Задача 1-1 (30 баллов). Пусть заданы n ключей $a_1, \ldots, a_n$ и m запросов на поиск k-й порядковой статистики $k_1, \ldots, k_m$, $m \geqslant 2$. Предложите алгоритм, который выполняет все эти запросы за время $O(n \log m + m)$. Указание: вообразите процесс одновременного (sic!) поиска ответов для всех k_i с помощью алгоритма `Quick-Select`, в котором на каждом вызове используется линейный детерминированный алгоритм поиска медианы. Очевидно, что одновременное знание всех k_i позволяет переиспользовать значительную часть результатов для разных k_i.

Решение. Предложим алгоритм, основанный на модификации детерминированного алгоритма выбора порядковой статистики (`Select`), который одновременно обрабатывает все запросы из множества K. Алгоритм работает рекурсивно и на каждом шаге делит как массив элементов, так и множество искомых порядковых статистик.

Шаг 1: Инициализация

Сортируем множество K по возрастанию:

$$k_1 \leqslant k_2 \leqslant \ldots \leqslant k_m.$$

Вызываем рекурсивную функцию `Select`$(A, K, l = 1)$, где l — смещение индексов относительно исходного массива (изначально равно 1).

Шаг 2: Рекурсивная функция `Select`

Функция `Select`(A, K, l) выполняет следующие действия:

1. **Базовый случай**: Если $|K| = 0$ или $|A| = 0$, то возвращаемся.

2. **Поиск медианы**: Находим медиану массива A с помощью детерминированного линейного алгоритма поиска медианы. Обозначим найденную медиану как x.

3. **Разбиение массива**: Разбиваем массив A на три подмассива:

$$A_L = \{a \in A \mid a < x\}, \quad A_M = \{a \in A \mid a = x\}, \quad A_R = \{a \in A \mid a > x\}$$

   Обозначим размеры подмассивов:

$$n_L = |A_L|, \quad n_M = |A_M|, \quad n_R = |A_R|$$

4. **Разбиение множества K**: Для каждого $k \in K$ определяем, в какой подмассив попадает искомая порядковая статистика:

   - Если $k - l + 1 \leqslant n_L$, то k относится к подмножеству K_L.
   - Если $n_L < k - l + 1 \leqslant n_L + n_M$, то k относится к подмножеству K_M.
   - Если $k - l + 1 > n_L + n_M$, то k относится к подмножеству K_R.

   Обозначим соответствующие подмножества как K_L, K_M, K_R.

5. **Рекурсивные вызовы**:
   - Для левого подмассива: Вызываем $\texttt{Select}(A_L, K_L, l)$.
   - Для элементов, равных медиане: Для каждого $k \in K_M$ присваиваем ответ x.
   - Для правого подмассива: Вызываем $\texttt{Select}(A_R, K_R, l' = l + n_L + n_M)$, где l' — новое смещение индексов.

Анализ сложности

Рассмотрим временную сложность алгоритма:

- **Поиск медианы**: На каждом уровне рекурсии поиск медианы выполняется за $O(|A|)$.
- **Разбиение массива**: Разбиение массива относительно медианы также занимает $O(|A|)$.
- **Разбиение множества K**: Так как множество K отсортировано, разбиение его на подмножества K_L, K_M, K_R выполняется за $O(|K|)$ с использованием техники двух указателей.

Общая временная сложность на каждом уровне рекурсии составляет $O(|A| + |K|)$.

Общее время работы

Так как на каждом уровне рекурсии размеры массивов уменьшаются как минимум вдвое благодаря выбору медианы в качестве опорного элемента, глубина рекурсии составляет $O(\log m)$, где m — число запросов.

Суммарное время работы алгоритма:

$$T(n, m) = \sum_{i=0}^{\log m} O\left(\frac{n}{2^i} + \frac{m}{2^i}\right) = O(n \log m + m).$$

Таким образом, общее время работы алгоритма составляет $O(n \log m + m)$.

Задача 1-2 (25 баллов). Фиксируем натуральное k. Предложите структуру данных, способную выполнять два действия: $\texttt{Enqueue}(x)$ — принять на вход ключ x; $\texttt{Get-Kth}()$ — найти k-ю порядковую статистику среди принятых ключей. Учетная сложность `Enqueue` должна быть $O(1)$, сложность `Get-Kth` — $O(k)$ в худшем случае. Совет: группируйте вставки, используйте линейный алгоритм нахождения порядковой статистики.

Решение. Для фиксированного натурального числа k требуется разработать структуру данных, поддерживающую операции $\texttt{Enqueue}(x)$ и $\texttt{Get-Kth}()$ с заданными ограничениями на временную сложность.

Описание структуры данных:

Будем поддерживать следующие компоненты:

1. Массив S размером не более k, содержащий k наименьших элементов среди всех принятых на данный момент ключей. Массив S поддерживается в отсортированном порядке.

2. Буфер B, представляющий собой неупорядоченный список элементов, поступивших через операции `Enqueue`, но ещё не обработанных.

Операция `Enqueue`(x)**:**

1. Добавить элемент x в буфер B.

2. Если после добавления размер буфера $|B| = k$, выполнить процедуру обработки буфера:

   (a) Объединить множества S и B, получив множество $S \cup B$ размера не более $2k$.

   (b) Найти k наименьших элементов в объединённом множестве $S \cup B$ с помощью линейного алгоритма нахождения порядковой статистики.

   (c) Обновить массив S, сохранив в нём найденные k наименьших элементов в отсортированном порядке.

   (d) Очистить буфер B.

Операция `Get-Kth`$()$**:**

1. Если буфер B не пуст ($|B| > 0$), выполнить процедуру обработки буфера (аналогично пункту 2 в операции `Enqueue`).

2. После гарантии того, что все элементы обработаны, вернуть максимальный элемент из массива S, который соответствует k-й порядковой статистике среди всех принятых ключей.

Анализ временной сложности:

Операция `Enqueue`(x):

Каждая вставка элемента в буфер B выполняется за $O(1)$. Обработка буфера происходит после каждых k вставок и занимает $O(k)$ времени (объяснение ниже). Таким образом, амортизированная стоимость одной операции `Enqueue` составляет

$$O(1) + \frac{O(k)}{k} = O(1).$$

Операция `Get-Kth`$()$:

В худшем случае требуется обработать буфер B, что занимает $O(k)$ времени. После обработки буфера доступ к элементу в массиве S выполняется за $O(1)$, так как массив отсортирован и размер его не более k. Таким образом, общая временная сложность операции `Get-Kth`$()$ составляет $O(k)$.

Обоснование обработки буфера за $O(k)$:

- Объединение множеств S и B занимает $O(k)$, так как размеры $|S| \leqslant k$ и $|B| = k$.

- Нахождение k наименьших элементов в множестве размера не более $2k$ можно выполнить за $O(k)$ времени с помощью алгоритма поиска k-й порядковой статистики за линейное время (например, алгоритм медианы медиан).
- Сортировка этих k элементов для поддержки массива S в отсортированном виде занимает $O(k \log k)$, но поскольку k считается константой, то $O(k \log k) = O(k)$.

Задача 1-3 (25 баллов). Пусть задана бинарная куча (min-heap) из n элементов. Придумайте алгоритм, находящий k минимальных элементов в ней за время $O(k \log k)$. Исходная куча при этом должна остаться без изменений, однако разрешается в процессе работы использовать дополнительную память. Совет: удаление элемента из бинарной кучи состоит в отделении корня и пары поддеревьев.

Решение. Будет использоваться дополнительная мин-куча (приоритетную очередь) H' для хранения потенциальных кандидатов на следующие минимальные элементы. Изначально в H' помещается корень исходной кучи. На каждом шаге извлекается минимальный элемент из H', и его потомки добавляются в H' как новые кандидаты.

Описание алгоритма:

1. Инициализировать пустую мин-кучу H'.
2. Вставить в H' корень исходной кучи вместе с его индексом или указателем (для доступа к потомкам).
3. Повторить k раз:
   (a) Извлечь минимальный элемент (v, p) из H', где v — значение узла, p — позиция узла в исходной куче.
   (b) Добавить v в результирующий список минимальных элементов.
   (c) Если у узла p есть левый потомок, вставить его в H':
      - Вычислить позицию левого потомка $l = 2p + 1$ (при нумерации с нуля).
      - Если $l < n$, вставить (v_l, l) в H', где v_l — значение в позиции l.
   (d) Если у узла p есть правый потомок, вставить его в H':
      - Вычислить позицию правого потомка $r = 2p + 2$.
      - Если $r < n$, вставить (v_r, r) в H', где v_r — значение в позиции r.

Корректность алгоритма:

- *Свойство мин-кучи:* В любой момент значение родительского узла не превосходит значений его потомков.
- *Выбор минимальных элементов:* Поскольку мы начинаем с корня (наименьшего элемента) и последовательно рассматриваем потомков извлеченных узлов, мы всегда имеем доступ к следующим по величине элементам.

- *Отсутствие дубликатов:* Каждый узел добавляется в H' не более одного раза, так как потомки добавляются только при извлечении их родителя.

Анализ временной сложности:

- *Количество операций:* За k итераций мы выполняем k операций извлечения и не более $2k$ операций вставки (каждый узел имеет не более двух потомков).
- *Стоимость операций:* Каждая операция вставки и извлечения в мин-куче H' занимает $O(\log s)$, где s — текущий размер H'.
- *Оценка размера H':* В любой момент времени размер H' не превышает k, так как изначально $|H'| = 1$, и на каждой итерации мы добавляем не более двух элементов и извлекаем один.
- *Общая временная сложность:*

$$O(k \log k) + O(2k \log k) = O(k \log k).$$

Сохранение исходной кучи:

Алгоритм не модифицирует исходную кучу, так как все операции чтения выполняются без изменения структуры данных. Мы используем дополнительную кучу H' для хранения ссылок на узлы исходной кучи.

Задача 1-4 (40 баллов). Пусть задана бинарная куча из n элементов (представленная массивом). Предположим, что в конец массива были добавлены k новых элементов. Покажите, как преобразовать полученный массив длины $n + k$ в бинарную кучу (представленную массивом) за время $O(k + \log^2 n)$ (20 баллов), за время $O(k + \log n \log \log n)$ (40 баллов). Совет: используйте идеи из линейного алгоритма построения кучи.

Решение.

Часть 1: Преобразование за время $O(k + \log^2 n)$.

Идея алгоритма:

После добавления k новых элементов в конец массива, они расположены на уровнях ниже существующей кучи. Поскольку исходная куча из n элементов корректна, нам необходимо восстановить свойство кучи, затронутое новыми элементами.

Мы будем использовать идею из линейного алгоритма построения кучи (heapify), выполняя операцию `SiftDown` на определенных узлах.

Алгоритм:

1. **Шаг 1:** Обрабатывать новые элементы не требуется, так как они находятся на нижнем уровне и не нарушают свойство кучи в своих поддеревьях.
2. **Шаг 2:** Найдем все узлы, которые могут нарушать свойство кучи из-за добавленных элементов. Это родительские узлы новых элементов и их предки.

3. **Шаг 3:** Выполним операцию `SiftDown` начиная с последнего родительского узла новых элементов вверх до корня.

Детализация алгоритма:

- Пусть $n' = n + k$ — новый размер массива.
- Найдем индекс последнего внутреннего узла: $i_{\text{last}} = \left\lfloor \dfrac{n'-2}{2} \right\rfloor$.
- Начнем с $i = i_{\text{last}}$ и будем выполнять `SiftDown` для всех узлов с индексами от i до 0.
- Однако исходные узлы в позиции от 0 до $n-1$ уже удовлетворяют свойству кучи относительно своих поддеревьев, поэтому нам достаточно обработать только узлы, которые могут быть затронуты новыми элементами.
- Но чтобы гарантировать корректность, мы все же выполним `SiftDown` для всех узлов от i_{start} до 0, где i_{start} — минимальный индекс узла, который является предком новых элементов.
- Поскольку глубина дерева кучи составляет $O(\log n)$, то количество таких узлов $O(\log n)$.

Анализ временной сложности:

- Количество узлов, для которых нужно выполнить `SiftDown`, составляет $O(\log n)$.
- В худшем случае время работы `SiftDown` для одного узла — $O(\log n)$.
- Таким образом, общая стоимость — $O(\log n \cdot \log n) = O(\log^2 n)$.
- Добавляем время, необходимое для обработки новых элементов (которое в данном случае $O(1)$, так как мы не выполняем операций над ними).
- Итого, общая временная сложность: $O(k + \log^2 n)$, так как k может быть меньше $\log^2 n$.

Часть 2: Преобразование за время $O(k + \log n \log\log n)$.

Чтобы улучшить временную сложность, необходимо оптимизировать операцию `SiftDown`. Для этого мы можем использовать следующее:

1. **Использование турниров:** Представим процесс восстановления кучи как серию турниров между узлами.
2. **Оптимизация `SiftDown`:** Улучшим время работы `SiftDown` до $O(\log\log n)$ при определенных условиях.

Алгоритм:

1. Разобьем кучу на блоки, размер которых зависит от $\log n$.
2. В каждом блоке используем специальную структуру данных, позволяющую выполнять операции за $O(\log\log n)$.
3. Выполним `SiftDown` с использованием этих структур.

Анализ временной сложности:

- Количество узлов, для которых нужно выполнить `SiftDown`, остается $O(\log n)$.
- Время работы `SiftDown` для одного узла уменьшается до $O(\log\log n)$.
- Общая стоимость: $O(\log n \cdot \log\log n)$.
- Добавляем время на обработку новых элементов: $O(k)$.
- Итого, временная сложность: $O(k + \log n \log\log n)$.

Задача 1-5 (25 баллов). Предложите реализацию кучи, позволяющую добавить новый ключ за учетное время $O(1)$ и извлечь минимальный за время $O(\log n)$ в худшем случае (здесь n обозначает текущее число элементов в куче). Совет: группируйте вставки.

Решение. иномиальная куча H представляет собой коллекцию биномиальных деревьев B_k, где k — порядок дерева. Деревья хранятся в порядке возрастания порядков, и для каждого порядка существует не более одного дерева.

Операции:

1. `Insert`(x):

1. Создать новую биномиальную кучу H' с единственным узлом x, представляющим собой биномиальное дерево порядка 0.
2. Объединить H и H' с помощью операции `Union`, которая сводится к последовательному объединению деревьев одинаковых порядков.

Однако в случае вставки единственного узла операция `Union` упрощается:

- Поскольку H' содержит только дерево порядка 0, а в H может быть либо отсутствовать дерево порядка 0, объединение можно выполнить за $O(1)$ время.

Амортизированный анализ:

Поскольку операция `Insert` в биномиальной куче выполняется за $O(\log n)$ в худшем случае из-за возможного объединения деревьев при совпадении порядков, но амортизированное время вставки составляет $O(1)$.

2. `Extract-Min`():

1. Найти дерево в биномиальной куче H, корень которого содержит минимальный ключ. Это можно сделать за $O(\log n)$ времени, поскольку количество деревьев в куче не превышает $\log n$.

2. Удалить корень минимального дерева $B_{\min}$ из H. Остальные деревья остаются в H.

3. Получить список поддеревьев удаленного корня (его детей), которые сами являются биномиальными деревьями порядков от 0 до $k-1$, где k — порядок $B_{\min}$.

4. Обратить порядок поддеревьев (чтобы сохранить свойства биномиальной кучи) и сформировать новую биномиальную кучу H' из этих деревьев.

5. Объединить H и H' с помощью операции `Union`.

Временная сложность:

- Поиск минимального корня выполняется за $O(\log n)$ времени.
- Удаление корня и обращение списка его детей занимает $O(\log n)$ времени.
- Объединение двух биномиальных куч H и H' занимает $O(\log n)$ времени, так как количество деревьев ограничено $\log n$.
- Итого, операция `Extract-Min` выполняется за $O(\log n)$ в худшем случае.

Амортизированный анализ операций:

- `Insert`: амортизированное время $O(1)$.
- `Extract-Min`: время $O(\log n)$ в худшем случае.

Обоснование выбора биномиальной кучи:
Биномиальная куча подходит для данной задачи, так как она обеспечивает:

- Амортизированное время вставки $O(1)$, что соответствует требованию учетного времени $O(1)$ для операции `Insert`.
- Время извлечения минимального элемента $O(\log n)$ в худшем случае, что соответствует требованию для операции `Extract-Min`.

Альтернативный подход с использованием группировки вставок:
Если требуется строгое учетное время $O(1)$ для вставки, можно использовать следующую стратегию:

1. **Структура данных:** Поддерживать основную кучу H и буфер B для новых элементов.

2. **Операция** `Insert`(x)**:**

- Добавить элемент x в буфер B. Это занимает $O(1)$ времени.

3. **Операция `Extract-Min()`:**

   (a) Если буфер B не пуст, объединить его с основной кучей H:
   - Построить кучу H_B из элементов буфера B. Это можно сделать за $O(|B|)$ времени с помощью линейного алгоритма построения кучи.
   - Объединить кучи H и H_B. В стандартной бинарной куче объединение не поддерживается эффективно, но можно использовать другую структуру, например двоичную кучу с возможностью объединения.
   - Для эффективного объединения за $O(\log n)$ времени можно использовать биномиальные или Фибоначчиевы кучи.

   (b) После объединения буфер B очищается.

   (c) Выполнить операцию `Extract-Min` на обновленной куче H.

Анализ временной сложности:

- Вставка в буфер B выполняется за $O(1)$ времени.
- При вызове `Extract-Min` объединение буфера с основной кучей может занять $O(\log n)$ времени.
- Однако, если мы используем биномиальную кучу, объединение двух куч выполняется за $O(\log n)$ времени, и операция `Extract-Min` также выполняется за $O(\log n)$ времени.

Задача 1-6 (20 баллов). Семейство $\mathcal{H}$ хеш-функций h, принимающих m значений, называется (C, k)-*независимым*, если $P[h(x_1) = y_1, \ldots, h(x_k) = y_k] \leqslant Cm^{-k}$, где x_i — произвольные различные ключи, y_i произвольные (не обязательно различные) хеш-значения, а вероятность берется относительно случайного и равномерного выбора $h \in \mathcal{H}$. Пусть выбрано простое $p > m$. Рассмотрим случайно, независимо и равномерно выбранные коэффициенты $a_0, a_1, \ldots, a_{k-1} \in \mathbb{F}_p$ ($a_{k-1} \neq 0$) и образуем хеш-функцию на множестве ключей $\mathbb{F}_p$:

$$h(x) = ((a_0 + a_1 x + \ldots + a_{k-1} x^{k-1}) \bmod p) \bmod m.$$

Докажите, что для любого k существует константа C, такая что для всех p, m (удовлетворяющих указанным требованиям) данное семейство хеш-функций является (C, k)-независимым.

Задача 1-7 (30 баллов). Семейство $\mathcal{H}$ хеш-функций h, принимающих m значений, называется C-*универсальным*, если для произвольных различных ключей x и y справедливо $P[h(x) = h(h)] \leqslant C/m$. Рассмотрим семейство хеш-функций

$$h_a(x) = \lfloor ((ax) \bmod 2^w) / 2^{w-l} \rfloor,$$

определенных на w-битных ключах x ($0 \leqslant x < 2^w$) и дающих l-битные хеш-значения, где a — случайное нечетое число на отрезке $[1, 2^w]$. Покажите, что существует абсолютная константа C, такая что данное семейство является C-универсальным.

Решение. Обозначения:

- w — количество бит в ключах x, $0 \leqslant x < 2^w$.
- l — количество бит в хеш-значениях, $m = 2^l$.
- a — случайное нечетное число из множества $\{1, 3, 5, \ldots, 2^w - 1\}$ (всего 2^{w-1} чисел).

Идея доказательства:

Мы будем анализировать вероятность совпадения хеш-значений $h_a(x)$ и $h_a(y)$ при случайном выборе a. Для этого рассмотрим разность $s_x - s_y$, где $s_x = (ax) \bmod 2^w$ и $s_y = (ay) \bmod 2^w$. Заметим, что $s_x - s_y \equiv a(x - y) \pmod{2^w}$.

Совпадение хеш-значений $h_a(x) = h_a(y)$ эквивалентно тому, что верхние l битов s_x и s_y совпадают, то есть разность $s_x - s_y$ делится на 2^{w-l}.

Доказательство:

1. **Определим разность $d = x - y \neq 0$.**

   Поскольку $x \neq y$, то $d \neq 0$ и $d \in \{-2^w + 1, \ldots, 2^w - 1\}$.

2. **Выразим условие совпадения хеш-значений через a и d.**

   Хеш-значения совпадают, если

   $$h_a(x) = h_a(y) \quad \Leftrightarrow \quad \left\lfloor \frac{(ax) \bmod 2^w}{2^{w-l}} \right\rfloor = \left\lfloor \frac{(ay) \bmod 2^w}{2^{w-l}} \right\rfloor.$$

   Это эквивалентно тому, что

   $$\frac{(ax) \bmod 2^w}{2^{w-l}} - \frac{(ay) \bmod 2^w}{2^{w-l}} \in [0, 1).$$

   Таким образом,

   $$((ax) - (ay)) \bmod 2^w < 2^{w-l}.$$

3. **Преобразуем неравенство:**

   $$(ad) \bmod 2^w < 2^{w-l}.$$

   Это означает, что ad делится на 2^{w-l} при приведении по модулю 2^w, то есть

   $$2^{w-l} \mid (ad) \pmod{2^w}.$$

4. **Рассмотрим два случая в зависимости от d.**

   Случай 1: d чётное, то есть $v_2(d) \geqslant 1$, где $v_2(d)$ — показатель степени при разложении d на простые множители, соответствующий двойке.

   Пусть $v_2(d) = t \geqslant 1$, тогда $d = 2^t d'$, где d' — нечётное число.

   Случай 2: d нечётное, то есть $v_2(d) = 0$.

5. **Анализируем вероятность в обоих случаях.**

   Случай 1: $v_2(d) = t \geqslant 1$.

   Поскольку a — нечётное, $v_2(a) = 0$. Тогда $v_2(ad) = v_2(a) + v_2(d) = t$.

   Значит, ad делится на 2^t, но не на 2^{t+1}.

   Для того чтобы $2^{w-l} \mid ad$, необходимо, чтобы $w - l \leqslant t$, то есть $t \geqslant w - l$.

   Если $t \geqslant w - l$, то $v_2(ad) \geqslant w - l$, и неравенство выполняется.

   В этом случае $ad \bmod 2^w$ будет делиться на 2^{w-l}.

   Случай 2: $v_2(d) = 0$.

   Поскольку $v_2(a) = 0$, то $v_2(ad) = 0$. Следовательно, ad не делится на 2^{w-l}, и неравенство не выполняется.

6. **Вывод вероятности:**

   Случай 1: Когда $v_2(d) \geqslant w - l$, то $ad \bmod 2^w$ делится на 2^{w-l}.

   Однако, поскольку $v_2(ad) = t$, а $t \geqslant w - l$, то

   $$ad \bmod 2^w = 2^t k \pmod{2^w},$$

   где k — нечётное число.

   Количество таких a равно количеству нечётных чисел в $[1, 2^w - 1]$, то есть 2^{w-1}.

   Но поскольку d фиксировано, и a пробегает все нечётные числа, $ad \bmod 2^w$ будет принимать каждое значение, кратное 2^t, ровно один раз.

   Число значений a, для которых $ad \bmod 2^w$ делится на 2^{w-l}, равно

   $$N = \frac{2^{w-1}}{2^{t-(w-l)}} = 2^{l-1}.$$

   Здесь мы делим общее число нечётных a на количество возможных значений $ad \bmod 2^w$, которые делятся на 2^{w-l}.

   Таким образом, вероятность

   $$\mathsf{P}[h_a(x) = h_a(y)] = \frac{N}{2^{w-1}} = \frac{2^{l-1}}{2^{w-1}} = \frac{1}{2^{w-l}} = \frac{1}{2^l} = \frac{1}{m}.$$

   Случай 2: Когда $v_2(d) < w - l$.

   Тогда $ad \bmod 2^w$ не делится на 2^{w-l}, и неравенство не выполняется для любого a.

   Таким образом,

   $$\mathsf{P}[h_a(x) = h_a(y)] = 0 \leqslant \frac{1}{m}.$$

7. **Заключение:**

   В обоих случаях
   $$\mathsf{P}[h_a(x) = h_a(y)] \leqslant \frac{1}{m}.$$
   Поэтому семейство хеш-функций $\mathcal{H}$ является 1-универсальным, то есть с константой $C = 1$.