Основные алгоритмы, комментарии по первому листку

Шибаев Иннокентий

6 февраля 2021 г.

1 По поводу теории

Определение 1.1 (O, Ω, Θ-нотация)**.** Будем говорить, что $f(n) = O(g(n))$, если $\exists N \in \mathbb{N}$, $\exists C > 0$:

$$\forall n > N : \ f(n) \leqslant Cg(n).$$

Аналогично, будем говорить, что $f(n) = \Omega(g(n))$, если $\exists N \in \mathbb{N}$, $\exists C > 0$:

$$\forall n > N : \ f(n) \geqslant Cg(n).$$

Наконец, будем говорить, что $f(n) = \Theta(g(n))$, если $\exists N \in \mathbb{N}$, $\exists C_1, C_2 > 0$:

$$\forall n > N : \ C_1 g(n) \leqslant f(n) \leqslant C_2 g(n).$$

Иначе можно сказать что $f(n) = \Theta(g(n)) \Leftrightarrow \begin{cases} f(n) = O(g(n)) \\ f(n) = \Omega(g(n)) \end{cases}$.

Важно отметить, что если $f(n) = P(n) = a_0 n^k + a_1 n^{k-1} + \ldots + a_k$ то $f(n) = \Theta(n^k)$ (т.е. любой полином можно оценивать порядком роста его наибольшей степени).

Определение 1.2 (Индуктивные функции)**.** Рассмотрим функции, которые определены на конечных последовательностях произвольной длины $(x_1, \ldots, x_n)$ с элементами из множества A, и принимают значение в множестве B. Функция f данного вида называется индуктивной, если существует $F : B \times A \to B$, такая, что

$$f(x_1, \ldots, x_n, x_{n+1}) = F(f(x_1, \ldots, x_n), x_{n+1})$$

По сути своей индуктивные функции (что логично из названия) можно вычислить через себя, достаточно на каждом шаге как-то обновлять значение полученное на предыдущем шаге с учетом новой информации.

Определение 1.3 (Индуктивное расширение)**.** Индуктивная функция $g : A \to B'$ называется индуктивным расширением функции $f : A \to B$, если существует такая $t : B' \to B$, что

$$t(g(x_1, \ldots, x_n)) = f(x_1, \ldots, x_n)$$

На самом деле это определение больше техническое. В лекции был пример про максимальную площадь треугольника со стороной на оси Ox. В такой постановке вычисляемая функция не индуктивна, мы не можем основываясь лишь на наибольшей предыдущей площади и новой точке вычислить новую лучшую площадь.

Функция g в определенном смысле "преобразует пространство", сохраняя информацию которую нам нужна для решения задачи – вместо всех точек в этой задаче она сохраняет три лучших, по которым уже можно вычислить площадь. И она индуктивна – эти три точки можно обновить считав новую.

Определение 1.4 (Онлайн-алгоритм)**.** Пусть дана некоторая последовательность $x_1, \ldots, x_n$, причем n заранее не известно. И есть некоторая функция f определенная на префиксах этой последовательности, т.е. можно вычислить $f(x_1),\ f(x_1, x_2)\ \ldots\ f(x_1, \ldots, x_n)$.

Алгоритм который для всех целых $i \in [1, n]$ на i-шаге вычисляет $f(x_1, \ldots, x_i)$ (т.е. вычисляет значение функции на новом префиксе после каждого считывания переменной) мы назовем *онлайн-алгоритмом.*

По сути своей определение выше говорит что онлайн-алгоритмы "готовы выдать ответ в любой момент" – у них нет отдельных этапов "считывания данных" и их "последующей обработки", ответы генерируются постоянно. Отдельно об этом можете прочитать в лекциях. Ниже в разделе с задачами я скажу пару слова также по поводу разобранной на семинаре задачи 4, в которой мы как-то опустили слова "онлайн-алгоритм" и "индуктивное расширение".

2 Задачи

Мы начнем с пары слов по поводу задачи 4 из первого листка.

Задача 2.1 (Задача 4 из листка 1, Шень 1.3.1(в,д))**.** Постройте линейный по времени онлайн-алгоритм, который вычисляет следующие функции или укажите индуктивные расширения для следующих функций:

в) второй по величине элемент последовательности целых чисел (тот, который будет вторым, если переставить члены в неубывающем порядке);

д) максимальная длина монотонного (неубывающего или невозрастающего) участка из идущих подряд элементов в последовательности целых чисел;

Хочется проговорить аккуратно здесь по поводу онлайн-алгоритмов и индуктивных расширений.

В пункте в) онлайновость заключалась в том что мы на каждом шаге обновляли первый и второй максимум, т.е. на каждом шаге мы готовы были выдать ответ вот так, сразу.

С индуктивным расширением все немного сложнее. Итак, на каждом шаге нам надо было вычислять функцию f вида "второе максимальное значение в последовательности". Эта функция не является индуктивной, т.к. мы не можем по этому значению и новому считанному заново вычислить второе максимальное – нам нужно и первое (проверьте это!). Поэтому нам нужно индуктивное расширение, нечто что можно индуктивно пересчитывать и при этом оно содержит всю необходимую нам информацию.

Поэтому наше индуктивное расширение это функция g вида "найти первое и второе максимальное значение в последовательности". Она из множества последовательностей бьет в множество пар элементов. А t (которая выше в определении была) тогда имеет вид "выдать второй элемент в паре". Т.е. то все выглядит вот так:

$$t(g(x_1, \ldots, x_n)) = t\left((x_{[n,1]}, x_{[n,2]})\right) = x_{[n,2]} = f(x_1, \ldots, x_n)$$

(здесь индекс у $x_{[n,1]}$ значит что это результат для последовательности длины n и это первый максимум в этой последовательности) и при этом g индуктивная, для нее существует процедура пересчета G (то что мы обсуждали на семинаре, с обновлением максимумов) такая, что

$$g(x_1, \ldots, x_n) = G(g(x_1, \ldots, x_{n-1}), x_n)$$

таким образом

$$f(x_1, \ldots, x_n) = t\left(G\left(g(x_1, \ldots, x_{n-1}), x_n\right)\right) = t\left(G\left(\left(x_{[n-1,1]}, x_{[n-1,2]}\right), x_n\right)\right).$$

Вот так с формальной точки зрения работает то что мы обсуждали. Нужно ли так расписывать постоянно – не думаю, особо смысла не имеет. Но сам формализм данный довольно интересен.

Задача 2.2 (Задача 5 из листка 1, Шень 1.3.2)**.** Даны две последовательности целых чисел $x[1], \ldots, x[n]$ и $y[1], \ldots, y[k]$. Выясните, является ли вторая последовательность подпоследовательностью первой, то есть можно ли из первой вычеркнуть некоторые члены так, чтобы осталась вторая. Число действий $O(n+k)$.

Решение. *TL;DR: Жадно ищем каждый новый символ из последовательности y в последовательности x.*

Для начала считаем обе последовательности (в данной задаче не требуется онлайн решения, и ограничений по памяти нет, вариант с тем как можно уменьшить потребление памяти рассмотрим потом). Так как мы считываем обе последовательности (одна длины n, а другая длины k) это требует некоторое $C_1(n+k)$ действий.

Предположим что последовательности y является подпоследовательностью x. Это значит, что есть некоторый набор возрастающих номеров $i_1, \ldots, i_k$ такой что $y[1] = x[i_1], \ldots, y[k] = x[i_k]$.

Теперь заметим, что, к примеру, если существует номер $j_1 < i_1$ такой что $y[1] = x[j_1]$ то мы вполне можем начать нашу подпоследовательность последовательности x с него и все равно построить y (потому что следующие номера такое действие никак не затронет).

Возникает идея – давайте будем смотреть на очередное $y[i]$ и искать в оставшейся последовательности x первое вхождение этого $y[i]$. После чего мы перейдем к $y[i+1]$, а вместо x будем рассматривать опять часть которая идет после вхождения $y[i]$ (суффикс последовательности). Это жадный алгоритм, который просто на каждом шаге старается "уместить"в последовательность x последовательность y как можно плотнее. Если же на очередном символе $y[i]$ последовательность x закончится – то выводим сообщение о том что вторая последовательность подпоследовательностью первой не является.

Мы описали **алгоритм**, теперь надо доказать его **корректность** и получить **оценку сложности**.

Начнем с **корректности**. Предположим что y является подпоследовательностью x. Как мы уже писали выше это значит что есть некоторый набор возрастающих номеров $i_1, \ldots, i_k$ такой что $y[1] = x[i_1], \ldots, y[k] = x[i_k]$.

Давайте заменим i_1 на $j_1 < i_1$ такое, что $y[1] = x[j_1]$ и j_1 наименьшее, т.е. фактически $x[j_1]$ является первым вхождением $y[1]$ в x. Теперь повторим это с i_2, только искать будем уже от $j_1 + 1$ до i_2, и т.д. Таким образом мы построим возрастающий набор индексов $j_1, \ldots, j_k$ такой что $y[1] = x[j_1], \ldots, y[k] = x[j_k]$. При этом j_1 это номер первого вхождения $y[1]$ в x, j_2 это номер первого вхождения $y[2]$ в x но начиная с j_1+1 символа и т.д. – т.е. ровно то что строит алгоритм предложенный нами. Таким образом если решение существует наш алгоритм его находит.

Теперь предположим что y **не** является подпоследовательностью x. Это значит что существует некоторая часть последовательности y (возможно пустая последовательность) вида $y[1], \ldots, y[i]$ что является подпоследовательностью x, но $y[1], \ldots, y[i+1]$ таковой уже не является. Опять же, алгоритм выше найдет индексы $j_1, \ldots, j_i$, но не сможет найти j_{i+1} (т.к. если бы он нашел такой мы бы вошли в противоречие с тем что $y[1], \ldots, y[i+1]$ не является подпоследовательностью). Таким образом он дойдет до конца последовательности x, и, значит, выведет сообщение о том что y подпоследовательностью x не является.

Осталось разобраться с **оценкой сложности**. На считывание мы потратили $C_1(n+k)$ действий. После этого мы идем по последовательности y и для каждого символа из нее ищем в последовательности x такой же символ, причем начиная с позиции предыдущего найденного символа $+1$ (в коде это были бы два итератора). Т.е. по обоим последовательностям мы идем только вперед, значит это опять же линейная сложность

– $C_2(n+k)$.

Итого получаем $(C_1+C_2)(n+k)$, значит $O(n+k)$ действий. □

Вообще эта задача довольно простая, просто выше мы привели решение постаравшись расписать все как можно подробнее. Настолько подробно наверное даже не надо, но из того что выше я надеюсь станет немного понятнее что имеется в виду когда говорят о том что надо расписывать решения достаточно подробно – вы должны привести алгоритм, доказательство корректности и оценку сложности. Причем доказательство корректности хоть и кажется очевидно следующим из алгоритма должно быть проведено достаточно аккуратно, в нем вы должны не полагаться на приведенный алгоритм, а пытаться его сломать, если так проще.

Еще одна вещь которую здесь можно обсудить это сложность по памяти. В данном случае мы просто считали обе последовательности, т.е. мы храним все $n+k$ чисел. Но на самом деле в решении мы это особо не используем. Если бы у нас было два источника последовательностей (вроде istream в $c++$) то нам бы было достаточно просто считывать один символ последовательности y, а затем считывать элементы последовательности x пока не найдется подходящий, таки образом по памяти получилось бы вообще $O(1)$.

Задача 2.3 (Задача 6 из листка 1)**.** На вход подается число k и последовательность из нулей и единиц, которая заканчивается специальным маркером конца ввода \$. Докажите, что любой онлайн-алгоритм, который проверяет, что на k-ом месте от конца последовательности стоит 1 использует $\Omega(k)$ битов памяти.

Решение. *TL;DR: Очевидный алгоритм для решения с $\Omega(k)$ – хранить просто последние k элементов последовательности и обновлять их на каждом шаге. Вопрос в том как показать что не хватит $k-1$ бита. Решение примерно такое – $k-1$ бит позволяют фактически поддерживать 2^{k-1} состояний, а последовательностей на k битах больше, значит по принципу Дирихле наш алгоритм может оказываться в одинаковых состояниях для разных строк, и дальше надо аккуратно расписать что все сломается.*

Здесь надо начать с небольшой оговорки по поводу онлайн-алгоритмов. В разделе с теорией выше мы написали что это алгоритм который вычисляет $f(x_1,\dots,x_j)$ после каждого считанного x_j. Тут это можно понимать так что "если алгоритм на встретил \$ то он не выводит ничего, и это и является результатом, если же x_j это спецсимвол то он должен что-то вывести". Короче ключевым для нас является то что мы не знаем изначально длину последовательности (т.е. то в какой момент будет нами считано \$).

Пойдем от противного – предположим что существует алгоритм который работает с использованием меньшего объема памяти, допустим используемая память это функция вида $g(k)$. Причем про ее порядок роста можно сказать то что $\lim\limits_{k\to+\infty}\frac{g(k)}{k}=0$, что в свою очередь значит что существует K такое, что $\forall k>K$ выполнено $g(k)\leqslant k-1$ (нам этого хватит).

Итак, мы предположили что есть более эффективный по памяти алгоритм, и как мы увидели из этого следует что для какого-то k он тогда должен будет потреблять меньше k бит памяти (последние два параграфа это просто попытка описать аккуратно почему мы можем вообще рассматривать алгоритм с $k-1$ битом памяти).

Теперь, алгоритм в каждый момент времени находится в пространстве состояний, определяемом тем какую память он имеет и что он сейчас считывает (по аналогии с машинами Тьюринга). Имея $k-1$ бит памяти это пространство содержит 2^{k-1} состояний.

Рассмотрим последние k элементов последовательности. Всего их вариантов могло быть 2^k. Но у нашего алгоритма возможных состояний меньше – 2^{k-1}! Это значит, что образовалась примерно следующая ситуация – алгоритм после считывания последних k символов двух разных последовательностей оказался в одном состоянии (разных именно в плане того что есть некоторый номер j из последних k такой что $x_j\neq y_j$).

Осталось только дальше выдать алгоритму последовательности одинаковые (т.е. после x_k и y_k продолжить выдавать одно и то же значение), т.е. в итоге имеем нечто вот такого вида:

$$\dots x_1,\, x_2 \dots x_{j-1},\, x_j,\, x_{j+1} \dots x_k,\, 0,\, 0,\, 0,\, 0,\, 0 \dots 0,$$
$$\dots y_1,\, y_2 \dots y_{j-1},\, y_j,\, y_{j+1} \dots y_k,\, 0,\, 0,\, 0,\, 0,\, 0 \dots 0.$$

Итак, после считывания строк до x_k или y_k включительно алгоритм оказался в одном и том же состоянии (в каждой ячейке памяти записано одно и то же). В силу детерминированности (а мы рассматриваем только детерминированные алгоритмы на данный момент) и того что дальше мы ему даем одно и то же (нули) он продолжает находится в одинаковом состоянии для обеих строк. Осталось в какой-то момент дать ему \$ так чтобы на k-й с конца позиции оказалось x_j и y_j соответственно. Как мы помним они не равны, так что алгоритм в одном из случаев совершит ошибку.

Тем самым алгоритма с порядком скорости роста по памяти медленнее чем линейной не существует. □

Тут еще можно обсудить сюжет связанный со скоростью работы алгоритма решающего эту задачу. Самое простое что можно придумать – хранить просто последние k считанных бит последовательности, и на каждом шаге делать обновление вида

$$m_k = m_{k-1},\ m_{k-1} = m_{k-2},\ \dots\ m_2 = m_1,\ m_1 = x_i$$

а в момент когда нам покажут \$ вывести m_k (здесь m от $memory$).

Однако это довольно так себе, потому что тогда на каждом шаге мы будем совершать k действий присвоения, и на последовательности длины n (ну т.е. в случае когда символ \$ находится на n-й позиции) алгоритм сделает $O(kn)$ операций. Можно однако еще завести переменную $last$ и на каждом шаге делать $last = last\%k + 1$ и считывать символы циклично в m_{last} (т.е. на каждом шаге делать $m_{last} = x_i$). Тогда когда мы встретим \$ надо будет как раз вывести m_{last}, это и будет ответом. И это уже позволяет говорить об $O(n)$ операциях.