18.310 lecture notes September 2, 2013

How to sort?

Lecturer: Michel Goemans

1 The task of sorting

1.1 Setup

Suppose we have n objects that we need to sort according to some ordering. These could be integers or real numbers we want to sort numerically, words we want to sort alphabetically, or diamonds we want to sort according to their size. The basic operation we can perform is to take two of these objects and compare them; we will then know which one is bigger and which is smaller. In fact, we typically do not compare the objects themselves, but rather an attribute or *key* of them. This could be their weight, value, size, etc. Comparing two keys does not necessarily mean comparing them numerically, but we have a way to tell which is bigger. For example, if we want to sort objects according to their weight, we only need to have at our disposal one of these old-fashioned scales, and not a modern one that displays the weight of an object. In this discussion, we will stick to

binary comparisons, that is, comparisons of one key with another. With an old-fashioned scale, for example, one could compare several objects at once, but we will not allow this. We also assume that no two keys are equal. In other words, when comparing two keys, we assume that only 2 outcomes (rather than 3) are possible since one of them will be strictly smaller than the other.

Our n keys are given to us in an arbitrary order. We want to rearrange them in increasing order, according to our sense of order. If we are given the following input of numerical values as keys

$$6 \ 2 \ 8 \ 12 \ 5 \ 1 \ 10$$

then the output would be

$$1 \ 2 \ 5 \ 6 \ 8 \ 10 \ 12.$$

1.2 Algorithms and their performance

An *algorithm* is a method for computing a task, which always terminate with a correct output, irrespective of what input it is given. For instance a *sorting algorithm* should always work: it should always output a sorted list, whatever initial list it is given as an input. Beside that, we would like our algorithm to perform our sorting task with the least amount of "effort" (measured in terms of number operations it requires). But how will we evaluate the performance of a sorting algorithm? Well, clearly it will take more operations to sort $1,000,000$ keys rather than just a mere $1,000$ keys. So we certainly want to evaluate its behavior as a function of n, the number of keys to sort. The number of operations will also depend on the input itself; n already sorted keys seem much easier to sort than n keys given in an arbitrary order. We will usually, but not always, adopt a worst-case perspective. This means we judge a sorting algorithm based on the number of operations it performs on the worst possible input of n keys. The worst input, of course, depends on the algorithm itself. One can view this worst-case analysis as a game with a malicious adversary. Once he (or she) knows the precise algorithm we will be using, (s)he will feed an input that makes our task as hard as possible. In practice, this worst-case type of analysis might be very conservative as the behavior on most inputs might be far from the worst-case behavior, but sitting right now on a flight over the Atlantic, I very much hope that a worst-case analysis of all the algorithms used in the control of the plane was performed...

1.3 Lower bound on the number of comparisons required to sort a list

The first question we ask is, given n keys, can we find useful upper and lower bounds on the number of (binary) comparisons we need to perform in order to sort them completely? We can find a *lower* bound by using the pigeonhole principle: the number of possible outcomes should be at least as large as the number of possible answers to our sorting problem. If we never perform more than k comparisons then the number of possible outcomes of the algorithm is at most 2^k (remember, we are not allowing equal keys). On the other hand, given n keys, any one of them can be the smallest, anyone of the remaining $n-1$ keys can come next, and so on, resulting in $n!$ possible answers. Therefore, the pigeonhole principle tells us that we have $n! \leq 2^k$; otherwise there would be two different orderings of the input that give the same list of answers to the comparisons (hence the algorithm would fail to return a correctly sorted list in one of the two cases). This means that, for any correct sorting algorithm, we have $k \geq \log_2(n!)$ where $\log_2$ denotes the logarithm taken in base 2. Since k must be an integer, we must therefore have that

$$k \geq \lceil \log_2(n!) \rceil$$

where $\lceil x \rceil$ denotes the smallest integer greater or equal to x.

To get an estimate of $\log_2(n!)$, we can use the easy fact that

$$\left(\frac{n}{2}\right)^{n/2} \leq n! \leq n^n,$$

or even better the more precise Stirling's formula which gives

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Thus, we see that for large values of n, $\log_2(n!)$ behaves like $n\log_2(n)$ (the other terms grow more slowly than $n\log_2 n$).

The table below shows the lower bound $k \geq \lceil \log_2(n!) \rceil$ for a few values of n:

n	k
2	≥ 1
3	≥ 3
4	≥ 5
5	≥ 7
6	≥ 10
7	≥ 13
$\vdots$	$\vdots$
100	≥ 525
$\vdots$	$\vdots$
1000	≥ 8530
$\vdots$	$\vdots$

Exercise. Show that for $n = 5$, it is possible to sort using 7 comparisons (tricky).

2 Some sorting algorithms

There are many different approaches for sorting a list of numbers. We can divide them into several general approaches.

1. **Sorting by inserting keys one at a time in a growing sorted list of keys.** The principle is to maintain a sorted list of keys. Initially we start with just one key. Once we have a sorted list of k keys, we can insert a new key into it in order to obtain a sorted list of $k+1$ keys, and repeat the process. This leads to INSERTION SORT and we will discuss later how to do the insertion.

2. **Sorting by merging.** The idea here is that if we have two sorted lists, it is rather easy to take their union and create a merged sorted list. This means that one can obtain larger and larger sorted lists. This leads to MERGE SORT, and we will describe it in more details as well.

3. **Sorting by pulling off the largest (or smallest) element.** If we have an efficient way to extract the largest element then we can repeat the process and obtain a sorted list of elements. This leads to several sorting algorithms; we will describe HEAP SORT, a rather beautiful and efficient way for sorting.

4. **Sorting by inserting keys one at a time in their correct final position.** The idea here is that we take the first key x of the list to be sorted and determine what will be its position in the sorted list. Then we place x at its correct position, we place all keys smaller than x before x and all keys larger than x after x. Now we can simply reiterate on the two sublists that come before and after x. This leads to QUICKSORT.

2.1 Insertion Sort

To describe Insertion Sort, we only need to say how we will go about inserting a new key, call it x, into a sorted list of size k.

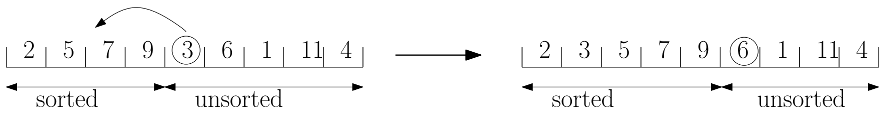


Figure 1: Inserting a new key in the Insertion Sort algorithm.

Let the sorted keys be denoted as $key_1, key_2, \cdots, key_k$. One possibility would be to compare x with each key, from smallest to largest, and insert it just before the first one that is larger than x (or after the last one, if no such key exists). This would not be very efficient since we would make k comparisons in the worst-case for one insertion, and thus about $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$ comparisons over all insertions in the worst-case.

Another possibility is to compare x to a middle key, say the one in position $m = \lfloor (k+1)/2 \rfloor$. 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 $key_1, key_2, \cdots, key_{m-1}$.

If x is bigger, we don't move anything but now insert x into the right hand half of the list $key_{m+1}, ., key_k$. 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 a cost as it happens when implementing such a sorting algorithm in a computer. It is a fine way when sorting a hand of cards (because there is no cost in "key movements" in that situation).

2.2 Merge Sort

The Merge Sort approach involves dividing the n keys into $n/2$ pairs, then sorting each pair, then combining the sorted pairs into sorted groups of size 4, then combining these into sorted groups of size 8, etc. For example, consider the following 8 keys to sort:

$$6 \;\; 2 \;\; 8 \;\; 12 \;\; 5 \;\; 1 \;\; 10 \;\; 11.$$

We can represent the groupings and the mergings as follows:

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 candidates for the front element of the merged lists are the front elements of both lists. Thus, with one comparison, we can find out the front of the merged list and delete this element from the corresponding list. We again have two sorted lists left, and can again pull off the smallest of the two fronts. If we keep on doing this until one list is completely depleted, we will have sorted the union of the two sorted lists into one. And if both lists have size k, we will have done at most $2k-1$ comparisons. Over each level, we do less than n comparisons, and since we have roughly $\log_2(n)$ levels, we get that we perform roughly $n \log_2(n)$ comparisons.

The trouble with this approach is that it requires extra space to put the keys that are pulled off from the front. 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.

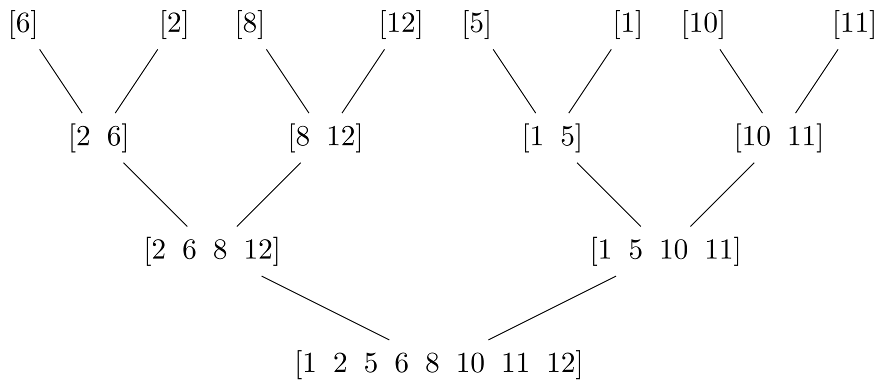


Figure 2: Groupings and mergings in the MERGE SORT algorithm.

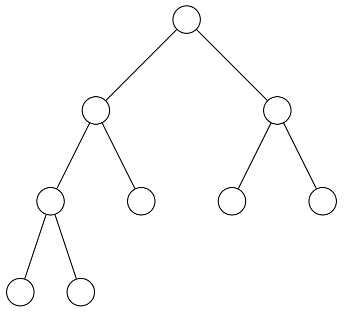
2.3 HEAP SORT

This is a **pull off at the top method** that has the virtue that it can sort everything in place, and performs essentially (up to a small factor) the optimum number of comparisons.

To describe it, we first need to define a *heap*. A *heap* is a tree whose nodes contains the keys to be sorted and which satisfies the conditions that

1. every node has 0, 1 or 2 children,
2. every level from the root down is completely full, except the last level which is filled from the left,
3. the key stored in any node is greater than the keys of (both of) its children (the *heap property*).

First, let's focus on conditions 1. and 2. A heap on 9 elements must therefore have the following structure:



The first three levels are filled (by 1, 2 and 4 nodes respectively) while the 2 nodes in the last level are as much to the left as possible. A heap with k full levels will have $1+2+4+\cdots+2^{k-1} = 2^k - 1$ nodes.

Below are two (out of many more) valid ways of putting the keys 1 up to 9 into a heap structure with 9 nodes. We should emphasize that the left and right children of a node serve the same purpose; both are supposed to have a key smaller than their parent. There is an easy way to store

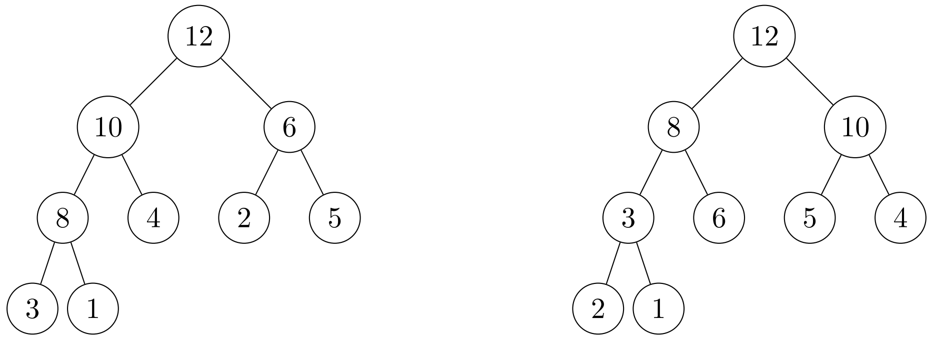


Figure 3: Two heap structures with 9 nodes

the keys of a heap without drawing the tree. Indeed, if we label the nodes of the heap from top to bottom, and from left to right in each level (see below), then notice that the children of the node labelled x will be labelled $2x$ and $2x + 1$ (if they exist).

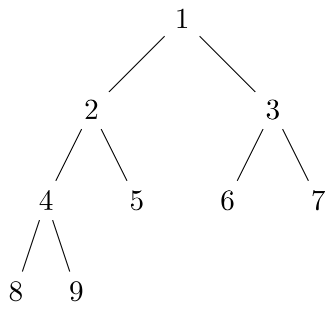
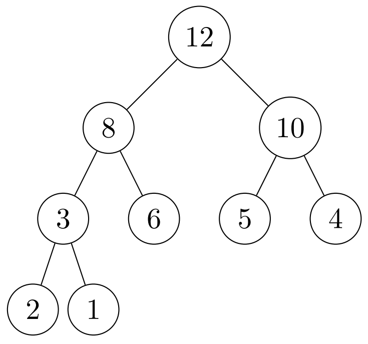


Figure 4: Labels of the nodes of a heap.

This property allows us to store a heap conveniently as an array of the same length as the number of keys. For example, Figure 5 shows a heap and its array representation. And knowing the array, one can easily rebuild the heap.

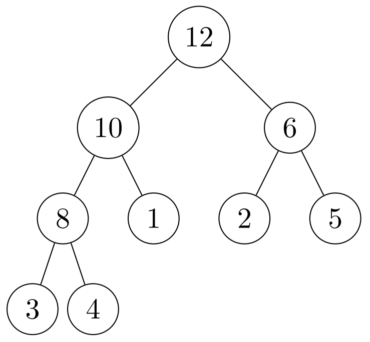


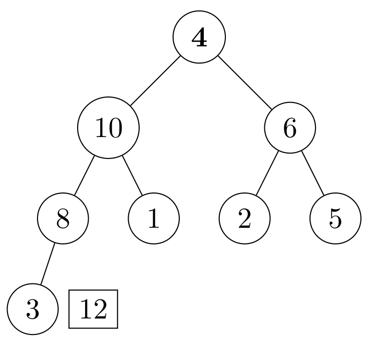
12	8	10	3	6	5	4	2	1

Figure 5: A heap and its array representation.

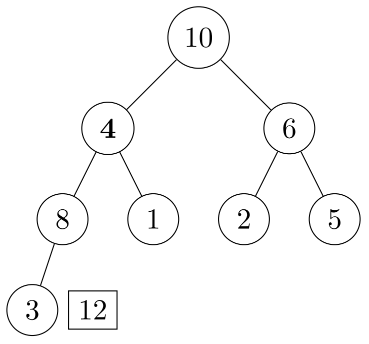
For now, let us postpone the discussion of how we initially arrange the given keys so that they satisfy the heap property. Once we have a heap, the root node contains a key which is larger than its children, which is larger than its own children, etc, so this means that the root contains the largest key. So this key (since it is the largest) should be placed at the end of the array. To do this, we can simply exchange the root key with the last key in the heap, and make this new last key inactive.

The problem is that the remaining active keys do not quite form a heap; indeed the heap property may be violated at the root. Let us call this a *headless heap*. Consider for example the following heap to the left, and suppose we exchange the root element and the last element to get the headless heap to the right.

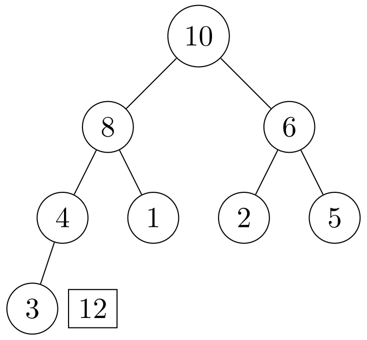




The heap property is indeed violated at the root. Key 4 is smaller than its children. This can be fixed rather easily. We can exchange key 4 with the key of its largest child, in this case, this is its left child with key 10 (but it could have been the right child as well). This gives:



By replacing the key 10 by the key 4, unfortunately, we are now violating the heap property at the node containing this key 4. In other words, the structure rooted at the node with key 4 is not a quite a heap but a headless heap. But we know how to fix this. In general, to fix a headless heap, we compare its root with its two children (by making 2 comparisons). If the root is larger than both its children, there is nothing to do and we are done. Otherwise, we exchange the root with the largest child, and proceed to fix the headless heap now rooted at what used to be this largest child. The headless heap problem has now trickled down one level. In our example, exchanging the key 4 with 8, we get the following heap, and we have recovered a (real) heap. In general, though, the headless problem may not end until we have reached a leaf.

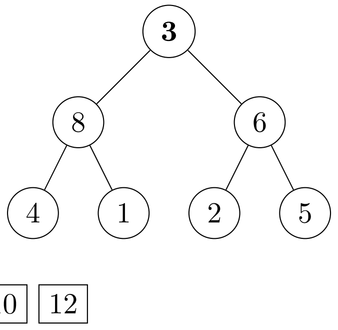


The array that corresponds to this heap, and the inactive 9th element is

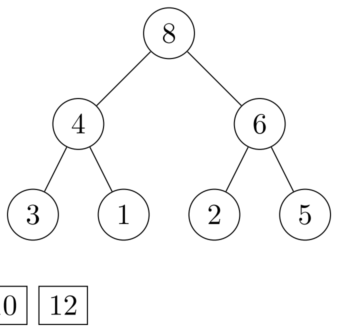
10	8	6	4	1	2	5	3	12

In general, the height of a heap is roughly equal to $\log_2(n)$, and fixing a headless heap therefore takes at most $2\log_2(n)$ comparisons (2 comparisons at each node on the path from the root to a leaf).

Now that we have another heap (with one fewer key), we can proceed and extract the maximum of the remaining active keys, which must sit at the root of the heap. Again, we can switch it with the last active element to get a headless heap:



And fix this headless heap problem by trickling it down the tree until we get a real heap:



The array that corresponds to this structure is now:

8	4	6	3	1	2	5	10	12

and we have made progress since the last 2 elements are correctly positioned.

Every time we extract an element, we need to perform at most $2\log_2(n)$ comparisons, and therefore we can extract all elements by doing at most $2n\log_2(n)$ elements.

But we have not discussed yet how to create the initial heap. There are two obvious ways to try it. One is a top-down approach. In this approach, we build up the heap one key at a time, starting by trying to place that key at the root. Another is a bottom-up approach. In this approach, we start with a binary tree with the keys placed arbitrarily at the nodes, and fix the levels one at a time, starting with the nodes closest to the leaves. The bottom-up approach takes roughly $c_1 n$ time, and the top-down approach takes roughly $c_2 n \log n$ time, for some constants c_1 and c_2.

Here is a sketch of the bottom-up approach: If we look at the leaves alone, they are heaps since there are no descendants to be bigger than them. If we go up one level then we may have headless heaps with two levels. And we know how to handle headless heaps! So we make them into heaps by curing their heads as done above. Now we can go up to the third level from the bottom: including these keys we now have headless heaps again and we know how to cure them! Each time we extend our heap property one level up the tree.

How do we cure a headless heap on the kth level of the tree? We've already shown how to do this, but let's go over it again. If we do two comparisons, we've fixed the keys on the kth 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. You can figure out how many steps building the original heap takes in the worst case from this description. It's only cn comparisons, though.

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