

# Randomised Algorithms

by

Gudmund S. Frandsen

*This lecture note was written for the course Pearls of Theory at University of Aarhus. Most recent revision, March 2, 1998.*

## ABSTRACT

Randomised algorithms offer simple and efficient solutions to a number of problems, though it can be a complex task to prove that a specific randomised algorithm has a desired property.

This note describes a simple technique for bounding the expected running time of randomised algorithms, namely analysis by indicator variables combined with linearity of expectation.

The technique is applied to quicksort, to randomised dictionaries and to two selected geometric algorithms: construction of a binary planar autopartition and construction of a convex hull in the plane. All algorithms are simple, but without the proper technique the analysis could be quite messy.

## 1. INTRODUCTION

By the notion of *randomised* algorithm we mean algorithms where we allow the use of coin tosses, or, more generally, a variable may be assigned a random value (according to some fixed distribution). The outcome of the coin tosses may affect the correctness of the algorithm, the running time, the space usage and other characteristics.

One might doubt the usefulness of randomisation if it leads to an incorrect algorithm, but one of the first modern examples of a randomised algorithm, the Rabin-Miller primality test, which may give a wrong answer with probability  $\leq \frac{1}{4}$ , is nevertheless used a lot in practice. It is much simpler and faster than the best known deterministic primality test, and the probability of making an error can be reduced to  $\frac{1}{4^n}$  by running the algorithm  $n$  times using *independent* coin tosses.

The examples of randomised algorithms in this note, will give correct output on every run, but their time usage is affected by the outcome of coin tosses. For a fixed input, we may therefore speak about the expected running time of our randomised algorithm. As usual we will be interested in worst case behaviour, i.e.

the maximum expected running time, where maximum is taken over all inputs, and expectation is taken over all coin tosses.

Presently, our local workstations have no facilities for tossing coins. This may seem to render our randomised algorithms useless, since, in general, what we can prove about the expected behaviour of an algorithm does not necessarily hold when we replace true randomness by what a built-in pseudo-random number generator can provide. In practice, however, it appears that some forms of pseudo-randomness work all right. In our complexity analysis, we will assume that we have true randomness for free, and then discuss the implementation issue further at the end of this note.

## 2. QUICKSORT

We start by considering a deterministic version of quicksort. It does sort correctly, but time usage depends on the actual input. For the particular version we have chosen, the worst case occurs when the input is already sorted.  $\Omega(n^2)$  comparisons are needed in that case.

Algorithm: Deterministic Quicksort (DQ)

Input:  $L = [a_1, \dots, a_n]$ , a list of distinct numbers

Output:  $L$  sorted in ascending order

Method: (sketch only)

If  $|L| \leq 1$  then return  $L$  otherwise:

1. Let  $e = a_1$
2. Split  $L' = L - \{e\}$  into the two sublists  
 $L_1 = [a_i \in L | a_i < e]$   
and  
 $L_2 = [a_i \in L | a_i > e]$   
by comparing  $e$  to each element of  $L'$ .
3. Recursively, sort  $L_1$  and  $L_2$ .
4. Return  
sorted- $L_1 \cdot [e] \cdot$  sorted- $L_2$

This bad behaviour can be removed by a simple randomisation, where the deterministic choice of  $e$  in line 1 is replaced with a randomised selection of  $e$ .

Algorithm: Randomised Quicksort (RQ)

Input:  $L = [a_1, \dots, a_n]$ , a list of distinct numbers

Output:  $L$  sorted in ascending order

Method: (sketch only)

If  $|L| \leq 1$  then return  $L$  otherwise:

1. Select  $e$  randomly from  $L$  using the uniform distribution

2. Split  $L' = L - \{e\}$  into the two sublists

$$L_1 = [a_i \in L | a_i < e]$$

and

$$L_2 = [a_i \in L | a_i > e]$$

by comparing  $e$  to each element of  $L'$ .

3. Recursively, sort  $L_1$  and  $L_2$ .

4. Return

$$\text{sorted-}L_1 \cdot [e] \cdot \text{sorted-}L_2$$

It turns out that the expected number of comparisons made by the modified algorithm RQ on any fixed input list (and therefore also on the worst case input) is  $O(n \log n)$ . We shall provide a particularly elegant argument for this upper bound.

### 3. ANALYSIS OF RANDOMISED QUICKSORT

In our analysis, we will count only the number of comparisons made.

**Exercise 1.** *Argue that the number of comparisons made is an upper bound on the actual running time (up to a constant factor) when using a reasonable model of computation.*

Since the procedure is recursive one may naturally get the idea of finding a recurrence describing the expected number of comparisons made, e.g.

$$C(n) = \sum_{j=1}^n \frac{1}{n} [n - 1 + C(j - 1) + C(n - j)] \quad \text{and} \quad C(0) = C(1) = 0$$

Although this recurrence can be solved, it is much more instructive to take a different approach and use *indicator variables*. The latter will also be used in our next example from computational geometry.

For the analysis of the RQ-algorithm, let some input list  $L_0 = [a_1, \dots, a_n]$  be given and let the corresponding output list be  $\text{sorted-}L_0 = [b_1, \dots, b_n]$ .

We are going to use the special binary random variables  $\{X_{ij}\}$  called *indicator variables*, defined by

$$X_{ij} = \begin{cases} 1, & \text{if the numbers } b_i \text{ and } b_j \text{ are compared (immediately or} \\ & \text{in a recursive call) when running RQ on input } L_0 \\ 0, & \text{otherwise} \end{cases}$$

We want to find an upper bound on  $\mathbf{E}[X]$ , where  $X$  is a random variable taking as value the total number of comparisons made, when running RQ on input  $L_0$ . Note that  $X = \sum_{ij} X_{ij}$ . Using that the expectation,  $\mathbf{E}[\cdot]$  is a linear function, we need only consider the indicator variables:

$$\begin{aligned} \mathbf{E}[X] &= \mathbf{E}\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}\right] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \mathbf{E}[X_{ij}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n (0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1]) \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \Pr[X_{ij} = 1] \end{aligned}$$

This leaves us to compute  $\Pr[X_{ij} = 1]$ . Assume in the following that  $j > i$ .

During the execution of RQ there will be a unique recursive call RQ' for which the input list  $L'$  contains both  $b_i$  and  $b_j$  but none of the sublists  $L'_1$  and  $L'_2$  contains both  $b_i$  and  $b_j$ .

$L'$  must contain all the elements  $b_i, b_{i+1}, \dots, b_{j-1}, b_j$ , and  $e'$  must be chosen among one of these  $j - i + 1$  elements, for  $b_i$  and  $b_j$  not to be put into the same sublist when making recursive calls.

If  $b_i$  is compared to  $b_j$ , it must occur in RQ' during the splitting of  $L'$  into  $L'_1$  and  $L'_2$ , and the  $e'$  selected in line 1 must be one of  $b_i$  and  $b_j$  out of the  $j - i + 1$  possible choices, i.e.

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$$

The upper bound on the expected number of comparisons follows from a final calculation:

$$\mathbf{E}[X] = \sum_{i=1}^n \sum_{j=i+1}^n \Pr[X_{ij} = 1]$$

$$\begin{aligned}
&= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\
&= 2 \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\leq 2 \sum_{i=1}^n \ln n \\
&= 2n \ln n
\end{aligned}$$

#### 4. COMPUTATIONAL GEOMETRY

The following two examples of randomisation will both be within the domain of geometric computations.

Computational geometry is a large independent discipline, and since our focus is randomisation, we do not want to get lost in details related to geometric algorithms only, and the following restrictions will make life much easier for us.

*No degeneracies.* We assume that the input to our algorithm is in a general position, i.e. no three points lie on a straight line, no four points lie on the same circle, no two line segments are parallel, etc. The absence of degeneracies allows us to discard the treatment of singular cases to obtain simpler algorithms.

*Exact real arithmetic.* In practice numerical precision on computers is limited. However, to avoid issues related to numerical analysis, we will assume the availability of exact real arithmetic.

Both the above restrictions are common in the computational geometry literature. The assumption of no degeneracies is not a serious restriction. It is usually fairly simple to extend an algorithm to handle degenerate input. However, the restriction to exact real arithmetic may be more serious. In practice, it can be a nontrivial task to massage a numerically unstable algorithm into a working program.

*2-dimensional world.* We restrict ourselves to the 2-dimensional version of the geometric problems, we consider. Generalisations to the 3-dimensional case can be found through the bibliography at the end of this note.

#### 5. PLANAR AUTO-PARTITION

Our first geometric example is the construction of a binary planar autopartition of a set of line segments.

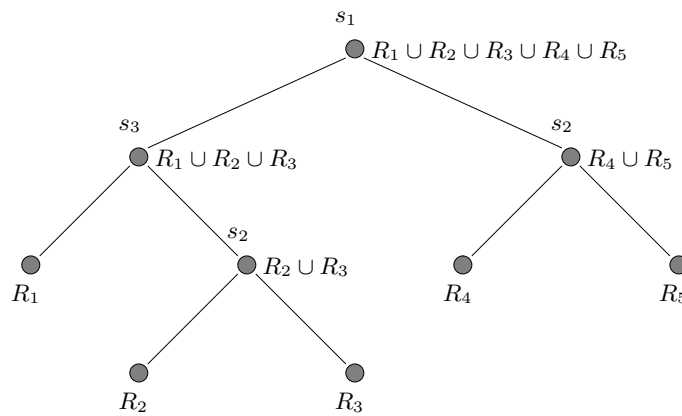
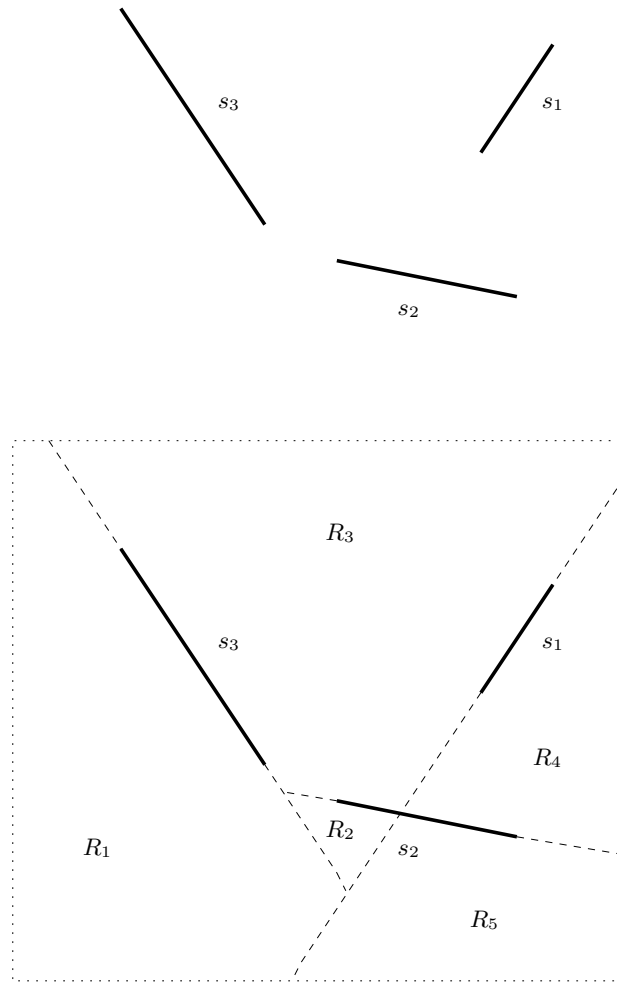


Figure 1: Binary planar autopartition of line segments

Figure 1 shows some line segments in the plane together with a binary tree representing a possible autopartition of the segments. Each node of the tree has associated a particular region of the plane such that the region corresponding to the root node is the entire plane. The region corresponding to a leaf does not contain (parts of) any line segments, but the regions corresponding to internal nodes do all contain (parts of) line segments. Each internal node has associated one of the line segments in its region. This line segment will split the region in two (if necessary through an extension of the line segment to infinity). The resulting two half-regions are associated with the sons. A region at node  $v$  is bounded by (extensions of) the line segments associated with the nodes on the path from  $v$  to the root.

An important application of partition trees is the *painters algorithm* for rendering a scene with hidden lines (surfaces) concealed. Consider a flight simulator. One may have a complex underlying fixed scenario that has to be projected onto a screen from rapidly changing points of view. We want no X-ray vision, i.e. only objects that are visible from a particular point of view must be visible on the screen. The basic idea of the painters algorithm is to paint the whole scene by first considering the most remote object and gradually progressing to nearer objects while painting *over* any more distant objects, ending up with a painting of what is actually visible. This must be repeated, when the point of view is changed.

When representing the objects by line segments, it turns out that an initial computation of a binary partition supports repeated applications of the painters algorithm: The region to be painted is represented by the partition tree. First paint the most remote subtree recursively, then paint the dividing line segment (or the part of it that is within the region considered) and finally paint the nearest subtree recursively. The time usage of the painters algorithm is proportional to the size of the partition tree. It is therefore important to have a small partition tree.

For a fixed set of  $n$  line segments there could be many different autopartitions, depending on which line segment is chosen to make a split when more than one is applicable. In the example of figure 1, the line segment  $s_2$  is split over two regions before any splitting occurs along  $s_2$  itself. This means that not one but two internal nodes are labelled  $s_2$ , implying that the partition tree has more than  $n$  internal nodes in total. For some constellations of line segments this situation is unavoidable in that all possible autopartition trees have several internal nodes associated with a single line segment.

**Exercise 2.** *Show how to place  $n$  disjoint line segments in the plane, so the smallest autopartition for them has more than  $n$  internal nodes.*

For application in the painters algorithm, it is important to have a small tree, so

how bad can the situation get?

We are going to present a simple recursive randomised algorithm to construct an autopartition. In the analysis, we will find that for any constellation of  $n$  line segments the expected size of the found tree is  $O(n \log n)$ .

This result has an important interpretation: Every constellation of  $n$  line segments has an autopartition of size  $O(n \log n)$ , since there is a positive probability of finding one! Such a nonconstructive proof of existence is said to use the *probabilistic method*. In a single run of the randomised algorithm, we may have bad luck and get a large autopartition, but we expect to get a small autopartition fast when running the algorithm repeatedly using *independent* random choices.

**Exercise 3.** *Show that we expect to use  $\leq 2$  runs of algorithm RA to find an autopartition of size  $\leq 2s(n)$ , if the expected size of an autopartition constructed in a single run of algorithm RA is  $s(n)$ .*

It is an open problem whether an autopartition of size  $O(n)$  always exists. Even the present bound of  $O(n \log n)$  is non-obvious.

**Exercise 4.** *Show how to place  $n$  disjoint line segments in the plane, so there exists a large autopartition for them of size  $\Omega(n^2)$ .*

Algorithm: Randomised Autopartitioning (RA)

Input:  $L = [s_1, \dots, s_n]$ , a list of nonintersecting line segments in some region  $R$  of the plane.

Output:  $T$ , an autopartition of  $L$

Method: (sketch only)

If  $|L| = 0$  return the empty tree, otherwise:

1. Select  $e$  randomly from  $L$  using the uniform distribution. (the extension of)  $e$  divides the region  $R$  into two subregions  $R_1$  and  $R_2$ .
2. Construct  $L_i$ , the restriction of  $L' = L - \{e\}$  to  $R_i$  for  $i = 1, 2$ .
3. Recursively, find auto-partitions  $T_1$  and  $T_2$  for  $L_1$  in  $R_1$  and  $L_2$  in  $R_2$ .
4. Return the tree with root labelled  $(e, R)$  and subtrees  $T_1$  and  $T_2$ .

## 6. ANALYSING RANDOMISED AUTO-PARTITIONING

In our analysis, we will count only the number of times (part of) a line segment is split between two regions.

**Exercise 5.** Argue that the number of nodes in an autopartition tree is bounded by  $O(n)$  plus the number of times (part of) a line segment is split between two regions.

Our analysis will be quite similar to the one we conducted for quicksort, so for the analysis of RA, let some input list  $L_0 = [s_1, \dots, s_n]$  be given and define the indicator variables

$$Y_{ij} = \begin{cases} 1, & \text{if the line segment } s_j \text{ is split by (the extension of) the} \\ & \text{line segment } s_i \text{ (immediately or in a recursive call) when} \\ & \text{running RA on input } L_0 \\ 0, & \text{otherwise} \end{cases}$$

Let  $Y$  be the random variable, whose value is the total number of segment splits made, when running RA on input  $L_0$ . We want to find an upper bound on  $\mathbf{E}[Y]$ :

$$\begin{aligned} \mathbf{E}[Y] &= \mathbf{E}\left[\sum_{i=1}^n \sum_{j=1}^n Y_{ij}\right] \\ &= \sum_{i=1}^n \sum_{j=1}^n \mathbf{E}[Y_{ij}] \\ &= \sum_{i=1}^n \sum_{j=1}^n \Pr[Y_{ij} = 1] \end{aligned}$$

We need to compute  $\Pr[Y_{ij} = 1]$ . Assume that the extension of  $s_i$  intersects  $s_j$  (otherwise  $\Pr[Y_{ij} = 1] = 0$ ). Let  $t$  be the point of intersection. Let  $\text{index}(i, j)$  be the number of line segments that  $s_i$  intersects before intersecting  $s_j$ . In the typical scenario of figure 2  $\text{index}(i, j) = 3$ .

During the execution of RA there will be a unique recursive call RA', for which the region  $R$  considered contains the intersection point  $t$  and for which the input list  $L'$  contains (parts of) both  $s_i$  and  $s_j$  such that these conditions do not both hold for any of the recursive calls on  $(R'_1, L'_1)$  and  $(R'_2, L'_2)$  initiated from RA'.

We will argue in terms of the example in figure 2, but the proof is valid in general.  $L'$  must contain (parts of) all the elements  $v_1, v_2, \dots, v_{\text{index}(i, j)}$  in addition to  $s_i$  and  $s_j$ .

If  $s_j$  is ever split by (the extension of)  $s_i$ , it must occur in RA' during the splitting of  $L'$  into  $L'_1$  and  $L'_2$ , and the  $e'$  selected in line 1 must be precisely  $s_i$  out of at least  $\text{index}(i, j) + 2$  possible choices that lead to separation of  $s_i$  and  $s_j$ , i.e.

$$\Pr[Y_{ij} = 1] \leq \frac{1}{\text{index}(i, j) + 2}$$

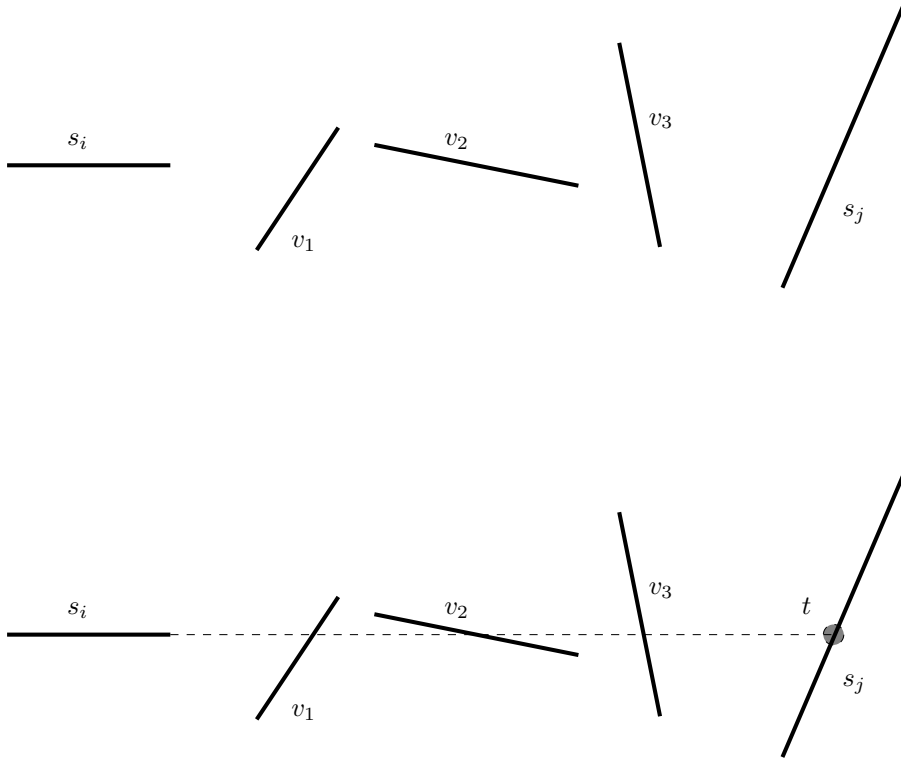


Figure 2:  $\text{Index}(i, j) = 3$ .

For a fixed segment  $s_k$  and number  $m \in \{0, 1, 2, \dots, n-2\}$ , there exist at most two segments  $s_l$  with  $\text{index}(k, l) = m$  (at most one segment to the right of  $s_k$  and at most one segment to the left of  $s_k$ , assuming that  $s_k$  is oriented east-west).

The upper bound on the expected number of segment splittings follows from a final calculation (we let  $\text{index}(i, j) = \infty$  when otherwise undefined).

$$\begin{aligned}
\mathbf{E}[Y] &= \sum_{i=1}^n \sum_{j=1}^n \Pr[Y_{ij} = 1] \\
&\leq \sum_{i=1}^n \sum_{j=1}^n \frac{1}{\text{index}(i, j) + 2} \\
&\leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \\
&\leq 2 \sum_{i=1}^n \ln n \\
&= 2n \ln n
\end{aligned}$$

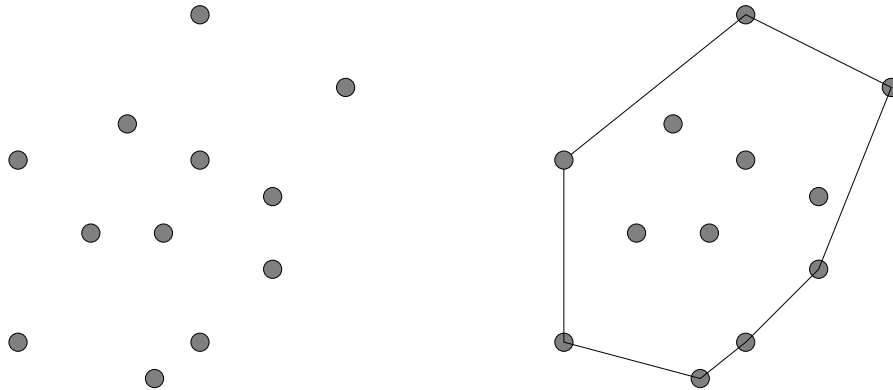


Figure 3: Cluster of points alone and with convex hull.

## 7. CONVEX HULL

Our third example of a randomised algorithm is the construction of a convex hull in the plane. Figure 3 shows some points in the plane and their convex hull (In two dimensions, one might be tempted to use a different term such as shortest enclosing fence).

de Berg et al. describe an application of convex hulls from the oil industry. In general, an oil well outputs a mixture of several components (In our 2-D world, we give name to only two of the components: A and B). The relative ratio of the components varies from well to well, but by mixing oil from different sources, one can usually construct a blend containing the relevant components in the proper ratios. Consider the following concrete problem; 4 oil wells produce A and B in the following ratios: (A: 10%, B: 20%), (A: 10%, B: 30%), (A: 15%, B: 35%) and (A: 20%, B: 35%), respectively. Is it possible from these 4 sources to mix a new oil containing 15% A, and 25% B? – No, it is not possible, since the point (15, 25) lies outside the convex hull of the points (10, 20), (10, 30), (15, 35) and (20, 35) (see Figure 4).

There are quite efficient and simple deterministic methods for constructing convex hulls, but since we have our focus on randomisation, we will present a randomised algorithm. It runs in expected time  $O(n \log n)$  on the worst case input, which we will demonstrate using *backwards analysis*.

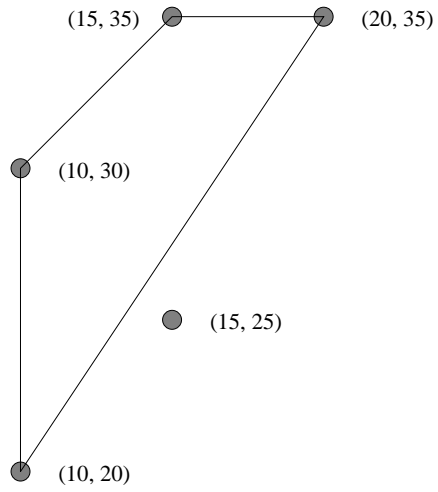


Figure 4: Mixing oil.

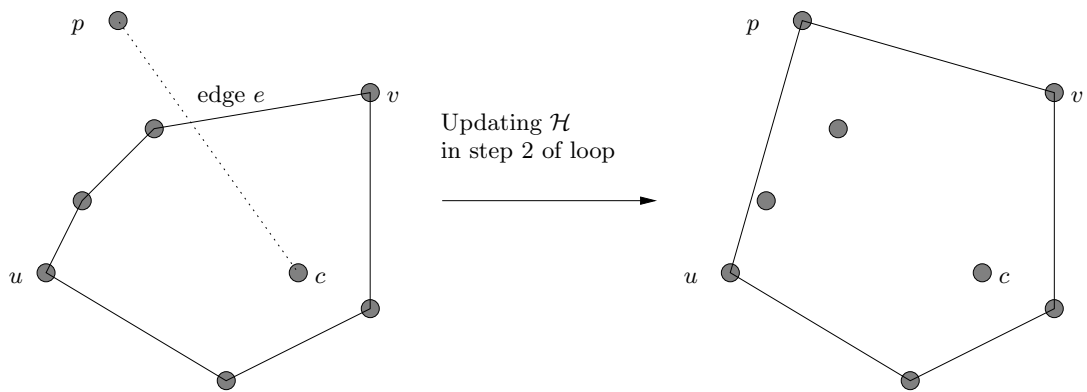


Figure 5: Updating hull after selection of point  $p$ .

Algorithm: Randomised Convex Hull Construction (RH)

Input:  $P = [p_1, \dots, p_n]$ , a list of points in the plane such that  $|P| \geq 3$ .  
Output:  $H$ , a circular list of edges, representing the convex hull of  $P$   
Method: (sketch only)  
(Init:) Let  $\mathcal{P} = [p_1, p_2, p_3]$ .  
Let  $\mathcal{H}$  be the triangle spanned by the points  $[p_1, p_2, p_3]$ .  
Let  $c$  be the center of  $\mathcal{H}$ .  
Compute a mapping  $T$  to satisfy the invariant below.  
(Loop:) Repeat the following steps until  $|\mathcal{P}| = n$  and return  $\mathcal{H}$ .

**Invariant:**

$\mathcal{H}$  is the convex hull of  $\mathcal{P}$ .

$c$  is a point inside  $\mathcal{H}$ .

$T$  is a mapping from  $P$  to  $\mathcal{H} \cup \{\text{"inside"}\}$  defined by (i) if  $p$  lies inside  $\mathcal{H}$  then  $T[p] = \text{"inside"}$  and (ii) if  $p$  lies outside  $\mathcal{H}$  then  $T[p] = e$  for some line  $e \in \mathcal{H}$  crossed by the line from  $c$  to  $p$ . We keep representations of both  $T$  and  $T^{-1}$ .

1. Select  $p$  randomly from  $P - \mathcal{P}$  using the uniform distribution.
2. If  $T[p] = e$  then
  - (i) Find  $e$  in  $\mathcal{H}$ , and follow  $\mathcal{H}$  in both directions from  $e$  to find all edges in  $\mathcal{H}$  that are visible from  $p$ . These edges forms a subpath  $S$  with endpoints  $u$  and  $v$ . Update  $\mathcal{H}$  by replacing  $S$  with the two edges  $(u, p)$  and  $(p, v)$ . (see figure 5)
  - (ii) Update  $T$ , i.e. for all  $f \in S$  and for all  $q \in T^{-1}(f)$  recompute  $T[q]$ .
3. Insert  $p$  into  $\mathcal{P}$ .

## 8. ANALYSING RANDOMISED CONVEX HULL CONSTRUCTION

In our analysis, we will count only the number of changes made to  $T$  in step 2(ii) of the loop that result in a value different from “inside”. Let us briefly argue that everything else takes linear time.

The combined time for initialisation and all visits to steps 1 and 3 of the loop is  $O(n)$ . In step 2(i) of the loop, we may occasionally have to deal with a long subpath  $S$ , but the combined length of all removed subpaths  $S$  is  $\leq 2n$ , since we insert at most 2 edges in each iteration and we start with an  $\mathcal{H}$  of size 3. Therefore all updates of  $\mathcal{H}$  takes total time  $O(n)$ . Clearly,  $T[p]$  can only be changed to “inside” once for each  $p$ , so the number of changes to  $T$  resulting in the value “inside” is also  $O(n)$ .

For notational convenience, let us call an update of  $T$  that changes the value of some  $T[p_i]$  from one non-“inside” value to another non-“inside” value for an *outside*-update.

We will use *backwards* analysis to find an upper bound on the expected number of outside-updates for a fixed input  $P$ . The underlying idea is that the number of outside-updates is the same, whether the algorithm is run forwards or backwards, but it turns out to be simpler to look at the latter case.

What should it mean to run the algorithm backwards? – If the random selection of a point  $p$  in step 1 of a specific iteration of the loop is *independent* of the random selections made in other iterations, then we are in fact adding the points of  $P - [p_1, p_2, p_3]$  to  $\mathcal{P}$  in random order. When running the algorithm backwards we remove the points of  $P - [p_1, p_2, p_3]$  from  $\mathcal{P}$  in random order, while updating  $\mathcal{H}$  and  $T$  to maintain the invariant. The total number of updates, will be the same, whether the algorithm is run forwards or backwards.

To avoid ambiguities, let the  $j$ th iteration of the loop (run forwards or backwards) be the iteration, where  $|\mathcal{P} - [p_1, p_2, p_3]|$  changes between  $(j - 1)$  and  $j$ .

Similar to the earlier analyses, we will use indicator variables

$$Z_{ij} = \begin{cases} 1, & \text{if the value of } T[p_i] \text{ is changed in an outside-update in} \\ & \text{the } j\text{th iteration of the loop.} \\ 0, & \text{otherwise} \end{cases}$$

Let  $Z$  be the random variable, whose value is the total number of outside-updates, when running RH on input  $P$ . We want to find an upper bound on  $\mathbf{E}[Z]$ :

$$\begin{aligned} \mathbf{E}[Z] &= \mathbf{E}\left[\sum_{i=4}^n \sum_{j=1}^{n-3} Z_{ij}\right] \\ &= \sum_{i=4}^n \sum_{j=1}^{n-3} \Pr[Z_{ij} = 1] \end{aligned}$$

We need to compute an upper bound of  $\Pr[Z_{ij} = 1]$ . We may assume that  $p_i$  is outside the hull computed so far, since  $Z_{ij} = 0$  otherwise; i.e. after the  $j$ th iteration  $T[p_i] = e$  for some edge  $e = (q, r)$ . Figure 6 illustrates a typical situation.

If  $Z_{ij} = 1$  then the edge  $e$  must be removed from  $\mathcal{H}$  when running the  $j$ th iteration of the loop backwards. This could happen only if either  $q$  or  $r$  is the random point selected (out of  $j$  possible points in  $\mathcal{P} - [p_1, p_2, p_3]$ ) to be removed in the  $j$ th iteration (run backwards). Therefore

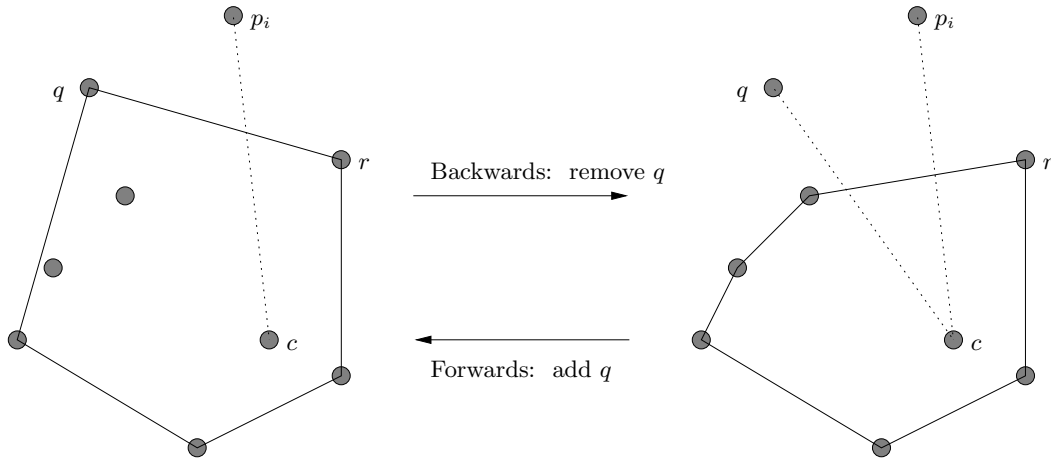


Figure 6: Backwards analysis.

$$\Pr[Z_{ij} = 1] \leq \frac{2}{j}$$

The upper bound on the expected number of outside updates follow from a final calculation

$$\begin{aligned} \mathbf{E}[Z] &= \sum_{i=4}^n \sum_{j=1}^{n-3} \Pr[Z_{ij} = 1] \\ &\leq \sum_{i=4}^n \sum_{j=1}^{n-3} \frac{2}{j} \\ &\leq 2 \sum_{i=4}^n \ln n \\ &\leq 2n \ln n \end{aligned}$$

By our earlier argument, the expected total time usage for the convex hull construction is  $O(n \log n)$ .

## 9. TREAPS

In this section, we introduce the treap (tree-heap), which is the basis of a very elegant randomised implementation of dictionaries. We describe the data structure here, but the analysis will be deferred to problem 2.

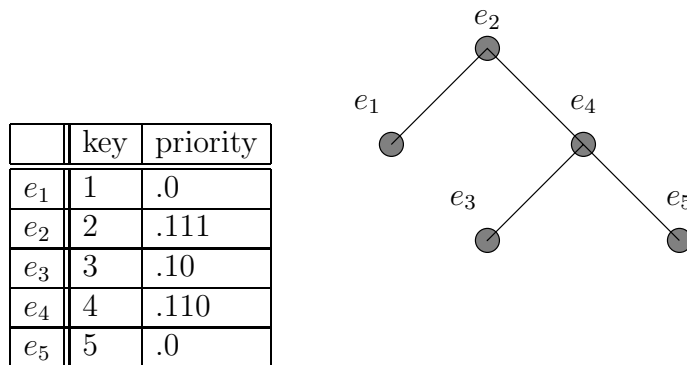


Figure 7: Treap with 5 elements.

A treap is a binary tree, where each node has associated two values, a key and a priority. The nodes are ordered in *in-order* after the keys and in *heap-order* after the priorities, i.e.

$$\text{key}(\text{left-son}) \leq \text{key}(\text{father}) \leq \text{key}(\text{right-son}),$$

and

$$\text{priority}(\text{son}) \leq \text{priority}(\text{father}).$$

Figure 7 shows 5 elements with keys and priorities and the corresponding treap. In fact the treap is unique when all keys are distinct and all priorities are distinct. To see this, observe that the top-priority element, say  $e$ , must be placed in the root. The remaining elements are split in two groups, those elements,  $L_1$  with smaller keys than  $e$ , and those elements,  $L_2$  with larger keys than  $e$ . By induction, there are unique treaps  $T_1$  and  $T_2$  for  $L_1$  and  $L_2$ , respectively. The combined tree with root  $e$ , left subtree  $T_1$  and right subtree  $T_2$  is unique.

In the dictionary problem we assume that the elements already have (distinct) keys, and for each element, we select a random priority from the uniform distribution on the real interval  $[0, 1]$ , and represent the dictionary by the resulting treap.

The treap will be unique, since all selected priorities are distinct with probability 1. It is not practical to represent a priority by its entire (infinite) binary expansion (such as .111000011101011100001101010...). However, a priority is only used for comparison with other priorities, so we need only know a sufficiently long prefix of each priority to check that the heap-order is satisfied. In figure 7 only such minimal prefixes of priorities are given.

When inserting a new element with a given key, we use the in-order on keys to look up the position in the tree, where it fits in as a leaf, and place it there (in the example of figure 7 an element with key 6 would be inserted as a right son of  $e_5$ ).

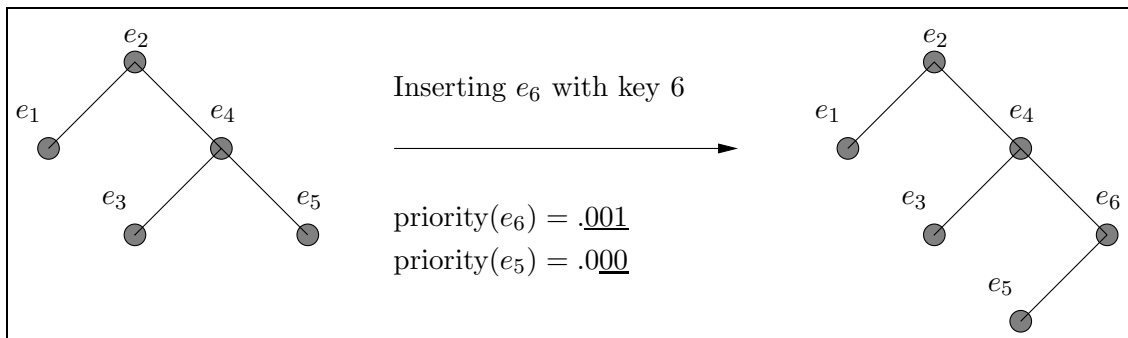


Figure 8: Inserting new element.

Afterwards, we possibly rotate it up in the tree to maintain heap-order, while selecting random bits for expanding priority-prefixes when needed for deciding comparisons (In the example of figure 8 one rotation was made on the basis of 5 new (underlined) bits).

Deletion of an element  $e$  can be understood as running the insertion process backwards: Change the priority of  $e$  to  $-\infty$  and rotate  $e$  down to maintain heap-order.  $e$  will then be a leaf that can be removed without destroying the treap properties.

Observe that since a treap is uniquely determined by the keys and priorities, then it is also transparent to its own history, i.e. the order in which elements are inserted or deleted does not influence the structure of the final treap.

It is left to the reader to prove that all the usual dictionary operations (look-up, insertion, deletion) can be done in expected time  $O(\log n)$  per operation (see problem 2).

## 10. IMPLEMENTING RANDOM SELECTION

So far, we have assumed that randomness is available for free, which may be unrealistic. We will here regard randomness as a resource, measured by the number of independent random bits used and review a typical random selection made in the quicksort algorithm:

“Select a random  $a$  from  $[a_0, a_1, \dots, a_{m-1}]$  using the uniform distribution.” (1)

(for technical convenience, we have shifted indices of  $a_i$ 's one down). If  $m = 2^b$ , then it suffices to draw  $b$  random bits, since the probability of getting any particular sequence of  $b$  bits is  $\frac{1}{2^b} = \frac{1}{m}$ . However, if  $m$  is not a power of 2 then no

fixed number  $b$  of random bits will suffice, since  $\frac{j}{2^b} \neq \frac{1}{m}$ , for all integer  $j$ . In this case we need to let the number of random bits used be a random variable itself!

Assume that the real number  $r$  is selected from the uniform distribution on the interval  $[0; 1]$ .  $r$  determines a unique  $k$  such that

$$\frac{k}{m} \leq r < \frac{k+1}{m} \quad (2)$$

Let the corresponding  $a_k$  be our random selection. Clearly, all  $a \in \{a_0, a_1, \dots, a_{m-1}\}$  have equal probability of being selected. Similarly to the generation of real priorities in the previous section on treaps, we need only generate the bits of a finite prefix of the infinite binary expansion of the real number  $r$  to determine the  $k$  for which (2) is valid.

*Example.* If  $m = 7$  and  $.1010$  is a prefix of the binary expansion of  $r$  then  $r \in [\frac{10}{16}; \frac{11}{16}]$  and since  $\frac{4}{7} < \frac{10}{16} < \frac{11}{16} < \frac{5}{7}$ , we can select  $k = 4$  using only 4 bits of the binary expansion of  $r$ .

Let the random variable  $Y_m$  denote the number of bits in the prefix of  $r$  that are needed to determine  $k$  uniquely. We want to find  $E[Y_m]$ .

After selection of  $t = \lceil \log m \rceil$  bits,  $r$  is confined to an interval  $I$  of length  $2^{-t}$ . If the interior of  $I$  contains no fraction of the form  $\frac{j}{m}$  for any  $j$  then we are done. Otherwise, the interior of  $I$  contains precisely one such fraction (since  $2^{-t} \leq \frac{1}{m}$ ), and by selecting one additional bit in the prefix of  $r$ , we have probability at least  $\frac{1}{2}$  of confining  $r$  to an interval containing no fraction of the form  $\frac{j}{m}$ . We conclude that the number of bits needed in addition to the first  $\lceil \log m \rceil$  bits are bounded by a variable that is geometrically distributed with parameter  $\frac{1}{2}$  and therefore has expectation 2. A more detailed analysis leads to:

$$E[Y_m] = \begin{cases} 0 & m = 1, \\ 1 + E[Y_{\frac{m}{2}}] & m \text{ is even,} \\ \lceil \log m \rceil + 2(\frac{m-1}{2^{\lceil \log m \rceil}}) & m \text{ is odd and } m \geq 3. \end{cases}$$

The expected number of bits used in the random selection (1) is therefore at most  $E[Y_m] \leq 2 + \log m$ .

A similar analysis is valid for the choice of line segments and points in the algorithms for constructing autopartitions and convex hulls, respectively. For the treap implementation of dictionaries, see problem 2.

Having constructed our random selections from truly random bits, the next question is, how do we get the random bits? Existing computers are quite deterministic in nature except for an occasional hardware error. The introduction of

randomness can be quite hard. However, it appears that the need has existed prior to the invention of electronic computers and native people have found their own solutions.

Anthropologists have discovered an example among Naskapi Indians. They roast an animal's shoulder blade to help find wild game: The burnings in the bone take the form of blackened spots and cracks. Imagination suggests the likeness of these marks produced by the heat, to rivers, lakes, mountains, trails and various animals. On this basis a direction for hunting is selected. The anthropologists argue that this may be a randomised algorithm giving the Indians an advantage over the game. Unwitting regularities exhibited by the Indians may provide a basis for anticipatory response from the animals that seek to avoid the hunters. This suggests that there is an advantage in avoiding a fixed pattern when hunting. Because the occurrence of cracks and spots in the shoulder blade and the distribution of game are in all likelihood independent events, it would seem that the practice of being guided by roasted shoulder blades could increase yields when hunting.

## PROBLEMS

**Problem 1.** *An instance of EXACT3SAT is a set of clauses  $C_1, \dots, C_k$ , where each clause  $C_i = (l_{i1} \vee l_{i2} \vee l_{i3})$  is a disjunction of precisely three distinct literals and each literal is a variable or a negated variable  $l_{ij} \in L = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ . A truth assignment is a mapping  $v : L \mapsto \{\text{true}, \text{false}\}$  satisfying that  $v(x_i) = \neg v(\bar{x}_i)$ .*

*Show that for a given instance of EXACT3SAT there is a truth assignment that makes at least  $\frac{7}{8}k$  clauses true.*

**Problem 2.** *Let  $T$  be a treap-based dictionary containing the elements  $[e_1, e_2, \dots, e_n]$ , where  $\text{key}(e_1) \leq \text{key}(e_2) \leq \dots \leq \text{key}(e_n)$ . Let  $e$  be a fixed element. Show that each of the operations  $\text{look-up}(e)$ ,  $\text{insert}(e)$  and  $\text{delete}(e)$  takes expected time  $O(\log n)$ . You may choose to follow the outline suggested here:*

*Define the indicator variables*

$$X_{ij} = \begin{cases} 1, & \text{if } e_j \text{ is the element of largest priority in the key-interval} \\ & \text{spanned by } e_i \text{ and } e_j, \text{ viz. the interval } [e_i, e_{i+1}, \dots, e_j] \text{ if} \\ & i < j \text{ and the interval } [e_j, e_{j+1}, \dots, e_i] \text{ if } j < i. \\ 0, & \text{otherwise} \end{cases}$$

*Let the random variable  $X_i = \sum_{j=1}^n X_{ij}$ .*

*(i) Argue that  $X_i$  is the length of the path from the element  $e_i$  to the root of the treap. Show that  $\mathbf{E}[X_i] \leq 2 \ln n$ .*

Define the indicator variables

$$Y_{ij} = \begin{cases} 1, & \text{if } e_i \text{ is the element of largest priority and } e_j \text{ is the element} \\ & \text{of second-largest priority in the key-interval spanned by} \\ & e_i \text{ and } e_j, \text{ viz. the interval } [e_i, e_{i+1}, \dots, e_j] \text{ if } i < j \text{ and} \\ & \text{the interval } [e_j, e_{j+1}, \dots, e_i] \text{ if } j < i. \\ 0, & \text{otherwise} \end{cases}$$

Let the random variable  $Y_i = \sum_{j=1}^n Y_{ij}$ .

(ii) Argue that  $Y_i$  is the number of rotations made when deleting  $e_i$  from the treap. Show that  $\mathbf{E}[Y_i] \leq 2$ .

(iii) Argue that the expected number of rotations made when inserting an element in a treap with  $n - 1$  elements, is the same as the expected number of rotations made when deleting an element from a treap with  $n$  elements.

(iv) Prove that the expected number of new random bits needed for the lazy generation of priorities when inserting an element is  $O(1)$ .

**Problem 3. [Oyster of the week]** The select algorithm sketched below finds the  $k$ 'th smallest element in a list. Show that the expected number of comparisons made by the algorithm for a fixed worst case input of length  $n$  is at most  $cn$  for a constant  $c = 2 + 2\ln 2 \approx 3.3863$ .

Algorithm: Randomised Selection (RS)

Input:  $L = [a_1, \dots, a_n]$ , a nonempty list of distinct numbers  
 $k$ , an integer such that  $1 \leq k \leq n$ .

Output:  $b$ , where  $b \in L$  and  $|\{a \in L : a \leq b\}| = k$

Method: (sketch only)

1. Select  $e$  randomly from  $L$  using the uniform distribution
2. Split  $L' = L - \{e\}$  into the two sublists  
 $L_1 = [a_i \in L | a_i < e]$   
and  
 $L_2 = [a_i \in L | a_i > e]$   
by comparing  $e$  to each element of  $L'$ .
3. If  $|L_1| = k - 1$  then return  $e$ .  
If  $|L_1| > k - 1$  then make a recursive call on  $L_1$  and  $k$ .  
If  $|L_1| < k - 1$  then make a recursive call on  $L_2$  and  
 $k - 1 - |L_1|$ .

## BIBLIOGRAPHY

This note draws partly on material from some of the following sources. These sources are also recommended for further studies.

1. Mark de Berg, Marc van Kreveld, Mark Overmars and Otfried Schwarzkopf, *Computational geometry. Algorithms and applications*. Springer-Verlag, Berlin, 1997
2. Rajeev Motwani and Prabhakar Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.
3. Ketan Mulmuley, *Computational Geometry. An Introduction through Randomized Algorithms*. Prentice-Hall, 1994.
4. Jeffrey Shallit, Randomised Algorithms in “Primitive” Cultures or What is the Oracle Complexity of a Dead Chicken. *SIGACT News* 23 (4), 1992, and update in *SIGACT News* 24 (1), 1993.
5. Aravind Srinivasan, *The Role of Randomness in Computation*. BRICS NS-95-6.