Лекция 7

Сортировка произвольных данных и порядковые статистики

7.1 Файл. Сортировка на четырех лентах

Вспомним, что *файл последовательного доступа* — это абстрактная структура данных, допускающая операции READ (с переходом к следующей записи), WRITE (с переходом к следующей записи) и REWIND. Его также можно рассматривать как ленту, по которой движется читающая/пишущая головка: каждый раз после чтения или записи она передвигается на одну позицию в направлении конца файла; по команде REWIND она возвращается к самому началу.

Разобьем исходный файл пополам на две ленты (последовательно считывая элементы, будем нечетные записывать на первую ленту, а четные — на вторую). Далее будем из двух лент, состоящих из отсортированных блоков по i элементов, составлять две ленты, состоящие из отсортированных блоков по $2i$ элементов (*распространенный прием*: для простоты будем считать, что количество элементов является степенью двойки 2^k, — в противном случае время работы вырастет заведомо не более, чем в константу раз, поскольку размер входа вырастет не более, чем в константу раз, даже если его округлить до степени двойки в большую сторону).

Делается это так: читаем поэлементно блоки с обеих лент (назовем эти ленты A и B), пишем блок удвоенной длины на одну ленту (назовем ее C) (а следующий — на другую, назовем ее D); при этом каждый раз на ленту C мы пишем наименьший элемент v из двух считанных (с ленты A и с ленты B) и читаем следующий элемент с той ленты, с которой взяли v (если текущий блок на ней еще не закончился).

Лемма 7.1. *Если ленты A, B длины 2^{k-1} состояли из блоков, отсортированных по i элементов, то после этой операции ленты C, D будут состоять из блоков, отсортированных по $2i$ элементов (и по-прежнему будут иметь длину 2^{k-1}). Эта процедура займет $O(n)$ операций считывания/записи элементов и $O(1)$ ячеек оперативной памяти.*

После этого ленты (A,B) и (C,D) меняются местами (читаем C и D, пишем на A и B). Очевидно, за t итераций ленты, отсортированные по 1 элементу, превратятся в ленты, отсортированные по 2^t элементов. Таким образом, мы доказали следующую теорему.

Теорема 7.1. *Приведенный алгоритм сортирует исходный файл за $O(n \log n)$ обращений к файлам.*

Замечание 7.1. Этот алгоритм полезно применять, когда данных так много, что они не помещаются в оперативную память (хотя дополнительной памяти на жестком диске он требует довольно много).

Также удобно этот алгоритм применять к спискам: в этом случае все происходит в оперативной памяти; при этом дополнительной памяти практически не требуется, так как вместо копирования данных на другую ленту происходит просто переброска указателей.

7.2 Массив

Абстрактное понятие массива. Линейный (полный) порядок. Сортировка массива.

ПРОБЕЛ В КОНСПЕКТЕ.

7.3 HeapSort

Замечание 7.2. Мы уже изучили один алгоритм, сортирующий файл за $O(n \log n)$ операций. Его можно применить и к массиву; однако, этот алгоритм использует $\Omega(n)$ ячеек дополнительной памяти. Следующий алгоритм использует лишь $O(1)$ ячеек (и также имеет сложность $O(n \log n)$). Однако при необходимости отсортировать большой файл следует использовать все же алгоритм сортировки на четырех лентах, поскольку он не требует нахождения всего массива в *оперативной* памяти.

Алгоритм HeapSort получил свое название от английского слова *heap* — куча. Неформально говоря, в этом алгоритме данные из массива организуются в виде «*кучи*»: двоичного дерева, в каждой вершине которого хранится элемент, не превосходящий элемента, хранящегося в родителе этой вершины.

Как нетрудно видеть, в таком представлении легко найти наибольший элемент массива: он находится в корне дерева. Удалив его из дерева и восстановив структуру кучи, мы сможем так же легко найти следующий по убыванию элемент, и т. д.

Таким образом, для достижения цели нам достаточно научиться строить кучу и восстанавливать правильность ее структуры после удаления ее корня. И то, и другое мы будем делать при помощи рекурсивной операции «*утапливания*» вершины: если в некоторой вершине хранится элемент, строго меньший элемента, хранящегося в одном из сыновей этой вершины, то этот элемент надо поменять местами с его сыном (с тем из двух сыновей, в котором ключ наибольший), а затем, если необходимо, продолжить его «утапливание».

Для того, чтобы описать этот алгоритм более строго, зафиксируем способ представления нашей кучи. Будем ее поддерживать в том же самом массиве, который нам дан. Укладка дерева в массив a производится следующим образом. Занумеруем дерево по уровням: корень — это $a[1]$, вершины следующего уровня — это $a[2]$ и $a[3]$, и т. д. При такой нумерации массив a содержит все элементы этого дерева, причем сыновья вершины $a[i]$ расположены в $a[2i]$ и $a[2i+1]$. Нам достаточно такого представления дерева, поскольку нам нужны лишь две операции: чтение конкретного элемента $a[i]$ и перестановка *содержимого* двух его вершин: $\mathrm{swap}(a[i], a[j])$.

Глобальная переменная *last* будет содержать номер последнего необработанного элемента массива (пока это просто n; потом, по мере обработки части массива, *last* будет уменьшаться), это поможет определить наличие (и количество) потомков у заданной вершины. Также будем использовать как глобальные переменные сам массив a и количество элементов в нем n.

«Утопим» вершину:

procedure pushnode (i : integer);
begin
  if $2i \le last$ then (* если i — не лист… *)
  begin
    (* выбрать из потомков i наибольший: *)
    $j := 2i$; (* это может быть $2i$ *)
    (* но это может быть и $2i+1$, если он есть и больше: *)
    if $2i < last$ then if $a[2i+1] > a[2i]$ then $j := 2i+1$;

    if $a[j] > a[i]$ (* *что неправильно!* *)
    then begin swap($a[i], a[j]$); pushnode(j) end
  end
end;

Построим правильную кучу (пусть всего в ней n вершин):

procedure pushall;
begin
  for i:=n downto 1 do pushnode(i)
end;

Лемма 7.2. *В дереве, построенном процедурой* pushall*, никакой потомок не превосходит родителя.*

Доказательство. Индукция по *убыванию* номера вершины (от n до 1). Иначе говоря, по построению дерева (добавлению корня к двум поддеревьям). На первом же шаге новая вершина становится больше всех своих потомков, на втором — единственная вершина, в которой что-то могло испортиться, также становится больше всех своих потомков, и т. д. □

Лемма 7.3. *Процедура* pushall *(вместе с вызовами процедуры* pushnode*) использует $O(n \log n)$ операций обмена (*swap*).*

Доказательство. Для каждой из n вершин вызывается процедура pushnode. Она делает не более $\log n$ рекурсивных вызовов (поскольку такова высота дерева), в каждом из них происходит лишь константное число обращений к элементам массива. □

Упражнение 7.1. Показать, что на самом деле используется лишь $O(n)$ операций (хотя для дальнейших рассуждений нам это не будет важно). □

Наконец, отсортируем массив:

```
procedure heapsort;
begin
    last := n;
    pushall;

    for i:=n downto 1 do
    begin
        swap(a[1], a[i]); (*a[1] — наибольший из оставшихся — в конец!*)
        last := i − 1;
        pushnode(1);                    (*ведь a[1] «испортился»*)
    end
end;
```

Теорема 7.2. *Процедура* heapsort *правильно сортирует массив и затрачивает на это лишь $O(n \log n)$ операций с элементами массива.*

Доказательство. **Время работы** складывается из времени работы pushall (см. лемму 7.3) и времени работы процедуры pushnode (в доказательстве леммы 7.3 мы уже видели, что это $O(\log n)$ операций), вызванной n раз.

Корректность *построения* кучи доказана в лемме 7.2. То, что на каждом шаге после отправки $a[1]$ в конец куча восстанавливается правильно, можно доказать аналогично индуктивному шагу в доказательстве леммы 7.2. Наконец, благодаря основному свойству кучи, на каждом шаге мы действительно «вытаскиваем» из нее (отправляем в конец массива) наибольший из оставшихся элементов.

□

Замечание 7.3. Теорема 7.2 справедлива для *любого* массива a (с *любыми* значениями). Таким образом, мы оценили время работы алгоритма *в наихудшем случае.* □

Упражнение 7.2. Точное время работы зависит от того, какие элементы мы сортируем. Какое время займет сортировка массива целых чисел на RAM-машине при помощи алгоритма heapsort? □

Замечание 7.4. Куча (heap) — один из вариантов реализации очереди с приоритетами.

ПРОБЕЛ В КОНСПЕКТЕ.

7.4 QuickSort

В этом разделе для простоты будем считать, что все элементы в массиве различны (анализ для случая, когда имеются одинаковые элементы, совершенно аналогичен).

Алгоритм QuickSort: возьмем какой-нибудь (скажем, первый в массиве) элемент, поставим его на нужное место i (так что все меньшие его элементы находятся слева, все бóльшие — справа) и рекурсивно отсортируем полученные массивы, состоящие из $i-1$ и $n-i$ элементов соответственно.

Теорема 7.3. *В алгоритме QuickSort количество операций над элементами массива в наихудшем случае составляет $O(n^2)$.*

Упражнение 7.3. Доказать теорему 7.3. □

Теорема 7.4. *В алгоритме QuickSort количество операций над элементами массива в наихудшем случае составляет $\Omega(n^2)$.*

Доказательство. Рассмотрим поведение алгоритма на уже отсортированном массиве. □

Теорема 7.5. *В алгоритме QuickSort количество операций над элементами массива* в среднем *составляет $O(n \log n)$.*

Доказательство. Пусть $t(\alpha)$ обозначает количество операций, затрачиваемое на массив, исходное упорядочение которого задано перестановкой α (как легко заметить, количество операций зависит только от этой перестановки, а не от конкретных элементов массива: $(9,5,7)$ и $(3,1,2)$ сортируются за одно и то же время). Количество операций, затрачиваемых в среднем на массивы размера n, обозначим через $T(n)$. Размер массива (или соответствующей перестановки) α обозначим через $|\alpha|$. Часть перестановки α со <u>значениями</u> от j до k (перенумерованными так, чтобы получилась правильная перестановка) обозначим через $\alpha[j:k]$.

$$\begin{aligned}
T(n) &= \frac{1}{n!}\sum_{|\alpha|=n} t(\alpha) = \\
&= \frac{1}{n!}\sum_{i=1}^{n}\sum_{|\alpha|=n,\,\alpha[1]=i} t(\alpha) \le \\
&\le \frac{1}{n!}\sum_{i=1}^{n}\sum_{|\alpha|=n,\,\alpha[1]=i} (cn + t(\alpha[1:i-1]) + t(\alpha[i+1:n])) = \\
&= cn + \frac{1}{n!}\sum_{i=1}^{n}\Bigg(i(i{+}1)\dots(n{-}1)\sum_{|\beta|=i-1} t(\beta) + \\
&\qquad\qquad + (n{-}i{+}1)(n{-}i{+}2)\dots(n{-}1)\sum_{|\gamma|=n-i} t(\gamma)\Bigg) = \\
&= cn + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{(i-1)!}\sum_{|\beta|=i-1} t(\beta) + \frac{1}{(n-i)!}\sum_{|\gamma|=n-i} t(\gamma)\right) = \\
&= cn + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)) = \\
&= cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \qquad (7.1)
\end{aligned}$$

Остается показать, что решение этого рекуррентного неравенства удовлетворяет условию $T(n) = O(n \log n)$. Предварительно убедимся, что

$$\sum_{i=2}^{n-1} i \ln i \le \int_2^n x \ln x \,\mathrm{d}x \le \frac{n^2 \ln n}{2} - \frac{n^2}{4}$$

(в этом можно убедиться при помощи интеграла — по монотонности функции $x \ln x$; или, вместо интеграла, по индукции).

Пусть $b = \max\{T(0), T(1)\}$, $k = 2b + 2c$. Покажем по индукции, что для всех $n \ge 2$ выполняется $T(n) \le kn \ln n$. База ($n = 2$) очевидна из

(7.1). Для $n \geq 3$ имеем

$$T(n) \leq cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \leq cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} T(i) \leq$$
$$\leq cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki\ln i \leq cn + \frac{4b}{n} + \frac{2k}{n}\left(\frac{n^2\ln n}{2} - \frac{n^2}{4}\right)$$
$$= kn\ln n + cn + \frac{4b}{n} - (b+c)n \leq kn\ln n.$$

□

7.5 Randomized QuickSort

Алгоритм Randomized QuickSort отличается от QuickSort тем, что на каждом шаге элемент выбирается случайным образом. Оценим время его работы *в наихудшем случае.* Оно будет зависеть от того, какие нам достанутся случайные числа.

Теорема 7.6. *Для* любого *входного массива математическое ожидание количества операций в алгоритме Randomized QuickSort над элементами массива составляет $O(n\log n)$.*

Доказательство.

ПРОБЕЛ В КОНСПЕКТЕ.

□

7.6 Поиск k-го элемента за линейное время в наихудшем случае

Для удобства сделаем так, чтобы количество элементов в массиве делилось на 10 (просто проверим лишние элементы по одному: найдем максимальный; если он не k-й — выкинем; это надо проделать не более 9 раз).

Разобьем массив на пятерки; возьмем медианы (третьи элементы) полных пятерок и вычислим их медиану. Полученным элементом и разобьем массив «пополам» (как в предыдущем алгоритме). Время работы (количество операций над элементами массива) в наихудшем случае составляет время на поиск медианы + время на поиск искомого элемента в одной из полученных «половинок»: $T(n) \le T(\frac{n}{5}) + T(\frac{7n}{10}) + cn$. Нетрудно по индукции доказать, что $T(n) \le 10cn$. Мы доказали следующую теорему.

Теорема 7.7. *Приведенный алгоритм затрачивает в наихудшем случае лишь $O(n)$ операций на поиск k-го элемента в массиве из n элементов.*

7.7 Поиск k-го элемента аналогично Randomized QuickSort

Заметим, что этот алгоритм совершает довольно много (хотя и линейное число) действий (в частности, сравнений). Рассмотрим похожий алгоритм, математическое ожидание количества сравнений в котором будет лишь $4n$.

Изменение заключается лишь в том, что элемент для "разделения" массива на половинки выберем случайно (как в Randomized QuickSort).

Теорема 7.8. *Для любого входного массива математическое ожидание количества операций в этом алгоритме над элементами массива составляет $4n$.*

Задача 7.1. Доказать теорему 7.8.

Замечание 7.5. Можно добиться $2n$ сравнений (но это непросто).