

2. Sorting

2.1 Introduction

Given a set of n objects, it is often necessary to sort them based on some characteristic, be it size, importance, spelling, etc. It is trivial for a human to do this with a small set, however, computer programs are frequently employed to do this and require specific algorithms. With very large sets, it is important to do this as efficiently as possible to reduce resource wastage. As such, we will examine this problem in detail and develop several algorithms.

Since the objects themselves may be arbitrarily large and difficult to move, it is best to sort the objects based on a unique “key” that represents the characteristic by which the objects are sorted. For example, sorting a list of files on a computer by size is best accomplished by sorting the sizes themselves. To do this, there are several operations that can be used:

- 1) Compare two keys and determine the larger of the two.
- 2) Move keys around in a list.
- 3) Compare and switch two keys. That is, compare two keys and place them side by side, with the larger on the right and the smaller on the left.

As mentioned, we desire to do the sorting as efficiently as possible. So first let’s answer the question: What is the largest number of comparisons we need to do to solve this problem? We can answer this using the pigeonhole principle. The number of possible outcomes to the comparisons must be greater than or equal the number of possible orders of the keys. With n keys, there are $n!$ possible orders, using simple combinatorics. If there are only two outcomes to a comparison (there are no ties) then the number of possible outcomes is 2^k , where there are k comparisons. Thus, we can determine a bound for k :

$$\begin{array}{|c} 2^k > n! \\ \hline k > \lg n! \end{array} \qquad (\lg \text{ refers to log base 2})$$

An approximate bound for k can be determined using Stirling’s Formula, which is developed later in the course. $n!$ is approximated by $c(n/e)^{n+1/2}$, where c is a constant.

With this in mind, let’s look at some actual algorithms.

2.2 Sorting Algorithms

Insertion Sort – The basic principle of insertion sort is to start with the unsorted set of keys, sort two of them using the compare and switch method, and then continually “insert” keys one by one into the sorted list.

We perform insertion by comparing the new key with a middle key (at position $n/2 + 1$, when there are n keys sorted). If the new key is smaller, the middle key and all greater keys are shifted over one position, and the new key is “inserted” into the left half

of the list (keys at positions 1, 2, ... n/2). If it's bigger, the key is inserted into the right (larger) half of the list (keys from n/2 +2 through n). Note that this is a recursive process, as are the other sort procedures.

For example, consider the following 5 unsorted keys: 3 2 5 1 -2. Start with the key 3 as the beginning of our sorted list, then insert 2 by comparing it to 3. 3 is shifted and 2 goes on the left, so our sorted list is now (2 3). Now insert five by comparing it to 3. It is larger so it goes on the right (2 3 5). Now insert 1 by comparing it to 3. It is smaller, so it must be inserted into the left side of the list, which is a list itself: (2). It is smaller than two, so it goes on the left (1 2 3 5). Now insert -2, comparing it to 3. It is smaller, so insert it into the left list (1 2). Compare it to 2. It is smaller, so insert into left list (1) by comparing it to 1. It is smaller so the new list is (-2 1 2 3 5). This list is now complete.

Merge Sort – Merge sort involves sorting the unsorted set into sorted pairs, using compare and switch. The pairs are then “merged” into lists of four, which are merged into eights, and so on.

To merge two lists of any size, compare the largest key in one list with the largest in the other list. The larger of the two is taken off and put at the head of a new list. Then repeat the process by taking the largest key remaining in each of the two old lists and compare. The larger is added to the end of the new list, just behind the first key added. This process is continued until one of the lists is empty. The keys remaining in the non-empty list are already sorted, so they are added to the end of the new list in their existing order.

Example: *Note that in the example below, the original lists are plain text, while the new list is italicized.*

Start with the unsorted set:	{ 1 5 6 2 }
Sort it into lists of two:	{ 1 5 } { 2 6 }
Now begin merging by comparing largest keys and form new list with bigger one (6).	{ 1 5 } { 2 } { 6 }
Continue process, add next largest to list (5):	{ 1 } { 2 } { 5 6 }
Continue merging...:	{ 1 } { 2 5 6 }
Now add remaining list to end:	{ 1 2 5 6 }
We're done.	

Heap Sort – A heap is a type of binary tree in which each vertex is larger than its two children. To clarify, a binary tree is a structure defined by vertices and their children. The tree starts with one vertex, which has either two children or none at all. Each of the children may also be a vertex, having two children, making the tree arbitrarily large. The structure gets its name because it can be drawn like an inverted tree, in which each vertex is above its children, which are written from left to right. When we consider the order of the elements in a tree in this section, the bottommost, rightmost elements are considered the last, and the topmost element is the first.

In a heap sort, the keys are first arranged into a heap, which is done by a procedure we will examine later. Once the keys are in a heap, we switch the top key (at the top of the heap so it must be the largest) with the last key in the heap (the bottom right of the tree). The former top key is said to be inactive now; the rest are still considered active.

The tree is now called a “headless” heap, since the topmost vertex is no longer larger than its children. In all other aspects it is still a heap, since all of the other vertices are larger than their children. We need to convert this headless heap back into a heap.

To do this, compare the topmost key’s children with each other, and compare and switch the larger of the two with the vertex key. Now, the top vertex contains the largest key, but now one its children is atop a headless heap.

Thus, the headless heap problem has moved down one rung in the tree and needs to be solved in the same way over and over until we reach the end of the tree. Since each propagation of the headless heap takes two comparisons, and the depth of a binary tree is $\lg N$, the number of comparisons to transform the entire headless heap into a heap is $2\lg N$.

Once the heap is back in proper form, we perform the same switch operation that we started with, switching the topmost element with the last active element (remember the former topmost elements have been made inactive). This continues until the heap is ordered from lowest to highest.

Example:

9 3 7 1 2 5 4	A heap.
4 3 7 1 2 5 9	Now headless, after 9 (inactive) is moved to the last position 4 replaces 9 at the top of the tree.
7 3 4 1 2 5 9	4 was switched with 7 (larger of 7 and 3), rendering the subtree with vertex 4 to be headless. 5 is the only active child of this vertex, so the two must be switched next.
7 3 5 1 2 4 9	All heaps are now proper, and a switch can occur again.
4 3 5 1 2 7 9	7 and 4 switched.
5 3 4 1 2 7 9	5 and 4 switched, no headless heaps remain and another switch can occur.

2 3 4 1 5 7 9	5 and 2 switched, heap is headless.
4 3 2 1 5 7 9	4 and 2 switched, no heap is headless.
1 3 2 4 5 7 9	4 and 1 switched, heap is headless.
3 1 2 4 5 7 9	3 and 1 switched, heap is not headless.
2 1 3 4 5 7 9	3 and 2 switched, heap is not headless.
1 2 3 4 5 7 9	1 and 2 switched, heap is not headless. Heap sort is complete.

The only question remaining is how to form a heap from the unsorted mess in the first place. First place the keys in an unsorted binary tree. If we look at the leaves (bottommost elements) alone, they are heaps since there are no descendents to be bigger than them.

If we go up one level, then we have headless heaps with two levels.

We know how to handle headless heaps!

So we make them into heaps as done ad nauseam above.

Now we can go up to the third level from the bottom. Including these keys we now have headless heaps again!

Well, fix them and keep rising in the same way.

Each time we extend our heapishness one level up the tree, we have fixed the heap up to the previous level. Thus, we need only fix headless heaps! As such, we can create a complete heap and then run the heap sort algorithm.

Tournament Sort – This method of sorting is quite efficient but requires memory of past comparisons. It involves several rounds of comparisons, much like a tournament among teams or players.

Let's assume that there are 32 keys to be sorted. In the first round, each key is compared with one other. In each subsequent round, the larger keys from the previous round are compared with one of the other larger keys. This continues until one key is determined to be the largest. If there are N keys (32), there will have been $N-1$ (31) comparisons so far to eliminate all the elements except the "winner." Also, the largest key must have been in $\lg N$ (5) comparisons.

To determine the second largest key, note that it must be one of the keys that was "beaten" by the largest key. There are only five such keys. The largest of these is found by comparing the first key eliminated with the second key eliminated, and the larger of these is compared with the third key eliminated, etc. Similarly, the third largest is among the keys beaten by the first or second largest, some of which may have already been compared with each other. In each round, there are fewer and fewer comparisons to be made, the maximum number of comparisons being $\lg N$. This is true provided that the keys that have been compared the fewest numbers of times thus far are compared in subsequent rounds before keys that have had more comparisons. For example, when comparing for the second largest key, the key beaten by the largest key in the first round is compared to the key beaten by the largest key in round 2 before the key beaten in the last round is compared to anything. Rationalize this principle as an exercise.

Example:

	<p>Begin by running a standard tournament. In the first round, compare 3 and 54, 23 and 7, etc. Then compare the "winners," or larger numbers. Continue comparing until only one number remains. That number is the largest. It is 54 in this case. To find the second largest, the only candidates are 3, 23, and 9.</p>
	<p>Run a tournament to find the largest of the candidates for 2nd largest overall. Note that we compare 3 and 23 first due to the order of elimination in the previous tournament. 23 is the 2nd largest; candidates for third are 3, 7, and 9.</p>
	<p>9 is the third largest. 2,4,7 are candidates for fourth.</p>
	<p>7 is fourth largest, 4 and 3 are candidates for fifth.</p>
	<p>4 is fifth largest. 2,3,1 are candidates for sixth.</p>
	<p>3 is the sixth largest. 1 and 2 are the candidates for seventh largest, but the comparison is already done, and 2 is seventh, 1 is eighth.</p>

Quicksort – The basic step in quicksort is selecting one key and placing it such that all keys to the left of it are smaller than it, and all keys to the right are larger. The purpose of this is to divide the problem into two smaller lists, which themselves are quicksorted.

We randomly select one key to be the comparison key. We consider all keys that have not been compared to this key to be “live” and all others “dead.” All keys begin “live.”

The key in question (key x) is compared with the furthest live key from it (key z) in the unsorted list. The two are then rearranged with the larger on the right and the smaller on the left, with *all other keys unmoved*. Key z is now dead, and x is compared with the next furthest element and rearranged appropriately. This process continues until all keys are dead. Key x should now be placed appropriately.

The problem is now divided into two problems, which are then quicksorted. In the first problem, there are $n-1$ comparisons since key x is compared to every key beside itself. In the second round there are $n-3$ comparisons, since there are $n-1$ unsorted keys remaining in two lists, and in each two lists, there is 1 less comparison than the number of keys in the list. If the lists are split evenly in half in each round, then there should be $\lg(n)$ rounds of sorting. Thus, since there are less than n comparisons per round, there are less than $n \lg(n)$ total comparisons, assuming each key sorted is in the exact middle of its list.

However, in the worst-case scenario of key selection, we choose a maximal or minimal value each time and there will be n sorting rounds, and n^2 steps.

Quicksort is often used, because when you pick the keys randomly, you can expect to take only a small multiple of $n \lg(n)$ steps.

Here is a plausibility argument for this: if you pick x at random, approximately $1/3$ of the time your x will be between the $n/3$ rd smallest and $n/3$ rd largest. If you only look at the effects of these choices, for each the problem addressed goes down in size from what it was, say m , to two problems, the larger of which has size at most $2m/3$; a reduction by a factor of $3/2$ at worst.

This means that after $\log n$ to the base $3/2$ of these good steps, the largest set should go down to size 1 and that means after sort of $3 \cdot \log_{3/2} n$ (“x insertion”) steps we should be done.

Consider the following example, in which the underlined number is the key being sorted, the bold numbers are dead keys, and the rest are live keys. Each line is one comparison step.

<u>6</u> 5 1 9 7 8 2
2 5 1 9 7 8 6
2 5 1 9 7 8 6
2 5 1 9 7 8 6
2 5 1 6 7 8 9

2 5 1 6 7 8 9

2 5 1 6 7 8 9

Thus, in this example the number 6 is placed in its proper place. Quicksort is then re-run on the resulting smaller lists, over and over again until completion.

2 5 1 6 7 8 9

1 5 2 6 7 8 9

1 2 5 6 7 8 9

1 2 5 6 7 8 9

Exercises

- Exercise 1* Explain each of the five algorithms in your own words, and give an example of each.
- Exercise 2* Estimate the number of comparisons needed in each of the five algorithms discussed above. Also estimate the number of other actions needed, such as movements of keys.
- Exercise 3* If you can program, write a computer program to implement each of these methods. If you do not program, write explicit directions in “pseudocode” to do this. Pseudocode means a set of instructions in English that are so explicit that a moron (i.e., a computer) could follow them and perform the sorting task.
- Exercise 4* What other ways are there to sort data? Are they as efficient as the algorithms outlined above? How many comparisons are needed to use them.

- Steven Kannan