Студент: Маргарита Чудова
Группа: 2
Дата: 27 апреля 2022 г.

---

1. Предположим, что существует некоторый алгоритм, умеющий делать extractMin() за $o(\log n)$. После извлечения минимума в общем случае, очевидно, свойство кучи может нарушиться, при этом единственный инструмент, доступный для восстановления порядка — это сравнения и свопы элементов.
   Тогда существует такая куча, которая умеет делать сортировку за $n \cdot o(\log n) = o(n \log n)$ (просто добавим в кучу элементы как обычно, а извлекать будем новым extractMin). Но это противоречит теореме о нижней оценке на сортировку сравнениями! Значит, такого быть не может.

2. (a) Найдем среди первых k элементов массива патронов минимум a_{i_1} — очевидно, он и будет первым элементом в отсортированном массиве, поэтому поменяем его с элементом, стоявшим на первом месте — a_1. После сдвинемся на 1 и повторим процедуру: на отрезке $[2, k+1]$ найдем минимум a_{i_2}. Этот элемент не может стоять на первом месте в массиве, так как первый элемент мы уже нашли, значит он будет вторым элементом, поменяем его с исходным вторым элементом a_2.
   Продолжаем так до тех пор, пока не закончится массив: находим минимум на отрезке, ставим его на самое левое место отрезка, поменяв с исходным самым левым элементом, продолжаем. Минимум ищем за $\mathcal{O}(k)$, итого сложность $\mathcal{O}(nk)$

   (b) Все то же самое, что в предыдущем пункте, но для поиска минимума используем кучу. На самом первом шаге кладем в кучу k элементов, потом будем извлекать минимум ($\mathcal{O}(\log k)$), добавлять следующий элемент массива и просеивать его ($\mathcal{O}(\log k)$). Итого $\mathcal{O}(n \log k)$

   (c) Сведем задачу к некоторому количеству сортировок сравнениями — нам надо отсортировать весь массив, сравнивая его элементы, так что это возможно. Рассмотрим подмассив $[1, \ldots, 2k]$, его размер $2k$. Отсортируем этот массив, после сортировки первые k элементов будут стоять так, как должны в отсортированном массиве, так как из них ни один элемент не мог отклониться больше чем на k, а это все есть в выбранном подмассиве. Меньший подмассив взять не можем, так как рискуем не учесть какой-нибудь элемент и неправильно их расставить, больший нет смысла. Последние k элементов подмассива не обязательно стоят в нужном порядке. Повторим процедуру: возьмем подмассив $[2k+1, \ldots, 3k]$. Для него по тем же причинам первые k элементов стоят так же, как должны в отсортированном массиве, а следующие k элементов не обязательно.
   Дойдем таким образом до конца массива, только после этого он окажется отсортирован. Таким образом, один массив мы сортировали за $\Omega(k \log k)$ в худшем случае по теореме о нижней оценке на сортировку сравнениями, а всего массивов не более $\frac{n}{k}$. Тогда оценка сложности всего алгоритма $\frac{n}{k}\Omega(k \log k) = \Omega(n \log k)$

3. Обозначим время, прошедшее от начала укладывания гномом до укладывания гнома i за T_i, а время пробуждения гнома k за W_k. Тогда:

$$T_i = \sum_{k=1}^{i} a_i$$

$$W_k = T_{k-1} + a_k + b_k$$

   Мы хотим, чтобы для любого $i < k$ выполнялось (то есть чтобы любой гном i, уложенный спать до гнома k, просыпался позже, чем уложат k):

$$W_k - T_i > 0 \qquad (1)$$

Отсюда следует:

$$T_i - T_{k-1} < a_k + b_k \tag{2}$$

Из этого неравенства видно, что будет разумно максимизировать сумму $a_l + b_l$ для каждого укладываемого гнома, так как мы хотим, чтобы время пробуждения было больше времени укладывания. Поэтому заведем новый массив

$$c_i = a_i + b_i,\ \forall i$$

и отсортируем его по убыванию. При этом для каждого следующего гнома k проверяем условие (1) относительно предыдущего гнома, то есть время укладывания следующего гнома не должно быть больше времени сна предыдущего. Запись нового массива — $\mathcal{O}(n)$, сортировка $\mathcal{O}(n \log n)$.

Докажем корректность. Что, если у нас есть оптимальный алгоритм, в котором нет такого порядка, то есть для какого-то m $a_m + b_m < a_{m+1} + b_{m+1}$? Докажем, что гномов m и $m+1$ можно поменять местами и привести этот алгоритм к нашему. Для этого нам надо показать, что условия (2) для них эквивалентны. Запишем и преобразуем их:

$$\begin{cases} T_i - T_{m-1} < a_m + b_m \\ T_i - T_m < a_{m+1} + b_{m+1} \end{cases} \tag{3}$$

Для первого эквивалентность следует мгновенно:

$$T_{m-1} - T_i < a_m + b_m < a_{m+1} + b_{m+1} \tag{4}$$

Для доказательства эквивалентности второго вычтем из первого a_m:

$$T_i - T_{m-1} - a_m < a_m + b_m - a_m \Rightarrow T_i - T_m < b_m < a_m + b_m \tag{5}$$

4. (a) Будем помечать вершины в антиклике синим цветом, а остальные — красным, и помещать вершины в словарик с пометкой blue или red соответственно. Возьмем вершину с максимальной степенью (d), пометим ее синим, а ее соседей — красным. Среди непомеченых вершин возьмем любую, сделаем с ней то же самое — ее красим в синий цвет и добавим в антиклику, каждого из соседей проверяем на наличие в списке красных вершин (синими не могут быть, так как всех соседей синих вершин уже обработали), это займет константное время для отдельного соседа, в итоге для каждой вершины это займет не более $\mathcal{O}(d)$ времени. Если сосед там есть, то мы в него не заходим, если соседа там нет, то помечаем его красным, кладем в соответствующий set и больше не трогаем. Таким образом, на каждом шаге мы красим не более $d+1$ вершины и проходимся не более, чем по d ребрам. Таким образом, в антиклике вершин будет не меньше $\frac{n}{d+1}$. При этом каждая вершина будет покрашена один раз и мы сделаем не более, чем $\frac{n}{d+1} \cdot d$ проверок ребер, значит время работы алгорима $\mathcal{O}(n)$.

7. Первый станок работает без перерывов, как только одна деталь готова, берем следующую. Посчитаем время T, которое не работает второй станок — это время надо минимизировать.
Сразу после подачи первой детали второй станок не сможет работать в течение времени a_1. Как только первая деталь будет готова, можем передать на второй станок вторую деталь, тем временем первую пускаем на второй станок. Если $a_2 > b_1$, то простой второго станка равен $a_2 - b_1$, если a_2 меньше, то нулю.
Перейдем к следующей детали. Будем считать, что для предыдущей детали был выполнен худший сценарий и второй станок простаивал. Тогда для третьей детали простой станка аналогично равен $\max(a_3 - b_2, 0)$. Итого получаем оценку сверху на T:

$$T_n \le a_1 + a_2 - b_1 + a_3 - b_2 + \ldots + a_n - b_{n-1} = \sum_{k=1}^{n} a_k - \sum_{k=1}^{n-1} b_k$$

Таким образом, у нас получается два ряда, причем один из них мы хотим минимизировать, а другой максимизировать.

По здравому смылу: если у детали очень маленькое время обработки на первом станке, это хорошо, потому что можем в первый станок накидать побольше деталей, если b больше, то нам не очень важно насколько. Но, если b меньше, чем a, то будет простой. Получается, нам выгодно сначала взять детали с самыми маленькими a, а детали с самыми маленькими b убрать в конец, потому что они могут оказаться слишком маленькими.

Напишем что-то вроде сортировки: для каждой детали i будем сортировать по $\min(a_i, b_i)$, причем если минимум a_i, то мы кладем деталь в один массив, а если b_i — в другой, причем по убыванию (то есть минимальный b в конце), потом соединяем эти массивы, берем детали в получившемся порядке.

Некая попытка доказать, что это работает:

Будет логично требовать, чтобы время простоя второго станка для детали j было меньше, чем для детали $j + 1$

$$T_{j+1} - T_j = T_{\text{след}} - T_{\text{пред}} = a_{j+1} - b_j > 0$$

Предположим, есть оптимальный алгоритм, где что-то работает по-другому и нет такого же порядка. Пусть в нем поменяны местами (a_j, b_j) и (a_{j+1}, b_{j+1}) относительно исходного алгоритма (гайки идут в порядке $1, \ldots, j + 1, j, \ldots$). Тогда

$$T_{\text{след}} - T_{\text{пред}} = a_j - b_{j-1} > 0$$

При этом знаем, что $a_j < a_{j+1}$, $b_j > b_{j+1} > b_{j-1}$ (по построению первого алгоритма). Тогда можем безболезненно поменять j и $j + 1$ элемент.

$$b_i > \sum_{j>i} a_j$$