Основные алгоритмы, комментарии по пятому листку (Сортировки II. Нижние оценки)

Шибаев Иннокентий

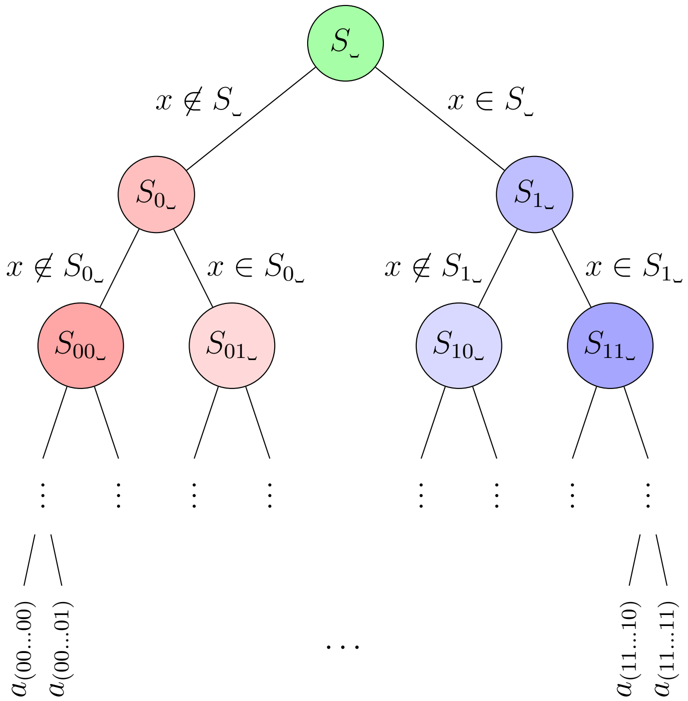
March 7, 2021

1 По поводу теории

1.1 Разрешающие деревья

Начать надо с описания того как вообще какой-либо алгоритм решает поставленную задачу. Алгоритм задает какие-то вопросы (которые мы позволяем задавать, это и есть модель, то множество алгоритмов, для которого мы ищем нижнюю оценку), и на основе выданной последовательности ответов на свои вопросы должен выдать какой-либо результат (к примеру нужную перестановку элементов массива для сортировки, или же загаданное число).

Описанное выше можно представить в виде *разрешающего дерева*. В узлах этого дерева находятся вопросы которые задает алгоритм, после каждого вопроса алгоритм переходит в одну из дочерних вершин. В листьях записаны результаты которые алгоритм сообщает после серии вопросов. Рассмотрим это дерево, к примеру, для случая игры с угадыванием числа на отрезке (загадывается натуральное число $x \in \{1, \ldots, N\}$, разрешается задавать вопросы вида «Принадлежит ли x множеству $S \subseteq \{1, \ldots, N\}$?»):



После каждого вопроса о принадлежности искомого элемента какому-то множеству мы переходим в соответствующее поддерево (причем поддеревьев 2 т.к. возможных ответов на такой вопрос всего 2 – «да» (этот результат ведет нас вправо и обозначается единицей в нижнем индексе следующих запросов, к примеру $S_{1\smile}$) или «нет» (соответственно цифра 0).

Примечание 1.1. Важно отметить, что в таком виде представляется **любой** алгоритм решения этой задачи в данной модели (т.е. в условиях когда мы можем задавать вопрос о принадлежности x множеству S). К примеру алгоритм который бы просто спрашивал на i-м ходу «x это i?» (т.е. $x \in S = \{i\}$) имел бы вид ветки длины n в каждом узле которой ответ «да» приводит в лист. Глубина такого дерева была бы равна n.

Т.е. фактически от не совсем формализованного в нашем курсе понятия «алгоритм» мы перешли к рассмотрению семейства каких-то деревьев с какими-то свойствами. Теперь надо понять что нам это дает.

Поговорим о том какой смысл имеет глубина этого дерева. Рассмотрим разрешающее дерево некоторого алгоритма. Пусть высота этого дерева равна h. Тогда мы можем утверждать, что алгоритм в худшем случае задаст h вопросов прежде чем выдать ответ. К примеру алгоритм из примечания в худшем случае будет работать n шагов (если $x = N$).

Итак, высота дерева это то сколько алгоритм будет работать в худшем случае. Теперь, если мы сможем доказать каким-либо образом, что высота дерева для данного множества алгоритмов не может быть меньше чем какое-то число мы получим оценку снизу – любой алгоритм будет совершать минимум такое же кол-во действий.

1.2 Различные способы построения нижних оценок

Мы обсуждали три различных способа построения нижних оценок, два на основе разрешающих деревьев и метод потенциалов. Кратко опишем их.

1. *Оценка по числу листьев.* В результате работы алгоритм должен выдать какой-то из возможных результатов. При этом к различным результатам должны вести различные наборы ответов на задаваемые алгоритмом вопросы – иначе два результата попадут в один лист в дереве, и алгоритм не сможет определить какой из них выдать. Таким образом если можно оценить высоту дерева исходя из числа листьев то это даст нижнюю оценку на число запросов алгоритмов – к примеру в бинарном дереве для кол-ва листьев k мы получаем что высота дерева не может быть меньше $\lceil \log_2 k \rceil$.

2. *Метод противника* (a.k.a. *сопротивляющийся оракул*). Идея состоит в том чтобы на каждый запрос алгоритма отвечать ему так, чтобы задержать его как можно сильнее, но при этом так, чтобы новый ответ не противоречил старым. К примеру для задачи угадывания числа, обсуждавшейся выше, на каждом шаге на вопрос о принадлежности $x \in S$ отвечать так, чтобы у алгоритма осталось бóльшее множество – таким образом мы будем его задерживать как можно сильнее.

3. *Метод потенциалов.* Суть в том чтобы ввести некоторую функцию f (потенциал), которая будет меняться после каждого запроса алгоритма. Затем надо проанализировать то как она меняется при различных запросах, и на основе этого понять насколько быстро она может меняться. Затем, если мы знаем каково должно быть значение функции когда алгоритм выдает ответ мы можем оценить кол-во действий k необходимое алгоритму примерно как

   $$k = \frac{f(\text{начало}) - f(\text{конец})}{\text{наибыстрейшее изменение за шаг}}.$$

   Его проще пояснять на примере, и это будет сделано ниже.

1.3 Примеры использования различных оценок

1.3.1 Оценка по числу листьев

К примеру для упомянутой выше задачи сортировки массива, для постановки «Алгоритм может задавать вопросы вида $a_i < a_j$» (a.k.a. сортировки сравнениями) мы можем заметить что алгоритм должен выдать один из $n!$ ответов (одну из возможных перестановок элементов). Таким образом в разрешающем дереве будет минимум $n!$ листьев.

Далее, каждый вопрос дает нам 1 бит информации, т.е. после каждого вопроса происходит ветвление на два случая – рассматриваемое дерево бинарное. Осталось заметить что высота бинарного дерева не может быть меньше чем $\lceil \log_2 n! \rceil = \Theta(n \log n)$.

Итак, никакая сортировка сравнениями не может работать быстрее чем $\Theta(n \log n)$. Осталось заметить что т.к. сортировка слиянием (MergeSort) работает как раз за $\Theta(n \log n)$ эта оценка уже не может быть улучшена.

Другой пример это игра с угадыванием числа – там всего N листьев, дерево опять же бинарное, так что в той постановке что обсуждалась выше оценка снизу будет иметь вид $\lceil \log_2 N \rceil$.

1.3.2 Оценка через метод противника

Опять же рассмотрим задачу с угадыванием числа – пусть нам уже задали вопросы о принадлежности x каким-то множествам. Значит сейчас у алгоритма есть информация, что $x \in \bigcap\limits_{i=1}^{k-1} S_i = A$. Теперь он задает нам вопрос о принадлежности x множеству S_k. Сделаем следующим образом: если $|\,(\{1, \ldots, N\} \backslash S) \cap A| \geqslant |S \cap A|$ то ответим «нет», иначе ответим что принадлежит. Фактически то что записано означает что мы переходим в бóльшую часть множества A (здесь вообще множество A возникает из-за того что нам надо чтобы наш новый ответ не противоречил старым).

Таким образом на каждом шаге множество A уменьшается не более чем в 2 раза. Т.е. какой бы алгоритм мы не придумали, он не сможет уменьшать множество ответов быстрее чем в два раза на каждом ходу. Значит быстрее чем за $\lceil \log_2 N \rceil$ решить эту задачу ни одним алгоритмом (в такой постановке) не получится.

1.3.3 Оценка через метод потенциалов

Задача 1.1 (Задача 3 из листка 5)**.** Есть n монет разного веса. За одно взвешивание можно сравнить по весу любые две монеты.

1. Найдите самую тяжёлую и самую лёгкую за $\frac{3}{2}n + O(1)$ взвешиваний.
2. Докажите, что нельзя найти среди n монет самую тяжелую и самую лёгкую монету за менее чем $\frac{3}{2}n + \mathrm{C}$ взвешиваний.

Решение. Нас здесь интересует второй пункт. Заметим, что после некоторого кол-ва запросов на сравнения все монеты можно условно разделить на 4 класса

- T_{max} – элементы которые потенциально могут оказаться максимумом;
- T_{min} – элементы которые потенциально могут оказаться минимумом;
- $T_{\text{отбр}}$ – элементы которые точно не могут оказаться ни минимумом, ни максимумом (т.е. которые попали в сравнения вида $a < x < b$);
- $T_{\text{ост}}$ – элементы которые еще не сравнивались ни с кем.

Любой алгоритм (в постановке описанной в условии) на каждом своем шаге сравнивает какие-то элементы из этих множеств, к примеру если он сравнивает два элемента из $T_{\text{ост}}$ то один перейдет в T_{max}, а другой в T_{min}.

Рассмотрим следующую потенциальную функцию

$$f\left(|T_{max}|, |T_{min}|, |T_{\text{отбр}}|, |T_{\text{ост}}|\right) = \frac{3}{2}\left|T_{\text{ост}}\right| + |T_{max}| + |T_{min}|$$

тогда в начале, до того как алгоритм начинает работу имеем $f(\ldots) = \frac{3}{2}n$, а в конце, когда алгоритм закончил работу имеем $|T_{max}| = 1$, $|T_{min}| = 1$, $|T_{\text{отбр}}| = n-2$, $|T_{\text{ост}}| = 0$ и, значит, $f(\ldots) = 2$.

Алгоритм который мы рассматривали на семинаре в начале разделяет все множество на пары (т.е. сравнивает между собой только элементы из $T_{\text{ост}}$, тратя на это $\frac{n}{2}$ действий, и переходит в состояние $\left\{\frac{n}{2}, \frac{n}{2}, 0, 0\right\}$ (считая что n – четно). При этом каждое такое сравнение уменьшает нашу функцию на 1, т.к. $|T_{\text{ост}}| \to |T_{\text{ост}}|-2, |T_{max}| \to |T_{max}|+1, |T_{min}| \to |T_{min}|+1$, т.е.

$$\frac{3}{2}\left|T_{\text{ост}}\right| + |T_{max}| + |T_{min}| \to \frac{3}{2}\left|T_{\text{ост}}\right| - 3 + |T_{max}| + 1 + |T_{min}| + 1.$$

Затем этот алгоритм ищет максимум во множестве T_{max} и минимум во множестве T_{min}, при этом опять же функция каждый раз уменьшается на 1. Таким образом мы переходим в конечное состояние за $\frac{3n/2-2}{1}$, и у нас уходит $\frac{3}{2}n-2$ шагов.

Но это для одного конкретного алгоритма. Давайте поймем что нужно сделать чтобы доказать что любой алгоритм решает эту задачу не быстрее. А для этого на самом деле надо доказать что остальные действия (сравнения элементов из других множеств, и из пар разных множеств) уменьшают функцию не быстрее – тогда получится что наилучший возможный способ уменьшаает ее на 1 на каждом ходу – а мы его как раз представили.

Построим таблицу возможных результатов (изменений функции) для всех пар сравнений (слева)

	T_{max}	T_{min}	$T_{\text{отбр}}$	$T_{\text{ост}}$
T_{max}	-1	$0; -2$	$0; -1$	$-\frac{3}{2}; -\frac{1}{2}$
T_{min}	$0; -2$	-1	$0; -1$	$-\frac{3}{2}; -\frac{1}{2}$
$T_{\text{отбр}}$	$0; -1$	$0; -1$	0	$-\frac{1}{2}$
$T_{\text{ост}}$	$-\frac{3}{2}; -\frac{1}{2}$	$-\frac{3}{2}; -\frac{1}{2}$	$-\frac{1}{2}$	-1

	T_{max}	T_{min}	$T_{\text{отбр}}$	$T_{\text{ост}}$
T_{max}	-1	0	0	$-\frac{1}{2}$
T_{min}	0	-1	0	$-\frac{1}{2}$
$T_{\text{отбр}}$	0	0	0	$-\frac{1}{2}$
$T_{\text{ост}}$	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$	-1

В некоторых клетках стоит по два числа – это случаи, когда то на сколько изменится функция зависит от результата сравнения. К примеру в случае сравнения $x \in T_max$ с $y \in T_{min}$ получаем что если $x < y$ оба элемента перейдут в $T_{\text{отбр}}$, иначе ничего не изменится.

При этом если есть два различных значения можно выбирать то что уменьшает функцию меньше (это в определенном смысле идея из метода противника, мы можем поменять вес камней так чтобы предыдущие сравнения с ним не изменились, но в этом результат был бы нужным (у камней из T_{min} вес модно сколь угодно уменьшать, а у T_{max}, соответственно, увеличивать).

И тогда (выбирая минимум в каждой клетке) получаем таблицу справа. В этой таблице все значения не превосходят 1, таким образом быстрее чем за $\frac{3}{2}n-2$ действий решить задачу мы действительно не сможем. $\square$

2 По поводу задач

Задача 2.1 (Задача 8^* из листка 4 (д/з))**.** Запишите рекуррентное соотношение для сложности и докажите по индукции, что трудоемкость алгоритма $Randomized-Qsort$ (в среднем) равна $O(n \log n)$.

Решение. В начале мы «выбираем случайный элемент из A», в данном случае это означает что мы выбираем i-й из элементов $A : |A| = n$ с вероятностью $\frac{1}{n}$ (равномерное распределение).

Нас просят найти сложность работы в среднем, т.е. нам надо написать рекурренту через матожидание суммы рекурсивных вызовов функции (от множеств B и C) взятое по случайной величине – индексу элемента выбранного опорным. Выглядит это в итоге так:

$$T(n) = \sum_{i=1}^{n} \left(T(i-1) + T(n-i)\right) \cdot p(i) + Cn.$$

Здесь $p(i)$ это вероятность того что именно i-й элемент будет выбран опорным, в нашей модели $p(i) = \frac{1}{n}$. Cn здесь идет от процедуры $Partition$, вообще время ее работы зависит от того какой элемент выбран опорным, но на порядок ($\Theta(n)$) это не влияет, так что обойдемся просто Cn. Подставляя значение $p(i)$ и переставляя члены в сумме получаем

$$T(n) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + Cn.$$

Теперь, воспользуемся индукцией. Предположим что $T(n) = O(n \log n)$. Попробуем подставить $T(n) \leqslant Dn \log n$ в правую часть:

$$T(n) \leqslant \frac{2D}{n} \sum_{i=0}^{n-1} k \log k + Cn.$$

Теперь надо придумать как оценить $\sum\limits_{i=0}^{n-1} k \log k$. Можно попробовать оценить как $\sum\limits_{i=0}^{n-1} n \log n = (n-1)n \log n$, но тогда справа будет получаться $2Dn \log n + \ldots$, что нам не подходит. Можно оценить немного аккуратнее, а именно $\sum\limits_{i=0}^{n-1} k \log k \leqslant \sum\limits_{i=0}^{n-1} k \log n = \frac{n(n-1)}{2} \log(n)$, тогда $T(n) \leqslant \frac{2D}{n} \frac{n(n-1)}{2} \log(n) + Cn = D(n-1)\log(n) + Cn$ – уже лучше но Cn все портит. Оценим еще аккуратнее:

$$\begin{aligned}
\sum_{i=0}^{n-1} k \log k &\leqslant \sum_{i=0}^{\lfloor (n-1)/2 \rfloor} k \log \frac{n}{2} + \sum_{i=\lfloor (n-1)/2 \rfloor + 1}^{n-1} k \log n = \\
&= \sum_{i=0}^{n-1} k \log n - \sum_{i=0}^{\lfloor (n-1)/2 \rfloor} k \log 2 \\
&= \frac{n(n-1)}{2} \log(n) - \frac{\lfloor (n-1)/2 \rfloor \cdot (\lfloor (n-1)/2 \rfloor + 1)}{2} \log 2 \leqslant \\
&\overset{\text{при достаточно больших } n}{\leqslant} \frac{n(n-1)}{2} \log(n) - \frac{n^2}{16}
\end{aligned}$$

и, подставляя это в рекурренту выше, наконец, получаем

$$T(n) \leqslant \frac{2D}{n} \frac{n(n-1)}{2} \log(n) - \frac{2D}{n} \frac{n^2}{16} + Cn = Dn \log(n) - D \log n - D\frac{n}{8} + Cn \overset{D=8C}{\leqslant} Dn \log(n)$$

и взяв $D = 8C$ получаем, наконец, $T(n) \leqslant Dn \log(n)$. □