Internal Sorting Methods

Rebekah Marie Bitikofer

Boise State University
Internal Sorting Methods

Rebekah Bitikofer (rebekahbitikofer@u.boisestate.edu)

(1) INTRODUCTION

Internal sorting methods are possible when all of the items to be accessed fit in a computer’s high speed internal memory. There are quite a few (Knuth’s third volume of *The Art of Computer Programming* covers 14 in total) but I will go over the four I found to be most versatile and useful.

Definitions

- $N$ will represent the number of items in our given file.
- Each item has a key which we can use to order the items.
- A sorting method is considered stable if it does not change the order of items with equal keys.
- A sorting method is comparison based sorting (CS) if the majority of the sorting is done by comparing items.
- A sorting method is non-comparison based sorting (NCS) if the majority of the sorting is done by without comparing individual items.

(2) BRIEF HISTORY OF SORTING

The history of sorting algorithms began with the United States Census of 1890. Herman Hollerith was a census worker/inventor and saw the need for machines to efficiently tabulate census information. With such a large data set, Hollerith’s machines were pivotal in providing a method for machines to efficiently tabulate census information. With such a large data set, Hollerith’s machines were pivotal in providing a method for machines to efficiently tabulate census information. With such a large data set, Hollerith’s machines were pivotal in providing a method for machines to efficiently tabulate census information. With such a large data set, Hollerith’s machines were pivotal in providing a method for machines to efficiently tabulate census information.

Since computers are great at storing large data sets, it followed that computer science sought methods to sort those data sets. In the early 1950s, methods like Bubble sort and other comparison sorting methods were being analyzed and refined. Shortly thereafter, non-comparison based algorithms were refined too and are optimal for large data sets.

Even today comparison and non-comparison sorting methods have their pros and cons. There is no algorithm that is better than the rest in every way so it is important to understand the benefits of many algorithms.

(3) STRAIGHT INSERTION CS

Straight Insertion is the simplest sorting method to code, does not require extra space on a machine, and works efficiently for $N \leq 25$. However, when $N \geq 25$, this algorithm is quite slow. Straight Insertion works by comparing two items in the list at a time. It compares the first two, sorts them, then compares the second item with the third. If an item is switched, it is then compared with the item to its left and switched if necessary. This continues until all necessary comparisons have been made.

Let $A = \{5, 3, 2, 1, 4\}$. If $A$ is sorted with Straight Insertion, it will follow as such:

$A_1 = \{3, 5, 2, 1, 4\}$ (one comparison)

$A_2 = \{2, 3, 5, 1, 4\}$ (two comparisons)

$A_3 = \{1, 2, 3, 5, 4\}$ (three comparisons)

$A_4 = \{1, 2, 3, 4, 5\}$ (one comparison)

With a small set like $A$, this is not difficult and only takes seven comparisons to complete.

(4) HEAPSORT CS

Heapsort is a very fast sorting method because it is guaranteed to be fast even though its average time and maximum time are about half the average run time for quicksort with about $2(N \log N)$ required. The heap from heapsort is a rooted binary tree with a bijection between the vertices and the items that are being sorted. Heapsort works by creating a heap from unsorted lists and creates a sorted list from comparing items in the heap. By using the binary tree to compare items, we can remove the items with the largest keys one by one to create our sorted list.

(5) RADIX SORTING NCS

Radix sorting refers to a method of list sorting that is very useful for keys which are short or have an unusual lexicographical sequence. Radix sorting works in stages and begins by looking at the first entry and sorts based on the first digit in the term or the last. If it is sorting by using the first digit, entries must include 0’s like the example below in order to maintain stability. $B = \{213, 001, 103, 320, 141, 025\}$

It is a stable method (the red and blue 001 entries will not be switched) which creates different "piles" for numbers to be sorted into. A number will end up in whichever pile corresponds with the digit that was just examined. It does not matter if the sorter starts on the left or right side of the number, but it is important that the sorter starts on one side and remains consistent. After sorting into the first set of piles, merge the piles together again and then start again by looking at the next digit.

With $B$, this is what it would look like working right to left.

$(0|250), (1|001, 0141), (2|132), (3|213), (4|304), (5|025)$

Then merge to get $B_2 = \{250, 001, 0141, 132, 213, 304, 025\}$

Now, look at the ten’s place:

$(0|001, 034), (1|213), (2|025), (3|132), (4|141), (5|250)$

merge $B_3 = \{001, 0141, 213, 205, 132, 141, 250\}$

And the hundred’s place:

$(0|001, 025), (1|132, 141), (2|213, 250), (3|304), (4|5)$

merge $B_4 = \{001, 0141, 213, 250, 132, 141, 250, 304\}$

This works most efficiently with large $N$.

(6) PARTITION EXCHANGE (QUICKSORT) CS

According to Knuth, Quicksort is the most useful general-purpose method for internal sorting. It requires minimum memory space and has an average run time that is better than most other internal sorting methods. However, if a bad partition is chosen, Quicksort can run very slowly. It is recommended to choose the median of three elements in order to avoid this worst-case scenario.

To sort via Quicksort, an item is selected and the list is divided into two sub-lists. The first list includes all items that should be placed before the selected item and the second list includes all items that should be placed after it. If the two sub-lists are close to equal, the number of comparisons will be close to $N \log_2 N$.

For my example, I will just choose the last element of $C$ to serve as the pivot item. While this type of selection can lead to less than ideal run times, it will suffice for the example.

Suppose that $C = \{9, 6, 3, 1, 2, 4, 8, 7, 5\}$. Our partition will be 5 so we will start by breaking $C$ into $C_1$ and $C_2$ such that $C_1$ contains all elements less than or equal to 5 and $C_2$ contains all elements greater than 5. Therefore, $C_1 = \{3, 1, 2, 4, 5\}$ and $C_2 = \{6, 8, 7, 9\}$.

Now, we will look at the items in $C_1$ and $C_2$ starting with the first entry and comparing it to the last entry and moving left to right. If the entry is greater than the last entry, we will switch them. When the last entry is the largest in the sub-list we will make comparisons to the entry directly to the right of it. This method will continue until there are no more switches to be made. Since $C$ is such a small list, this will be quite short.

$C_1 = \{2, 1, 3, 4, 5\}$, $C_2 = \{6, 8, 9\}$

After two switches, $C_1$ is sorted.

$C_2 = \{7, 6, 8, 9\}$, $C_3 = \{6, 7, 8, 9\}$

Similarly, $C_2$ has also been sorted after just two switches. Now that $C_1$ and $C_2$ are sorted, we can merge them together by adding $C_1$ to the end of $C_2$. Now, $C = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

(7) CONCLUSION

Each algorithm that I have covered has a specific benefit that merits its use in computer science. Some have faster run times (Heapsort), simpler code (Straight Insertion), run with a smaller memory space (Quicksort), or work well with large sets (Radix Sorting). Different sorting tasks will lead users to unique sorting algorithms and so we have many variations of organization systems.

(8) REFERENCES
