

Лекция по алгоритмам #1

Тема: вступительная лекция, разбор вступительного теста

2 сентября

Собрано 12 сентября 2014 г. в 00:12

Содержание

1	Общая информация по курсу	2
2	Общие слова про время работы, про тестирование	2
3	Оценка качества алгоритма	3
4	Разбор теста	4
5	Разбор задачи 8	6

1 Общая информация по курсу

- **Преподаватель:** Сергей Владимирович Копелиович
- **Способы связи:**
 - vk.com/burunduk1 (быстрое общение, постоянный контакт)
 - burunduk30@gmail.com (деловая переписка по учебе, сдача домашних заданий)
 - телефон: +7-921-331-99-56
- **Dropbox:** <https://www.dropbox.com/sh/mh48b3jjv0roi4p/AAA58cpLM6tYNZV4y3YHjjHea?dl=0>
- **Тестирующая система:** <http://acm.math.spbu.ru/tsweb/>
- **АСМ:** тренировки по субботам в СПб АУ, с 10:00 до 15:00, затем обед и разбор. Более подробная информация здесь <http://acm.math.spbu.ru/~sk1/mm/au/2014/>
- **Язык программирования:** на этом курсе наш язык C/C++ с STL.
- **Источники информации:**
 - wiki, вики, <http://e-maxx.ru>, <http://neerc.ifmo.ru/wiki>, отдельные статьи на хабре
 - Книги: Кормен; Бабенко и Левин; Дасгупта и Вазирани.
- **Отчетность:**
 - Одна глава электронного конспекта в \LaTeX . Дедлайн 2 недели.
 - Теорэкзамен общий с базовым курсом.
 - Допуск к экзамену: зачет по практической части курса.
 - Практическая часть: каждую неделю получаете констест и теорзадачи.
 - Реализация и тестирование одного из описанных алгоритмов. С `codereview`, с промежуточным дедлайном 2 недели и окончательным дедлайном 1 месяц.
 - Будет необязательно задание по гуглению. Вопросы из серии: за какое оптимальное время умеют решать задачу о `maximum weighted matching`?

2 Общие слова про время работы, про тестирование

- Если вы знаете в теории некоторый алгоритм, но не понимаете, как его реализовать, знание менее ценно.
- Если вы реализовали алгоритм, а он работает “не всегда”, такая реализация никому не нужна. Алгоритм должен на всех возможных случаях работать корректно.
- Если вы хотя бы один раз самостоятельно реализовали алгоритм, знание осядет глубже, надолго. Во время реализации нового алгоритма вы убедитесь, что алгоритм понимаете до конца, и, скорее всего, встретите пару тонких моментов, о которых до этого не задумывались.
- Реализация может быть “доказательством корректности”. Для этого в код нужно вставить побольше `assert`-ов, то есть, проверок утверждений, которые заведомо должны быть верными. Например, можно насчитать число сравнений, сделанных алгоритмом `MergeSort` и проверить в конце, что

это значение не больше $n \log_2 n$ (`assert(cnt <= n*log(n)/log(2));`). Кроме этого код следует протестировать на случайных тестах (случайная перестановка, n случайных чисел от 1 до m), на крайних случаях (все числа в массиве равны, тождественная перестановка, обратная перестановка). Если вы придумали новый сложный алгоритм, даже если есть проверенное письменное доказательство, рекомендуется алгоритм реализовать и протестировать, могут обнаружиться спецэффекты.

- Пусть вы поняли алгоритм, но не уверены, что до конца в нем разобрались. Есть два хороших способа проверить наверняка — реализовать алгоритм, или, что обычно быстрее, запустить его руками (на бумажке) на небольшом тесте.
- Если вы думаете, что поняли алгоритм, вас просят его реализовать, а вы говорите, что за час не справитесь, значит, или это очень-очень сложный алгоритм, или вы поняли максимум общую идею, но не весь алгоритм, все его этапы и случаи.
- Современные машины делают $\approx 10^9$ элементарных операций в секунду. Это значит, что не “элементарных” операций, например, `a[b[i]] = k++`; за секунду мы успеем сделать $\approx 10^8$ (здесь ошибка может быть в 2 – 3 раза в обе стороны). По алгоритму и размеру данных нужно уметь быстро оценивать, насколько алгоритм быстро сработает.
Пример #1: за сколько вычисляется 20-е число Фибоначчи рекурсивной функцией? $f_{20} \leq 2^{20}$, значит мгновенно, сильно быстрее одной десятой секунды.
Пример #2: за сколько сортируется массив из 10^7 элементов? $10^7 \log 10^7 \approx 240 \cdot 10^6$, значит, около одной секунды (здесь может быть ошибка в несколько раз, но порядок мы оценили).
- При вычислении времени работы полезно умение быстро переводить из двоичной в десятичную. $10^3 \approx 1024 = 2^{10} \Rightarrow 10^9 \approx 2^{30}$ и $10^8 \approx 2^{20} \cdot 100 \approx 2^{26}$.

3 Оценка качества алгоритма

При сравнение алгоритмов используется несколько критериев:

- Время работы. Обычно это сложный фактор, нужно оценивать:
 1. Время выполнения в секундах на макс-тесте. До реализации оценивать сложно.
 2. Асимптотику. Но не всегда этого достаточно:
хеш-таблица, умножение матриц за $\mathcal{O}(n^{2.37})$.
 3. Константу. Количество операций: $10n$ определенно хуже, чем $2n$.
 4. Тип операций: хеш-таблица работает долго не из-за того что операций много.
 5. При детальном анализе алгоритмов конкретного типа, например сортировок, отдельно оценивают число сравнений, число присваиваний и т.д. Так как в разных случаях нужны бывают разные сортировки. Примеры: суффиксный массив за $\mathcal{O}(n \log^2 n)$, сортировка $100GB$ файла на жестком диске.

- Количество использованной дополнительной памяти (размер входных данных не считается, иногда также не учитывается размер выходных данных). Память считается более дорогим, чем время, ресурсом.
- Возможность распараллеливания на несколько машин. Это относится особенно к структурам данных.
- Поведение алгоритма можно оценивать в худшем случае, в среднем случае (матожидание по входным данным), в лучшем случае. Если алгоритм вероятностный, как QuickSort, можно оценивать матожидание по случайным битам.
- Количество крайних случаев. Это особенно видно на геометрических алгоритмах. Вот примеры задач, идея решения которых несложна, но умение реализовать эти идеи считается серьезным мастерством.
 1. Является ли диагональ невыпуклого многоугольника нестрогой внутренней? $\mathcal{O}(n)$
 2. Найти длину пересечения прямой и невыпуклого многоугольника. $\mathcal{O}(n)$
 3. Найти площадь пересечения двух невыпуклых многоугольников. $\mathcal{O}(n^2 \log n)$
 4. Получение планарного графа, представляющего собой объединение/пересечение k невыпуклых многоугольников. $\mathcal{O}(n^2 \log n)$
 5. Построение диаграммы Вороного в 2D за $\mathcal{O}(n \log n)$ методом заметающей прямой.

Если вы придумали новый алгоритм и в нем масса крайних случаев, пожалуйста, не спешите публиковаться, не плодите ад!

- Длина кода (LOC = lines of code) и сложность идеи. Опытным программистам может показаться, что это не так важно — один раз написал, сто лет работает, все пользуются и рады. Но если сложным и длинным в реализации алгоритмом начинают пользоваться как черным ящиком, часто нет возможности модифицировать алгоритм изнутри для решения схожих задач. Также бывает, что из-за повышенной сложности алгоритма кроме очень узкого круга специалистов никто до конца не знает, как же это все работает, в итоге тема не развивается, а иногда даже забывается.
- Число проходов по входным данным. Однопроходные алгоритмы.
- Online/Offline структуры данных (см. следующую лекцию).
- Амортизированное, real time время работы (см. следующую лекцию).

4 Разбор теста

- **Задача #1.**
 - (a) $\mathcal{O}(n^2)$ — сортировка вставками, выбором, пузырьком
 - (a) $\mathcal{O}(n \log n)$ — merge sort, quick sort, heap sort
 - (b) $\mathcal{O}(n + m)$ — сортировка подсчетом.

```

1. int cnt[M]; // заполнен нулями
2. for (i = 0; i < n; i++) // O(n)
3.     cnt[a[i]] += 1;
4. for (i = 0; i < m; i++) // O(m)
5.     for (j = 0; j < cnt[i]; j++) // O(n) в сумме по всем i
6.         print(i); // выводим cnt[i] раз число i

```

(b) $O(n), O(m)$ — формально не верно, правильно $O(n + m)$.

(b) $O(n + n \log_n m)$ — цифровая сортировка (radix sort, digital sort).

(b) Дополнительные баллы всем, кто вспомнил про длинные числа.

• **Задача #2.**

(a) Дек, реализация на массиве, $O(1)$

(b) Массив, $O(1)$

```
i = rand() % n; swap(a[i], a[n-1]); return a[--n];
```

(c) Куча с обратными ссылками + массив, $O(\log n)$

Node p[]; p[i] — позиция в куче элемента, добавленного в i -й момент времени

(d) Двусвязный список + массив, $O(1)$

Node p[]; p[i] — позиция в списке элемента, добавленного в i -й момент времени

(e) HashMap, рандомизированное $O(1)$

(f) куча с обратными ссылками + HashMap, $O(\log n)$

HashMap<int, Node> p; — по элементу за $O(1)$ узнаем его позицию в куче.

• **Задача #3а.**

```

1. max = a[0], result = 0;
2. for (i = 1; i < n; i++) {
3.     if (a[i] + max >= S)
4.         result = 1;
5.     if (a[i] > max)
6.         max = a[i];
7. }

```

• **Задача #3б.**

Решение, за $O(\sqrt{n})$:

```

1. for (a = 1; a * a <= n; a++) {
2.     b = sqrt(n - a * a);
3.     res += (a * a + b * b == n);
4. }

```

Решение, за $O(\sqrt{n})$ элементарных арифметических операций с целыми числами:

```

1. for (a = 1, b = sqrt(n); a <= b; a++) {
2.     while ((tmp = a * a + b * b) > n) //  $2\sqrt{n/2}$  умножений
3.         --b;
4.     if (tmp == n)

```

```

5.         res += (a == b ? 1 : 2);
6.     }

```

- **Задача #4а.**

Одна из возможных идей — предподсчет за $\mathcal{O}(n^2)$, можно положить все элементы матрицы в хеш-таблицу и за $\mathcal{O}(1)$ отвечать на поступающие запросы. Если предподсчет запрещен, то на один запрос можно отвечать за время $\mathcal{O}(n)$:

```

1.     i = n - 1, j = 0;
2.     while (i >= 0 && j < n) {
3.         if (a[i][j] == x)
4.             return 1;
5.         (a[i][j] < x) ? ++j : --i;
6.     }

```

Заметим, что без предподсчета быстрее чем за $\mathcal{O}(n)$ на запрос не ответить, так как на побочной диагонали могут стоять произвольные числа.

- **Задача #4б.**

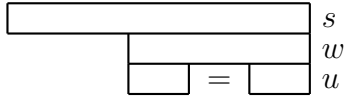
Существуют решения за $\mathcal{O}(n)$, это или алгоритм Манакера (аналог Z -функции), или построение дерева палиндромов. Оба алгоритма просты в реализации и встретятся в курсе позднее. Дерево палиндромов также позволяет за $\mathcal{O}(n)$ решить задачу #8а — число различных подстрок-палиндромов. Здесь описаны лишь более простые решения. Итак, решения за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$:

1. У каждого палиндрома есть центр. Центр находится или в позиции i , или между позициями i и $(i + 1)$. Центр можно перебрать.
2. Решим задачу для фиксированного центра x . Если есть палиндром с центром в x и длиной L , то есть и палиндромы с центром в x и длинами $L - 2, L - 4, \dots$. Поэтому достаточно найти максимальное L .
3. Поиск максимального L за линейное время: `while (s[x+L] == s[x-L]) L++;`
4. Поиск максимального L за $\mathcal{O}(\log n)$: бинарный поиск, а внутри сравнение подстрок за $\mathcal{O}(1)$ с помощью полиномиальных хешей.

5 Разбор задачи 8

- **Задача #8а.**

Докажем, что в строке длины n различных подстрок-палиндромов не более n . База индукции: в строке длины 1 есть ровно 1 подпалиндром. Шаг индукции: докажем для строки s длины $n + 1$, $s = wa$, $|w| = n$, для строки w мы уже знаем, что различных подпалиндромов не более n . Рассмотрим все суффиксы s , являющиеся палиндромами. Самый длинный суффикс z мог оказаться новым подпалиндромом, покажем, что любой более короткий суффикс-палиндром u встречался, как подстрока, в строке w :



Чтобы доказать равенство, мы пользуемся тем, что строки w и u — палиндромы.

• **Задача #86.**

Опишем решение за 1.38^n . Рассмотрим произвольную вершину v , мы ее или возьмем в ответ, или не возьмем. Если возьмем, то мы не можем брать её соседей. Получаем рекурсивное решение, которое в первой ветке удаляет вершину v и запускается от оставшейся части графа (не берёт v в ответ), а во второй ветке удаляет вершину v и всех её соседей, после чего запускается от остатка (берёт v в ответ). Пусть в исходном графе n вершин, тогда время работы нашего решения $T(n)$, можно оценить как $T(n) = T(n - 1) + T(n - d - 1)$, где d — степень вершины v . Чем d больше, тем лучше. Осталось заметить, что если в графе присутствуют только вершины степени не более двух, то наш граф — набор путей и циклов, и максимальное независимое множество ищется на нем жадно. Получили $T(n) \leq T(n - 1) + T(n - 4) \Rightarrow T(n) = \mathcal{O}(1.38^n)$.

КОНЕЦ