Sorting

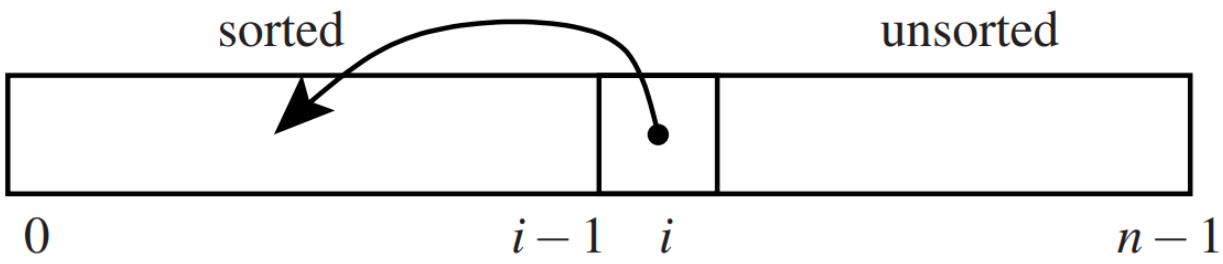
Problem: *Sort a list L of length n over a totally ordered data type D*

We look at algorithms which use comparisons, i.e. take any two members of the list and see which is below the other. There are other methods which can be employed if D is of certain special kinds. We wish to discover what are the most efficient sorting algorithms, and whether there are optimal algorithms. Just as with searching, we will do two things:

- Measure the time complexity of known algorithms
- Obtain a lower bound for the amount of work required

We start by considering a simple sorting algorithm and measuring its worst-case complexity.

Insertion Sort works as follows. Suppose that $\mathrm{L}[0..\mathrm{i}-1]$ is known to be sorted (where $1 \leq \mathrm{i} < \mathrm{n}$). Then insert $\mathrm{L}[\mathrm{i}]$ into $\mathrm{L}[0..\mathrm{i}-1]$ in its correct position. When this is done, $\mathrm{L}[0..\mathrm{i}]$ will be sorted.



Algorithm:

```
i = 1
while i < n:
        # insert L[i] into L[0..i-1]
        j = i
        while L[j - 1] > L[j] and j > 0:
                Swap(j - 1, j)
                # swaps L[j - 1] and L[j]. L[j] will always be the
value to be inserted
```

```
                j = j - 1
        i = i + 1
```

How many comparisons does Insertion Sort take? The insertion can be done by comparing $L[i]$ successively with $L[i-1], L[i-2], \ldots$ until we find the first $L[j]$ such that $L[i] \geq L[j]$. Once this is found $L[i]$ should be inserted between $L[j]$ and $L[j+1]$.

In the best case, $L[i] \geq L[i-1]$ (so that $L[0..i]$ is already sorted).

Only one comparison is needed. However in worst case we need to perform i comparisons. This will happen if $L[i] < L[0]$. Therefore in worst case we need to perform i comparisons for $i = 1, 2, \ldots, n-1$. So for Insertion Sort:

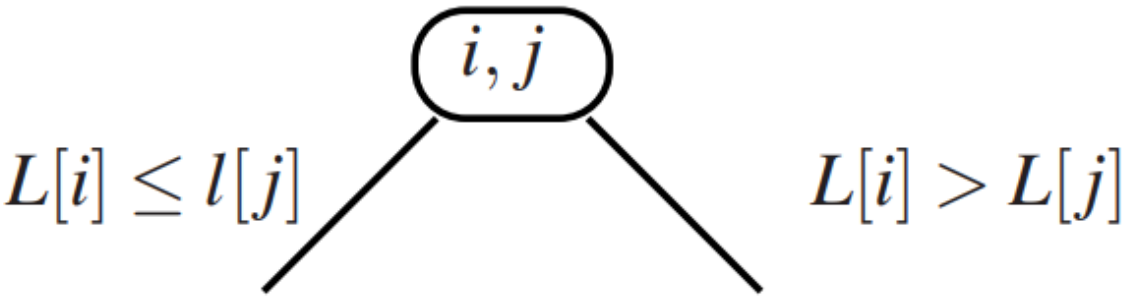
$$W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

The worst case arises when sorting a list which is in reverse order. So we have a sorting algorithm which is $\Theta(n^2)$. We shall see that there are faster algorithms. First we establish a lower bound for sorting.

Lower Bound for Sorting by Comparison

Say that a node of a tree is a leaf if it has no successor nodes. Otherwise it is internal.

Any algorithm for sorting by comparison can be expressed as a decision tree. A decision tree for sorting a list L of length n is a binary tree with the following properties. Every internal node of the tree consists of a pair i, j where $i, j < n$. This represents comparing $L[i]$ with $L[j]$. The left- hand path is followed if $L[i] \leq L[j]$ and the right-hand path if $L[i] > L[j]$.



Every leaf node represents a possible outcome expressed as a permutation of $[0, 1, \ldots, n-1]$. For example if n = 4 a possible leaf will be $[1, 0, 3, 2]$. This means that the original list can be sorted by placing $L[0], L[1], L[2], L[3]$ in the order $L[1], L[0], L[3], L[2]$. Thus, once a permutation is given the list is in effect sorted. Of course the actual algorithm may well rearrange L during the stage when the comparisons are carried out.

Example:

The algorithm 3-sort for sorting a list of length 3 is described by the following decision tree:

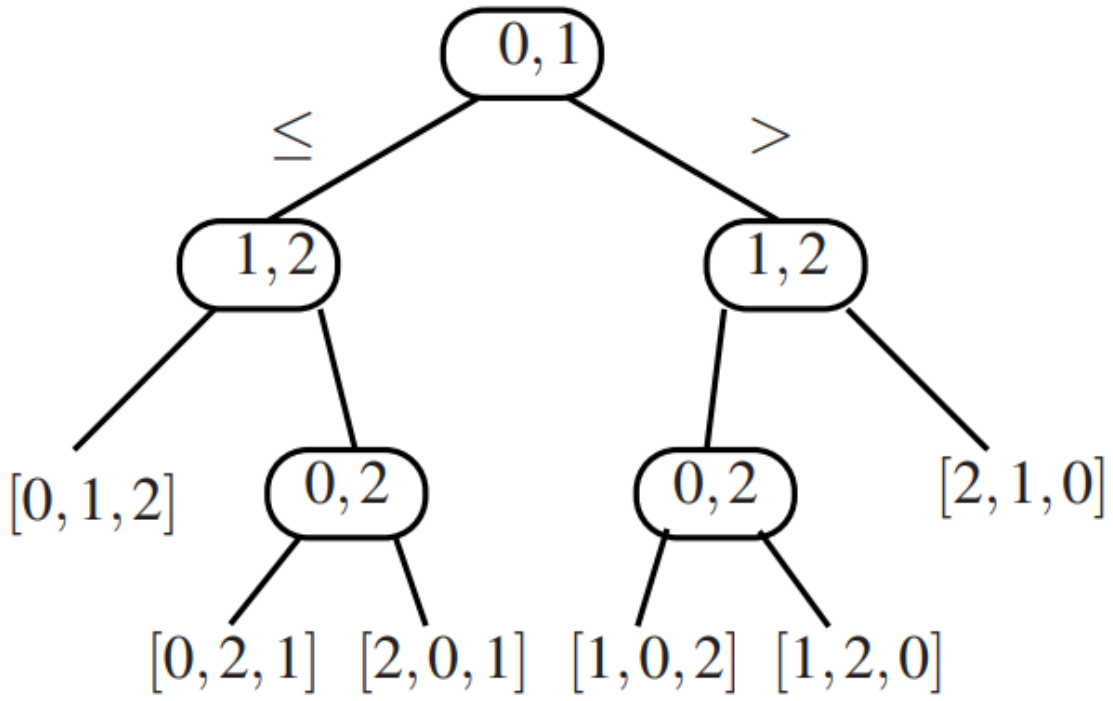


Figure 2.3: Decision tree for 3-sort

We note the following:

- The worst-case number of comparisons is 3
- The best is 2
- There are 6 leaf nodes, one for each of the 6 possible permutations of [0,1,2]

Now we argue that any sorting algorithm for $n = 3$ must do at least 3 comparisons in worst case:

- First note that the algorithm should have every possible permutation of [0,1,2] as a leaf in its decision tree
- If it did not have a certain permutation, then it could not correctly deal with a list which was jumbled in that particular way
- So its tree must have at least 6 leaves
- Now the worst-case number of comparisons is equal to the depth of the tree
- If a binary tree has depth 2 then it can have no more than 4 leaves
- So the tree must have depth at least 3, and the algorithm must do at least 3 comparisons in worst case
- Therefore 3-sort is optimal

Now we generalise what we have just done to obtain a lower bound for all values of n.

If a binary tree has depth d then it has $\leq 2^d$ leaves.

Proof. By induction on d.

Base case $d = 0$. Then no more than $1 = 2^0$ leaves.

Induction step. Assume true for all trees of depth $\leq d$. Suppose the tree has depth $d+1$. The root has one or two successors, which have depth $\leq d$. By the inductive hypothesis, the successors have at most 2^d leaves each. So the total number of leaves in the tree $\leq 2^d + 2^d = 2^{d+1}$.

Consider a decision tree for sorting a list of length n. It must have at least one leaf for every permutation of $[0, \ldots, n-1]$. Now there are $n!$ such permutations. So the tree needs at least $n!$ leaves.

The worst-case performance of the algorithm is exactly the depth d of the tree. d must satisfy $n! \leq 2^d$. Hence $d \geq \log n!$

Since d is an integer we can improve this to $d \geq \lceil \log n! \rceil$

Therefore,
Lower Bound for Sorting in worst case: Any algorithm for sorting a list of length n by comparisons must perform at least $\lceil \log n! \rceil$ comparisons in worst case.

We now give a table showing the lower bounds for n = 1 to 10 compared with the worst-case performance of Insertion Sort:

n	1	2	3	4	5	6	7	8	9	10
$\lceil \log(n!) \rceil$	0	1	3	5	7	10	13	16	19	22
$n(n-1)/2$	0	1	3	6	10	15	21	28	36	45

Although Insertion Sort is optimal for $n \leq 3$, there is a big difference as n gets larger.

Therefore, there is room for improvement, either by finding a faster sorting algorithm, or by finding a better lower bound argument.

However first we consider the lower bound for average-case analysis. One might expect this to be significantly less than the worst-case bound, but we shall find that it is almost the same.

Let us return to the example 3-sort. There are 6 possible rearrangements of a list of length 3. From the tree (Figure 2.3) we see that $[0, 1, 2]$ and $[2, 1, 0]$ can be sorted in 2 comparisons, while the other four take 3 comparisons. It is reasonable to assume that each of the 6 is equally likely. Therefore the average number of comparisons is:

$$(2+2+3+3+3+3)/6 = 2\,2/3$$

which is scarcely less than the worst-case number 3.

We now generalise to arbitrary decision trees

Call the sum of all the depths of the leaf nodes in a tree its *total path length*.

This is closely related to the average number of comparisons. Suppose that the tree has $n!$ leaves and total path length b. Then the average number of comparisons is $b/n!$

So for a given number of leaves ($n!$) we want to find a lower bound on the total path length

Now notice that the tree for 3-sort has depth 3, and all its leaves have depth 3 or 2.

A binary tree of depth d is **balanced** if every leaf node is of depth d or $d-1$.

If a binary tree is **unbalanced** then we can find a balanced tree with the same number of leaves without increasing the total path length.

Proof. Suppose a tree of depth d is unbalanced. Then it has a leaf of depth d, and a leaf of depth $d' \leq d-2$. In this case we can construct another tree with the same number of leaves, which is "more balanced" in a sense that we do not make precise, and where the total path length has not increased.

Take a node of depth d. There are two cases:

1. If this node has no sibling (i.e. it is the only child of its parent) then remove it. An example is shown on the left-hand side of Figure 2.4. The parent becomes a leaf node of depth $d-1$. The total path length has reduced by 1. The depth of the tree may or may not have been reduced. The number of leaves is unchanged. The tree is "more balanced".

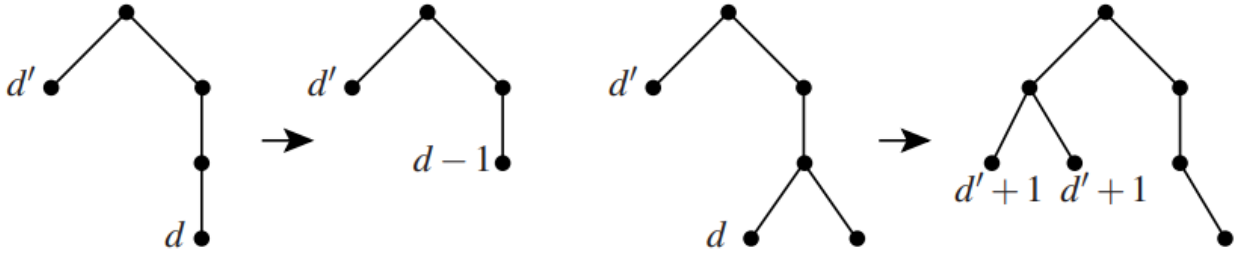


Figure 2.4: Balancing binary trees

2. If the node of depth d has no sibling, remove it and suspend two new leaf nodes below the leaf of depth d'. An example is shown on the right-hand side of Figure 2.4. The old two leaves contributed $d + d'$ to the total path length, while their replacements contribute $2(d' + 1)$. So the total path length has not increased (and will have been reduced if $d' < d - 2$). The number of leaves is unchanged. Again the tree is "more balanced".

We repeat the above procedure until we can no longer find a pair of nodes of depth d and $\mathrm{d}' \leq \mathrm{d} - 2$. The tree is now balanced

The number of leaves is unchanged and the path length has not increased (and may have reduced)

In view of the proposition, if we are trying to find the minimum total path length of a tree with $\mathrm{n}!$ leaf nodes, we can restrict our attention to balanced trees. But in this case the average depth of a leaf node must be only slightly less than the depth of the tree. In fact if the tree has depth d, then the average leaf depth must be between $\mathrm{d} - 1$ and d. We conclude that the lower bound for average-case is almost the same as for worst case.

Lower Bound For Sorting In Average Case: Any algorithm for sorting a list of length n by comparisons must perform at least $\lfloor \log \mathrm{n}! \rfloor$ comparisons in average case.

We have not mentioned the possibility that the list to be sorted will contain repeats. This will reduce the number of possible different permutations. However it does not alter the work we have done above in any essential way.

Merge sort

We would now like to see how various well-known algorithms match up against the lower bounds we have calculated. In this section we consider Merge sort. Merge sort splits the list into two roughly equal halves, and sorts each of these recursively before merging the two sorted sublists. When sorting a list L of length n, the following procedure will be called initially with $\mathrm{left} = 0, \mathrm{right} = \mathrm{n} - 1$

Algorithm:

```
procedure Mergesort(left,right):
        if left < right:
                mid = ⌊(left+right)/2⌋
                # left ≤ mid < right
                Mergesort(left,mid)
                Mergesort(mid+1,right)
                Merge(left,mid,right)
```

The procedure $\mathrm{Merge}(\mathrm{left}, \mathrm{mid}, \mathrm{right})$ merges the two sorted sublists $\mathrm{L}[\mathrm{left}..\mathrm{mid}]$ and $\mathrm{L}[\mathrm{mid}+1..\mathrm{right}]$ to form a new sorted list, and puts the result in the space occupied by those two lists

It works by writing the merged list into an auxiliary array and then copying it back into L when the merging is over. We require $\mathrm{left} \leq \mathrm{mid} < \mathrm{right}$ as a precondition for calling Merge.

We now see how many comparisons Merge performs. This depends on how Merge works, but we shall assume that it follows the following procedure:

- Generate the merged list in ascending order by repeatedly removing the current least value from the two lists to be merged
- Since the lists to be merged are sorted, it is enough to compare the leftmost elements of each list until one of the lists is exhausted, after which the remainder of the other list is transferred automatically
- We see that each element is placed in the merged list at the expense of at most one comparison, apart from the last element, which can always be transferred automatically
- So the number of comparisons is one less than the length of the merged list, i.e. $\mathrm{right} - \mathrm{left}$. It can be shown that this form of merge is **optimal** in worst case

We now obtain the following recurrence relation for the worst-case behaviour of Merge sort. The “n−1” is the number of comparisons to perform the merge.

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n-1+W(\lceil n/2 \rceil)+W(\lfloor n/2 \rfloor) \end{aligned}$$

$W(\lceil n/2 \rceil)$ = First half of list
$W(\lfloor n/2 \rfloor)$ = Second half of list

This allows us to calculate $W(n)$ for $n = 1, \ldots, 10$, and to compare this with the lower bounds we obtained earlier.

n	1	2	3	4	5	6	7	8	9	10
$\lceil \log(n!) \rceil$	0	1	3	5	7	10	13	16	19	22
$W(n)$	0	1	3	5	8	11	14	17	21	25

It is apparent that there is a fairly tight, though not exact fit.

In particular, we can be certain that Merge sort is optimal for $n \leq 4$.

We would like to discover whether Merge sort continues to be close to the lower bound as n gets large. We therefore solve the recurrence relation for $W(n)$ by repeated expansion. To simplify matters we assume $n = 2^k$ $(k \geq 0)$

$$\begin{aligned} W(n) &= n-1+2W(n/2) \\ &= (n-1)+2(n/2-1)+2^2W(n/2^2) \\ &= n+n-(1+2)+2^2W(n/2^2) \\ &\cdots \\ &= n+n+\cdots+n-(1+2+2^2+\ldots+2^{k-1})+2^kW(n/2^k) \end{aligned}$$

There are k ns in the sum. Also $1 + 2 + \ldots + 2^{k-1} = 2^k - 1 = n - 1$. Hence:

$$W(n) = k.n-(n-1)+0\ .$$

$k = \log n$, therefore:

$$W(n) = n\log(n) - n + 1$$

This is for n a power of 2. It can be shown that for general n the solution is:

$$W(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

We now wish to compare $\mathrm{W(n)}$ for Merge sort with the lower bound we obtained earlier. The two functions are of the same order. Notice:

$$\log(n!) = \sum_{i=1}^{n} \log i$$

This is roughly the area under the curve y = $\log \mathrm{x}$ between 1 and n, so (as n gets large):

$$\sum_{i=1}^{n} \log i \approx \int_{1}^{n} \log x \, dx$$

$\log \mathrm{x}$ is a constant multiple of $\ln \mathrm{x}$, and

$$\int_{1}^{n} \ln x \, dx = [x\ln(x) - x]_{1}^{n} = n\ln(n) - n + 1$$

Therefore, Merge sort is of the same order as the lower bound we obtained.

Master Theorem

Worst case comparisons for Binary Search:

$$W(1) = 1$$
$$W(n) = W(n/2) + 1$$

Worst case comparisons for MergeSort:

$$W(1) = 0$$
$$W(n) = 2W(n/2) + (n-1)$$

Number of arithmetic operations for Strassen's Algorithm:

$$A(1) = 1$$
$$A(n) = 7A(n/2) + 18(n/2)^2$$

All three algorithms are examples of Divide and Conquer algorithms. The general form for Divide and Conquer:

- Work $\mathrm{T(n)}$ for input size n
- Split into a sub-problems of size $\mathrm{n/b}$
- Non-recursive work (split and combine) is $\mathrm{f(n)}$

The recurrence relation for $\mathrm{T(n)}$ is:

$$T(n) = aT(n/b) + f(n)$$

(plus base cases)

We shall show how to solve such a recurrence relation up to Θ. It is helpful to associate recursion trees with our recurrence relations. As an example, consider:

$$T(1) = 1$$
$$T(n) = aT(n/2) + n$$

Create a recursion tree as follows:

- Start with input of size n
- Each unfolding of the recursion takes us down one level to a subproblems each with size halved
- Each node of the tree records the size and the non-recursive work done

Suppose n = 2^k

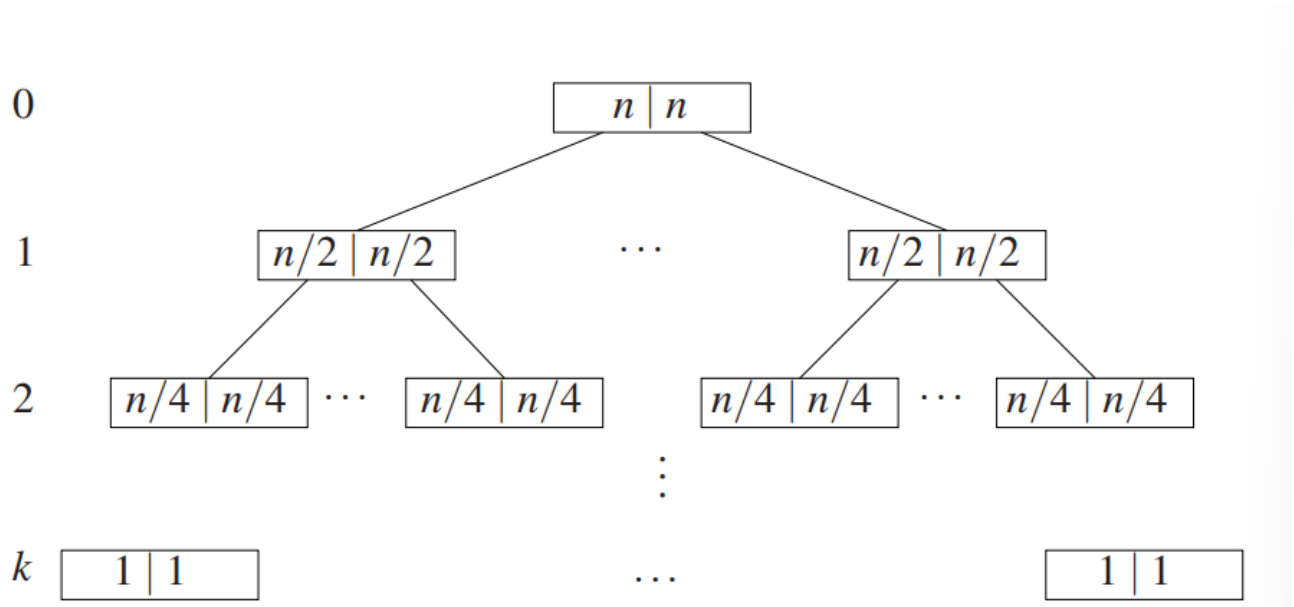


Figure 2.5: Recursion tree.

The total work done is the sum of the work at each of the $k + 1$ levels.

$$n + a(n/2) + a^2(n/2^2) + \cdots + a^{k-1}n/(2^{k-1}) + a^k$$
$$= n + (a/2)n + (a/2)^2 n + \cdots + (a/2)^{k-1}n + a^k$$

This is a geometric series.

$$\sum_{i=o}^{k} ar^i = \frac{a(r^{k+1} - 1)}{r - 1}$$

Provided $r \neq 1$

Clearly the sum is $(k + 1)a$ if $r = 1$

Corollary 2.6.9 *Let $t(n)$ be the largest term in the geometric progression*

$$a, ar, ar^2, \ldots, ar^k$$

where r is non-negative, $r \neq 1$ and r does not depend on n (though a and k can depend on n). Then

$$\sum_{i=o}^{k} ar^i = \Theta(t(n))$$

Clearly $t(n)$ is either a (if $r < 1$) or ar^k (if $r > 1$)

We can apply the Corollary to $n + (a/2)n + (a/2)^2 n + \cdots + (a/2)^{k-1} n + a^k$

Here $r = a/2$

There are three cases depending on a

- If $a < 2$ the greatest term is n
  - $T(n) = \Theta(n)$
  - Here the non-recursive work at level 0 dominates
- If $a = 2$ then $T(n) = (k + 1)n = \Theta(n \log n)$
  - Here the work is (roughly) evenly spread at all levels. (similar to MergeSort)
- If $a > 2$ the greatest term is $a^k = a^{\log n} = n^{\log a}$
  - $T(n) = \Theta(n^{\log a})$
  - Here the base cases (the leaves of the recursion tree) dominate

We now work towards the general case. Consider

$$T(n) = aT(n/b) + f(n)$$

(plus base cases).

The recursion tree will have $1 + \log_b n$ levels

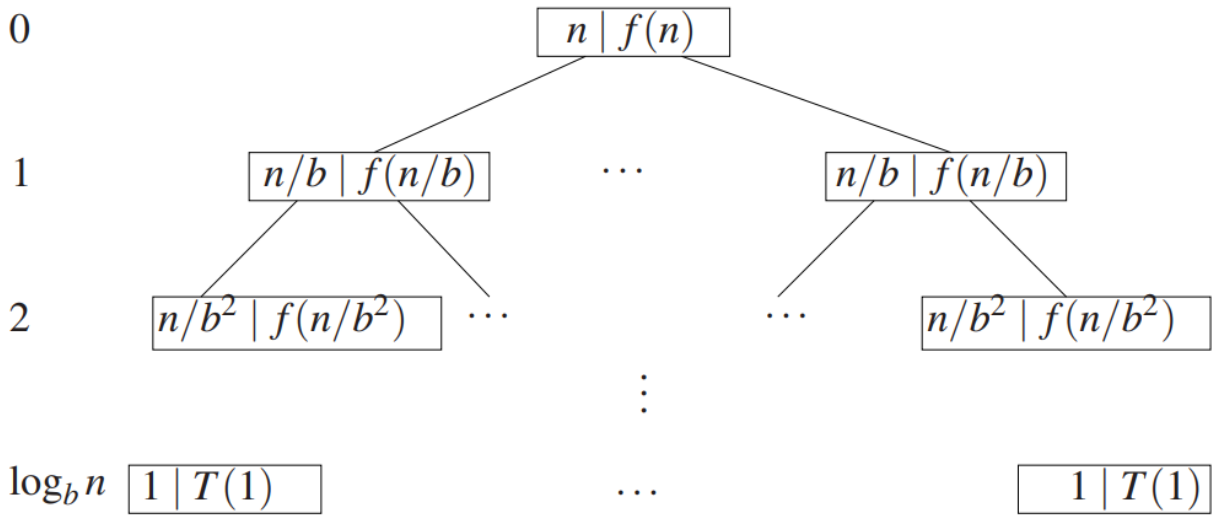


Figure 2.6: Recursion tree.

Level 0: work $f(n)$
Level 1: work $af(n/b)$
Level 2: work $a^2 f(n/b^2)$
. . .
Level $\log_b n$: work $\Theta(a^{\log_b n})$

For the bottom level we just need to know the number of leaves, as the work for each is constant.

Suppose that $f(n) = n^c$. Then the ratio is $r = a/b^c$
Let the critical exponent be $E = \log_b a = \log a / \log b$

Then $r > 1 \iff a > b^c \iff \log_b a > c \iff E > c$. There are three cases:

- $E < c \implies T(n) = \Theta(f(n))$
- $E = c \implies T(n) = \Theta(f(n) \log_b n) = \Theta(f(n) \log n)$
- $E > c \implies T(n) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) = \Theta(n^E)$

The Master Theorem:

$$T(n) = aT(n/b) + f(n)$$

has solutions as follows, where $E = \log a / \log b$ is the critical exponent:

1. If $n^{E+\varepsilon} = O(f(n))$ for some $\varepsilon > 0$ then $T(n) = \Theta(f(n))$
2. If $f(n) = \Theta(n^E)$ then $T(n) = \Theta(f(n) \log n)$
3. If $f(n) = O(n^{E-\varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^E)$

We look at some examples, starting with the worst case comparisons for Binary Search:

$$W(n) = W(n/2) + 1$$

Here $a = 1$ and $b = 2$ and $f(n) = \Theta(n^0)$
Then $E = \log a / \log b = 0$
So

$$W(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Next, the worst case comparisons for MergeSort:

$$W(n) = 2W(n/2) + (n - 1)$$

Here $a = 2$ and $b = 2$ and $f(n) = \Theta(n^1)$
Then $E = \log a / \log b = 1$
So

$$W(n) = \Theta(n \log n)$$

Finally, the number of arithmetic operations for Strassen's Algorithm:

$$A(n) = 7A(n/2) + 18(n/2)^2$$

Here $a = 7$ and $b = 2$, $f(n) = \Theta(n^2)$
Then $E = \log a / \log b = \log 7 > 2$
So

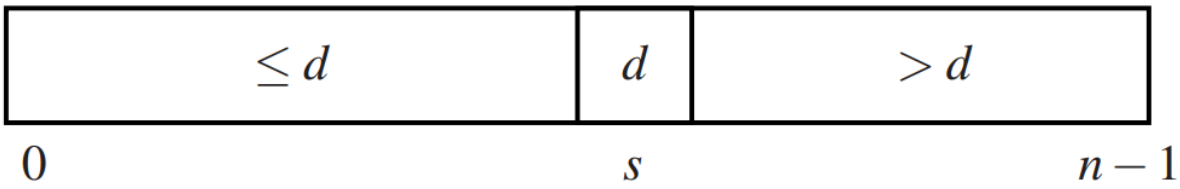
$$A(n) = \Theta(n^{\log 7})$$

Note that any improvement to $f(n)$ here will not help with the order of $A(n)$.

Quicksort

Quicksort works by “splitting” the list L around a member of L, which we shall take to be $L[0] = d$

This involves comparing every $L[k]$ $(k \neq 0)$ with d, and placing it before or after d depending on whether $L[k] \leq d$ or $L[k] > d$



s is the new index of the element d

The portions before and after d are then sorted recursively, after which the whole list is sorted

Algorithm Quicksort:

```
procedure Quicksort(left,right):
        if left < right:
                s = Split(left, right)
                # left ≤ s ≤ right
                # L[s] is now in correct position
                Quicksort(left,s-1)
                Quicksort(s+1,right)
```

`Split(left, right)` splits `L[left..right]` around `d = L[left]`, and returns the index s of the splitpoint. The result is a reordering of $L[left..right]$ with $left \leq s \leq right$ and

- $left \leq k \leq s$ implies $L[k] \leq d$
- $s < k \leq right$ implies $L[k] > d$

Algorithm `Split(left, right)`:

```
# pre-condition: left < right
d = L[left] # pivot
i = left + 1
j = right
# Invariant:
```

```
# left < i ≤ j + 1
# j ≤ right
# if left ≤ k < i then L[k] ≤ d
# if j < k ≤ right then L[k] > d
while i ≤ j:
        if L[i] ≤ d:
                i = i + 1
        else:
                Swap(i, j)
                j = j - 1
# i = j + 1
Swap(left, j)
return j
```

Applying `Split` to a list of length n takes $n - 1$ comparisons.

Quicksort is harder to analyse than MergeSort because the list can be split into sublists of varying lengths. The split might be roughly in the middle, or it might be near to one end.

To minimise the number of times that Quicksort is called recursively, we want the split to occur at the midpoint of the list.

The worst case is when the split occurs at one end. This gives us the following recurrence relation for the worst-case:

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n-1+W(n-1) \end{aligned}$$

The solution to this is:

$$W(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

This is no better than Insertion Sort. Ironically the worst case arises in particular when L is already sorted.

However Quicksort performs well in practice because the split is unlikely to be near one of the ends, and if it is near the middle then we get performance similar to that of Mergesort.

Earlier we said that average-case complexity is often of the same order as worst-case complexity. Quicksort is an exception, with a significant improvement for average-case complexity.

We now obtain a recurrence relation for A(n), the average number of comparisons to sort a list of length n.

When `Quicksort(0, n - 1)` is invoked it will split the list at some position s. We assume that s is equally likely to take each of the possible values $0, \ldots, n-1$. Each of these values therefore has a probability of $1/n$. If the split is at s, then $\mathrm{Quicksort}(0, s-1)$ and $\mathrm{Quicksort}(s+1, n-1)$ will be invoked, taking $A(s)$ and $A(n-s-1)$ comparisons respectively. Hence:

$$\begin{aligned} A(0) &= 0 \\ A(1) &= 0 \\ A(n) &= n - 1 + \frac{1}{n}\sum_{s=0}^{n-1}(A(s) + A(n-s-1)) \end{aligned}$$

This can be simplified to:

$$\begin{aligned} A(1) &= 0 \\ A(n) &= n - 1 + \frac{2}{n}\sum_{i=2}^{n-1} A(i) \end{aligned}$$

This is not easy to solve. However, $A(n)$ is $\Theta(n \log n)$

Proof. We show that $A(n) \leq 2n \ln n$. This is enough to show $A(n)$ is $O(n \log n)$, since logarithms to different bases differ by a constant factor. We then deduce that $A(n)$ is $\Theta(n \log n)$, using the fact that the lower bound for sorting in average case is $\Theta(n \log n)$.

By induction. Clearly holds for $n = 1$. Assume that for all $i < n$ we have $A(i) \leq 2i \ln i$.

$$
\begin{aligned}
A(n) &\leq n - 1 + \frac{2}{n} \sum_{i=2}^{n-1} A(i) \\
&\leq n + \frac{2}{n} \sum_{i=2}^{n-1} 2i \ln i \quad \text{(by Induction Hypothesis)} \\
&\leq n + \frac{2}{n} \int_{2}^{n} 2x \ln x \, dx \quad \text{(since } x \ln x \text{ is monotonically increasing)} \\
&\leq n + \frac{2}{n} [x^2 (\ln x - \frac{1}{2})]_2^n \\
&\leq n + \frac{2}{n} [n^2 (\ln n - \frac{1}{2}) - 2^2 (\ln 2 - \frac{1}{2})] \\
&\leq n + 2n(\ln n - \tfrac{1}{2}) \quad (\text{since } \ln 2 > \tfrac{1}{2}) \\
&\leq 2n \ln n
\end{aligned}
$$

The following table shows (to two decimal places) the lower bounds for the average case and the values of A(n) for Quicksort. The figures are very similar to those given for the worst-case lower bounds and W(n) for Mergesort. Therefore, Quicksort gives good performance in average case.

n	1	2	3	4	5	6	7	8	9	10
Lower bound	0	1	2.67	4.67	6.93	9.58	12.37	15.37	18.56	21.84
$A(n)$	0	1	2.67	4.83	7.40	10.30	13.49	16.92	20.58	24.44

It might seem that Mergesort should be preferred to Quicksort, since W(n) for Mergesort is almost as good as A(n) for Quicksort.

However Quicksort can be performed “in place” without using extra memory space, whereas Mergesort needs extra memory space to perform the merges.

Therefore, Quicksort is preferred in practice.

Heapsort

We start by defining min and max binary heaps.

A **heap structure** is a left-complete binary tree.

Left-complete means that if the tree has depth d then:

- All nodes are present at depth $0, 1, \ldots, d-1$
- And at depth d no node is missing to the left of a node which is present.

An Example:

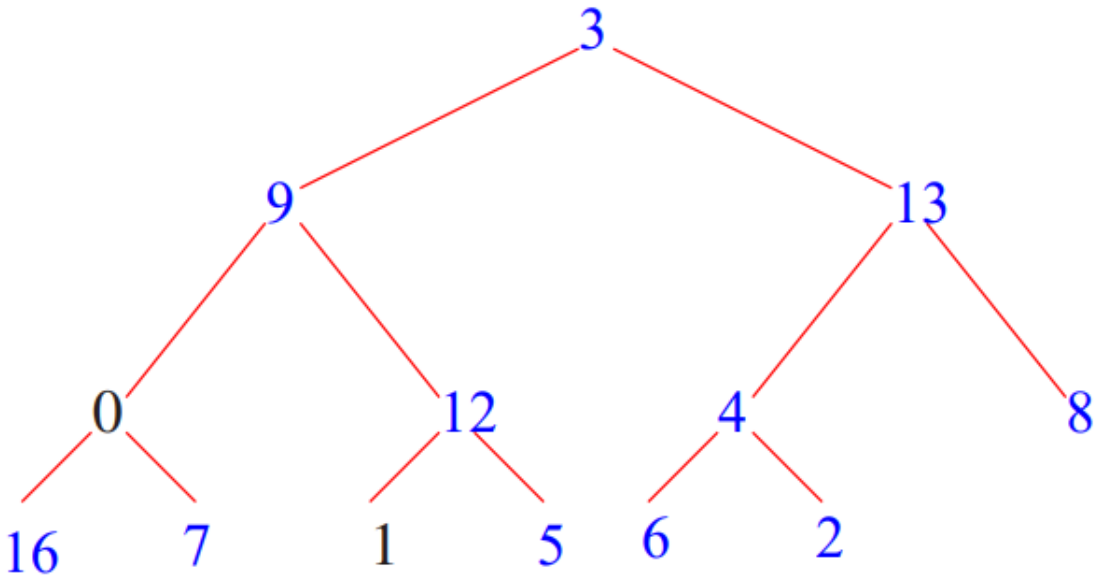


Figure 2.7: Example of a heap structure.

We call the rightmost node at depth d the last node.

We say that a tree T is a *minimising partial order tree* if the key at any node $\leq$ the keys at each child node (if any).

A min heap is a heap structure with the minimising partial order tree property.

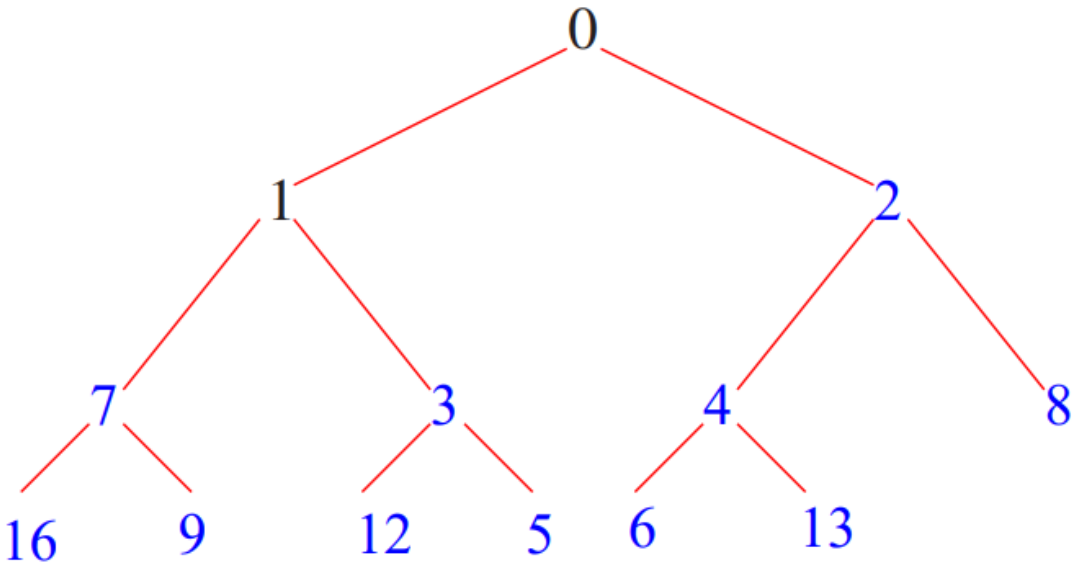


Figure 2.8: Example of a min heap.

Note that for any node of a min heap, the left and right subtrees below the node are also min heaps.

A tree T is a **maximising partial order tree** if the key at any node $\geq$ the keys at each child node (if any).

A max heap is a heap structure with the max partial order tree property.

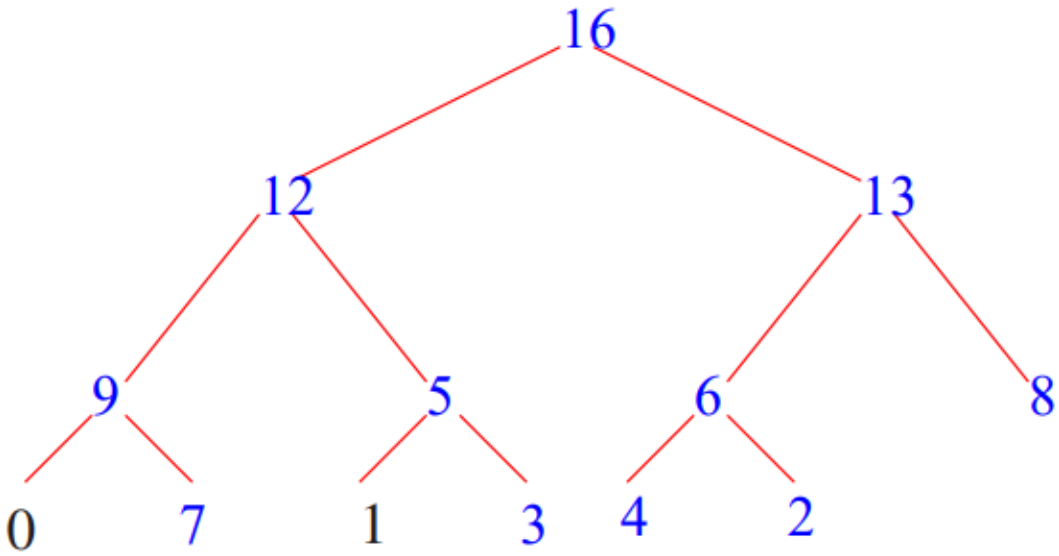


Figure 2.9: Example of a max heap.

Note that for any node of a max heap, the left and right subtrees below the node are also max heaps.

Clearly the largest key is at the root of the tree, which allows us to use max heaps to perform sorting.

Algorithm Heapsort scheme:

```
Build max heap H out of an array E of elements
for i = n to 1:
        max = getMax(H)
        deleteMax(H)
        E[i] = max
```

We repeatedly find the maximum element in the heap and add it to the sorted array E starting at the righthand end and filling the array to the left.

Here `getMax(H)` just reads the root node of H. We have to do more work for `deleteMax(H)`, which removes the maximum element and then takes the remainder and turns it into a heap again (with one fewer element).

Algorithm `deleteMax(H)`:

```
copy element at last node into root node
remove last node
fixMaxHeap(H)
```

Note that copying the last element into the root and removing the last node both preserve the property that the left and right subtrees of the root are max heaps.

We use `fixMaxHeap(H)` to restore the heap property. Starting from a heap structure H where the left and right subtrees of the root are heaps, this makes H into a heap (again).

Algorithm `fixMaxHeap(H)`:

```
if H not a leaf:
        largerSubHeap = the left or right subheap with the larger
root
        if root(H).key < root(largerSubHeap).key:
                swap elements at root(H) and root(largerSubHeap)
                fixMaxHeap(largerSubHeap)
```

If the heap has depth d we see that `fixMaxHeap` takes at most $2d$ comparisons. Since a heap with n elements has depth $\lfloor \log n \rfloor$, `fixMaxHeap` takes $O(\log n)$ comparisons.

Before starting heapsort proper, we need to build the initial max heap. Here is a scheme whereby, starting from a heap structure H that does not necessarily have the partial order property, we build a max heap.

Algorithm `buildMaxHeap(H)`:

```
if H not a leaf:
        buildMaxHeap(left subtree of H)
        buildMaxHeap(right subtree of H)
        fixMaxHeap(H)
```

This is an example of a divide and conquer algorithm.

Analysis of `buildMaxHeap`: Suppose for simplicity $n = 2^k - 1$ so that the heap structure is a complete binary tree with depth $k - 1$. Let W(n) be the worst-case number of comparisons for `buildMaxHeap`.

$$W(n) = 2W((n-1)/2) + 2\log n$$

Apply the Master Theorem with $a = 2, b = 2$ and $f(n) = 2\log n$. The critical exponent $E = 1$, and so

$$W(n) = \Theta(n^E) = \Theta(n)$$

So we can build the heap in linear time.
We can now analyse Heapsort overall:

Algorithm `Heapsort`:

```
Build max heap H out of an array E of elements O(n)
for i = n to 1:
        max = getMax(H) O(1)
        deleteMax(H) O(log n)
        E[i] = max
```

This gives overall $O(n \log n)$ comparisons, the same as MergeSort.

Thus Heapsort is an optimal algorithm for sorting by comparisons.

Heapsort has an advantage over MergeSort, in that it can be carried out entirely in place, like Quicksort, **if we implement heaps using arrays**.

The idea is to store the heap level by level in an array starting at index 1. Starting at index 1 rather than 0 simplifies the arithmetic. Then the left and right children of node at index i are at positions $2i$ and $2i + 1$, respectively.

The parent node of the node at index i is at $\lfloor i/2 \rfloor$. There is no need for pointers.

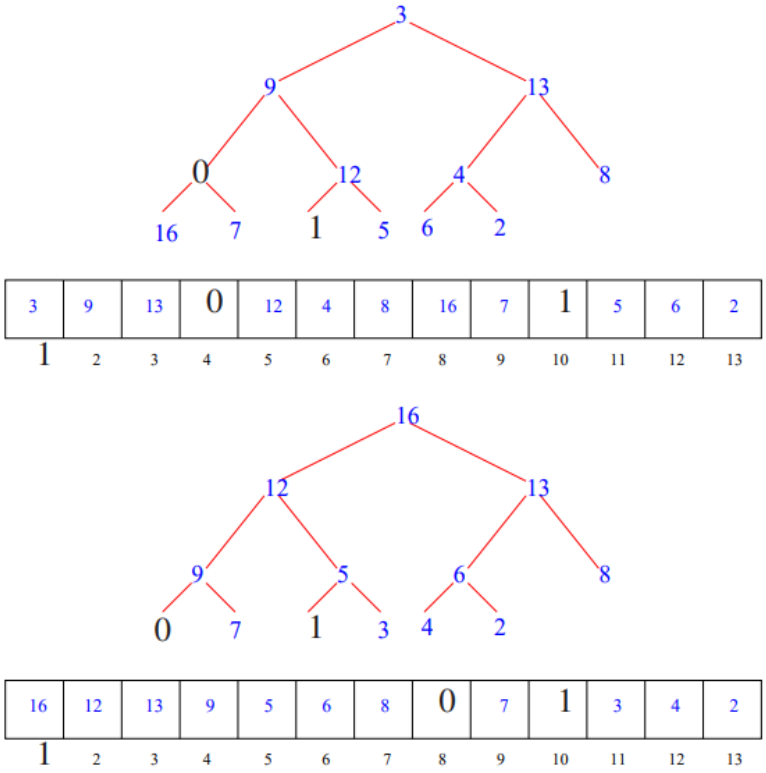


Figure 2.10: Examples of (1) a heap structure and the corresponding array (2) a max heap and the corresponding array.

Next we present Heapsort on an array:

Algorithm `Heapsort(E,n)`:

```
# perform Heapsort on elements 1..n of an array E of elements
heapsize = n
buildMaxHeap(1,heapsize)

# Invariant:
# max heap in E[1..heapsize] of first heapsize-many elements
# of sorted list
# elements in heapsize+1 to n are correctly sorted

while heapsize > 1:
        swap(1,heapsize)
        heapsize = heapsize - 1
        fixMaxHeap(1,heapsize)
```

Algorithm `buildMaxHeap(root,heapsize)`:

```
left = 2 * root
right = 2 * root + 1
if left ≤ heapsize:
        # root is not a leaf
        buildMaxHeap(left, heapsize)
        if right ≤ heapsize:
                # there is a right subtree
                buildMaxHeap(right, heapsize)
        fixMaxHeap(root,heapsize)
```

Algorithm `fixMaxHeap(root, heapsize)`:

```
left = 2 * root
right = 2 * root + 1
if left ≤ heapsize:
        # root is not a leaf
        if left = heapsize:
                # no right subheap
                largerSubHeap = left
        elif E[left].key > E[right].key:
                # favours right subheap if equal
                largerSubHeap = left
        else:
                largerSubHeap = right
        if E[root].key < E[largerSubHeap].key:
                swap(root,largerSubHeap)
                fixMaxHeap(largerSubHeap,heapsize)
```

Priority queues can be implemented as binary heaps.

Let us recall the Min PQ as used for Prim's algorithm:

- Each item x of the queue has a priority key$[x]$
- Items removed lowest key first

The operations are:

- `Q = PQcreate()`
- `isEmpty(Q)`
- `insert(Q, x)`
- `getMin(Q)`
- `deleteMin(Q)`

- `decreaseKey(Q,x,newkey)` — updates `key[x] = newkey`

We should use a min binary heap. We sketch how to perform the PQ operations using an array implementation of min binary heaps. Suppose we have an array E plus the `heapsize` parameter with the heap in `E[1..heapsize]` (inclusive).

Some operations are simple to implement:

- `Q = PQcreate()`
  - Create an empty array E of a suitable size with $\text{heapsize} = 0$
- `isEmpty(Q)`
  - Check if $\text{heapsize} = 0$
  - Time $O(1)$
- `getMin(Q)`
  - Return $E[1]$
  - Time $O(1)$
- `deleteMin(Q)`
  - `E[1] = E[heapsize]`
  - decrement `heapsize`
  - `fixMinHeap(1, heapsize)`
  - Time $O(\log n)$

Algorithm `insert(Q, x)`:

```
heapsize = heapsize + 1
E[heapsize] = x
# allow x to percolate towards the root until heap property is
restored
percolateup(heapsize)
```

```
procedure percolateup(c):
        if c > 1:
                parent = ⌊c/2⌋
                if E[c].key < E[parent].key:
                        swap(c, parent)
                        percolateup(parent)
```

Time taken is $O(\log n)$

It may be possible to build the queue in one go (time $O(n)$) rather than inserting elements individually (time $O(n \log n)$)

The operation `decreaseKey(Q,x,newkey)` is harder.

If we know the location c of x in the heap then we can change its key to `newkey` and use `percolateup(c)` to restore the heap. The problem is to locate x efficiently – in time $O(\log n)$ rather than $O(n)$

One possible solution is as follows:

- Suppose that each element has an identifier `id`
- Suppose that identifiers are integers in a compact range `[1..maxid]`
- Use a supplementary array `xref` to store the location of `id`:
  - so `xref[id] = k` means that element is at location k in heap
- We need to add code to `percolateup` to keep `xref` up to date as swaps occur