k.
Чтобы найти размер максимальной клики, достаточно применить бинпоиск по ответу.
Это общая техника, применимая для максимизации/минимизации численной характеристики.

• Сведение search задач к decision задачам

Последовательно фиксируются биты (части) подсказки y.

Пример: выполняющий набор для SAT.
Пусть $M(\varphi)$ проверяет выполнимость формулы φ (M – решение SAT).
$\varphi[x_i{=}e]$ – формула, полученная из φ подстановкой x_i`:=`e. Индукция:
`while` n `> 0:`
    `if` $M(\varphi[x_n{=}0]) = 1$ `then` $r_n = 0$ `else` $r_n = 1$
    $\varphi \leftarrow \varphi[x_n{=}r_n]$`; n--`

Пример: k-INDEPENDENT.
Взять или выкинуть первую вершину?
`if` $(G \setminus \{1\}) \in k$-INDEPENDENT `then` G `\=` $\{1\}$
`else k--`, ans `∪=` $\{1\}$, G `\=` $\{N(1)\}$ (выкинуть соседей 1-й вершины)

• Решение NP-полных задач

Пусть вам дана NP-полная задача. С одной стороны плохо – для неё нет быстрого решения. С другой стороны её можно свести к SAT, для которого несколько десятилетий успешно оптимизируются специальные SAT-solvers. Например, вы уже можете решать k-CLIQUE, построив вход к задаче SAT и скормить его `python3` пакету pycosat.
А ещё можно принять участие в соревновании.

0.8. Гипотезы

Гипотеза 0.8.1. $P \neq NP$

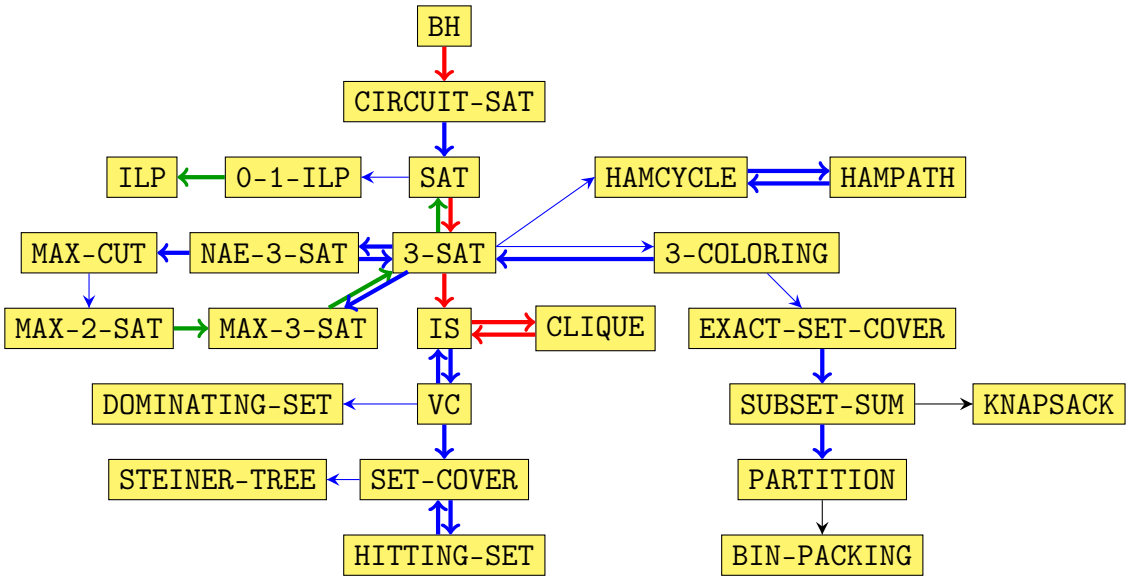
Гипотеза 0.8.2. ETH (exponential time hypothesis): $\nexists$ решения за $2^{o(n)}$ для 3-SAT

Гипотеза 0.8.3. SETH (strong ETH): $\forall \varepsilon\, \exists k$: $\nexists$ решения за $(2-\varepsilon)^n$ для k-SAT

Lm 0.8.4. SETH $\Rightarrow$ ETH $\Rightarrow$ P $\neq$ NP

0.9. Дерево сведений

Легенда:
- **Красное** – было на лекции.
- **Синее** – было в задачах практики/дз, нужно знать на экзамене.
- Бледносинее – было в задачах практики/дз, не будет на экзамене.
- **Зелёное** – очевидное вложение.
- Чёрное – будет доказано и использовано в будущем.



Лекция #1: (*) Дополнительная сложность

29 января 2021

1.1. (*) Алгоритм Левина

Есть универсальный алгоритм, который для любой задачи из $\overline{\text{NP}}$ работает за время близкое к оптимальному. Рассмотрим любую задачу поиска $L \in \overline{\text{NP}}$.

Пусть $M(x, y)$ – полиномиальный чекер для L. Рассмотрим любой корректный алгоритм A, решающий L. $\exists S_A$ – программа на питоне, имплементирующая A. S_A – строка, то есть, число в 256-ичной системе счисления. Рассмотрим соответствующее ей число N_A.
N_A – константа. Пусть $t(|x|)$ – время работы A.

Будем перебирать все программы и время работы.

```
1 solve(x):
2     for (T = 1;; T *= 2) // время работы T
3         for (P = 1; 2^P <= T; P++) // программа P
4             y = call(P, T, x) // запускаем T шагов программы P на входе x
5             if M(x, y)
6                 return y
```

Если решений нет, наш алгоритм успешно зависнет. Далее обозначим $|x| = n$.
Если решение $\exists$, как только $T \geqslant \max(2^{N_A}, t(n)) = F = \Theta(t(n))$, мы найдём решение.

Оценим время работы.
Одна итерация внешнего `for` работает за $\mathcal{O}(M \cdot T \log T)$. Суммарное время:
$\sum_T (M \cdot T \log T) = \Theta(M \cdot F \log F) = \Theta(M \cdot t(n) \log t(n))$ (помним, что 2^{N_A} – просто константа).

Итого, мы получили алгоритм, который для любой $\overline{\text{NP}}$ задачи при существовании решения находит его за время $\Theta(M \cdot t \log t)$, где t – время работы оптимального алгоритма.
Например, для гамильтонова пути это будет $\Theta(V \cdot t \log t)$.

Полученный алгоритм называется *алгоритмом Левина*.

Следствие 1.1.1. Если P = NP, то алгоритм Левина решает все $\overline{\text{NP}}$ задачи за полином.

1.2. (*) Расщепление

Мы будем решать задачу поиска максимального независимого множества (IS). Но, конечно, все наши идеи подходят также для максимальной клики и минимального вершинного покрытия.

Итак, простейший алгоритм расщепления:
берём вершину v и или берём её в независимое множества (первая ветка рекурсии), или не берём (вторая ветка). Берём $\Rightarrow$ удалим и v и соседей. Не берём $\Rightarrow$ удалим v.

Время работы $T(n) = T(n - deg[v] - 1) + T(n-1) \Rightarrow$ выгодно взять v: $deg[v] = \max$.

Осталось заметить
1. Если $\exists$ вершина степени 1 её выгодно жадно взять в IS.
2. Если $\exists$ вершина v степени 2. Если мы берём ровно одного из её соседей, можно заменить его на v $\Rightarrow$ есть только два разумных варианта – или взять v, или её обоих соседей $\Rightarrow$ модифицируем наш граф: удалим v, смерджим её соседей a, b в одну вершину ab. Вариант «брать v» перешёл в «не брать ab», вариант «брать a, b» перешёл в «брать ab».

Ответ уменьшился ровно на 1, а число вершин на 2. При оценке времени нам важно будет, что как только появляется вершина степени 2, сразу $n \to n-2$.

Остался случай $\forall v\ deg[v] \geqslant 3$. Уже получили асимптотику $\boxed{T(n) = T(n-1) + T(n-4) \leqslant 1.3803^n}$.
В дальнейшем анализе рекурренты $T(n-a_1) + T(n-a_2) + \dots$ будем обозначать $[a_1, a_2, \dots]$.

Замечание 1.2.1. На самом деле все асимптотики α^n нужно записывать как $\mathcal{O}^*(\alpha^n)$, так как на каждом шаге присутствует ещё полиномиальное время на выбор нужного случая и, возможно, модификацию графа. Сейчас мы сосредоточены на минимизации α, про полином не думаем.

• **Разбор случаев** $deg[v] \geqslant 3$

3a. $\forall v\ deg[v] = 3 \Rightarrow$ после расщепления сразу попадаем в случай (2): $\exists v\ deg[v] = 2$,
итого $T(n) \leqslant T(n-1-2) + T(n-1-3-2) = [3, 6] \leqslant 1.174^n$. На самом деле появится сразу много вершин степени 2, оценку можно сильно улучшить, нам хватит такой.
3b. $\exists v\ deg[v] = 3 \Rightarrow$ по непрерывности $\exists u\ deg[u] \geqslant 4 \wedge$ у u есть сосед степени 3. Расщепляемся по u: $T(n) \leqslant T(n-1-2) + T(n-1-deg[u]) = [3, 5] \leqslant 1.194^n$. Из случаев (3b) и (3a) хуже (3b) $\Rightarrow$ если появляются вершины степени 3, можно считать, что это именно (3b).
4a. $\forall v\ deg[v] = 4 \Rightarrow$ после расщепления сразу попадаем в случай (3*b*): $\exists v\ deg[v] = 3$, итого $T_{4a}(n) \leqslant T_{3b}(n-1) + T_{3b}(n-5) = [1+3, 1+5, 5+3, 5+5] = [4, 6, 8, 10] \leqslant 1.239^n$.
4b. $\exists v\ deg[v] = 4 \Rightarrow$ по непрерывности $\exists u\ deg[u] \geqslant 5 \wedge$ у u есть сосед степени 4. Расщепляемся по u: $T_{4b}(n) \leqslant T_{3b}(n-1) + T(n-6) = [4, 6, 6] \leqslant 1.2335^n$.
5a. $\forall v\ deg[v] = 5 \Rightarrow$ после расщепления сразу попадаем в случай (4*b*): $\exists v\ deg[v] = 4$, итого $T(n) \leqslant T_{4b}(n-1) + T_{4b}(n-6) = [5, 7, 7, 10, 12, 12] \leqslant 1.2487^n$.
Поскольку $T_{4a} \geqslant T_{4b}$, важно, что именно 4*b*, это точно так, если $n > 1+5+25$, то есть, не $\mathcal{O}(1)$.
5b. $\exists v\ deg[v] = 5 \Rightarrow$ по непрерывности $\exists u\ deg[u] \geqslant 6 \wedge$ у u есть сосед степени 5. Расщепляемся по u: $T(n) \leqslant T_{4b}(n-1) + T(n-7) = [5, 7, 7, 7] \leqslant 1.2413^n$.
6a. $\forall v\ deg[v] = 6 \Rightarrow$ после расщепления сразу попадаем в случай (5*b*): $\exists v\ deg[v] = 5$, итого $T(n) \leqslant T_{5b}(n-1) + T_{5b}(n-7) = [6, 8, 8, 8, 12, 14, 14, 14] \leqslant 1.24499^n$.
7. $\exists v\ deg[v] \geqslant 7 \Rightarrow T(n) \leqslant T(n-1) + T(n-8) = [1, 8] \leqslant 1.23206^n$

Итого время работы $= \max\{T_{3a}, T_{3b}, T_{4a}, T_{4b}, T_{5a}, T_{5b}, T_{6a}, T_7\} = T_{5a} = [5, 7, 7, 10, 12, 12] \leqslant \boxed{1.2487^n}$
P.S. Заметим, что $[1, 7] \approx 1.2554^n \Rightarrow$ все части анализа необходимы.

• **Алгоритм**

Анализ сложен. А алгоритм прост. Выбираем ребро (a, b): $deg[a] = \min, deg[b] = \max$.
В первую очередь минимизируем $deg[a]$, при равных максимизируем $deg[b]$).
Если $deg[a] \leqslant 2$, применяем жадность, иначе расщепляемся по b.

Наша оценка 1.2487^n не точна, её можно улучшать и улучшать. Например, на самом деле $T_{5a} \approx \max(T_{4b}, T_{4a}, T_{3b}, T_{4a})$, так как в обеих ветках рекурсии мы очень не скоро получим что-то кроме этих четырёх случаев. Но получим, в (2) степени увеличиваются, $deg_{ab} = deg_a + deg_b - 2$.

1.3. (*) Оценка с использованием весов

Повторим анализ. Сперва более простую версию.
1. Если $\exists$ вершина степени 1 её выгодно жадно взять в IS.
2. $\forall v\ deg[v] = 2 \Rightarrow$ пути и циклы, задача решается жадно.
3. $\exists v\ deg[v] \geqslant 3 \Rightarrow T(n) = T(n-1) + T(n-deg-1) \leqslant T(n) = T(n-1) + T(n-4) \leqslant 1.3803^n$

Новая идея: сопоставим вершинам разной степени разные веса.

$w_1 = 0, w_2 = 0.5, w_3 = 1, w_4 = 1, \dots$ (веса больших степеней тоже 1).
Для вершин степени 1 мы по-прежнему планируем запускать жадность.

Заметим:
a) $k = \sum_v w_{deg[v]}[v] \leqslant n$
b) $k = 0 \Rightarrow$ граф пуст, ответ известен
c) $w_i \leqslant w_{i+1} \Rightarrow$ после любых расщеплений $k \searrow$

Заменим число вершин n на суммарный вес вершин k. $k \leqslant n \Rightarrow T(k) \leqslant T(n)$.

3. $\max_v deg[v] = 3 \Rightarrow$ расщепляемся по 3-ке. Степень её соседей уменьшится, их вес уменьшится на 0.5 и в случае $3 \to 2$, и в случае $2 \to 1$. В ветке с удалением соседей то же произойдёт с их соседями. Итого $T(k) \leqslant T(k - 1 - 3{\cdot}0.5) + T(k - 1 - 5{\cdot}0.5) = [2.5, 3.5] \leqslant 1.263^k$.
4. $\max_v deg[v] = 4 \Rightarrow$ расщепляемся по 4-ке. Разберём 2 случая.
Все соседи 3-ки: $T(k) \leqslant T(k - 1 - 4{\cdot}0.5) + T(k - 1 - 4{\cdot}0.5) = [3, 3] \leqslant 1.25993^k$.
Все соседи 4-ки: $T(k) \leqslant T(k - 1) + T(k - 1 - 4) = [1, 5] \leqslant 1.3248^k$.
Случаев больше, их разбор здесь не представлен, худшим из них будет «все 4-ки».

• Применяем идею весов по полной.

1. Если $\exists$ вершина степени 1 её выгодно жадно взять в IS.
2a. Если $\exists$ две смежных вершины a, b степени 2, заменим $u - a - b - v$ на $u - v$.
2b. Если $\exists$ вершина степени 2, удалим её, стянем её соседей.
Если их степени были 3, 3, новая вершина имеет степень $3{+}3{-}2 = 4$.

Подберём теперь веса. Самые простые: $w_1 = w_2 = 0, w_3 = 0.8, w_4 = 1, w_5 = 1, \dots$
Заметим, что попадая в состояние $\exists v\ deg[v] \leqslant 2$, мы сразу уменьшим k хотя бы на $2w_3 - w_4 = 0.6$.

3a. $\forall v\ deg[v] = 3 \Rightarrow T(k) \leqslant T_2(k - 4 \cdot w_3) + T_2(k - 6 \cdot w_3) = [3.2 + 0.6, 4.8 + 0.6] = [3.8, 5.4] \leqslant 1.165^k$.
3b. $\exists v\ deg[v] = 3$ и максимальный сосед тройки это 4. Расщепляемся по соседу. Случаи.
Соседи 3333: ...
Соседи 3444: ...
3c. $\exists v\ deg[v] = 3$ и максимальный сосед $\geqslant 5$. Расщепляемся по соседу. Случаи.
Соседи 33333: ...
Соседи 34444: ...
Соседи 35555: ...
4. ...

Как вы понимаете, разбор случаев весьма трудоёмкий. Может быть, его можно как-то запрограммировать? =)

Ссылки: [wiki] [Robson'2001, 1.1888^n]

Кстати, $1.1888^n \approx 2^{n/4}$, то есть, для $n \approx 100$ алгоритм точно не плох.

Лекция #2: Рандомизированные алгоритмы

4 февраля 2024

2.1. Случайные числа в C++. Немного про rand().

Начиная с C++11 для генерации случайных чисел принято использовать вихрь Мерсенна:

```
1 mt19937 gen;
2 for (int i = 0; i < n; i++) // n целых чисел из [0, M)
3     a[i] = gen() % M; // gen() даст 32-битное целое случайное число.
```

Более подробный пример про случайные числа в C++ сможно посмотреть здесь.

Обратите внимание, что функциями `rand`, `srand`, `random_shuffle` уже давно не пользуются. Одна из причин: `rand()` возвращает псевдослучайное число от `0` до `RAND_MAX`, под Windows `RAND_MAX` $= 2^{15} - 1 = 32767$, что слишком мало, а в Linux `RAND_MAX` $= 2^{31} - 1$, что убивает потенциальную кроссплотформенность.

2.2. Определения: RP, coRP, ZPP

Рандомизированными называют алгоритмы, использующие случайные биты.

Первый тип алгоритмов: «решающие decision задачи, работающие всегда за полином, ошибающиеся в одну сторону». Строго это можно записать так:

Def 2.2.1. $\mathrm{RP} = \{L \colon \exists M \in \mathrm{PTime} \begin{cases} x \notin L \Rightarrow M(x,y) = 0 \\ x \in L \Rightarrow \Pr_y[M(x,y) = 1] \geqslant \frac{1}{2} \end{cases} \}$

x – вход, y – подсказка из случайных бит. Расшифровка: RP = randomized polynomial time. То есть, если $x \notin L$, M не ошибается, иначе работает корректно с вероятностью хотя бы $\frac{1}{2}$. Если для какого-то y алгоритм M вернул 1, то это точно правильный ответ, $x \in L$.

Def 2.2.2. $\mathrm{coRP} = \{L \colon \exists M \in \mathrm{PTime} \begin{cases} x \in L \Rightarrow M(x,y) = 1 \\ x \notin L \Rightarrow \Pr_y[M(x,y) = 0] \geqslant \frac{1}{2} \end{cases} \}$

Если для какого-то y алгоритм M вернул 0, то это точно правильный ответ, $x \notin L$.

• **Сравнение классов** NP, RP

Если ответ 0, оба алгоритма для $\forall$ подсказки выдадут 0. Если ответ 1, для NP-алгоритма $\exists$ хотя бы одна подсказка, а RP-алгоритм должен корректно работать хотя бы на половине подсказок.

• **Понижение ошибки**

Конечно, алгоритм, ошибающийся с вероятностью $\frac{1}{2}$ никому не нужен.

Lm 2.2.3. *Пусть M – RP-алгоритм, ошибающийся с вероятностью p. Запустим его k раз, если хотя бы раз вернул 1, вернём 1. Получили алгоритм с ошибкой p^k.*

Например, если повторить 100 раз, получится вероятность ошибки $2^{-100} \approx 0$.

Если есть алгоритм, *корректно* работающий с близкой к нулю вероятностью p, ошибающийся с вероятностью $1 - p$, то повторив его $\frac{1}{p}$ раз, получим вероятность ошибки $(1-p)^{1/p} \leqslant e^{-1}$.

• ZPP (zero-error probabilistic polynomial time)

ZPP – класс задач, для которых есть никогда не ошибающийся вероятностный алгоритм с полиномиальным матожиданием временем работы.

Def 2.2.4. $\text{ZPP} = \{L\colon \exists P(n),\ M \textit{ такие, что } E_y[Time(M(x,y))] \leqslant P(|x|)\}$

> Про алгоритмы и для RP, и для ZPP говорят «вероятностные/рандомизированные».
> Чтобы подчеркнуть принадлежность классу, уточняют ошибку.
> RP-алгоритм обладает односторонней ошибкой. ZPP-алгоритм работает без ошибки.

Мы определили основные классы только для задач распознавания, только для полиномиального времени. Можно определить аналогичные классы для задач поиска и для любого времени.

Важно помнить, что в RP, coRP, ZPP алгоритмы работают на *всех* тестах.
Например, вероятность ошибки в RP *для любого теста* не более $\frac{1}{2}$.

2.3. Примеры

Все наши примеры про поиск и не «за полином», а гораздо быстрее.

• Часто встречающееся число

Поиск числа, которое встречается в массиве больше половины раз.
У решения этой задачи есть две версии.
ZPP-версия: «*мы уверены, что такой элемент есть, пробуем случайные, пока не попадём*».
RP-версия: «*хотим определить, есть ли такой элемент в массиве, делаем одну пробу*».

• Поиск квадратичного невычета

Для любого простого p ровно $\frac{p-1}{2}$ ненулевых остатков обладают свойством $a^{\frac{p-1}{2}} \equiv -1 \bmod p$.
Такие a называются «квадратичными невычетами».

Задача: дано p, найти невычет.
Алгоритм: пока не найдём, берём случайное a, проверяем.
Время одной проверки — возведение в степень ($\log p$ для $p \leqslant 2^w$).
Сколько в среднем понадобиться проверок? $E = 1 + \frac{1}{2}E \Rightarrow E = 2$.
Итого ZPP-версия работает за $\mathcal{O}(\log p)$.

2.4. Проверка на простоту

• Тест Ферма

Малая теорема Ферма: $\forall p \in \mathbb{P}\ \forall a \in [1..p-1]\ \ a^{p-1} \equiv 1 \bmod p$.
Чтобы проверить простоту p, можно взять случайное a и проверить $a^{p-1} \equiv 1 \bmod p$

Lm 2.4.1. $\exists a\colon a^{p-1} \not\equiv 1 \bmod p \Rightarrow$ *таких* a *хотя бы* $\frac{p-1}{2}$.

Доказательство. Пусть b такое, что $b^{p-1} \equiv 1 \bmod p \Rightarrow (ab)^{p-1} = a^{p-1}\,b^{p-1} \not\equiv 1 \bmod p$. ■

Мы показали, что если тест вообще работает, то с вероятностью хотя бы $\frac{1}{2}$.
К сожалению, есть составные числа, для которых тест вообще не работает – числа Кармайкла.

Утверждение 2.4.2. У числа Кармайкла a есть простой делитель не более $\sqrt[3]{a}$.

Получаем следующий всегда корректный вероятностный алгоритм:
проверим возможные делители от 2 до $\sqrt[3]{a}$, далее тест Ферма.

• **Тест Миллера-Рабина**

```
1  bool isPrime(int p):
2      // p − 1 = 2^s t,  t -- нечётно
3      if (p < 2) return 0; // 1 и 0 не простые
4      p → (s, t)
5      a = randInt(2, p-1) // [2, p-1]
6      g = pow(a, t, p) // возвели в степень по модулю
7      for (int i = 0; i < s; i++) {
8          if (g == 1) return 1;
9          if (g % p != -1 && g * g % p == 1) return 0;
10         g = g * g % p;
11     }
12     return g == 1; // тест Ферма
```

В строке 8 мы проверяем, что нет корней из 1 кроме 1 и -1.
Время работы $\mathcal{O}(\log p)$ операций «умножение по модулю».

Утверждение 2.4.3. $\forall a$ вероятность ошибки не более $\frac{1}{4}$.

• **Изученные алгоритмы**

(*) k-я порядковая статистика учит нас:
«*вместо того, чтобы выбирать медиану, достаточно взять случайный элемент*».
(*) 3-LIST-COLORING: вычеркнуть из каждого клоза случайный элемент, получить 2-SAT. Вероятность успеха этого алгоритма $\frac{2^n}{3^n}$, чтобы получить вероятность $\frac{1}{2}$, нужно было бы повторить его $1.5^n \ln 2$ раз, поэтому к RP-алгоритмам он не относится.
Ещё про него полезно понимать «*работает на* 2^n *подсказках из* 3^n *возможных*».

• (*) **Проверка** $AB = C$

Даны три матрицы $n \times n$, нужно за $\mathcal{O}(n^2)$ проверить равенство $AB = C$. Все вычисления в $\mathbb{F}_2$. Вероятностный алгоритм генерирует случайный вектор x и проверяет $A(Bx) = Cx$.

Lm 2.4.4. *Вероятность ошибки не более* $\frac{1}{2}$.

Доказательство. Пусть $AB \neq C \Rightarrow D = AB - C \neq 0$.
Посчитаем $\Pr_x\left[A(Bx) = Cx\right] = \Pr_x\left[Dx = 0\right]$. $\exists i, j\colon D_{i,j} \neq 0$. Зафиксируем произвольный x. $(Dx)_i = D_{i,0}x_0 + \cdots + D_{i,j}x_j + \cdots + D_{i,n-1}x_{n-1}$. $D_{i,j} \neq 0 \Rightarrow$ меняя значение x_j, меняем значение $(Dx)_i \Rightarrow \Pr_x\left[(Dx)_i = 0\right] = \frac{1}{2} \Rightarrow \Pr_x\left[Dx = 0\right] \leqslant \frac{1}{2}$. ■

2.5. ZPP = RP ∩ coRP

Lm 2.5.1. *Неравенство Маркова:* $x \geqslant 0 \Rightarrow \forall a\ \Pr[x > aE(x)] < \frac{1}{a}$

Доказательство. Пусть $\Pr[x > aE(x)] \geqslant \frac{1}{a}$, тогда матожидание должно быть больше чем произведение $aE(x)$ и $\frac{1}{a}$ (так как «есть хотя бы $\frac{1}{a}$ x-ов со значением $aE(x)$). Формально: $\Rightarrow E(x) > \big(aE(x)\big) \cdot \frac{1}{a} + 0 \cdot (1 - \frac{1}{a}) = E(x)$?!? ■

Следствие 2.5.2. $x \geqslant 0 \Rightarrow \forall a\ \Pr[x \leqslant aE(x)] \geqslant \frac{1}{a}$

Теорема 2.5.3. ZPP = RP ∩ coRP

Доказательство. Пусть $L \in$ ZPP $\Rightarrow \exists$ алгоритм M, работающий в среднем за $P(n)$ $\overset{Cons\ 2.5.2}{\Rightarrow}$

$\Pr[Time(M) \leqslant 2P(n)] \geqslant \frac{1}{2} \Rightarrow$ запустим M с будильником на $2P(n)$ операций. Если алгоритм завершился до будильника, он даст верный ответ, иначе вернём $0 \Rightarrow$ RP или $1 \Rightarrow$ coRP.

Пусть $L \in \text{RP} \cap \text{coRP} \Rightarrow \exists M_1(\text{RP}), M_2(\text{coRP})$, работающие за полином, ошибающиеся в разные стороны, которые можно запускать по очереди $(M_1 + M_2)$. Пусть $x \notin L \Rightarrow M_1(x) = 0$, $\Pr[M_2(x) = 0] \geqslant \frac{1}{2}$. Сколько раз нужно в среднем запустить $M_1 + M_2$? $E \geqslant 1 + \frac{1}{2}E \geqslant 2$ (всегда запустим 1 раз, затем с вероятностью $\frac{1}{2}$ получим $M_2(x) = 1$ и повторим процесс с начала). ■

Lm 2.5.4. $\text{P} \subseteq \text{coRP} \cap \text{RP} = \text{ZPP} \subseteq \text{RP} \subseteq \text{NP}$

При этом строгие ли вложения неизвестно (про все три вложения).
Зато известна масса задач, для которых есть простое RP-решение, но неизвестно P-решение.

2.6. Двусторонняя ошибка, класс BPP

Def 2.6.1. $\text{BPP} = \{L\colon \exists M \in \text{PTime} \begin{cases} x \notin L \Rightarrow \Pr_y[M(x,y) = 0] \geqslant \alpha \\ x \in L \Rightarrow \Pr_y[M(x,y) = 1] \geqslant \alpha \end{cases}\}$, *где* $\alpha = \frac{2}{3}$.

Другими словами алгоритм всегда даёт корректный ответ с вероятностью хотя бы $\frac{2}{3}$.
Для BPP тоже можно понижать ошибку: запустим n раз, выберем ответ, который чаще встречается (majority). На самом деле α в определении можно брать сколь угодно близким к $\frac{1}{2}$.

Lm 2.6.2. *О понижении ошибки. Пусть* $\beta > 0, \alpha = \frac{1}{2} + \varepsilon,\ \varepsilon > 0 \Rightarrow \exists n = poly(\frac{1}{\varepsilon})$ *такое, что повторив алгоритм* n *раз, и вернув majority, мы получим вероятность ошибки не более* β.

(*) Доказательство. Пусть $x \in L$, мы повторили алгоритм $n = 2k$ раз.
Если получили 1 хотя бы $k{+}1$ раз, вернём 1, иначе вернём 0.
Вероятность ошибки $\texttt{Err} = \sum\limits_{i=0}^{i=k} p_i$, где p_i – вероятность того, что алгоритм ровно i раз вернул 1.
Выпишем в явном виде формулу для p_i, при этом удобно отсчитывать i от середины: $i = k - j$.

$$p_i = p_{k-j} = \tbinom{n}{i}(\tfrac{1}{2} + \varepsilon)^{k-j}(\tfrac{1}{2} - \varepsilon)^{k+j} = \tbinom{n}{i}(\tfrac{1}{4} - \varepsilon^2)^k(\tfrac{1/2-\varepsilon}{1/2+\varepsilon})^j \leqslant \tbinom{n}{i}(\tfrac{1}{4} - \varepsilon^2)^k$$

$$\texttt{Err} = \sum_{i=0}^{i=k} p_i \leqslant \sum_{i=0}^{i=k} \binom{n}{i}(\tfrac{1}{4} - \varepsilon^2)^k = \tfrac{1}{2}2^n(\tfrac{1}{4} - \varepsilon^2)^k = \tfrac{1}{2}(1 - 4\varepsilon^2)^k$$

Хотим, чтобы ошибка была не больше β, для этого возьмём $k = (1/4\varepsilon^2) \cdot \ln\frac{1}{\beta} = poly(\frac{1}{\varepsilon})$. ■

Замечание 2.6.3. Люди не знают, кто больше BPP или NP. Из $\text{NP} \subset \text{BPP}$ следует плохое, в это люди не верят. [[zoo]] [[wiki]]

2.7. Как ещё можно использовать случайные числа?

• Идеальное кодирование

Алиса хочет передать Бобу некое целое число x от 0 до $m - 1$. У них есть общий ключ r от 0 до $m-1$, сгенерированный равномерным случайным распределением. Тогда Алиса передаст по открытому каналу $y = (x + r) \bmod m$. Знание **одного** такого числа не даст злоумышленнику ровно никакой информации. Боб восстановит $x = (y - r) \bmod m$.

• Вычисления без разглашения

В некой компании сотрудники стесняются говорить друг другу, какая у кого зарплата.
Зарплату i-го обозначим x_i. Сотрудники очень хотят посчитать среднюю зарплату $\Leftrightarrow$ посчитать сумму $x_1 + \cdots + x_n$. Для этого они предполагают, что сумма меньше $m = 10^{18}$, и пользуются следующим алгоритмом:

0. Первый сотрудник генерирует случайное число $r \in [0, m)$.
1. Первый сотрудник передаёт второму число $(r + x_1) \bmod m$.
2. Второй сотрудник передаёт третьему число $(r + x_1 + x_2) \bmod m$.
3. ...

n. Последний сотрудник передаёт первому $(r + x_1 + x_2 + \cdots + x_n) \bmod m$.

$*$. Первый, как единственный знающий r, вычитает его и говорит всем ответ.

• Приближения

На практике будут разобраны $\frac{1}{2}$-OPT приближения для MAX-3-SAT и VERTEX-COVER.

2.8. Парадокс дней рождений. Факторизация: метод Полларда

«В классе 27 человек $\Rightarrow$ с большой вероятностью у каких-то двух день рождения в один день»
Пусть p_k – вероятность, что среди k случайных чисел от 1 до n все различны. Оценим p_k снизу.

Lm 2.8.1. $1 - p_k \leqslant \frac{k(k-1)}{2n}$

Доказательство. $f = \#\{(i,j)\colon x_i{=}x_j\}$, $1 - p_k = \Pr[f{>}0] \leqslant E_k[f] = \frac{k(k-1)}{2} \cdot \Pr[x_i{=}x_j] = \frac{k(k-1)}{2}\frac{1}{n}$ ■

Следствие 2.8.2. При $k = o(\sqrt{n})$ получаем $1 - p_k = o(1) \Rightarrow$ с вероятностью ≈ 1 все различны.

Теперь оценим p_k при $k = \sqrt{n}$.

Lm 2.8.3. $k = \sqrt{n} \Rightarrow 0.4 \leqslant p_k \leqslant 0.8$

Доказательство. $p_k = \underbrace{\frac{n}{n} \cdot \frac{n-1}{n} \cdot \ldots \cdot \frac{n-\sqrt{n}/2}{n}}_{(n-\sqrt{n}/2)/n \leqslant x \leqslant 1} \cdot \underbrace{\frac{n-\sqrt{n}/2-1}{n} \cdot \ldots \cdot \frac{n-\sqrt{n}}{n}}_{(n-\sqrt{n})/n \leqslant x \leqslant (n-\sqrt{n}/2)/n} \Rightarrow \left(\frac{n-\sqrt{n}/2}{n}\right)^{\sqrt{n}/2}\left(\frac{n-\sqrt{n}}{n}\right)^{\sqrt{n}/2} \leqslant$
$p_k \leqslant (1)^{\sqrt{n}/2}\left(\frac{n-\sqrt{n}/2}{n}\right)^{\sqrt{n}/2} \Rightarrow e^{-1/4}e^{-1/2} \leqslant p_k \leqslant e^{-1/4} \Rightarrow 0.4723 \leqslant p_k \leqslant 0.7788$ ■

$\forall k$ можно оценить p_k гораздо точнее:

$p_k = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k+1}{n} = \frac{n!}{n^k(n-k)!} \approx \frac{(n/e)^n}{n^k((n-k)/e)^{n-k}} = \frac{1}{((n-k)/n)^{n-k}e^k} = \frac{1}{(1-k/n)^{n-k}e^k} \approx \frac{1}{(e^{-1})^{(n-k)k/n}e^k}$

Первое приближение сделано по формуле Стирлинга, второе по замечательному пределу.

• Поллард

Пусть у n минимальный делитель p, тогда $p \leqslant \sqrt{n}$. Сгенерируем $\sqrt[4]{n} \geqslant \sqrt{p}$ случайных чисел.
По парадоксу дней рождений какие-то два (a и b) дадут одинаковый остаток по модулю p.
При этом с большой вероятностью у a и b разный остаток по модулю $n \Rightarrow gcd(a-b, n)$ – нетривиальный делитель n. Осталось придумать, как найти такую пару a, b.

```
1 x = random [2..n-1]
2 k = pow(n, 1.0 / 4)
3 for i=0..k-1: x = f(x)
```

Здесь функция f – псевдорандом на $[0..n)$.
Например (без док-ва) нужными нам свойствами обладает функция $f(x) = (x^2 + 1) \bmod n$.
Так мы получили x, теперь переберём y.

```
1 y = f(x)
2 for i=0..k-1:
3     g = gcd(x - y, n)
4     if (g != 1 && g != n) return g
5     y = f(y)
```

Почему так можно? Рассмотрим последовательность $x_i = f^i(x_0)$. Она зацикливается по модулю p, что выглядит как буква «ρ». Наша цель — найти две точки друг над другом. Сделав достаточно много шагов из стартовой точки окажемся на цикле. После чего остаётся пройти ещё один раз весь цикл.

Полученный нами алгоритм не работает на маленьких n. Все такие n можно проверить за $n^{1/2}$. Также стоит помнить, что вероятность его успеха $\approx \frac{1}{2}$, поэтому для вероятности ≈ 1 всю конструкцию нужно запустить несколько раз.

Сейчас асимптотика – $\mathcal{O}(n^{1/4} \cdot T(gcd))$.

Чтобы получить чистое $\mathcal{O}(n^{1/4})$, воспользуемся тем, что $gcd(a,n) = 1 \wedge gcd(b,n) = 1 \Rightarrow gcd(ab \bmod n, n) = 1 \Rightarrow$ можно вместо `gcd(x-y,n)` смотреть на группы по $\log n$ разностей: `gcd(`$\prod_i$`(x-y`$_i$`),n)`, и только если какой-то `gcd` $\neq 1$, проверять каждый `y`$_i$ отдельно.

2.9. 3-SAT и random walk

• Детерминированное решение за $3^{n/2}$

Пусть решение X^* существует и в нём нулей больше чем единиц. Начнём с $X_0 = \{0, 0, \ldots, 0\}$, чтобы из X_0 попасть в X^* нужно сделать не более $\frac{n}{2}$ шагов. Если X_i не решение, то какой-то клоз не выполнен, значит в X^* одна из трёх переменных этого клоза имеет другое значение. Переберём, какая. Получили рекурсивный перебор глубины $\frac{n}{2}$, с ветвлением 3. Если в X^* единиц больше, начинать нужно с $X_0 = \{1, 1, \ldots, 1\}$. Нужно перебрать оба варианта.

• Рандомизированное решение за $3^{n/2}$

Упростим предыдущую идею: начнём со случайного X_0. Матожидание расстояния от ответа равно $\mathsf{E} = n/2$. с вероятностью $\frac{1}{2}$ он на расстоянии $\leqslant \frac{1}{2}n$ от ответа. — это неверно Сделаем $\frac{1}{2}n$ шагов, выбирая каждый раз случайное из трёх направлений. На каждом шаге с вероятностью $\frac{1}{3}$ мы приближаемся к ответу. Итого с вероятностью $\frac{1}{n+1} \cdot \frac{1}{3^{n/2}}$ мы придём в ответ. Повторим $\mathcal{O}(3^{n/2}(n+1)) \approx \mathcal{O}(1.73^n)$ раз.

• Рандомизированное решение за 1.5^n

Почти такой же алгоритм. Сделаем теперь не $\frac{n}{2}$, а n шагов.

В процессе доказательства нам пригодится знание $\forall \alpha \ \sum_k (\binom{n}{k}\alpha^k) = (1+\alpha)^n$.
Доказательство: раскроем скобочки.

Анализ вероятностей. Если перебор за $3^{n/2}$ перебирал все варианты, то мы перебираем 1 вариант и угадываем с вероятностью $\geqslant p = \sum_k Pr[k]\frac{1}{3^k}$, где k – расстояние Хэмминга от X_0 до X^*, а $Pr[k] = \binom{n}{k}/2^n$ – вероятность того, что при выборе случайного X_0 расстояние равно k.
$p = \sum_k \frac{1}{2^n}\binom{n}{k}\frac{1}{3^k} = \frac{1}{2^n}(1+\frac{1}{3})^n = (\frac{2}{3})^n \Rightarrow$ повторим процесс 1.5^n раз.
Кстати, поскольку $Pr[k \leqslant \frac{n}{2}] = \frac{1}{2}$, достаточно делать даже не n, а $\frac{n}{2}$ шагов.

• Рандомизированное решение за 1.334^n

Schoning's algorithm (1999). Получается из предыдущего решения заменой $\frac{n}{2}$ шагов на $3n$ шагов. Вероятность успеха будет не менее $(3/4)^n \Rightarrow$ время работы $\mathcal{O}^*(1.334^n)$. Доказательство. Статья.
В той же статье дан более общий результат для k-SAT: $(\frac{2}{1+1/(k-1)})^n = (\frac{2(k-1)}{k})^n$.

Теорема 2.9.1. *Вероятность успеха* $(3/4)^n$

Доказательство. В прошлой серии мы находились на расстоянии k от ответа и хотели дви-

гаться всё время к ответу. А теперь посчитаем вероятность p того, что мы за первых $3k$ шагов сделали k шагов в неверную сторону, $2k$ шагов в верную сторону (итого пришли в ответ).

$$p = \binom{3k}{k} (\tfrac{1}{3})^{2k} (\tfrac{2}{3})^{k}$$

Приблизим $n! \approx \left(\frac{n}{e}\right)^n \Rightarrow \binom{3k}{k} \approx \frac{(3k)^{3k}}{(2k)^{2k}k^k} = \left(\frac{27}{4}\right)^k \Rightarrow p \approx \left(\frac{27}{4}\right)^k (\frac{1}{9})^k (\frac{2}{3})^k = (\frac{1}{2})^k$.

Последний шаг такой же, как в 1.5^n: $Pr[success] = 2^{-n} \sum_k \binom{n}{k} 2^{-k} = 2^{-n}(1+\frac{1}{2})^n = \left(\frac{3}{4}\right)^n$. ■

Лучшее сегодня решение 3-SAT: алгоритм PPSZ за 1.308^n от Paturi, Pudlak, Saks, Zani (2005).

• Поиск хороших кафешек

Пусть Вы в незнакомом городе, хотите найти хорошее кафе. Рассмотрим следующие стратегии:

1. Осматривать окрестность в порядке удаления от начальной точки
2. Random walk без остановок
3. Random walk на фиксированную глубину с возвратом

Первый, если Вы начали в не очень богатом на кафе районе не приведёт ни к чему хорошему. У второго будут проблемы, если в процессе поиска вы случайно зайдёте в промышленный район. Третий же в среднем не лучше, зато минимизирует риски, избавляет нас от обеих проблем. Для решения 3-SAT мы использовали именно третий вариант.

Замечание 2.9.2. Random Walk помогает находить гамильтонов путь в случайных графах.

Angluin,Valiant'1979 (`[AV79]`): (random walk + posa-rotation) за $\mathcal{O}(n \log^2 n)$ дают гамильтонов путь для случайных графов, где $m \geqslant C \cdot n \log n$.
В 2021-м идею довели до $\mathcal{O}(n)$ `[Nenadov'Steger'Su]`, в этой же статье есть описание `[AV79]`.

2.10. Лемма Шварца-Зиппеля

Lm 2.10.1. *Пусть дан многочлен P от нескольких переменных над полем $\mathbb{F}$.*

$$(P \not\equiv 0) \Rightarrow \Pr_x[P(x) = 0] \leqslant \tfrac{degP}{|\mathbb{F}|}$$

Доказательство. Индукция по числу переменных.
База: многочлен от 1 переменной имеет не более $\deg P$ корней. Переход: wiki. ■

Следствие 2.10.2. Задача проверки тождественного равенства многочлена нулю $\in$ coRP.

Доказательство. Подставим случайный x в поле $\mathbb{F}_q$, где (q – простое) $\wedge$ ($q > 2 \deg P$). ■

• (*) Совершенное паросочетание в произвольном графе

Дан неорграф, заданный матрицей смежности c. Нужно проверить, есть ли в нём совершенное паросочетание.

Def 2.10.3. *Матрица Тамма T:* $T_{ij} = -T_{ji}, T_{ij} = \begin{cases} 0 & c_{ij} = 0 \\ x_{ij} & c_{ij} = 1 \end{cases}$

Здесь x_{ij} – различные переменные. Пример: $c = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} 0 & x_{12} & x_{13} \\ -x_{12} & 0 & 0 \\ -x_{13} & 0 & 0 \end{bmatrix}$

Теорема 2.10.4. $\det T \not\equiv 0 \Leftrightarrow \exists$ *совершенное паросочетание*

Теорему вам докажут в рамках курса дискретной математики, нам же сейчас интересно, как проверить тождество. Если бы матрица состояла из чисел, определитель можно было бы посчитать Гауссом за $\mathcal{O}(n^3)$. В нашем случае определитель – многочлен степени n. Подставим во все x_{ij} случайные числа, посчитаем определитель по модулю $p = 10^9 + 7$. Получили вероятностный алгоритм с ошибкой $\frac{n}{p}$ и временем $\mathcal{O}(n^3)$.

Следствие 2.10.5. Мы умеем искать размер максимального паросочетания за $\mathcal{O}(n^3 \log n)$

Доказательство. Бинпоиск по ответу. Чтобы проверить, есть ли в графе паросочетание размера хотя бы k рёбер ($2k$ вершин), добавим $n-2k$ фиктивных вершин, которые соединены со всеми и проверим в полученном графе наличие совершенного паросочетания за $\mathcal{O}(n^3)$. ■

• Равенство мультимножеств

На практике изучим, как за $\mathcal{O}(n)$ времени и $\mathcal{O}(1)$ памяти проверить равенство двух мультимножеств. В будущем мы это ещё раз сделаем в теме «хеширование».

• (*) Гамильтонов путь

В 2010-м году в неорграфе научились проверять наличие гамильтонова пути за $\mathcal{O}^*(1.657^n)$. Можно почитать в оригинальной статье Бьёркланда (Björklund).

2.11. Random shuffle

```
1 void Shuffle(int n, int *a)
2     for (int i = 0; i < n; i++)
3         swap(a[i], a[rand() % (i+1)]);
```

Утверждение 2.11.1. После процедуры `Shuffle` все перестановки a равновероятные.

Доказательство. Индукция. База: после фазы $i = 0$ все перестановки длины 1 равновероятны. Переход: выбрали равновероятно элемент, который стоит на i-м месте, после этого часть массива $[0..i)$ по индукции содержит случайную из $i!$ перестановок. ■

• Применение

Если на вход даны некоторые объекты, полезно их первым делом перемешать.

Пример: построить бинарное дерево поиска, содержащее данные n различных ключей. Если добавлять их в пустое дерево в данном порядке, итоговая глубина может быть n, а время построения соответственно $\Theta(n^2)$. Если перед добавлением сделать `random shuffle`, то матожидание глубины дерева $\Theta(\log n)$, матожидание времени работы $\Theta(n \log n)$. Оба факта мы докажем при более подробном обсуждении деревьев поиска.

2.12. Дерево игры [stanford.edu]

Пусть есть игра, в которую играют два игрока. Можно построить дерево игры (возможно бесконечное) – дерево всех возможных ходов. Теперь игра – спуск по дереву игры. Для некоторых простых случаев хорошо известны «быстрые» способы сказать, кто выиграет.

• Игра на 0-1-дереве

Задача: в листьях записаны числа 0 и 1, первый победил, если игра закончилась в 1, иначе

победил второй. Пусть дерево – полное бинарное глубины n.

Решение: простой рандомизированный алгоритм «*сперва сделаем случайный ход, а второй сделаем только если не выиграли первым ходом*». Асимптотика $\mathcal{O}(1.69^n)$. Анализ в практике.

• Игра на OR-AND-дереве

Близкая по смыслу, но не игровая, а вычислительная задача. Дано полное бинарное дерево, в листьях 0 и 1. В промежуточных вершинах чередующиеся `AND` и `OR` гейты от детей.

Задача равносильна игре на 0-1-дереве: `AND` означает «чтобы первый выиграл, второй в обеих ветках должен проиграть», `OR` означает «чтобы второй проиграл, первый в одной из веток должен выиграть». Чередование `AND`/`OR` значит «ходят по очереди».

• Игра на min-max-дереве

Полное бинарное дерево. В листьях записаны $\mathbb{Z}$ числа из $[0, m)$.
Первый игрок максимизирует результат игры, второй минимизирует.

Решение #1: бинпоиск по ответу и игра на 0-1-дереве
Решение #2: (*) альфа-бета отсечение
Решение #3: (*) комбинация этих идей – алгоритм MTD-f (статья, wiki). MTD-f в своё время стал прорывом в шахматных стратегиях.

2.12.1. k-путь

Задача. Хотим найти k-путь – простой путь длины ровно k **вершин** из a в b. Для $n = k$ это ровно гамильтонов путь, мы умеем его искать за $2^n n$. При малых k мы пока не умеем ничего лучше перебора: n^k или точнее deg^k. В этом разделе мы предложим вероятностный алгоритм за $exp(k) \cdot poly(n)$.

• Алгоритм

Покрасим вершины в k цветов, каждую в случайный. Теперь за $\mathcal{O}(2^k E)$ найдём разноцветный k-путь, если он есть (динамика как для гамильтонова пути).
Если k-путь есть, покрасить его k^k способов, а разноцветно покрасить $k!$ способов.
Вероятность успеха $\frac{k!}{k^k} \geqslant e^{-k} \Rightarrow$ повторим $\mathcal{O}(e^k)$ раз. Итого: $\mathcal{O}^*((2e)^k)$.

2.13. (*) Квадратный корень по модулю

Задача: даны простое p и a, найти x: $x^2 = a \bmod p$.

Хорошо известны два решения: алгоритм Тоннелли-Шенкса'1973 и алгоритм Циполла'1907. Оба алгоритма вероятностны и требуют угадать квадратичный невычет.
В этой главе мы изучим только второй [1][2], как более быстрый и простой в реализации.

Алгоритм: возьмём $P(x) = (x + i)^{(p-1)/2} - 1$ и $A(x) = x^2 - a = (x - x_1)(x - x_2)$.
Посчитаем $gcd(P(x), A(x))$, если он содержит ровно один из $x - x_1$ и $x - x_2$, мы победили.

Теорема 2.13.1. *Вероятность успеха* $\geqslant \frac{1}{2}$. *Время работы* – $\mathcal{O}(\log n)$ *делений чисел порядка* p.

Корни существуют $\Leftrightarrow$ символ Лежандра $a^{(p-1)/2} = 1$. В реализации вместо gcd будем вычислять $R(x) = P(x) \bmod A(x)$ – возведение в степень с умножением похожим на комплексные числа:

$$(p_1 x + q_1)(p_2 x + q_2) = (p_1 q_2 + p_2 q_1)x + (p_1 p_2 a + q_1 q_2)$$

Пусть $R(x) = bx + c$, тогда при $b \neq 0$ пробуем корень $-cb^{-1}$.

```
1 def root(a, p):
2     if pow_mod(a, (p-1)/2, p) != 1: return -1 # символ Лежандра
3     while True:
4         i = random [0..p)
5         bx+c = pow_mod(x+i, (p-1)/2, x^2-a) # многочлен в степени по модулю
6         if b != 0:
7             z = -c/b
8             if z^2 = a \bmod p: return z
```

Доказательство Thm 2.13.1. Для начала посмотрим на символ Лежандра и заметим, что $(\alpha/\beta$ – невычет) $\Leftrightarrow$ (ровно один из $\{\alpha, \beta\}$ – вычет). Теперь исследуем корни многочлена из решения $P(x) = (x+i)^{(p-1)/2} - 1$. Ими являются такие z, что $(z-i)$ – квадратичный вычет. Мы хотим оценить вероятность того, что «ровно один из $\{x_1, x_2\}$ является корнем $P(x)$» $\Leftrightarrow$ «$(i-x_1)/(i-x_2)$ – невычет». $\forall x_1 \neq x_2$ $i \xrightarrow{f} \frac{i-x_1}{i-x_2} \Rightarrow f$ – биекция. Невычетов $(p-1)/2$. ■

Замечание 2.13.2. Если длина p равна n, то E(времени работы) при «умножении по модулю» за $\mathcal{O}(n \log n)$ получается $\mathcal{O}(n^2 \log n)$.

Замечание 2.13.3. Алгоритм можно обобщить на извлечение корня k-й степени.

Лекция #3: Модели вычислений

12 апреля 2024

3.1. RAM

Регистры – произвольные целые числа.
Какие операции разрешены? `move/add/sub/mul/div/ifzero/jump` и, конечно, `halt/in/out`
Остальное выражается через них, например цикл `for` n раз:

```
i = n
loop:
...
sub i 1
ifzero end // если результат последей операции 0, выйти из цикла
jump loop // перейти в начало цикла
:end
```

Важно, что память неограничена, к любой ячейке с целым номером i обращение за $\mathcal{O}(1)$. Обозначать можно, как $[i]$. Например $x = [i]$. В ячейках хранятся неоргранченные целые числа, что приводит к некоторому пространсту для оптимизаций: можно взять два массива целых чисел, закодировать из в два длинных целых числа, и сделать за $\mathcal{O}(1)$ операцию `+/-/OR/AND/XOR/</>` с массивами за $\mathcal{O}(1)$.

• Флойд в RAM за $\mathcal{O}(V^2)$

Например Флойд:

```
for k=1..n
    for i=1..n
        if d[i][k]
            for j=1..n
                d[i][j] |= d[k][j]
```

Можно заменить на

```
for k=1..n
    for i=1..n
        if ((d[i] >> k) & 1) == 1 // операция с числами, O(1)
            d[i] |= d[k] // операция с n-битными числами, O(1)
```

Аналогично `bfs` и `dfs` можно реализовать за $\mathcal{O}(V)$. См. практику.

• Сумма в массиве за $\mathcal{O}(\log n)$

Пусть массив = целое число. Тогда мы можем проделать тот же трюк, что и подсчёте числа единичных бит: $x \to y = (x \texttt{ AND } 01010101) + ((x \texttt{ >> } 1) \texttt{ AND } 01010101) \to z = (y \texttt{ AND } 00110011) + ((y \texttt{ >> } 2) \texttt{ AND } 00110011) \to \dots$ После i операций в группе из 2^i бит у нас записана их сумма. Индукционный шаг: сложить параллельно две соседние группы. Для подсчёта единичных бит трюк даёт $\mathcal{O}(\log w)$ битовых операций, для сложения чисел в массиве, закодированным, как целое числа $\mathcal{O}(\log n)$ арифметических операций в RAM.

3.2. RAM-w

Word RAM. RAM с ограничением «за $\mathcal{O}(1)$ арифметические операции происходят лишь с числами из $[0, 2^w)$, и ячейки памяти также w-битовые». w-битовое число = word = машинное слово. Мы привыкли к 64-битным компьютерами, которые, как раз RAM-w для $w = 64$.

• **Флойд в RAM за $\mathcal{O}(V^3/w)$**

Аналогичный код из раздела про RAM будет работать за $\mathcal{O}(V^3/w)$, так как n-битное число в RAM-w = bitset длины w, в котором биты разбиты на группы по w, и все операции за $\mathcal{O}(n/w)$. Заметим, мы везде предполагаем, что операции с числами от 1 до n происходят за $\mathcal{O}(1)$ $\Rightarrow$ $n < 2^w \Rightarrow \log n < w \Rightarrow w$ с точки зрения асимптотики w явно не константа. Форма записи $\mathcal{O}(V^3/\log V)$ тоже корректна, но $\mathcal{O}(V^3/w)$ – более точно и понятно.

Лекция #4: Модели вычислений и диагонализация

11 апреля 2024

4.1. Машина Тьюринга

[wiki] [итмо-конспекты]

Зададим какой-то конечный алфавит Σ. Машина Тьюринга состоит из трёх лент, где одна лента состоит из бесконечной последовательности символов, проиндексированных натуральными числами. Первая лента для входных данных, вторая для выходных, а третья рабочая, на которую можно писать что угодно. Также у самой машины есть состояние q из конечного множества Q. Обычно это состояние кодирует «позицию в программе» или какое-то константное число бит памяти.

В каждый момент у машины Тьюринга есть три головки, по одной на каждую ленту. Головка указывает на конкретный символ. За один шаг машина Тьюринга читает все символы под головками и на основании этого принимает решение сдвигать ли головки, и какие символы стоит записать под их старым положением.

Формально, задано какое-то конечное множество состояний Q машины Тьюринга, а шаг вычисления описывается функцией:

$$\delta \colon Q \times \Sigma^3 \to Q \times \Sigma^3 \times \{Left, Stay, Right\}^3$$

Т.е. по состоянию и символам принимаем решение, что делать дальше.

Возможно это неинтуитивно, но в такой модели можно запрограммировать любую программу, которую можно было написать для «обычного» компьютера, причём получится разве что в полином хуже.

Это верно и для других разумных моделей, так что получается, что вне зависимости от рассматриваемой модели классы, `P`, `NP` выражают одинаковое множество алгоритмов/задач.

4.2. Недетерминизм и рандом

Как бы формализовать класс `NP`? Можно добавить ещё одну ленту для y. А можно определить недетерминированную машину тьюринга.

Она будет отличаться от обычной тем, что функций перехода будет две: δ_0 и δ_1, и алгоритм как бы недетерминированно делает переход в лучшей из них (формально, в decision задаче, машина допускает слово x если есть хотя бы один способ выбирать переход каждый раз).

Несложно видеть, что это определение эквивалентно старому: если есть подсказка, то её можно интерпретировать как выбор по каким δ делать переходы. В другую сторону: в начале сделаем достаточно много шагов вычислений, которые просто записывают полученный недетерминизм в подсказку y.

Как бы формализовать рандомизированные вычисления? Да также. Хотя в случае рандома обычно говорят о вероятностной ленте. Вероятностная лента это бесконечная лента заполненная случайными битами. Каждый раз, когда нужно немного случайности, можно её просто читать вперёд.

4.3. RAM-машина

RAM-машина это более мощный и естественный способом формализовать процесс вычисления.

RAM-машина состоит из бесконечного количества регистров $r_0, r_1, r_2, \ldots$ и кода, который она выполняет. Возможные инструкции выглядят следующим образом:

1. `mov` r_i r_j, копировать один регистр в другой
2. `mov` r_i $const$,
3. `mov` r_i $*r_j$, разыменование указателя, скопировать регистр с номером, лежащим в r_j.
4. `mov` $*r_i$ r_j,
5. `add/mult/sub/div/mod` r_i r_j, арифметические операции
6. `jump 17`, перейти на другую строку программы
7. `halt`, завершить исполнение

Обычно считают, что арифметические операции выполняются за $\mathcal{O}(1)$, вне зависимости от битности чисел.
Однако в таком случае можно за полиномиальное время создать и работать с числами экспоненциальной длины, что как-то странно. Поэтому часто добавляют требование о том, что потребляемая память составляет не более, чем полином от времени работы.

4.4. Строгое и слабое полиномиальное время

Предположим, что входные данные задачи даны как последовательности целых чисел (int), а решаем мы задачу на RAM-машине.

Def 4.4.1. *Алгоритм решения называется строго полиномиальным, если*

1. *Время работы можно ограничить полиномом от* ***количества*** *целых чисел во входе.*
2. *Алгоритм потребляет не более чем полиномиальное количество памяти (в битах), относительно входа.*

Все остальные алгоритмы, работающие за полиномиальное время, называются *слабо полиномиальными.*
Пример: QuickSort работает за строго полиномиальное время, а алгоритм Евклида — нет.
Есть задачи для которых известно полиномиальное решение, но существование строго полиномиального алгоритма является нерешённой задачей.

4.5. Диагонализация: доказательство теоремы о временной иерархии

Напомним теорему:

Теорема 4.5.1. $DTime(f) \subsetneq DTime(f \log^2 f)$

Вложение очевидно, будем доказать именно неравенство. Доказывать будем для машин Тьюринга. Нам понадобится идея о том, что существуют программы, способные симулировать выполнение других программ.
Такая конструкция называется универсальной машиной тьюринга. По описанию другой машины M, её входных данных x и таймеру $t(|x|)$, она способна выполнить $t(|x|)$ шагов этой машины за время $\mathcal{O}(t(|x|) \log t(|x|))$.
И перед тем как приступить к доказательству сделаем ещё несколько замечаний.

1. Чтобы показать неравномощность нужно предъявить язык, который принадлежит правой части, но не левой. Скоро мы такой язык предъявим.

2. Любой язык можно определить его характеристической функцией, т.е. $L = \{x \mid f(x) = \text{ true}\}$
3. Доказывать утверждение мы будем в частном случае: $DTime(n) \subsetneq DTime(n^{1.5})$. Несложно будет увидеть, что общий случай доказывается аналогично.

Доказательство. Пронумеруем все возможные программы (машины Тьюринга) последовательными целыми числами (машину Тьюринга можно задать строкой, а множество строк счётно). Получилась последовательность машин $M_1, M_2, \ldots, M_n, \ldots$. Аналогично пронумеруем все возможные битовые строки, которые могут лежать в языке. Получилась последовательность $s_1, s_2, \ldots, s_n, \ldots$.

Зададим язык характеристической функцией:

$$f(s_k) = \text{not } M_k(s_k) \text{ запущенная с будильником } t = |s_k|^{1.4}$$

Если машина в определении не успеет отработать, то скажем, что $f(s_k) = \text{ false}$.

Заметим, что вычислить f мы можем за время $\mathcal{O}(n^{1.4}\log)$, а значит этот язык лежит в $DTime(n^{1.5})$. Покажем, что не существует машины, решающей его за время $\mathcal{O}(n)$ или $\leqslant Cn$ для какой-то C.

Основная идея: пусть машина M_k решает этот язык за время $\leqslant Cn$. Тогда $f(x) = M_k(x)$ для всех x. С другой стороны, $f(s_k) = \text{not } M_k(s_k)$, а значит пришли к противоречию.

К сожалению, это не совсем верно, в случае если M_k не успеет завершиться за время $t = |s_k|^{1.4}$ шагов. Т.е. если константа C достаточно велика.

Однако каждая машина встречается в списке машин бесконечное число раз (в любой код можно добавить незначащих комментариев или бессмысленных действий), т.е. $M_k = M_{i_1} = M_{i_2} = \ldots$. И если пропустить достаточно много машин в этом списке, то асимптотика при $n^{1.4}$ таки задавит $\mathcal{O}(n)$, а значит аргумент выше приведёт к честному противоречию. ■

Такой трюк называется диагонализацией. Используя похожий аргумент, можно, например, доказать, что $|\mathbb{N}| \neq |2^{\mathbb{N}}|$.

Лекция #5: Доп лекции

апрель-мая 2024

5.1. Квадратный корень по простому модулю

См. конспекты прошлых лет.

5.2. Кармаркар-Карп для взлома полиномиальных хешей

Задача. Для любой пары $\langle p, m\rangle$ до 2^{128} легко строится контрпример.

Алгоритм. См. конспект прошлых лет. Кратко: Возьмём числа $p^i \mod m$, и будем последовательно применять операцию `sort` и заменить на разности $a_1 - a_0, a_3 - a_2, \ldots$ Восстановление ответа: обходим дерево, получаем для каждого $p^i \mod m$ знак `+/-`, чтобы сумма была 0. Оценка времени: $\mathcal{O}(n)$ умножений по модулю $+\mathcal{O}(sort(n))$. Оценка на n: $n = 2^k, 2^{k(k+1)/2} = m \Rightarrow n = 2^{\sqrt{2\log m}}$ для модуля m. Реальная оценка: $m \leqslant 2^{61}$, 4 100 `WA`, 4 200 `OK`.

Другое решение: делаем строки длины 64.
Простой способ: накидаем $m^{1/2}$ случайных хешей, там будут одинаковые. Более умный способ: делим на 2 половины, идём двумя указателями. Получаем $\mathsf{E} = m^{1/3}$. Умный способ: делим на 4 четверти, идём двумя указателями. Получаем $\mathsf{E} = m^{1/4}$. Если замкнём эту идею рекурсивно, получим лучше, чем Кармаркар-Карп, там $\frac{1}{n^{\log n}}$, у нас в 2^k раз меньше, но работает за $\mathcal{O}(2^k)$ (см. `approx/karmarkar-karp/karmarkar-karp.cpp`).

5.3. RMQ-сверху: только $2n$ памяти.

[CF]

Запишем вершины в порядке Эйлерова обхода (сперва выписываем корень, затем левое поддерево, затем правое). Если мы i, наш левый сын $i{+}1$, наш правый сын $i{+}1{+}size$, где $size$ – размер левого поддерева. В рекурсии знаем $L, R \Rightarrow$ знаем и длину отрезка слева len, а $size = 2len{-}1 \Rightarrow$ правый сын $= i{+}2len$.

5.4. LCA: двоичные подъёмы за $\mathcal{O}(n)$ памяти

См. конспект 3-го курса (2324s_au3).

5.5. HLD за $\mathcal{O}(\log n)$

[ЛКШ]

Построим вместо дерева отрезков BST. Пусть S – «размер того, что висит под путём».
Делимся по вершине, что слева и права висит $\leqslant \frac{S}{2}$. Получаем фиксированное дерево, структура меняться не будет. Это именно «BST по неявному ключу» – исходный массив не в листьях, а в промежуточных вершинах и листьях.

Ответ на запрос – подъём по исходному дереву. При подъёме по исходному дереву сперва поднимаемся до корня BST, затем прыгаем в отца корня в исходном дереве. Прыжков вверх по BST-шкам в сумме на больше $\log n$, т.к. каждый раз в 2 раза увеличивается размер того, что «висит под отрезком-поддеревом BST», прыжков между BST тоже не больше $\log n$, т.к. это HLD, рёбра прыжков лёгкие.