MULTILEVEL SORTING ALGORITHMS

LONG CHEN

ABSTRACT. Two multilevel sorting algorithms, merge-sort and quick-sort, are briefly discussed in this note. Both of them are divide and conquer algorithms and have average complexity $\mathcal{O}(N \log N)$ for a list of size N.

We shall discuss two multilevel sorting algorithms to sort ascendingly a list of N values represented by an array (or a list) `a(1:N)`. The philosophy is "divide and conquer". A divide-and-conquer algorithm usually consists of three steps:

(1) divide the problem into sub-problems with smaller sizes.
(2) solve each sub-problem.
(3) merge the solutions of sub-problems.

When solving the sub-problems, i.e., in Step 2, the same procedure can be recursively applied which results in a multilevel algorithm. Therefore we only need to describe a two-level method in detail. The dividing step is usually refer to top-to-bottom and the merge step is bottom-to-top.

Before we move on to the multilevel algorithms, we review briefly two classic sorting algorithms: insertion sort and bubble sort. Insertion sort inserts elements from the list one by one in their correct position into a new sorted list. It is simple and relatively efficient for small lists. But insertion is expensive, requiring shifting all following elements over by one. Bubble sort compares the adjacent elements and swaps them if they are not in order. It continues doing this for each pair of adjacent elements until no swapping needed.

These two algorithms' average and worst-case complexity is $\mathcal{O}(N^2)$. Both of them suffers from the local operations in the finest scale. The merge sort or quick-sort can be thought of as insertion sort or bubble sort applied to multilevel scales, respectively.

1. MERGE SORT

1.1. **Algorithm.** Merge sort is a natural and intuitive multilevel algorithm. It was discovered by von Neumann in 1945 and rediscovered by many researchers.

Algorithm: Mergesort

```
(1) Divide the input list into two almost equal-size sub-lists
(2) Sort each sub-list
(3) Merge two sorted sub-lists to get the sorted list
```

In Step 2, the algorithm can be recursively called to sort each sub-list. The recursion stops when each sub-list contains one element which is considered as sorted. A more efficient criterion is that the size is less than 30 and then simple sort algorithms, e.g., insertion-sort can be applied. By doing this way, it can take the advantage of the speed of insertion sort on small data sets.

Date: Sept 1, 2015, updated on September 21, 2015.

Dividing step is trivial. Just split the input list into two almost equal-size sublist. Let $m = \lfloor n/2 \rfloor$. The two sublists are simply `a(1:m)` and `a(m+1,n)`.

The tricky part is the merge step. Given two sorted lists `a` and `b`, they can be merged into a sorted list with operations proportional to the size of `a` and `b`. A possible implementation is as follows. Maintain two pointers i and j and initialize to 1. The pointer i moves forward (left to right, i.e., 1 to end) in `a` and stops when `a(i)>b(j)`. Then j moves forward in `b` until `b(j)>a(i)`. Record the values (before the stop) along the path. Either i or j reaches the end, the merge is finished. The length of the path of i and j is at most the length of the list and thus the complexity of the merge is bounded by `length(a) + length(b)`. The merge step can thought of as an insertion of a list of various lengths not just one by one.

Merge-sort requires additional $\mathcal{O}(N)$ (memory) space for the sorted output and involves data movement due to the tricky merge part.

1.2. **Complexity.** Let N be the size of the input list. Then it takes $\log_2 N$ dividing steps to get sub-lists with one element. In merge phase, merging sub-lists in each level requires N operations. So in total, the complexity of merge-sort is $N \log_2 N$ sorting algorithm.

Another complexity analysis is as follows. Let $T(N)$ be the operation count of merge sort for a list of size N. Then the recurrence

$$T(N) = 2T(N/2) + N$$

follows from the definition. The closed form of $T(N)$ can be obtained accordingly.

Exercise 1.1. *Implement the merge-sort.*

2. Quick Sort

2.1. **Algorithm.** Quick sort is invented by Tony Hoare in 1960 and the original algorithm and several of its variations is presented in 1962 [1]. Sedgewick refined and popularized quicksort and analyzed many versions of quicksort in 1978 [3].

Algorithm: Quicksort

(1) (a) Choose an element (called a pivot) from the list;
    (b) Partition the list into left and right sub-lists such that all elements in the left list are smaller than the pivot and all elements in the right list are larger.
(2) Sort the left and the right sub-lists.
(3) Merge the sorted sub-lists into a sorted list.

In Step 2, the algorithm can be recursively called to sort each sub-list. The recursion stops at a certain level. An obvious one is the list contains only one element. A better one is when the size is less than 30. Then use insertion-sort which is faster than quick-sort in this level.

Compare with the general three steps of divide-and-conquer, the merging phase is trivial. The focus is the dividing. While in the mergesort, the dividing is trival but the merge requires some work.

The choice of a pivot is the key. In the worst scenario, the pivot happens to be the largest or the smallest number. Then it is as slow as bubble sort and insertion sort. The ideal pivot would be the median such that the left and right sublists have almost equal size. However, it is not cheap to get the median for an unsorted list.

The simplest choice is to select an element from a fixed position of the input list: the first, the last, or the middle item. A better strategy is to approximate the median of `a` by computing the median of a small subset of a. For example, the median-of-three method: chose the median of the left, middle and right of the list. A fair choice is to randomly select an element from the input list. The randomness reduces the average complexity and introduce a core algorithm concept: randomized algorithms. Indeed, many randomized algorithms in computational geometry can be viewed as variations of quick-sort [2].

Partitioning step is straight-forward if an additional array is allocated. We simply scan the input array and save the left list from the left to right (forwards) and the right list backwards in the new array.

An in-place (i.e. requiring small $\mathcal{O}(1)$ additional space) partition can be realized by swapping [3]. We use two pointers i and j for the left and right list. The pointer i moves forwards and j backwards. i stops when $a(i) > p$ and j stops when $a(j) < p$. If $i < j$, then we swap $a(i)$ and $a(j)$ and continue. The partition will be achieved when $i > j$. The pivot can be first swapped to the leftmost or rightmost location.

When the input list contains a lot of duplicated elements, the partition performs not well. A simple fix is a 3-way partition. Just add one sub-list to store all elements equals the pivot value.

Because of the partition may reorder elements within a partition, it is not a stable sort, meaning that the relative order of equal sort items is not preserved.

2.2. **Complexity.** The average complexity of quick-sort is $\mathcal{O}(N \ln N)$ and the worst case is $\mathcal{O}(N^2)$. Here complexity is simply the operation count. The dominate operation could be chosen as comparison or swaps. The speed depends also on other issues, e.g., the data movement and its spatial locality (cache-efficiency). For example, the complexity of the merge-sort is always $\mathcal{O}(N \log N)$ but in practice quick-sort performs better than merge-sort due to the large data movement and additional space needed in the merge-sort.

The worst case is straight-forward. To analyze the average case, we assume the pivot is randomly chosen from the input list. And to simplify the discussion, we assume distinct values, i.e., no duplication in the input list.

We first follow Ross [4] to show the average complexity is $\mathcal{O}(N \ln N)$. Let X denote the number of comparisons. We are interested in the expectation $E[X]$. Let s_i be the i-th number in the sorted list, i.e., $s_1 < s_2 < \cdots < s_N$, and let $I(i,j) = 1$ if s_i and s_j are directly compared, and $I(i,j) = 0$ otherwise. Using this notation, we can express X as

$$X = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} I(i,j)$$

and the expectation can be simplified as

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} I(i,j)\right] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} E\left[I(i,j)\right] \\ &= \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \Pr\{s_i \text{ and } s_j \text{ are compared}\}. \end{aligned}$$

To be directly compared, the set of numbers $s[i:j] := \{s_i, s_{i+1}, \dots, s_{j-1}, s_j\}$ should be in the same list. Otherwise if the pivot satisfies $s_i < p < s_j$, then s_i and s_j will be in different lists. If the pivot is not in the set $s[i:j]$, these numbers will be still in the same list. At a certain level, the pivot will be in $s[i:j]$. When it happens, $I[i,j] = 1$ only if either s_i or s_j is selected. So

$$\Pr\{s_i \text{ and } s_j \text{ are compared}\} = \frac{2}{j-i+1}$$

and

$$E[X] = \sum_{i=1}^{N-1}\sum_{j=i+1}^{N} \frac{2}{j-i+1} \approx \sum_{i=1}^{N-1}\int_{i+1}^{N} \frac{2}{x-i+1}\,\mathrm{d}x \approx 2N\ln N.$$

In the ideal case, it takes $\log_2 N$ levels to reach a list of size 1. Each level consist of N comparison. Thus the ideal case takes $N\log_2 N$ comparision while the average case is $2N\ln N \approx 1.39N\log_2 N$ which is only about 39% worse than in its best case.

Exercise 2.1. *Justify the recurrence*

$$T(N) = \frac{1}{N}\left[\sum_{i=0}^{n-1}(T(i)+T(n-i))\right] + cN$$

and, together with $T(1) = \mathcal{O}(1)$, derive the formulae

$$T(N) = \mathcal{O}(N\log N).$$

A stronger result shows the quick-sort algorithms takes $\mathcal{O}(N\ln N)$ operations with high probability, which is called a high concentration result. We follow [2] to present the following proof. We first unfold the recursion into a tree of partitions where each level of the tree corresponds to a partition. The leafs of the tree are sub-lists with size less than or equal to 30.

Lemma 2.2. *Let e be an arbitrary element of the input list. Let n_j be the size of the list containing e at the end of the jth level. We set $n_0 = N$. Then*

$$\Pr\{n_{j+1} \geq 7/8\, n_j\} \leq 1/4 \textit{ for any } j \geq 0.$$

For a fixed element e, we call the jth partitioning step successful if $n_j < 7/8\, n_{j-1}$. After k successful partitioning steps, the size is smaller than $(7/8)^k N$. Therefore, e can be involved in at most $\log_{8/7}(N/30)$ successful partitioning steps. To simplify the notation, the log in the sequel is $\log_{8/7}$.

Lemma 2.3. *Among $20\log N$ partitioning steps, the probability that an element e goes through $20\log N - \log(N/30)$ unsuccessful partitioning steps is $\mathcal{O}(N^{-7})$.*

Proof. Let X be a random variable denoting the number of unsuccessful partitioning steps among $20L$ steps with $L = \log N$. The random choices of pivots are independent and thus X is a Bernoulli random variable. Consequently

$$\Pr\{X > 20L - L + \log 30\} \leq \Pr\{X > 19L\} \leq \sum_{j>19L}\binom{20L}{j}\left(\frac{1}{4}\right)^{j}\left(\frac{3}{4}\right)^{20L-j}.$$

Then use the tail bound of the Bernoulli random variable to get the estimate. □

Theorem 2.4. *With probability $1 - cN^{-6}$, the size of sub-lists is less than or equal to 30 after $\mathcal{O}(\ln N)$ levels and consequently, with probability $1 - cN^{-6}$, the complexity is $\mathcal{O}(N\log N)$.*

Proof. For one element, the unsuccessful probability is cN^{-7}. By the union bound, for N elements, the probability is cN^{-6}. In other words, with probability $1 - cN^{-6}$, the size of sub-lists is less than or equal to 30 after $\mathcal{O}(\ln N)$ levels. □

REFERENCES

[1] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5:10–16, 1962.
[2] J. Jaja. A Perspective on Quicksort. *Computing in Science and Engineering*, 2(1):43–49, 2008.
[3] R. Sedgewick. Implementing Quicksort programs, 1978.
[4] R. Sheldon. *A first course in probability*. Pearson Education India, 2009.