

# Lecture 1 – Sorting and Selection

Uriel Feige

Department of Computer Science and Applied Mathematics

The Weizman Institute

Rehovot 76100, Israel

`uriel.feige@weizmann.ac.il`

Nov 2 and 9, 2017

## 1 Sorting

When designing an algorithm for a computational problem, we shall be concerned with correctness and with complexity. The measure that we shall often use for complexity is the number of basic operations as a function of the input size. This roughly corresponds to the running time of the algorithm, but not exactly. The actual running time also depends on the data structures used, on hierarchical structure of memory and locality of reference, on the instruction set of the particular processor and the relative costs of various operations, on the optimizing compiler, and so on. Giving a detailed analysis of running time is beyond the scope of this course. Instead, we shall illustrate the issues involved through the example of sorting.

The input to a sorting problem is a list of  $n$  items  $X = x_1, \dots, x_n$ . There is a total order relation on the items (e.g., by size, or by lexicographic order). For simplicity and w.l.o.g. we may assume here that no two items are equal. The *order* in  $X$  of an item  $x_i$  is equal to the number of items in  $X$  not larger than  $x_i$ . The goal of the sorting algorithm is to come up with a permutation  $\pi : \{1, n\} \rightarrow \{1, n\}$  such that for every  $i$ , the order of  $x_i$  is  $\pi(i)$ .

We shall consider algorithms based on comparisons, and our measure of complexity will be the number of comparisons.

### 1.1 Insertion sort

**Insertion sort:** Consider the items of  $X$  one by one. By the end of phase  $i$ , we hold a list  $X_i$  that is the sorted version of the prefix  $x_1, \dots, x_i$ . In phase  $i + 1$ , use binary search to insert  $x_{i+1}$  in its correct location in  $X_i$ , obtaining the list  $X_{i+1}$  which is the sorted version of the prefix  $x_1, \dots, x_{i+1}$ . After phase  $n$ , output  $X_n$ .

**Complexity:** The number of comparison operations needed to insert item  $x_i$  is roughly  $\log i$  ( $\log$  denotes logarithms to the base 2). Hence the total number of comparisons is roughly  $\sum_{i=2}^n \log i \leq n \log n$ .

In terms of number of comparison operations, insertion sort is almost optimal w.r.t. comparison based algorithms. As computation branches only as the result of comparisons,

every comparison based sorting algorithm needs to perform at least  $\log n! \simeq n \log n$  comparisons so as to be able to produce each of the  $n!$  possible outputs. (A similar lower bound holds also for comparison based randomized algorithms.)

However, in practice, insertion sort is not very efficient. The reason is that it needs to pay a lot in data movement operations. Assume for example that  $X_i$  is held in an array of length  $i$ . Then in order to insert  $x_{i+1}$  and create the array  $X_{i+1}$ , we may need to move  $\Omega(i)$  items, giving a total of  $\Omega(n^2)$  data movements. For large values of  $n$ , the time needed to move data around may be a dominant factor in the running time for insertion sort, making the number of comparisons an inadequate measure of complexity.

Using more complicated data structures, it is possible to reduce the amount of data movement, perhaps at the expense of additional comparisons. This leads to algorithms such as *heapsort*. Hence the basic idea behind insertion sort does lead to algorithms that are of practical significance, though going from this basic idea to a practical algorithm involves additional design stages of the type that we will not be concerned with in this course.

## 1.2 Quicksort

We now consider another well known sorting algorithm.

**Quicksort:** Choose a *splitting* item  $x_i$ . Compare every item to  $x_i$  and split the list  $X - \{x_i\}$  into  $X_l$  (the items smaller than  $x_i$ ) and  $X_h$  (the items larger than  $x_i$ ). Sort each of the lists  $X_l$  and  $X_h$  recursively.

The complexity of quicksort depends on the choice of splitting item. Ideally, for every sublist  $X'$  generated during a run of the algorithm, the splitting item for  $X'$  is the median of  $X'$ , splitting  $X'$  into two equal size parts. This gives  $\log n$  levels of recursion. As each item is compared at most once in each level, the number of comparisons then is roughly  $n \log n$ .

However, if the splitting item is always the largest item in  $X'$ , we have  $n$  levels of recursion, requiring  $\Omega(n^2)$  comparisons.

One reason why quicksort works well is that the average case behaves more like the best case than like the worst case, as we shall see shortly. But the average case does require more comparisons than insertion sort. So why is quicksort preferable? The reason is that quicksort pays very little overhead on top of comparisons. It is relatively easy to implement quicksort in a way that at most one data item is moved per comparison. (For example, use two pointers advancing from both sides on  $X'$ , and think of the splitting item  $x$  as lying in between. Pointers stop on items that are misplaced with respect to  $x$ , these items are swapped, and then the pointers continue to advance.) Moreover, quicksort exhibits high locality of reference, paying relatively little overhead for paging/caching.

**Expected complexity:** We consider here randomized quicksort, in which the splitting item for  $X'$  is chosen uniformly at random from the items of  $X'$ . Intuitively, in this case  $X'$  is expected to split at a ratio of roughly  $(1/4, 3/4)$ , giving  $O(\log n)$  levels of recursion, and  $O(n \log n)$  comparisons. It is possible to use “brute force” and give a detailed analysis based on this intuition. However, we shall seek a simpler analysis.

Let us first prove again an upper bound on the number of comparisons. Consider an arbitrary pair of items. Quicksort compares them at most once. Hence the total number

of comparisons is at most  $\binom{n}{2}$ . (A similar upper bound holds for most comparison based sorting algorithms.)

Now let us use similar principles to bound the expected number of comparisons. For simplicity of notation, let  $x_1, \dots, x_n$  denote the sorted order of  $X$  (which is what the algorithm is trying to find). Let  $y_{ij}$  denote the indicator random variable that is 1 if  $x_i$  and  $x_j$  were directly compared during a run of quicksort, and 0 otherwise. (The random variable depends on the choices of random splitters.) Hence we are interested in the expectation  $E[\sum_{i < j} y_{ij}]$ . Working with expectations is very convenient, because of the *linearity of expectation*, which holds even for random variables that depend on each other. Hence we have  $E[\sum_{i < j} y_{ij}] = \sum_{i < j} E[y_{ij}]$ .

Observe that  $E[y_{ij}] = \sum_k k \text{Prob}[y_{ij} = k]$  is exactly equal to the probability that  $x_i$  and  $x_j$  were directly compared. This probability is exactly  $2/(j-i+1)$ . (Consider the first time when one of  $x_i, \dots, x_j$  happens to be a splitting item. If either  $x_i$  or  $x_j$  are the splitting item, they are directly compared. Otherwise, they end up in different sublists and are never compared.) Hence

$$\sum_{i < j} E[y_{ij}] = \sum_{i < j} \frac{2}{j-i+1} \simeq \sum_{i=1}^{n-1} 2 \ln(n-i) \simeq 2n \ln n$$

Hence the expected number of comparisons in randomized quicksort is roughly  $2n \ln n$ .

### 1.3 Randomized algorithms

Randomized quicksort is a randomized algorithm. It is of the ‘‘Las Vegas’’ type, meaning that it always produces the correct answer, and randomization only effects the running time. It is possible to run a deterministic version of quicksort in which the splitting item is chosen by some deterministic rule (e.g., the first item in  $X'$ , or the middle item of  $X'$ ). In this case the running time would depend on the input. For most inputs, the running time would be  $O(n \log n)$ , but for some rare inputs, the running time may be as high as  $\Omega(n^2)$ . The purpose of choosing the splitting item at random is so as to have quicksort have expected  $O(n \log n)$  running time for *every* input.

When analysing the running time of a randomized algorithm, one may be interested in the whole distribution (for every  $t$ , the probability that the algorithm runs for  $t$  steps). However, in most cases, the expectation is the major measure to consider. There are several reasons for this.

1. If the algorithm is run many times on many inputs, then by the laws of large numbers, the average run time will converge to the expectation. (The number of runs needed so as to observe such a convergence depends on other measures such as the variance. An implication of the items below is that the variance in the running time of randomized algorithms does not behave badly, implying rather quick convergence.)
2. Let  $T$  be the running time of a randomized algorithm. Because  $T \geq 0$ , we trivially have that for every  $c > 1$ ,  $\text{Prob}[T \geq cE[T]] \leq 1/c$ . Hence we also have tail bounds on the probability of exceptionally long runs.

3. To get better control of the tail bounds, one may stop a Las Vegas algorithm once it runs for  $2E[T]$  steps and restart it on the original input with fresh random coin tosses. In each period of  $2E[T]$  steps it has probability at least  $1/2$  of finishing. Hence in this case,  $\text{Prob}[T \geq cE[T]] \leq (1/2)^{c/2}$ . The probability of exceptionally long runs decays exponentially.
4. For many Las Vegas algorithms (such as quicksort), restarting on the original input is not really necessary. We can restart on the concatenation of sublists that ended the previous phase (as this is a legal input), or better still, just continue the algorithm without interruption. Hence for randomized quicksort, the probability of exceptionally long runs decays exponentially.

## 2 Selection

The largest item can be selected in  $n - 1$  comparisons, e.g., by a tree-like tournament. The second largest can be selected in  $n - 1 + \log n$  comparisons, by having a tournament among those who were directly eliminated by the largest item.

The most challenging item to select is the median. For deterministic algorithms, the minimum number of comparisons that suffices for finding the median is known to lie strictly between  $2n$  and  $3n$  (the exact value is open). We present a simple randomized median selection algorithm with linear complexity, often referred to as *quickselect*.

Given a list  $X$  with  $n$  items in which one seeks the item of order  $k$ , choose an item  $x_i$  uniformly at random to be the *splitting item*. Compare every other item to  $x_i$ , creating two lists  $X_h$  (of items of value higher than  $x_i$ ) and  $X_\ell$  (of items of value lower than  $x_i$ ). If  $|X_h| = k - 1$  then  $x_i$  was of order  $k$ . If  $|X_h| \geq k$ , continue recursively with  $X_h$  instead of  $X$ , updating  $n$  to be  $|X_h|$ . If  $|X_h| < k - 1$ , continue recursively with  $X_\ell$  instead of  $X$ , updating  $n$  to be  $|X_\ell|$  and  $k$  to be  $k - 1 - |X_h|$ .

Let us analyse the expected number of comparisons of the above randomized algorithm. A simple way of obtaining an upper bound that is not tight is as follows. (Note: the upper bound explained here is tighter and more elegant than the one shown in class.) We prove by induction that  $T(n) = 4n$  is an upper bound on the expected number of comparisons needed to select the element of rank  $k$  among  $n$  items. Consider a phase starting with  $n$  items. A splitting item is chosen,  $n - 1$  comparisons are made, and the algorithm continues to the next phase with one of the lists  $X_h$  or  $X_\ell$  (or already ends if the splitting item happened to be of rank  $k$ ). Hence the number of items continuing to the next phase is at most  $\max[1, |X_h|, |X_\ell|] \leq \max[|X_h|, |X_\ell|]$ . Observe that if the splitting item is chosen uniformly at random, then the expectation satisfies  $E[\max[|X_h|, |X_\ell|]] \leq \frac{3n}{4}$ . Using the fact that both expectation is linear and the upper bound  $4n$  is linear we get that:

$$T(n) \leq n - 1 + 4 \frac{3n}{4} \leq 4n$$

Hence  $4n$  is indeed an upper bound on  $T(n)$ .

The exact expectation (up to low order terms) can be derived using the following approach, though it involves more tedious computations. Recall the variables  $y_{ij}$  from the analysis of quicksort, with  $i < j$ . Consider several cases.

1.  $i < j \leq n/2$ . Elements  $i$  and  $j$  will be directly compared only if a splitting item is either  $i$  or  $j$  before there is a splitting item in the range  $i$  to  $n/2$  (which would separate  $i$  from the list containing the median). Hence  $E[y_{ij}] = \frac{2}{n/2-i}$ . Fixing  $i$  and summing over all  $j$  for this case we have  $\sum_{j|i < j \leq n/2} E[y_{ij}] \leq 2$ . Summing over all  $i < n/2$  we get  $n$  comparisons for this case.
2.  $n/2 \leq i < j$ . By symmetry with the above case, this case also contributes at most  $n$  comparisons (in expectation).
3.  $i < n/2 < j$ . Here  $E[y_{ij}] = \frac{2}{j-i+1}$ . Let us provide a rough estimate of  $\sum_{i < n/2 < j} \frac{2}{j-i+1}$ . For fixed  $i$  summing over all  $j$  gives at most  $2(\ln(n-i) - \ln(n/2-i) + O(1))$ . Summing over all  $i$  gives at most (after rearranging)  $2\sum_{i=n/2}^n \ln i - 2\sum_{i=1}^{n/2} \ln i + O(n) = 2\sum_{i=1}^n \ln i - 4\sum_{i=1}^{n/2} \ln i + O(n)$ . Changing to integrals and using  $\int \ln x dx = x \ln x - x$  we get at most  $2n \ln n - 2n - 4(\frac{n}{2} \ln \frac{n}{2} - 4\frac{n}{2}) + O(n) \simeq 2n \ln 2 + O(n)$ .

Hence overall the expected number of comparisons made by this algorithm is at most  $(2 + 2 \ln 2 + O(1))n$ . More careful estimations for the case  $i < n/2 < j$  can determine the exact constant contributed by the  $O(1)$  notation, but we refrain from doing this.

## 2.1 Selecting the median deterministically

We assume for simplicity that all numbers divide with no remainders in the description of algorithms, and hence omit ceiling and floor notations. (In the end of this section we explain how to change the algorithm so as to avoid this assumption.)

We present here a deterministic algorithm that outputs the median. Moreover, for every other item it also determines if it is larger or smaller than the median. (Any comparison based algorithm has this additional property. See Section 2.3. However, more general classes of algorithms need not have this property. For example, if one could do arithmetic on items, then the comparison  $2x_1 = x_2 + x_3$  shows that  $x_1$  is the median among  $\{x_1, x_2, x_3\}$ , but does not determine which of  $x_2$  and  $x_3$  is smaller than  $x_1$ .)

The input to the algorithm is a set  $X$  of  $n$  items and a parameter  $k$ , and the desired output is the  $k$ th largest item.

Partition the set  $X$  into  $n/5$  groups  $R_1, R_2, \dots$ , each of size 5. For each group  $R_i$  find its median  $m_i$ . Consider only the median items  $m_1, m_2, \dots$  of each group (hence there are  $n/5$  items). Recursively, find the median among these median items. Let  $m$  be this median item. Compare  $m$  against each of the original items, other than those for which the outcome of the comparison is already known. Specifically, there is no need to compare  $m$  against any of the other median items (by our assumption that our median finding algorithm determines for other items whether they are larger than the median). Likewise, if  $m_i < m$  then  $m$  needs to be compared only against those two items in  $R_i$  that are larger than  $m_i$ , and if  $m_i > m$  then  $m$  needs to be compared only against those two items in  $R_i$  that are smaller than  $m_i$ .

The above completes one phase of the algorithm.

Let  $X_\ell$  denote the set of items smaller than  $m$ , and let  $X_h$  denote the set of items larger than  $m$ . If  $|X_h| = k - 1$  then  $m$  was of order  $k$ . If  $|X_h| \geq k$ , continue recursively with  $X_h$

instead of  $X$ , updating  $n$  to be  $|X_h|$ . If  $|X_h| < k - 1$ , continue recursively with  $X_\ell$  instead of  $X$ , updating  $n$  to be  $|X_\ell|$  and  $k$  to be  $k - 1 - |X_h|$ .

Clearly, the above algorithm outputs the median. To analyse its running time, we use a recurrence relation. Let  $T^*(n)$  be the best possible upper bound (taken over all possible values of  $k$ ) on the time to select the  $k$ th largest item out of  $n$  items. Then we have the following recurrence relation:

$$T^*(n) \leq \frac{n}{5}T^*(5) + T^*\left(\frac{n}{5}\right) + \left(\frac{2n}{5} - 2\right) + T^*\left(\frac{7n}{10}\right)$$

The first term is for finding the medians in the  $n/5$  groups. The second term is for finding  $m$ , the median of medians. The third term is the additional comparisons made by  $m$  (two items in each group, except for the group containing  $m$  itself). The fourth term is for doing selection in either  $X_\ell$  or  $X_h$  (where-ever needed). Observe that there are  $\lceil n/10 \rceil$  medians  $m_i$  smaller or equal to  $m$ , and each of them certifies that two additional items are smaller than  $m$ . Hence  $|X_h| \leq \frac{7n}{10}$ , and by symmetry, the same holds for  $|X_\ell|$ .

Rather than finding  $T^*(n)$  (which might be a fairly complicated function of  $n$ ), let us find some upper bound on it, namely, some  $T(n)$  that satisfies  $T(n) \geq T^*(n)$  for all  $n$ . Observe that

$$T^*(n) \leq \frac{n}{5}T(5) + T\left(\frac{n}{5}\right) + \left(\frac{2n}{5} - 2\right) + T\left(\frac{7n}{10}\right)$$

and hence if  $T(n)$  satisfies the reverse inequality

$$T(n) \geq \frac{n}{5}T(5) + T\left(\frac{n}{5}\right) + \left(\frac{2n}{5} - 2\right) + T\left(\frac{7n}{10}\right)$$

then surely  $T(n)$  is an upper bound on  $T^*(n)$ .

The above inequality does not allow for a solution of the form  $T(x) = cx$  for some constant  $c$ . To overcome this problem, the key insight is that for sufficiently large  $c$ , we know that  $T(5)$  is in fact much smaller than  $cn$ , and hence can be replaced by a smaller expression. In particular, five items can be sorted by insertion sort using only 8 comparisons, and the median can be found by only 7 comparisons. Hence we can replace the inequality by:

$$T(n) \geq 7\frac{n}{5} + T\left(\frac{n}{5}\right) + \left(\frac{2n}{5} - 2\right) + T\left(\frac{7n}{10}\right)$$

Consider a solution of the form  $T(x) = cx$ . We get

$$cn \geq 7\frac{n}{5} + c\frac{n}{5} + \left(\frac{2n}{5} - 2\right) + c\frac{7n}{10} = \frac{9c}{10}n + \frac{9}{5}n - 2$$

Taking  $c = 18$  we see that  $T(n) = 18n$  is a valid upper bound.

One might be concerned that rounding issues (the fact that at various stages  $n$  is not divisible by 5) may introduce errors in our analysis of  $T(n)$  and that they build up. There is sufficient slackness in the analysis to address them. Here is one way of doing this. Express  $n = 5d + r$ , where  $d \geq 1$  is integer, and  $r \in \{0, 1, 2, 3, 4\}$ . Run the corresponding phase only on  $5d$  items, putting  $r$  items aside. Then compare the  $r$  items with  $m$  (to see which of them continues to the next phase), and the phase ends. The desired inequality becomes:

$$T(n) = T(5d + r) \geq 7d + T(d) + (2d - 2) + T\left(\frac{7d}{2} + r\right) + r$$

For  $r \in \{3, 4\}$ ,  $T(n) = 18n$  no longer is a valid solution to the above inequality. Hence it seems as if we should seek an upper bound larger than  $18n$ . However, this is misleading. In fact, we can prove an upper bound smaller than  $18n$ , as it strengthens the inductive hypothesis. We need to be careful that the upper bound holds for all  $n < 5$  (which serve as the base case for the recursion). Specifically, we can take  $T(n) = 18n - 2$  and then we get that

$$90d + 18r - 2 \geq 7d + (18d - 2) + (2d - 2) + (18\left(\frac{7d}{2} + r\right) - 2) + r = 90d + 18r - 6 + r$$

and the inequality holds (because  $r \leq 4$ ).

## 2.2 An asymptotically optimal randomized median selection algorithm

Pick at random a set  $S$  of  $\epsilon n$  items in  $X$ , where  $\epsilon < \frac{1}{8}$  will be determined later.

In  $S$ , use any linear time selection algorithm to select an item  $h$  of order  $(\frac{1}{2} - \epsilon)|S|$  and an item  $\ell$  of order  $(\frac{1}{2} + \epsilon)|S|$ .

Compare every item  $x_i \in X$  first with  $h$ , and if  $x_i < h$ , then also compare  $x_i$  with  $\ell$ . Let  $Y$  be the set of items satisfying  $\ell < x_i < h$ . Let  $X_\ell$  be the set of items smaller than  $\ell$ , and let  $X_h$  be the set of items larger than  $h$ .

Check that  $(\frac{1}{2} - 2\epsilon)n \leq |X_h| < \frac{n}{2}$  and  $(\frac{1}{2} - 2\epsilon)n \leq |X_\ell| < \frac{n}{2}$ . If either one of these four inequalities fail, then restart the algorithm.

Else, the median in  $X$  is the item of order  $n/2 - |X_h|$  in  $Y$ . Search for this item in  $Y$  using any linear time deterministic selection algorithm.

Let  $cn$  be the running time of the deterministic selection algorithm. Then if the algorithm does not restart, its running time is at most:  $2c\epsilon n + n + (\frac{1}{2} + 2\epsilon)n + 4c\epsilon n$ . The first term comes from selecting two items in  $S$  (in fact, once one item is selected, the second item can be selected within a set smaller than  $S$ , but we ignore such improvements), the second term is for comparisons with  $h$  (also here, items of  $S$  need not be compared with  $h$ ), the third term is for comparison with  $\ell$  (cannot be possibly done more than  $(\frac{1}{2} + 2\epsilon)n$  times, because then  $X_h$  is too small and the algorithm restarts), and the fourth term is for selection within  $Y$ . So altogether the running time with no restarts is at most  $T_1 = \frac{3n}{2} + (6c + 2)\epsilon n$ .

Let  $p$  denote the probability that the algorithm restarts. Then the probability that the algorithm is run for the  $k$ th time is  $p^{k-1}$ , and hence the expected number of times that the algorithm is run is  $\sum_{i \geq 0} p^i = \frac{1}{1-p}$ . Altogether, the expected running time of the algorithm is at most  $\frac{T_1}{1-p}$ .

Let us bound  $p$  from above. For this we use the following concentration result for independent random variables.

**Theorem.** Let  $X = \sum X_i$  where the  $X_i$  are independent Boolean random variables,  $X_i \in \{0, 1\}$ ,  $p_i = Pr[X_i = 1]$ , and  $\mu = E[X] = \sum p_i$ . For  $0 < \delta \leq 1$  we have the Chernoff bounds

$$Pr[X \geq (1 + \delta)\mu] < \left[\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}}\right]^\mu < e^{(-\delta^2/2 + \delta^3/6)\mu} \leq e^{(-\delta^2/3)\mu}$$

and

$$\Pr[X \leq (1 - \delta)\mu] < e^{-\delta^2\mu/2}.$$

Suppose for simplicity that we choose  $S$  with repetition. To have  $|X_h| \geq (\frac{1}{2} - 2\epsilon)n$  we need  $h$  to be of order at most  $(\frac{1}{2} - 2\epsilon)n$  in  $X$ . This will happen if  $S$  has at most  $(\frac{1}{2} - \epsilon)|S|$  items among the first  $(\frac{1}{2} - 2\epsilon)n$  of  $X$ . The probability that this fails can be upper bounded using the Chernoff bound, by setting  $\mu = (\frac{1}{2} - 2\epsilon)|S| \geq \frac{\epsilon n}{4}$  and  $\delta \geq 2\epsilon$ . We get a probability of failure not more than  $e^{-\epsilon^3 n/3}$ . Altogether, there are four types of bad events to exclude, and for each of them its probability can be upper bound in the same manner. One obtains that  $p \leq 4e^{-\epsilon^3 n/3}$ .

Picking  $\epsilon = n^{-1/4}$ , the expected number of comparisons made by the median selecting algorithm becomes  $\frac{3n}{2} + o(n)$ .

### 2.3 A lower bound for selecting the median

A lower bound of  $\frac{3n}{2} - o(n)$  is known for the expected number of comparisons needed by any randomized algorithm for selecting the median. Here we shall show a similar lower bound for deterministic algorithms (which is simpler to prove, and stronger deterministic lower bounds are known).

Let the items be  $\{1, \dots, n\}$ , given as input in some arbitrary order, and one needs to select the median item  $n/2$ . For item  $i < n/2$ , a *useful comparison* is one that compares  $i$  with some  $j$  in the range  $i < j \leq n/2$ . For item  $i > n/2$ , a *useful comparison* is one that compares  $i$  with some  $j$  in the range  $n/2 \leq j < i$ . Consider the set of comparisons made when an algorithm for selecting the median ends. We claim that every item  $i \neq n/2$  must be involved in a useful comparison. Suppose otherwise for some  $i < n/2$  (the case  $i > n/2$  is similar). Then the total order  $\{1, \dots, i-1, i+1, \dots, n/2, i, n/2+1, \dots, n\}$  is also consistent with the set of comparisons, and in this order  $i$  is of rank  $n/2$ . This means that the algorithm errs if in its input order the items holding values  $i$  up to  $n/2$  are permuted cyclically ( $i$  replaces  $n/2$ , and  $j+1$  replaces  $j$  for  $i < j \leq n/2$ ).

It follows that every median finding algorithm (including randomized ‘‘Las Vegas’’ ones) must make  $n - 1$  useful comparisons. To prove a nontrivial lower bound on the number of comparisons, we show that it must also make some non-useful comparisons. We refer to these as *wasted comparisons*.

The lower bound for the deterministic case is based on an adversary argument. Rather than fixing the values of all items in advance, this is done ‘‘on the fly’’. At every step, the algorithm points to two items that it wishes to compare. If both were already involved in previous comparisons, then their values are already fixed, and the reply is the result of comparing these values. However, if one or both of these items were not involved in previous comparisons, then the adversary picks values for them (among values not picked earlier for other items), and then answers the comparison. The goal of the adversary is to pick values in such a way that would delay the end of the algorithm as much as possible. (More generally, rather than picking values, the adversary may just pick the outcome of the comparison, in a way consistent with all previous comparisons, but without actually picking values for the items. This may lead to stronger lower bounds, but will not be used here.)



Here is a simple strategy for the adversary. For every comparison that involves a new item, give the new item a value that will make the comparison *wasted*. How many such comparisons can the adversary waste? As long as there is at least one item smaller than  $n/2$  and at most one item larger than  $n/2$  that are not fixed, the adversary can continue with his strategy. Each wasted comparison fixes at most one item smaller than  $n/2$  and at most one item larger than  $n/2$ . Hence the adversary can force  $n/2 - 1$  wasted comparisons. Together with the useful comparisons, this gives a lower bound of  $3n/2 - 2$  comparisons.

**Open Question.** Is there a deterministic median selection algorithm that makes at most  $5n/2$  comparisons.

### 3 Lower bounds for randomized algorithms

#### 3.1 Yao's principle

A convenient way to lower bound the expected number of comparisons made by randomized algorithms is to use *Yao's principle*. Namely, one exhibits a distribution  $D$  over inputs such that no deterministic algorithm has a small expected number of comparisons. Observe that every randomized algorithm  $R$  is a distribution (call it  $D_R$ ) over deterministic algorithms, where each possible setting of the coin tosses of  $R$  gives a deterministic algorithm. (Remark: for every  $n$  there are only finitely many non-redundant deterministic algorithms. Hence  $D_R$  has finite support even if there is no bound on the number of coin tosses that  $R$  makes.)

Suppose that every deterministic algorithm makes in expectation at least  $t$  comparisons, where expectation is taken over a random input sampled from  $D$ . Then this also holds for every algorithm in the support of  $D_R$ . By linearity of expectation, the expected number of comparisons made by  $R$  (expectation both over random coin tosses of  $R$  and selection of input from  $D$ ) is at least  $t$ . Hence for some input in  $D$ , the expected number of comparisons made by  $R$  is at least  $t$ .

#### 3.2 The principle of deferred decisions

The principle of deferred decisions is useful both for analysing randomized algorithms and for proving lower bounds on randomized algorithms. The basic idea is that when a random choice needs to be made (e.g., the input is a random permutation; the algorithm permutes the items at random), the outcome of the choice is revealed gradually on a "need to know" basis. For example, this principle shows that if randomized quicksort is implemented by first permuting all items at random (and thereafter continuing deterministically), item 4 in Section 1.3 still applies (the probability of exceptionally long runs decays exponentially).

#### 3.3 Lower bound for randomized sorting

We have seen that every deterministic (comparison based) sorting algorithm requires at least  $\log n!$  comparisons on some input. Let us now prove the same for the expected number of comparisons for randomized sorting algorithms. By Yao's principle, it suffices to show that there is a distribution over inputs such that every deterministic algorithm requires at least  $\log n!$  comparisons on this distribution. As a distribution, we shall take the uniform

distribution over all possible outcomes. We shall now prove a more general statement, for which the randomized sorting lower bound is only a special case.

Suppose that a problem has a set  $S$  of  $N$  possible inputs that all have distinct solutions. Suppose further that every *query* step in which an algorithm accesses the input has at most two possible outcomes, depending on the input (e.g., a comparison has only two possible outcomes). If the input instance is chosen uniformly at random from  $S$ , then every randomized algorithm requires an expected number of at least  $\log N$  queries.

To prove the above claim, let  $T(N)$  denote a lower bound on the expected number of queries, and we shall prove  $T(N) \geq \log N$  by induction. For the base case observe that  $T(1) = 0 = \log 1$  (no query is required if there is only one possible input) and  $T(2) = 1 = \log 2$  (at least one query is required if there are two possible inputs with distinct solutions). Now we proceed to the inductive step. On having  $N$  possible inputs (with distinct solutions) the algorithm makes a query. Regardless of the query, there is some  $0 \leq p \leq 1$  (that depends on the query) such that on exactly  $pN$  inputs the answer is 0 and on  $(1-p)N$  queries the answer is 1. Given that the input instance is selected uniformly at random, and using the *principle of deferred decisions*, the probability that the answer is 0 is  $p$ . Hence any  $T(N)$  that satisfies the following inequality is a lower bound:

$$T(N) \geq 1 + pT(pN) + (1-p)T((1-p)N) \geq 1 + p \log pN + (1-p) \log(1-p)N$$

$$= 1 + \log N + p \log p + (1-p) \log(1-p) = 1 + \log N - H(p) \geq \log N$$

as desired. Here  $H(p) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$  is the *entropy* function which is known to be maximized at  $p = \frac{1}{2}$ .