

Экзаменационные билеты

Алгоритмы. 5 курс. Весенний семестр.

Горбунов Егор Алексеевич

13 июня 2016 г.

1 Декартово дерево

Определение 1. Декартово дерево – это бинарное дерево, в каждой вершине которого хранится пара (k, p) , причём декартово дерево является деревом поиска по ключу k и кучей по приоритету p .

Лемма 1. Пускай нам дан набор пар $(k_1, p_1), (k_2, p_2), \dots, (k_n, p_n)$, причём все p_i различны. Тогда существует единственное декартово дерево, построенное по этому набору пар.

Доказательство. Будем рекурсивно (по индукции) строить декартово дерево. Все приоритеты p_i различны, а значит среди пар можно выбрать единственную, у которой приоритет $p_i = p_{\max}$ максимальный. Относительно этого приоритета все пары мы можем однозначно разбить на две группы: в «левую» группу отправим пары у которых $p_i < p_{\max}$, а в правую те, у которых $p_i \geq p_{\max}$. Таким образом разделили исходную задачу размера n на более мелкие две. Построим соответственно декартово дерево на «левой» и «правой» группе и присоединим к корневой паре с приоритетом p_{\max} . Дерево построено, значит существует. На каждом уровне выбор корня делается единственным способом (тут используем единственность приоритетов), а значит дерево единственно. \square

Замечание 1. Алгоритм, описанный в доказательстве леммы отработает за $\mathcal{O}(n^2)$.

1.1 Построение за линейное время

Пускай нам дан уже отсортированный по ключам в порядке возрастания набор пар $(k_1, p_1), \dots, (k_n, p_n)$. Построим по ним декартово дерево быстро – за $\mathcal{O}(n)$. Декартово дерево – бинарное дерево поиска по ключам k_i , а это значит, что при добавлении ключей в порядке возрастания, добавляемая вершинка должна оказаться самой правой (у нас в левой веточке ключи $< k$, а в правой $\geq k$). Заметим таким образом, что мы всегда можем добавлять новую вершину x в правую ветвь декартова дерева:

1. В самой правой ветви найти такие две вершины a и b (b – непосредственный сын a), что $a.p > x.p$ и $x.p > b.p$ (всякие граничные случаи где $a = NIL$ или $b = NIL$ не берём в голову, сейчас пофиг)
2. Перекинуть ссылки на сынишек: $x.l = b$, $a.r = x$

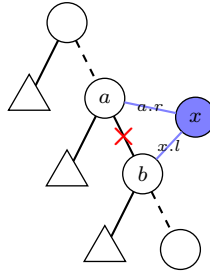


Рис. 1: Вставка вершины x при построении декартова дерева

Как находить такие вершины a и b ? Можно спускаться от корня вниз и вправо. Но тогда представим себе такую ситуацию: на вход подали набор пар $(1, 1), (2, 2), (3, 3), \dots, (n, n)$. Пары отсортированы как надо. На каждой итерации алгоритма построения дерева новая вершина будет добавляться в самый конец пути вправо, таким образом для нахождения вершины a мы каждый раз будем спускаться вниз проходя по i вершин на i -ой итерации. Это как-то квадратично, а мы хотим за $\mathcal{O}(n)$.

Будем искать место для вставки с конца правой ветки. Почему это разумно? А потому, что веточка снизу ломается влево каждый раз при добавлении новой вершины там, куда её добавили. Это наводит на разные амортизационные мысли. Имеем дерево T , которое строится.

$$\Phi_i(T) = \text{длина правой ветки дерева перед } i\text{-ой итерацией}$$

Тогда амортизационная стоимость итерации (добавления одной вершинки) равна (тут t — это то, сколько мы прошли по правой ветки снизу в поисках места вставки)

$$\tilde{c}_i = t + \Delta_i \Phi = t + ((l - t + 1) - l) = t - t + 1 = 1 = \mathcal{O}(1)$$

Таким образом амортизационная оценка на построение всего дерева: $\mathcal{O}(n)$!

1.2 Операции вставки и удаления

Для вставки и удаления введём 2 дополнительные операции: Split и Merge

- Процедура *Split* принимает на вход дерево T и ключик k , а возвращает 2 декартовых дерева: в первом дереве все ключики $< k$, а во втором $\geq k$.

```

1 def Split(T, k):
2     if T == nil: return (nil, nil) # nil is empty tree or something like this...
3     if T.k < k:
4         (T.r, R) = Split(T.r, k)
5         return (T, R)
6     else:
7         (L, T.l) = Split(T.l, k)
8         return (L, T)

```

Логика проста: если ключ в корне дерева меньше k , то этот корень уже находится в искомой «левой» половине разбиения (сплита, разреза, как хотите!), а значит «правую» половину раз-

биения мы берём из рекурсивного разбиения правого сына, а левые остатки присоединяем к итоговой левой части.

- Процедура *Merge* принимает на вход два дерева L и R , причём таких, что ключи в первом дереве меньше либо равны ключам во втором и возвращает декартово дерево содержащее ключи обоих деревьев. Опять просто и рекурсивно:

```
1 def Merge(L, R):
2     if size(L) == 0: return R
3     if size(R) == 0: return L
4     if L.p > R.p:
5         L.r = Merge(L.r, R)
6         return L
7     else:
8         R.l = Merge(L, R.l)
9         return R
```

Тут мы просто уменьшаем задачу валидным способом...

- Вставка в дерево: процедура *Insert*. Процедура будет принимать дерево и новую вершину, а возвращать изменённое дерево.

```
1 def Insert(T, node)
2     L, R = Split(T, node.k)
3     return Merge(Merge(L, node), R)
```

Заметим, что мы используем тут целых 2 слияния и 1 сплит. Можно чуть лучше. Если у *node* приоритет выше, чем у *T*, то достаточно посплитить *T*, а потом присоединить результат к *node*. Если же приоритет меньше, то мы можем спуститься по *T* как по дереву поиска до той вершины, где её приоритет будет выше.

```
1 def Insert(T, node):
2     if node.p > T.p:
3         node.l, node.r = Split(T, node.k)
4         return node
5     if node.k < T.k:
6         T.l = Insert(T.l, node)
7     else:
8         T.r = Insert(T.r, node)
9     return T # root of result tree is T
```

- Удаление из дерева: процедура *Remove*. Принимаем дерево и ключ, а возвращаем дерево, из которого вершинка с ключиком была удалена.

```
1 def Remove(T, k)
2     L, R = Split(T, k)
3     D, R = Split(R, k)
4     return Merge(L, R)
```

Тут делается 2 сплита и 1 слияние, но можно лучше. Ясно, что если $T.k == k$, то нам просто нужно вернуть слитых левого и правого сына *T*. Этим и воспользуемся:

```

1  def Remove(T, k)
2      if T.k == k:
3          return Merge(T.l, T.r)
4      if k < T.k:
5          T.l = Remove(T.l, k)
6      else:
7          T.r = Remove(T.r, k)
8      return T

```

Замечание 2. Легко видеть, что время работы операций Split и Merge — $\mathcal{O}(h)$, где h — высота дерева. А значит аналогично и время работы вставки и удаления.

1.3 Дуча (Treap, Дерамида)

Определение 2. Дуча — это декартово дерево, приоритеты в котором случайные и равномерно распределённые.

Лемма 2. Математическое ожидание высоты дучи равно $\mathcal{O}(\log n)$, где n — число вершин в ней.

Доказательство. Обратимся к рекурсивному (тот, что за $\mathcal{O}(n^2)$) алгоритму построения декартова дерева. Заметим, что в силу случайности приоритетов мы равновероятно выбираем любой ключ при построении дерева, относительно которого все остальные ключи будут разбиты по группам $<$ или \geq . Это в точности та же процедура, которая происходит при применении быстрой сортировки. И дерево рекурсии для этих задач идентичны. Умеем доказывать, что дерево рекурсии QuickSort будет иметь высоту, в среднем (из-за случайности процесса, а не по входам), логарифмическую! \square

Выводы: таким образом Treap — это дерево поиска, операции на котором будут работать за $\mathcal{O}(\log n)$.

2 Задачи RMQ и LCA

Определение 3. Дан массив A . Задача **RMQ** или Range Minimum Query заключается в том, что по запросу $[l, r]$, $l \leq r$, $l, r \in \mathbb{N}$ нужно найти $\min_{i \in [l, r]} A[i]$.

Определение 4. Пусть дано дерево T . Задача **LCA** или Least Common Ancestor заключается в том, что по запросу (v, u) , где v и u есть вершины дерева T , нужно найти вершину $p \in T$ такую, что p — общий предок v и u и при этом он самый близкий к ним, т.е. нету вершины p' , которая общий предок v и u и при этом её глубина больше глубины p .

Задачи RMQ и LCA можно решать в разных условиях: когда исходный массив или дерево могут меняться (динамическая задача) или когда они даны и неизменны (статическая задача).

Замечание 3. Заранее скажем, что существует круговорот RMQ и LCA в природе, а именно: RMQ сводится к LCA, а LCA сводится к $\text{RMQ} \pm 1$, что есть частный случай задачи RMQ.

Замечание 4. В лоб задачи LCA и RMQ решаются за $\mathcal{O}(n)$ (ответ на один запрос, на дерево никаких ограничений в духе сбалансированности нет). Но мы научимся (возможно, описано тут некоторое не будет) решать быстрее:

	Дерево отрезков	Полный предподсчёт	Sparse Table
предподсчёт	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
запрос	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
динамически?	да	нет	нет

Таблица 1: Задача RMQ

	Двоичные подъёмы	Полный предподсчёт	Ladder decomposition
предподсчёт	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
запрос	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
динамически?	нет	нет	нет

Таблица 2: Задача LCA, если решать именно как LCA :) Под Ladder Decomposition понимается разложения дерева на «длиннейшие» пути, а потом наворачивание поверх этого двоичных подъёмов (самый обычных), в этом документе про это точно не будет.

Так же мы научимся сводить RMQ к LCA и наоборот за линейное $\mathcal{O}(n)$ время, так что методы решения одной задачи можно использовать для решения другой.

2.1 Динамический RMQ и RSQ. Дерево отрезков

Пусть у нас есть ассоциативная операция \circ , например \min или $+$. Заметим тогда, что при вычислении $A[l] \circ A[l+1] \circ A[l+2] \circ \dots \circ A[r]$ мы можем в силу ассоциативности расставлять скобки как хотим. Построим такую структуру данных, которая нам будет разбивать отрезок запроса на некоторое число *непересекающихся* подотрезков, на которых ответ уже посчитан. Тогда, в силу ассоциативности, мы можем получить ответ на всё отрезке. Далее в качестве операции будем рассматривать, например, $+$.

Определение 5. *Дерево отрезков* — это такое бинарное дерево, что его корень ассоциирован со всем массивом $A[1..n]$, а дети вершины, которой соответствует кусок массива $A[l..r]$, ассоциированы с кусками массива $A[l..m]$ и $A[m+1..r]$, где $m = \frac{l+r}{2}$. См. рисунок 2

В каждой вершине такого дерева мы можем хранить ответ на запрос на отрезке, которому эта вершина соответствует.

2.1.1 Построение дерева отрезков

Построить дерево можно за линейное время. Считаем, что длина исходного массива $n = 2^k$. Всего вершин в дереве $2n$, т.к. это полное бинарное дерево на n листьях (можешь доказать это через лемму о рукопожатиях, например). Само дерево можно хранить в массиве $tree[2*n]$ так, что $tree[1]$ — это корень дерева отрезков, который соответствует всему массиву. А дети i -ой вершины — это вершины $tree[2*i]$ и $tree[2*i+1]$. Т.е. отец вершины $tree[j]$ — это $tree[j / 2]$.

```

1 def build(arr):
2     tree = [0] * (2 * len(arr) + 1)
3     tree[len(arr) + 1:] = list(arr)
4     for i in reversed(range(1, len(arr)+1)):
5         tree[i] = tree[2 * i] + tree[2 * i + 1]
6     return tree

```

Ясно, что строим за $\mathcal{O}(n)$.

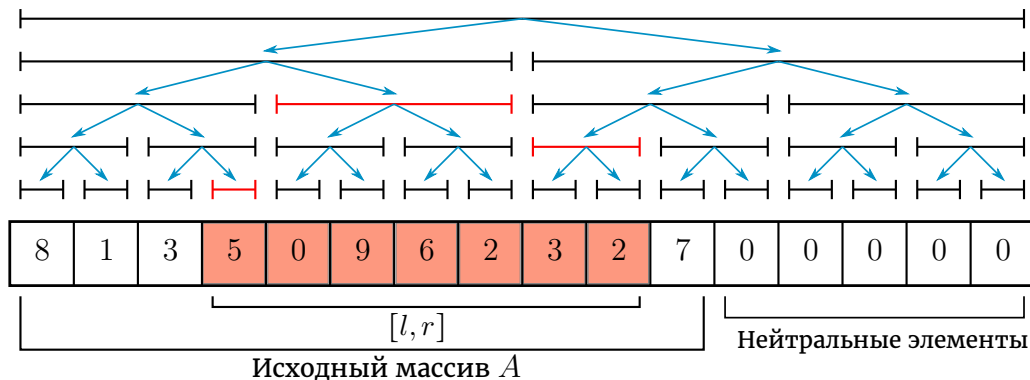


Рис. 2: Дерево отрезков для данного массива A . Если размер A не степень двойки, то добъём до степени двойки нейтральными элементами относительно той операции, которую хотим считать на отрезках массива.

2.2 Ответ на запрос и оценка на время работы

Отвечать на запрос о сумме (минимуме, ...) на отрезке будем декомпозируя искомый отрезок на куски, которые представляют из себя вершины дерева отрезков.

```

1 def sum(v, l, r, ql, qr):
2     """
3     v - индекс вершины дерева отрезков в массиве tree
4     l, r - границы отрезка, за которые вершина v отвечает
5     ql, qr - текущие границы запроса, ql >= l и qr <= r
6     """
7     if ql > qr:
8         return 0
9     if l == ql and r == qr:
10        return tree[v]
11    m = (l + r) / 2
12    return sum(2*v, l, m, ql, min(m, qr)) + sum(2*v+1, m+1, r, max(m+1, ql), qr)

```

Нужно понять, как быстро это работает?

Лемма 3. Вызов `sum(1, 0, n-1, l, r)` отработывает за $\mathcal{O}(\log n)$.

Доказательство. Заметим следующую вещь. Всего может быть 2 случая:

1. Если какая-то граница отрезка $[l, r]$ совпала с какой-то границей отрезка $[0, n-1]$. Видим

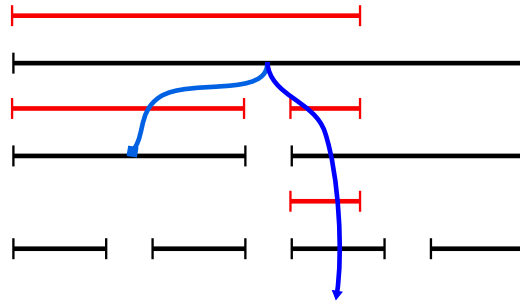


Рис. 3: Хотя бы одна из границ совпала.

тогда, что у нас один из ветвей рекурсии сразу же остановится и останется ещё всего лишь одна. Т.е. можно считать, что число ветвей рекурсии не увеличивается.

2. Если у нас ни одна из границ не совпала и при этом отрезок $[1, r]$ не пересекает середину отрезка, соответствующего вершине, то тогда у нас опять же число ветвей рекурсии на этом шаге не увеличивается.
3. Если же ни одна из границ не совпала, а отрезок $[1, r]$ пересекает середину, то ситуация получается такой: Видим тогда, что у нас увеличилось число ветвей рекурсии. Их стало две. Но

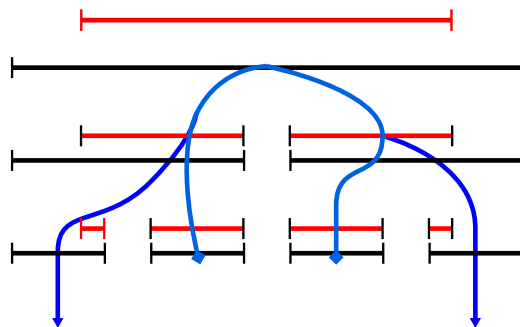


Рис. 4: Отрезок $[1, r]$ пересекает середину отрезка вершины

заметим тогда, что обе эти ветви рекурсии удовлетворяют случаю 2.. А значит, что ни одна из порождённых ветвей рекурсии не расплодится (больше чем на один шаг).

Получили таким образом, что максимальное число ветвей рекурсии равно 4 (это видно на рисунке 4), но это значит, что время работы ответа на запрос пропорционально высоте дерева отрезков, а высота этого дерева, очевидно, $\log n$. Вот и получили $\mathcal{O}(\log n)$ на запрос. \square

Замечание 5. Из леммы так же следует, что мы можем любой отрезок разбить на $\mathcal{O}(\log n)$ отрезков из дерева (канонических отрезков). Иначе мы бы не могли получить такой асимптотики.

2.2.1 Обновление элемента массива

Тут всё просто. Каждый элемент массива покрывается ровно $\log n$ каноническими отрезками. Это, к слову, хорошо видно из рисунка 2. Таким образом нам нужно просто обновить значение на всех этих отрезках. Это делается либо проходом сверху вниз, либо снизу вверх (тут нам помогает наш способ хранения дерева через массив и удобная адресация к родителю вершины).

```

1 def update(v, l, r, idx, val):
2     if l == r and l == idx:
3         tree[l] = val
4     else:
5         m = (l + r) / 2
6         if idx <= m:
7             update(2*v, l, m, idx, val)
8         else:
9             update(2*v+1, m+1, r, idx, val)
10        tree[v] = tree[2*v] + tree[2*v+1] # or min(tree[2*v], tree[2*v+1])

```

Тут совсем ясно, что мы тратим ровно высоту времени, т.е. $\mathcal{O}(\log n)$

Замечание 6. Быстрее делать обе операции (запрос и изменение) нельзя. Если бы было можно, то и массив бы мы отсортировали быстрее, чем за $\mathcal{O}(n \log n)$ (достаём минимум, заменяем элемент по его индексу на ∞), а это теоретически невозможно.

2.3 Статический RMQ. Sparse table

Пусть у нас теперь исходный массив фиксирован и не меняется. Нужно лишь уметь отвечать на запросы минимум на отрезке $[l, r]$.

1. Полный предподсчёт. Можно просто взять и для всех пар подсчитать ответ за $\mathcal{O}(n^2)$, после чего отвечать на каждый запрос за $\mathcal{O}(1)$.
2. **Sparse Table** или Разреженная таблица. В данной ситуации нам важно, что операция, которую мы хотим считать на массиве, *идемпотентна*. Заметим такую вещь: если исходный запрос $[l, r]$ мы разобьём на 2 пересекающихся подотрезка $[l, l+2^k]$ и $[r-2^k, r]$, то $\min_{[l, r]} = \min(\min_{[l, l+2^k]}, \min_{[r-2^k, r]})$. Тут 2^k максимальное такое, что $r - l + 1 \geq 2^k$. Суть в том, что такие 2 отрезка гарантированно покроют весь отрезок $[l, r]$. Суть метода Sparse Table в том, что мы предподсчитываем ответы на запросы на всех отрезках, длина которых — степень двойки. Вплоть до длины $\log n$. Таким образом предподсчёт у нас займёт $\mathcal{O}(n \log n)$. Идемпотентность нам нужна, т.к. отрезки, на которые мы разбиваем запрос, могут пересекаться.
3. Для операций, у которых есть обратная операция, мы можем предподсчитать за $\mathcal{O}(n)$ результаты на префиксах массива, а потом отвечать за $\mathcal{O}(1)$.

2.4 LCA. Эйлеровы обходы. Сведение к RMQ и наоборот

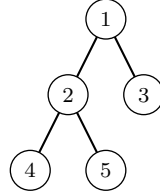
2.4.1 Сведение RMQ к LCA

Пусть у нас есть массив $A[1..n]$. Рассмотрим набор пар $(A[i], i)$ и построим по нему декартово дерево, где i (индексы в массиве) будут ключами, а приоритетами будут $A[i]$. Ясно, что т.к. полученное дерево будет деревом поиска по индексам и кучей по элементам массива, то наибольший общий предок

вершины $(A[l], l)$ и вершины $(A[r], r)$ — это такая вершина $(A[i], i)$, что $l \leq i \leq r$ и $A[i] \leq A[l], A[r]$. Это ясно по построению и свойствам дерева.

2.4.2 Эйлеров обход

Эйлеровы обходы бывают разные. Пускай у нас есть дерево: Эйлеров обход — это своеобразный про-



токол обхода в глубину данного дерева. Эйлеровы обходы бывают следующие:

- (а) *Обход по рёбрам.* Обходим дерево начиная с корня по рёбрам слева направо. Проходя по ребру добавляем его к обходу.

$(1, 2), (2, 4), (4, 2), (2, 5), (5, 2), (2, 1), (1, 3), (3, 1)$

- (b) *Обход по вершинам с повторениями.*

$1, 2, 4, 2, 5, 2, 1, 3, 1$

- (с) *Обход по вершинам, где вершина добавляется в обход при входе в её поддерево и при выходе.*

$1, 2, 4, 4, 5, 5, 2, 3, 3, 1$

Будем эйлеровым обходом называть обход по вершинам с повторениями. Заметим следующую вещь. Пронумеруем все вершины в порядке обхода *в ширину*, т.е. чтобы у вершины с меньшей глубиной всегда был больший номер. Теперь пусть мы хотим найти LCA вершин a и b . Тогда посмотрим на индекс i первого вхождения a в эйлеровом обходе и индекс r первого вхождения j в эйлеровом обходе. Не умаляя общности пускай $i < j$. Что может находиться между позициями i и j в эйлеровом обходе? В силу устройства эйлерова обхода, если b не находится в поддереве a , то между i и j может лежать единственный общий предок a и b и причём он будет наименьшим (least common ancestor) и он будет иметь наименьший номер. Если же b лежит в поддереве a , то a и будет ответом на запрос. Заметим, что в силу нумерации получается, что для ответа на запрос нам нужно найти минимум на отрезке $[i, j]$ в эйлеровом обходе. Таким образом будем хранить массив $first[1..n]$ первых вхождений каждой вершины. Если потребуется найти $lca(a, b)$, то нужно спрашивать $rmq(first[a], first[b])$. Свели.

Замечание 7. Какова длина эйлерова обхода? Если смотреть на то, как мы ходим по рёбрам, то можно заметить, что в обход мы добавляем конец каждого ребра (когда идём вниз) и начало каждого ребра (когда поднимаемся вверх), плюс ещё корень, который появляется вначале, потому что с него начинаем (очевидно). Итого $2(n - 1) + 1 = 2n - 1$. А значит сведение работает за $\mathcal{O}(n)$.

Замечание 8. Существует алгоритм решения $RMQ_{\pm 1}$ за $\mathcal{O}(n)$ предподсчёт и $\mathcal{O}(1)$ на запрос. А т.к. мы всё ко всему умеем сводить, то для LCA и RMQ в общем случае тоже можно получить такую асимптотику.

3 Хеширование 1

3.1 Прямая адресация. Хеширование. Коллизии

3.2 Методы разрешения коллизий

3.3 Гипотеза равномерного хеширования

3.4 Оценка времени поиска в хеш-таблице