Теоретическое домашнее задание
по алгоритмам и структурам данных
Задание №2

Автор: *Ряжских Дмитрий*

МФТИ 2024

1 Задача 1 (Вдвое больший справа)

1.1 Идея решения

Создаём массив (B[1:n]) пар вида { A[i], i }. Отсортируем этот массив в обратном порядке и по отсортированному массиву будем последовательно брать элементы, добавляя их в очередь обработки (inProcess), при выходе из которой их индексы будут добавляться в декартовое дерево (possibleIndexes).

1.2 Алгоритм

- Общий алгоритм
  1. Создаём описанные выше структуры: B, inProcess, possibleIndexes
  2. Заполняем массив B парами: B[i] = { A[i], i }
  3. Сортируем B по убыванию любой сортировкой, асимптотика которой не хуже O(NlogN), например, MergeSort
  4. Для каждого элемента current из B выполняем пункты 5-8:
  5. Пока крайний элемент очереди (firstInQueue) не меньше, чем current[1] * 2, удаляем из очереди крайний элемент и добавляем в possibleIndexes значение firstInQueue[2]
  6. Если possibleIndexes пустая, то ответом для элемента A[current[2]] будет None
  7. Иначе ответом для A[current[2]] будет possibleIndexes.next(current[2])
  8. Добавляем в inProcess текущий элемент (current)
- Реализация possibleIndexes::next(value)
  1. Заводим переменную result = None
  2. Начинаем обходить ДД, начиная с корня (текущую вершину обозначим за node):

3. Если node $\neq$ None выполняем шаги 4-5
4. Если значение в текущей вершине больше value, то result = min(result, node.value) и переходим в левого ребёнка node
5. иначе переходим в правого ребёнка node
6. В качестве ответа возвращаем result
7. Примечание: None является нейтральным элементом по операции min, то есть min между None и любым другим другим значением равно другому значению

1.3 Краткое объяснение решения

Так как мы обрабатываем значения в порядке убывания, и все обработанные значения хотя бы в 2 раза больше элемента, для которого считаем ответ, то если для этого элемента есть ответ, то он находится среди индексов обработанных значений, так как любые другие значения не подходят под условие. Именно для этого мы и храним индексы в куче, для быстрого поиска ближайшего справа среди всех доступных.

1.4 Асимптотика

Создание массива B занимает O(N), а его сортировка O(NlogN). Дальше каждый элемент B обрабатывается в очереди не больше чем 2 раза: при добавлении и удалении, то есть за O(1), а ответ для него считается за O(logN), так как используется функция possibleIndexes::next. То есть, несмотря на то, что за одну итерацию из очереди может удалится до N элементов, суммарно за все итерации удалится не более N элементов. Постобработка удалённого из очереди элемента занимает O(logN) (добавление в ДД). Итоговая асимптотика по времени получается O(N + NlogN + N($1^{*}logN$ + logN + 1)) = O(NlogN).

Каждая из структур B, possibleIndexes и inProcess содержат в моменте не более N элементов, то есть асимптотика решения по памяти O(N).

2 Задача 2 (Прямая сумма и ее статистика)

2.1 Идея решения

Будем хранить кучу с минимальной суммой в корне; будем добавлять и удалять из неё элементы так, чтобы при i-м удалении мы получали i-ю порядковую статистику.

2.2 Алгоритм

- Инициализация кучи
  1. Создаём кучу, хранящяя элементы вида $\{ a_i + b_j, \text{j} \}$
  2. Куча поддерживает минимум по первому значению в элементе
  3. Для всех i от 1 до min(k, |a|) добавляем элемент $\{ a_i + b_1, 1 \}$ в кучу
- Подсчёт порядковой статистики
  1. Для i от 1 до k повторяем следующие шаги:
  2. берём минимальный элемент из кучи ($\{ (a \oplus b)_i, \text{j} \}$, где $(a \oplus b)_i$ обозначает i-ю порядковую статистику $(a \oplus b)$)
  3. Добавляем элемент $\{ a_i + b_{j+1}, \text{j+1} \}$ в кучу
  4. Последний взятый после k итераций элемент будет ответом

2.3 Доказательство решения

Так как массивы отсортированы, для ∀i, j выполняется $a_i + b_j \leq a_i + b_{j+1}$ и $a_i < a_{i+1}$. Из второго условия следует что для ∀i > k, j верно $a_i + b_j \geq$

$a_k + b_1 \geq a_{k-1} + b_1 \geq ... \geq a_1 + b_1$, то есть $a_i + b_j$ не может иметь k-ю порядковую статичтику, так как найдётся хотя бы k элементов меньше его, и его можно не рассматривать. Это обосновывает инициализацию кучи.

Обоснуем теперь подсчёт. Из первого условия абзаца выше следует, что, так как для каждого a_i в куче хранится минимальный из нерассмотренных в порядковой статистике b_j, в куче на данный момент хранится минимальный элемент $(a \oplus b)$, не считая уже исключённые.

2.4 Асимптотика

Инициализация стека требует не более O(k) времени, а при каждой итерации подсчёта добавление и удаление элемента из кучи происходит за O(logk). Так как при подсчёте происходит k операций, итоговая асимптотика решения по времени выходит O(klogk).

Куча требует O(k) дополнительных ячеек памяти, так как её максимальный размер равен k.

3 Задача 3 (Параллельный поиск)

3.1 Идея решения

Отсортируем запросы k_i по возрастанию, рекурсивно, начиная со среднего элемнта, будем сортировать через quickSort ключи так, чтобы на a_{k_i} месте был ответ на k_i запрос.

3.2 Алгоритм

- Функция **findOrders**(a[], k[], left, right)

Принимает на вход a[] и уже отсортированный массив k[], возвращает массив ans[], где ans_i - ответ на k_i запрос. left и right - соответственно левая и правая граница подмассива массива k[], который сейчас обрабатывается

1. Если left == right, ничего не делаем, иначе:
2. current = $k_{\frac{left+right}{2}}$ - какую порядковую статистику ищем
3. $ans_{\frac{left+right}{2}}$ = quickSelect(a[left:right], current - left)
4. Рекурсивно вызываем findOrders(a[left:current], k[], left, current) и findOrders(a[current+1:right], k[], current, right)

- Функция **quickSelect**

Принимает на вход a[] и некоторый k_i, возвращает k_i-ю порядковую статистику a[]

1. Берём случайный элемент из a[] (обозначим его pivot) и перенесём его в конец массива
2. Заведём счётчик-указатель на индекс leftPartEnd = 1
3. Для каждого элемента из a[] сравним его со значением pivot
4. Если он меньше либо равен, поменяем его местами с элементом с индексом leftPartEnd и увеличим leftPartEnd на 1
5. Если по итогу обхода всех элементов leftPartEnd оказался равен k_i, возвращаем значение pivot
6. Если leftPartEnd $< k_i$, то рекурсиво ищем порядковую статистику справа от pivot: return quickSelect(a[leftPartEnd+1:], k_i - leftPartEnd)
7. Если leftPartEnd $> k_i$, то рекурсиво ищем порядковую статистику слева от pivot: return quickSelect(a[:leftPartEnd], k_i)

- **Основной алгоритм**

---

1. Сортируем k[] любой сортировкой с асимптотикой не хуже O(NlogN) (Например, с помощью quickSort)
2. ans[] = findOrders(a[], k[], 0, |k|) - массив ответов на все запросы
3. Примечание: Если требуется отвечать на запросы последовательно в исходном порядке, то можно создать вспомогательный массив соответствий старый индекс-новый индекс на моменте сортировки k[]. На итоговую асимптотику это не повлияет

- *Подразумевается что в функцию и из функции всегда передаются ссылки так, как это происходит в Java*

3.3 Обоснование решения

quickSelect делит исходный массив на две части: меньше k-й порядковой и больше неё. Следовательно для любого i $<$ k i-я порядковая будет лежать в первой части, а для i $>$ k - во второй части, поэтому их можно рассматривать отдельно и независимо, что собственно и делает findOrders.

leftPartEnd указывает на pivot, так как pivot будет последним подходящим под условие в пункте 4 элементом, потому что стоит в конце рассматриваемой части массива.

3.4 Асимптотика

k[] сортируется за O(MlogM)

quickSelect без учёта рекурсии работает за O(N), а в рекурсию передаётся не весь массив, а только часть, которая в среднем равна $\frac{N}{2}$. То есть, асимптотика quickSelect равна

$$O(\sum_{i-0}^{\dots}(\frac{N}{2^i})) = O(N)$$

Теперь разберёмся с findOrders. При каждом вызове findOrders, она единожды вызывает quickSelect, а потом дважды рекуррентно себя, но от двух независимых частей массива a[]. То есть каждый следующий "уровень"рекурсии работает всреднем с массивом длиной $\frac{N}{2}$, где N - средняя длина массива на предыдущем "уровне". Несложно заметить, что уровней всего $log_2 M$. Откуда получаем асимптотику findOrders:

$$\sum_{i=0}^{log_2 M} 2^i * qs(\frac{N}{2^i}) = O(NlogM)$$

где qs(N) - асимптотическая сложность по времени функции quickSelect при передаче в неё массива длиной N.

Итоговая асимптотика: O(MlogM + NlogM) = O(NlogM), так как по условию M $\leq$ N

4 Задача 4 (Splay-explained)

4.1 Идея решения

—

4.2 Алгоритм

—

4.3 Обоснование решения

—

4.4 Асимптотика

—

5 Задача 5 (Непересекающиеся)

5.1 Идея решения

Будем использовать четыре множества (их можно построить, например, на ДД), которые поддерживают операции добавления элемента, нахождения k-й порядковой статистики и операций нахождения ближайшего большего/ближайшего меньшего элемента за O(logN). Первые два множества будут хранить левые и правые концы S_1, а другие два - S_2. Будем считать количество пар непересекающихся отрезков и вычитать их из общего числа пар отрезков.

5.2 Алгоритм

1. Заводим множество leftEndsOfS1, rightEndsOfS1, leftEndsOfS2 и leftEndsOfS2 - описанные выше множества концов

2. Заводим переменную countNotIntersections = 0, в которой будем хранить количество непересекающихся пар отрезков.

3. Пусть left и right - концы добавляемого отрезка (для определённости пусть мы добавляем в $|S_1|$, в противном случае всё симметрично с точностью до индексов)

4. Добавляем left и right в leftEndsOfS1 и rightEndsOfS1 соответственно

5. leftest = leftEndsOfS2.next(right) - самое левое начало, правее добавляемого отрезка

6. countMoreRight = |leftEndsOfS2| - leftEndsOfS2.order(leftest) + 1 - ищем количество отрезков из S_2, которые лежат правее добавляемого

7. rightest = rightEndsOfS2.previous(left) - самый правый конец, левее добавляемого отрезка

8. countLessLeft = rightEndsOfS2.order(rightest) - ищем количество отрезков из S_2, которые лежат левее добавляемого

9. countNotIntersections += countMoreRight + countLessLeft

10. Ответом на новый запрос будет $(|S_1| * |S_2| - countNotIntersections)$

5.3 Обоснование решения

Функции next и previous работают аналогично описанной в задании 1. Функция order за логарифмическое время находит порядковую статистику элемента. Для её работы достаточно в узле множества хранить количество детей слева и общее количество детей (задача поиска порядковой статистики в ДД за логарифм была в контесте). k-я порядковая статистика в случае пункта 8 означает, что начиная с k+1-го элемента отрезки имеют пересечение или лежат правее добавляемого, а ровно k отрезков лежат левее, в случае пункта 6 она означает, что начиная с этого номера все концы лежат правее.

5.4 Асимптотика

Добавление концов в leftEndsOfS1 и rightEndsOfS1 занимает $O(\log S_1)$, а подсчёт непересекающихся отрезков $O(\log S_2)$. Итоговая асимптотика $O(\log S_1 + \log S_1)$.