

2. Sorting

The problem we address is as follows:

We have n objects, which we can compare with one another and judge between them, and when we do so we always assume we have a method for determining that one is bigger than the other.

We actually usually do not sort the objects themselves which objects could be huge, and moving one at all could be a large chore. Instead we represent each object by a **key**. Given two keys, we can go back to the objects and make whatever appropriate comparison we want among the two objects, and determine which key is bigger in the sense we are concerned with.

The keys are given to us in a random order. We want to rearrange them in increasing order, given our sense of order.

In this problem, we will usually, but not always discuss sorting schemes by their “worst case” behavior. This means we judge a method by the performance it achieves on the possible answer that it does the worst on. Imagine a gremlin arranges the ordering to make your task as hard as possible, consistent with your previous comparisons. In real life, we are often content with schemes which usually work out well, but can occasionally screw up. If they do so we can try something else.

Our tools are the ability to compare keys: given keys a and b , to determine which of the two is bigger; and to move keys around. In particular, we can easily perform an operation called “compare and switch by which we mean: compare the two keys a and b , put the larger one on the right and the smaller one on the left.

Our object is to do this with the least amount of effort. There are many different definitions of “effort” which give rise to different problems. Thus we can measure effort by the number of comparisons used in our ordering scheme; or by the number of comparisons and the least key motion; or by number of time steps to perform the scheme when comparisons can be done in parallel. We can also try to sort while using a minimum of space beyond that which the unsorted keys originally occupied.

We will stick to binary comparisons, that is, comparisons of one key with another, in this discussion.

Recall that in the weighing problem used our balance to compare several coins with several

others.

The weighing problem is actually a special case of a sorting problem, in which we know from the beginning that all but one of our coins are equal. When we assume, as we will do here, that no two of our keys are equal in our sense, then the three possible outcomes of a comparison reduce to two; there is no possibility of getting equality in a comparison.

The first question we ask is, given N keys, can we find useful upper and lower bounds on the number of comparisons we need to sort them completely?

We can find a lower bound on the number of comparisons exactly as we did in the weighing problem, namely by use of the pigeon-hole principle.

The number of possible outcomes must exceed the number of possible answers to the sorting problem.

Given N keys, any one of them can be biggest, and given that one, any one of the $N-1$ other keys can come next, etc.,

Thus the number of possible arrangements (called permutations) of the keys that are possible answers is $N!$.

Since each comparison has two possible answers, the number of possible outcomes of k comparisons is 2^k .

We must therefore have, for N at least 3, $N! < 2^k$ which means $\log n! < k$, where here the log is taken to base 2. (The inequality is strict for N at least 3 because $N!$ is never a power of 2 for N at least 3.)

As we shall see a bit later, we can find an expression for $n!$ that is quite accurate for our purposes. We claim here, with proof deferred that $n!$ has the form $c(n/e)^{n+1/2}$, where c is a constant.

Exercise:1.1 This yields a bound for k . Figure it out.

Now let's consider some schemes for actually sorting. It turns out that almost any conceivable approach can be used to sort, and there are many reasonably efficient ways to sort,

Here are some examples of general approaches.

1. **Sorting by insertion:** You start with an unsorted mess of keys,

First you pick out two of them and sort them by comparing and putting the larger one on the right, say.

Then you **insert**

another key to the sorted list, increasing its length by 1. You repeat this action until the list of sorted keys grows to N in length.

After doing this insertion step k times you have $k+2$ sorted keys. To complete our description of the method we need only describe how to insert a new key in a sorted list of other keys.

We will do that soon.

2. **Sorting by merging:**

you start with your unsorted mess. Then you sort them into pairs. Then you sort pairs of sorted pairs into sorted quartets; then

You sort pairs of sorted quartets into sorted octets, and so on.

The key and only step here is that of merging two sorted lists into one sorted list. We will discuss several ways to do this. There is a simple way which r

3. **Sorting by pulling off the biggest (or smallest), successively:**

To do this you find the biggest (or, if you prefer, the smallest) key, remove it from the list, then find the next biggest, then the next biggest, etc., until you have them all.

We will look at two such methods: **tournament sort, and heap sort.**

4. **Sorting by inserting keys one at a time in their correct positions.**

When inserting key x , we do so in such a way that all the keys smaller than x lie to x 's left, which implies that all the keys to x 's right are larger than x .

We will begin by describing two sorting methods which both have the advantage that they require no extra space and use only simple operations for their implementation. Of these perhaps the most commonly used is called quicksort, which is of the fourth type here,

Quicksort

The basic step, starting with N unsorted keys arranged in a line, is:

Pick one of them, call it x , and by comparing x with all $n-1$ other keys, rearrange the line so that each key smaller than x is to x 's left and each larger key is to x 's right.

Having done this, you have reduced the problem of sorting the whole mess into two smaller

problems, These are: sorting the keys to the left of x , and separately sorting the keys to x 's right.

Here is a small example: suppose $x=6$ and the other 7 keys (for $n=8$) are, in initial order, 5, 1, 9, 45, 7, 8, 2. Our first task is to get 6 into the fourth position, with 5, 1 and 2 to its left, and the rest to its right.

There are three questions to address:

1. How do we choose x ?

2. How do we arrange it so that smaller keys are to the left of x and larger keys to x 's right?

3. Finally, what does this accomplish for us?

Here are some answers:

1. In practice people usually pick a key at random to be x . What you want is to get a key whose position in the sorted list is near its middle. If x is near the middle of the list, this step will reduce the sorting problem to two problems each of roughly half the size of the original problem. In worst case this is a horrible thing to do, because every choice of x you make will cause your gremlin to make that an end key, and you will have reduced the problem from sorting n keys to sorting $n-1$ keys by this step. But this method is the most commonly used sorting method. If you want a worst case efficient version of quicksort you can get one by using some sort of preprocessing before picking x .

2. We insert x where it belongs as follows.

We start by putting x at one end of the list and all the other keys on it are said to be **alive**.

We then **compare x with the live key (call it z) furthest from x** . We rearrange those two keys only, leaving the others fixed. We do so by putting the larger of x and z on the right and the smaller on the left. We then declare z dead.

We repeat this step until all keys other than x are dead.

Then our task is accomplished.

Let us see what happens with our example.

We start with **6** 5 1 9 45 7 8 2.

Marking dead keys in boldface, we proceed as follows

First we compare 6 with 2 the key farthest from 6; 6 is bigger and they therefore switch and 2 dies, giving us **2** 5 1 9 45 7 8 **6**.

Now we compare 6 and 5 (5 is now the furthest live key from 6), there is no switch, and 5 dies; We compare 6 with 1 and 1 similarly dies, We then compare 6 with 9, 6 switches, and 9 dies, Then we compare 6 successively with 8 then 7 then 45, each of which dies at the comparison.

Writing out every single step the procedure looks like

6 5 1 9 45 7 8 2
2 5 1 9 45 7 8 6
2 **5** 1 9 45 7 8 **6**
2 **5** **1** 9 45 7 8 **6**
2 **5** **1** 6 45 7 8 **9**
2 **5** **1** 6 45 7 **8** **9**
2 **5** **1** 6 45 **7** **8** **9**
2 **5** **1** 6 **45** **7** **8** **9**

So with $n-1$ comparisons (you have to compare x with every other key to be sure that that key ends up on the correct side of x) we get x in its right place.

So what good does this do us?

We have split the problem into two smaller ones, the left hand one (here? sorting 2 5 1, a three key problem) and a right hand one (here a 4 key problem) In general, the sum of the of the sizes of the two problems will be $n-1$, (since we can now forget about x itself)

And what good does this do us?

if
we are lucky (or skillful) and x is near the middle of the ordering of the keys it does us lots of good;

We can use the same procedure again on both of the two remaining unsorted blocks, and therefore with $n-2$ additional comparisons, break the keys into **four** smaller groups, and so on.

If we break it evenly each time, we will be done after $\log n$ (to base 2) rounds of this procedure. This will require a total of roughly $n \log n$ steps.

On the other hand we could have very bad luck and x could always be maximal or minimal every time we pick it. If so, (in worst possible case) this method could be quadratic, since all the comparisons to insert x could reduce the problem in size merely from n to $n-1$.

Quicksort

is often used, because if you are able to pick the keys randomly, you can expect to take only a small multiple of $n(\log n)$ steps, and it requires no extra space. The only key movement involves interchanging two keys at a time.

Here is a plausibility argument for t : if you pick x at random, at 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 should go down to 1 and that means after sort of $3 \cdot \log_{3/2} n$ ($?x$ insertion?) steps we should be done. And we can more or less expect to be done in at most this many steps.

This gives us a horribly crude performance estimate of an upper bound that is only a small constant off from the best.

Heap Sort

This is a **pull off at the top method**, that has the virtue that it can sort everything in place, has no need of remembering anything, uses only comparison and/or comparison and switch plus $n-1$ additional switches of keys. At the end the keys are all lined up in order.

How is it done?

First we define a heap: it is an arrangement of keys with a single root key, and each key has either 0, 1 or 2 keys as its “children”. In it, each key is “larger” in the sense of our desired ordering, than each of its children.,

Here are the steps.

First, number the keys as they are given to you and imagine they are arranged in the form of the vertices of a balanced binary tree (with vacancies perhaps on the bottom row.)

What does this mean?: key number 1 is at the top and is the root of the arrangement; it has two children, keys 2 and 3; these have children, 2 has keys numbered 4 and 5, and 3 has keys numbered 6 and 7 as children. Remember that this numbering is not the ordering of the keys that we seek, but instead describes the initial positions of the keys to be ordered.

In general, **the children of key x will be keys $2x$ and $2x+1$** , unless of course we started with fewer keys than $2x+1$; if there are fewer than $2x$ keys, then x is at the bottom of the tree.

The first major step is to rearrange the keys in order to make this tree into a heap. This means, **we arrange it so that each key is bigger than its children.** We will discuss how to do this shortly. It is not the big step, since it takes a number of actions only of order n , not $n \log n$.

Once we have a heap, the top (active) key is bigger than its children which are each bigger than their own, and so on, so that the top key is bigger than **every (active) key**.

What we do then is to take that top key and switch it with the last active key, and then make the former top key inactive.

The active keys then form a "headless heap"; they are heap-like except for the top key, which is not necessarily bigger than its children.

The key step then, which gets repeated over and over again, is to convert a "headless heap" back into a heap.

We will see that this can be done in at most $2 \log n$ steps (log to base 2), (often somewhat fewer) so that this method takes at most roughly twice the minimum number of steps.

So how do we convert a headless heap into a heap?

The possibly wrongly placed key is in position 1;

We first compare its children, (in positions 2 and 3), and determine that the bigger child is in position 2 (**say**);

we

then compare and switch 1 with 2 putting the bigger of the two keys in 1. Now the key in 1 is definitely bigger than that in 2, and since it is at least as big as the former 2 key, it is bigger than the key in 3 as well.

There is still a problem;

If the new key in 2 is the key that was in 1, the sub-tree whose top key is in position 2 may be a headless heap.

We treat this subtree exactly the same way we treated the whole tree before; namely we compare the children of 2, (in positions 4 and 5) and determine which is bigger;

We then compare and switch the key now in 2 with the bigger of those in 4 and 5, and by the same argument, the new key in 2 will be bigger than the new ones in 5 and 4, and the headless heap problem is now at worst in the subtree headed by either 5 or 4 **but not both.**

We continue this process, at each step, **which consists of a comparison and a comparison and switch**, and each time we do so the headless heap problem trickles **one level down** in the tree.

When the possibly bad head becomes a leaf among the active keys, the problem disappears: **a leaf is automatically a heap since it has no children to be bigger than it.**

The depth of our tree is roughly $\log_2(n)$; so in at most $2\log_2 n$ comparisons, we cure the headless heap problem; and are ready to make another switch.

Suppose we start with 7 keys, and have made them into the following heap:

$$k_1 = 15$$

$$k_2 = 11 \quad k_3 = 6$$

$$k_4 = 7 \quad k_5 = 8 \quad k_6 = 3 \quad k_7 = 1 \quad k_4 = 7 \quad k_5 = 8 \quad k_6 = 3 \quad k_7 = \underline{15}$$

We start with the switch between k_1 and k_7 after which the new k_7 dies.

We underline the dead keys, put the suspect head location in boldface along with the largest of its child's keys;

$$\mathbf{k_1} = 1$$

$$k_2 = 11 \quad k_3 = 6$$

Now the problem is that k_1 is not bigger than its children; we compare its children, k_2 and k_3 and find k_2 to be bigger;

$$\mathbf{k1} = 1$$

$$k2 = \mathbf{11} \quad k3 = 6$$

So we now compare and switch k1 with k2 to get:

$$k1 = 11$$

$$\mathbf{k2} = 1 \quad k3 = 6$$

$$k4 = 7 \quad k5 = 8 \quad k6 = 3 \quad k7 = \underline{\mathbf{15}}$$

Now the problem is that k2 is not bigger than its children; we compare its children and find k5 bigger

$$k2 = 1$$

$$\mathbf{k4} = 7 \quad k5 = 8$$

So we compare and switch k2 with k5 to get:

$$k1 = 11$$

$$k2 = 8 \quad k3 = 6$$

$$k4 = 7 \quad k5 = 1 \quad k6 = 3 \quad k7 = \underline{\mathbf{15}}$$

We now have a heap again and can switch again:

$$\mathbf{k1} = 3$$

$$k2 = \mathbf{8} \quad k3 = 6$$

$$k4 = 7 \quad k5 = 1 \quad k6 = \underline{\mathbf{11}} \quad k7 = \underline{\mathbf{15}}$$

We now have our headless problem at the top: we compare k2 and k3 and find that k2 is bigger; so we next compare and switch k1 and k2; combining these two steps we push the headless problem to level 2, to k2 getting

$$k1 = 8$$

$$\mathbf{k2} = 3 \quad k3 = 6$$

$$k4 = 7 \quad k5 = 1 \quad k6 = \underline{11} \quad k7 = \underline{15}$$

Comparing $k2$'s children, $k4$ and $k5$, we find $k4$ larger and so next compare and switch $k2$ and $k4$, getting

$$k1 = 8$$

$$k2 = 7 \quad k3 = 6$$

$$\mathbf{k4} = 3 \quad k5 = 1 \quad k6 = \underline{11} \quad k7 = \underline{15}$$

We now have a heap again and can switch; getting

$$\mathbf{k1} = 1$$

$$k2 = 7 \quad k3 = 6$$

$$k4 = k5 = \underline{8} \quad k6 = \underline{11} \quad k7 = \underline{15}$$

Again the problem is at the top; we compare $k2$ and $k3$, find $k2$ bigger, and compare and switch $k2$ and $k1$ to get

$$k1 = 7$$

$$\mathbf{k2} = 1 \quad k3 = 6$$

$$k4 = 3 \quad k5 = \underline{8} \quad k6 = \underline{11} \quad k7 = \underline{15}$$

This time $k2$ has only one live descendent, so we can just compare and switch it with $k4$, which again gives a heap among the now few live keys:

$$k1 = 7$$

$$k2 = 3 \quad k3 = 6$$

$$\mathbf{k4} = 1 \quad k5 = \underline{8} \quad k6 = \underline{11} \quad k7 = \underline{15}$$

After one more switch we get

$$\mathbf{k1} = 1$$

$$k2 = 3 \quad k3 = 6$$

$$k_4 = \underline{7} \quad k_5 = \underline{8} \quad k_6 = \underline{11} \quad k_7 = \underline{15}$$

Now we compare k_2 with k_3 , find k_3 large, and compare and switch k_1 with k_3 , and k_3 is now a leaf, so we have a heap again.

$$k_1 = 6$$

$$k_2 = 3 \quad k_3 = 1$$

$$k_4 = \underline{7} \quad k_5 = \underline{8} \quad k_6 = \underline{11} \quad k_7 = \underline{15}$$

Now we make our last switch and now have to do only one comparison and switch between k_1 and k_2 and a switch and we are done, getting

$$k_1 = \underline{1}$$

$$k_2 = \underline{3} \quad k_3 = \underline{6}$$

$$k_4 = \underline{7} \quad k_5 = \underline{8} \quad k_6 = \underline{11} \quad k_7 = \underline{15}$$

Notice that after every switch step except the next to last, the first compare and switch step is unnecessary; the suspect top key, having come from the bottom, is definitely smaller than one of the level two keys, and so it will always switch.

Subsequent comparison and switches may not switch, and if so the children's heaps below need not be fixed at all.

Now how do get the original keys into a heap in the first place?

Here is one way: if we look at the leaves 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 by curing their heads 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, we cure, them and keep rising in the same way.

Each time we extend our heapishness one level up the tree, having cured to the previous level, we need only cure headless heaps!

How do we cure a headless heap on the k th level of the tree? We've already shown how to do this, but let's go over it again. If we do two compares, we've fixed the keys on the k th level, but we might have created one headless heap on the next level closer to the bottom. We then might have to apply two compares to this one, and so on. I'll let you figure out how many steps building the original heap takes total in the worst case from this description. It's only $c \cdot n$, though.

Exercise: Show that the number of comparisons necessary to create a heap in the first place is a constant times n .

Another way is to put all the small keys at the bottom level, then all the next smallest keys at the next level, etc.

In our case that involves finding the 4th largest key out of our 7 and putting it and the smaller keys in the last row, then finding the middle key of the top 3 and then putting the bottom two in level two, and the biggest at the top.

For a full tree like we have here, this requires finding the median of the keys, (the middle element) and the median of the top half, etc. We will see how to do that with a number of comparisons linear in n .

Tournament Sort

Tournament sort

is a pull-off-top method that is very efficient but applying it involves remembering which keys have been compared to which others and using that information to determine what to do next. This takes some effort; it really isn't too bad otherwise.

First you have a regular tournament among the players, oops among the keys.

At first they play in pairs. The winners advance to the next round and again play each other. Winners again advance, etc., until there is one winner.

Assume that we started with a power of two number of keys, like 32. Then after one round there are 16 winners, second round 8 third round 4, fourth round 2, and at last 1 winner.

Notice that the winner plays 5 matches, (in our case key comparisons) and everyone else plays at most 4 matches with players other than the winner.

At this point there have been $16+8+4+2+1 = 31$ comparison and we have 1 winner.

But notice that the second best must have lost only to the winner.

We can therefore find the second best by comparing the 5 players that the winner beat.

So we have the first key eliminated by the winner play the second, the winner of that play the third eliminated of the winner? victims, etc.

You can show that we can determine the second winner with at most 4 contests and **again, there will be at most 5 players that the second winner beat.** And these are the only possible candidates for 3rd best.

And these can fight it all out in 4 more contests, etc, so we can find the second third fourth fifth etc, winners, all with at most 4 contests each.

The number of comparisons to find the next winner after the first here is 4 or $\log_2 32 - 1$. In general we end up with somewhat fewer than $\log_2 n$ comparisons per new top key after the first winner is found.

Please convince yourself that if you always have the players who were eliminated after playing the fewest games play first. the keys that are eligible to become k-th biggest are always at most 5 in number.

Notice that as you go along, more and more of the keys have their ranks determined, so that the number of keys left to compare with to determine the new winner goes down from the 4 above to 3 then 2 then 1.

Simple Merging

This approach involves sorting into pairs, then combining the pairs in pairs into sorted 4s then combining these into sorted 8s then etc..

At each stage you take two sets of keys of the same size, each of which is already sorted, and combine them together.

Notice that the only candidate for top of both of two sorted lists are **the tops of the two lists**.

If we compare them and pull off the biggest, we again have two sorted lists left, and can again pull off the biggest of the two tops.

If we keep on doing this until one list is completely depleted, we will have sorted the pair of lists into one.

The trouble with this approach (which at one time was used commercially) is that it requires extra space to put the keys that are pulled off the top. Since you do not know which list will be depleted when you merge them, you can't use the partially depleted list space very efficiently.

Insertion Sort

To describe **insertion sort**, you need only say how you will go about inserting a new unknown key, call it x , in a sorted list of size k .

Let the sorted keys be denoted as $key_1, key_2, \dots, key_k$

What we want to do is to compare x to a middle key, and the one in position $\lfloor k/2 \rfloor + 1$? namely? $key_{(\lfloor k/2 \rfloor + 1)}$ will do.

If x is smaller we want to move all the keys from this middle and beyond over by 1 to make room for x , and insert x into the list $key_1, key_2, \dots, key_{(\lfloor k/2 \rfloor)}$.

If x is bigger, we don't move anything but now insert x into the right hand half of the list: $key_{(\lfloor k/2 \rfloor + 2)}, \dots, key_k$.

This can be shown to require a minimum number of comparisons, but it requires lots of key movement, which can be bad if key movement has cost.

We can describe this algorithm for insertion of x into L as follows, if we call it $I(x, L)$ and call moving a list L over $M(L)$

$$I(x, \{1, \dots, k\}) = \text{if } (x < key_{(\lfloor k/2 \rfloor + 1)}) \text{ } I(x, \{1, \dots, \lfloor k/2 \rfloor\}) \text{ and } M\{\lfloor k/2 \rfloor + 1, k\}$$

Otherwise $I(x, \{[k/2]+2, ?, k\})$

Exercises:

1. 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.
2. 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 (ie, a computer) could follow them and perform the sorting task.)