Построение и анализ алгоритмов – Лекция 5. Сортировка за линейное время

Олег Смирнов
oleg.smirnov@gmail.com

27 октября 2011 г.

Содержание

Цель лекции

- Нижняя граница сортировки сравнениями
- Сортировка подсчётом (Counting sort)
- Поразрядная сортировка (Radix sort)
- Карманная сортировка (Bucket sort)

1 Нижняя граница сортировки

Насколько быстро можно отсортировать массив? Это зависит от *модели вычислений* – то есть того, что разрешено делать с элементами.

Сортировка вставкой, сортировка слиянием (Merge sort) и сортировка Хоара (quicksort) являются примерами сортировки сравнениями, т.е., когда сравнения используются, чтобы определить относительный порядок элементов. Наилучшим результатом, который пока удалось достичь, является $O(n \log n)$. Эту границу можно доказать с помощью *деревьев решений*.

2 Деревья решений

Предположим, что нам нужно решить задачу сортировки трёх чисел.

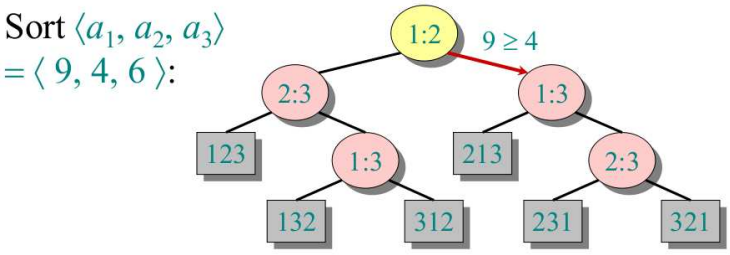
Запишем алгоритм не в обычной текстовой форме, а в виде дерева. Для того, чтобы определить порядок элементов, мы сравниваем их. Внутренние узлы дерева, помеченные как $i : j$, представляют собой сравнение двух элементов – a_i, a_j:

- Если $a_i \leqslant a_j$, то выполнение идёт по левой ветке.
- Если $a_i > a_j$, то по правой.

В листьях дерева находятся перестановки $\langle \pi(1), \pi(2), \ldots \pi(n) \rangle$, такие, что

$$a_{\pi(1)} \leqslant a_{\pi(2)} \leqslant \ldots \leqslant a_{\pi(n)}$$

Рассмотрим пример, когда массив состоит из чисел 9, 4 и 6:



Деревом решений можно смоделировать любой алгоритм сортировки сравнениями:

- Одно дерево для каждого размера входных данных – n.
- Можно рассматривать алгоритмы как набор сравнений, причём каждое сравнение – это очередное ветвление дерева.
- Дерево содержит все возможные варианты выполнения алгоритма.
- Время выполнения алгоритма = длина выбранного пути от корня до листа с финальной перестановкой-решением.
- Наихудшее время выполнения = высота дерева.

С помощью этой модели можно доказать следующую теорему: любое дерево решений, сортирующее n элементов, имеет высоту $\Omega(n \lg n)$.

Доказательство: обозначим количество листьев дерева как l. Поскольку дерево должно уметь сортировать все возможные варианты входных данных, в нём должно быть как минимум $n!$ листьев, отсюда неравенство $l \geqslant n!$. Тем не менее, поскольку дерево бинарное, количество его листьев ограничивается 2^h, где h – высота дерева, т.е. $l \leqslant 2^h$. Из этих двух неравенств получаем третье: $2^h \geqslant n!$

Поскольку логарифм – монотонно возрастающая функция, то:

$$h \geqslant \lg n! \geqslant (\text{по формуле Стирлинга}) \geqslant \lg(n/e)^n = n\lg(n/e) = n\lg n - n\lg e = \Omega(n \lg n)$$

Таким образом, можно сделать вывод, что сортировка слиянием является асимптотически оптимальной.

3 Сортировка подсчётом (Counting sort)

Эта сортировка применима в том случае, если нужно отсортировать целые числа в заданном (не очень большом) диапазоне.

Вход: массив $A[1 \ldots n]$, где $n \in \{1, 2, \ldots, k\}$

Результат: отсортированный массив $B[1 \dots n]$

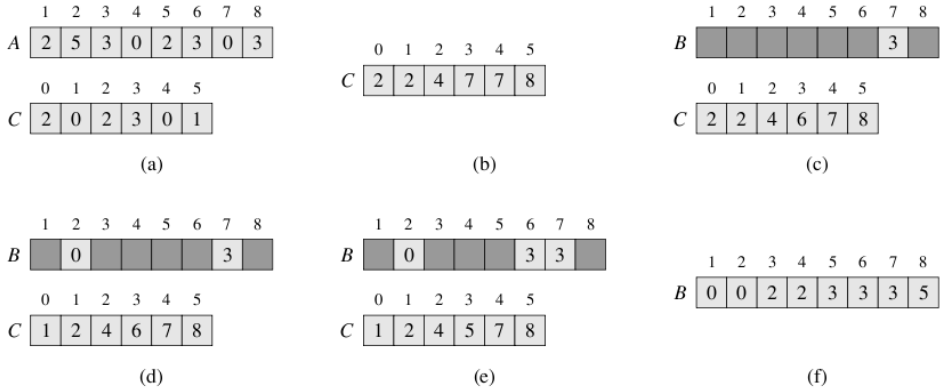
Вспомогательный массив: $C[1 \dots k]$

Идея состоит в подсчёте количества элементов, меньших, чем заданный, и последующем выборе места для элемента на основании этих данных. Например, если известно, что 15 элементов меньше рассматриваемого элемента, то очевидно, что его место в отсортированном массиве будет равно 16. Однако в массиве может быть несколько одинаковых элементов, потому необходимо несколько модифицировать алгоритм.

```
1  for i ← 1 to k
2        do C[i] ← 0
3  for j ← 1 to n
4        do C[A[j]] ← C[A[j]] + 1
5  for i ← 2 to k
6        do C[i] ← C[i] + C[i − 1]
7  for j ← n downto 1
8        do B[C[A[j]]] ← A[j]
9           C[A[j]] ← C[A[j]] − 1
```

В первом цикле обнуляется массив-счётчик, во втором цикле определяется количество вхождений каждого элемента. В третьем – берутся префиксные суммы, то есть определяется количество элементов меньших или равных заданному. В четвёртом цикле происходит расстановка элементов по местам согласно значению массива-счётчика.

Пример:



Анализ: первый и третий циклы требуют $O(k)$, второй и четвёртый – $O(k)$, таким образом время выполнения алгоритма в целом составляет $O(n + k)$.

Если $k = O(n)$, то сортировка подсчётом работает за линейное время. Это возможно, т.к. сортировка подсчётом не использует сравнения.

Важное свойство сортировки подсчётом: стабильность.

Стабильность сортировки означает, что она сохраняет относительный порядок равных элементов. Это важно, к примеру, для использования сортировки подсчётом в поразрядной сортировке.

4 Поразрядная сортировка (Radix sort)

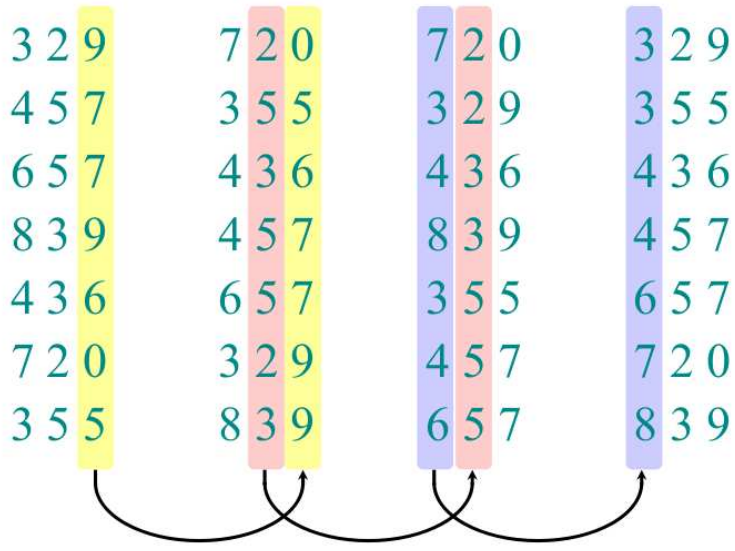
4.1 Идея

Алгоритм был предложен Германом Холлеритом для переписи населения в США в 1890-м году, т.о. является чуть ли не самой старой имплементацией сортировки.

Сортировались перфокарты (размером 80 на 16). Нужно было отсортировать каждый столбец.

Первая идея: сортировать карты по первой, самой старшей цифре. Плохая идея, т.к. после такой сортировки нужно было разбивать всю стопку карт на 10 котейнеров и продолжать сортировку в них, снова разбивая и т.д. Тяжело поддерживать сразу много контейнеров.

Хорошая идея: сортировать по младшей цифре, используя стабильную сортировку как подпрограмму, тогда можно обойтись только одним контейнером.



4.2 Корректность

Доказательство корректности алгоритма: математическая индукция по позиции цифры (по колонке).

Гипотеза: Пусть $t-1$ колонок (числа, состоящие из младших $t-1$ цифр) уже отсортированы и мы сортируем по t-й цифре (колонке).

Предположим, что мы сравниваем две цифры в t-й колонке. Есть два варианта:

- Они равны. Тогда поскольку наша вспомогательная сортировка стабильна – мы сохраним порядок. А по гипотезе индукции предыдущие цифры уже были отсортированы. Потому в итоге получим, что наши числа окажутся в правильном порядке.
- Они неравны. Тогда наша сортировка расставит числа в правильном порядке, поскольку значение числа определяется его старшей цифрой, которую мы как раз и отсортируем в этом шаге.

4.3 Анализ

Будем использовать в качестве вспомогательной стабильной сортировки сортировку подсчётом.

Предположим, что нам нужно отсортировать n чисел, каждое из которых состоит из b бит.

Плохая идея работать с чисто битовым представлением, т.к. предположим, что наши числа находятся в диапозоне от 0 до n и их n, тогда двоичное представление таких чисел будет занимать $\lg n$ и, соответственно, нам надо будет пройтись по ним $\lg n$ раз, то есть суммарное время выполнения будет $\mathcal{O}(n \lg n)$, что хуже линейного.

Но у нас есть контроль над размером "цифры". Можно сгруппировать биты в группы и назвать их "цифрами". Предположим, что мы группируем биты в группы по r бит, таким образом получаем, что наши числа записаны в системе счисления с основанием 2^r. Тогда наше число записано с помощью b/r цифр.

Пример: 32-битное слово разбиваем на группы по 8 бит. Таким образом получаем $r = 8, b/r = 4$ прохода по цифрам в системе счисления с базой 2^8

Вопрос в том, как оптимально разбить биты на группы.

Вспомним, что сортировка подсчётом работает за $O(n+k)$ для чисел в диапазоне от 0 до $k-1$. Если мы будем разбивать на группы по r,

то максимально возможной цифрой будет 2^r, и время выполнения сортировки подсчётом будет $O(n+2^r)$. Количество проходов алгоритма же равно количеству "цифр", то есть b/r. То есть финальная оценка равна $T(n,b) = \Theta((b/r)(n+2^r))$

Попробуем минимизировать $T(n,b)$, управляя r. Минимум можно найти с помощью дифференцирования по r, но мы сделаем проще. Увеличение r означает уменьшение количества проходов, однако в правом множителе 2^r растёт гораздо быстрее.

Мы не хотим, чтобы $2^r >> n$, поэтому мы выбираем $r = \lg n$. Таким образом получается, что $T(n,b) = \Theta(bn/\lg n)$. Для чисел в диапазоне от 0 до $n^d - 1$ количество бит $b = d\lg n$, потому время выполнения будет равно $\Theta(dn)$.

5 Карманная сортировка (Bucket sort)

Если входные данные подчиняются равномерному закону распределения, то ожидаемое время алгоритма карманной сортировки линейно зависит от количества входных элементов.

Предположим, что необходимо отсортировать массив $A[1\ldots n]$, все элементы которого равномерно распределены в интервале $[0,1)$. Идея заключается в том, чтобы разбить интервал на k одинаковых интервалов, или карманов (buckets), а затем распределить элементы между этими карманами. Если k выбрано удачно и элементы распределены равномерно, то в каждый карман попадёт не много элементов. Затем можно отсортировать содержимое каждого кармана за линейное время и объединить результаты в выходной массив.

Рассмотрим пример для $k = n$:

```
1  n ← length(A)
2  for i ← 1 to n
3        do append(B[[nA[i]], A[i])
4  for i ← 0 to n − 1
5        do insertion_sort(B[i])
6  merge(B[0], B[1] ... B[n])
```

Здесь функция *append* добавляет элемент в хвост массива, а $insertion\_sort$ – сортирует массив вставкой.

Поведение алгоритма во многом зависит от выбора числа k. Если в каждый карман попадёт ровно по одному элементу, то алгоритм превратится в сортировку подсчётом. Если $k = 2$, а каждый из двух карманов

затем сортируется рекурсивно, то алгоритм станет Quicksort-ом. Таким образом, карманную сортировку можно считать обобщением этих алгоритмов.