AN OVERVIEW OF RANDOMISED ALGORITHMS

P Raghavan

Rapporteur: Richard Achmatowicz

An overview of randomised algorithms

Prabhakar Raghavan [*]

September 17, 1996

Abstract

We survey some of the basic ideas in randomised algorithms, at a level suitable for advanced undergraduates. We assume that the reader is familiar with the basics of the analysis of algorithms, and probability theory.

1 Overview

A *randomised algorithm* is one that makes random choices during its execution. As a result its behaviour may vary, even on a fixed input. In designing a randomised algorithm, the goal is to ensure that the algorithm is likely to do well on *every input*. For instance, we may design a randomised algorithm and analysis, showing that its expected running time is $O(n^2)$ on *every input*.

It is important to distinguish this from the *probabilistic analysis of algorithms*. Here the intent is to study the behaviour of an algorithm whose input is chosen from a probability distribution. Any probabilistic statement here is conditioned on the randomly chosen input; thus one may for example infer that an algorithm's running time is $O(n^2)$ (say) on *most inputs*.

There are two principal benefits to randomisation in algorithms: simplicity and speed. For many applications a randomised algorithm is the simplest, or the fastest, or both. For instance, we know of no deterministic algorithm that will determine whether a given integer is prime, in time polynomial in the number of digits in the number (i.e., the length of the input). We do however know of a randomised algorithm that, given any integer, will determine its primality correctly with probability exceeding 1/2 in time polynomial in the number of digits. Subsequent independent repetitions of this primality test can be used to drive the probability of failure down exponentially.

[*]IBM Almaden Research Center, 650 Harry Road, San Jose CA 95120, USA.

The reader is referred to the recent book by the author with R. Motwani [1] for many more details and examples of randomised algorithms.

2 Randomised Quicksort

In this section we study a simple randomised sorting algorithm. The algorithm will always correctly sort a set S of n numbers. We will show that its expected running time (the expectation being over the random choices made by the algorithm) is $O(n \log n)$, although it may on rare occasions take time n^2 to run to completion.

If we could find a member y of S such that half the members of S are smaller than y, then we could use the following scheme. We partition $S \setminus \{y\}$ into two sets S_1 and S_2, where S_1 consists of those elements of S that are smaller than y, and S_2 has the remaining elements. We recursively sort S_1 and S_2, then output the elements of S_1 in ascending order, followed by y, and then the elements of S_2 in ascending order. In particular, if we could find y in cn steps for some constant c, we could partition $S \setminus \{y\}$ into S_1 and S_2 in $n-1$ additional steps by comparing each element of S with y; thus, the total number of steps in our sorting procedure would be given by the recurrence

$$T(n) \leq 2T(n/2) + (c+1)n, \tag{1}$$

where $T(k)$ represents the time taken by this method to sort k numbers on the worst-case input. This recurrence has the solution $T(n) \leq c'n \log n$ for a constant c', as can be verified by direct substitution.

The difficulty with the above scheme in practice is in finding the element y that splits $S \setminus \{y\}$ into two sets S_1 and S_2 of the same size. Examining (1), we notice that the running time of $O(n \log n)$ can be obtained even if S_1 and S_2 are *approximately* the same size — say, if y were to split $S \setminus \{y\}$ such that neither S_1 nor S_2 contained more than $3n/4$ elements. This gives us hope, because we know that every input S contains at least $n/2$ candidate splitters y with this property. How do we quickly find one?

One simple answer is to choose an element of S at random. This does not always ensure a splitter giving a roughly even split. However, it is reasonable to hope that in the recursive algorithm we will be lucky fairly often. The result is an algorithm we call Randomised Quicksort.

Algorithm Randomised Quicksort:

Input: A set of numbers S.

Output: The elements of S sorted in increasing order.

1. Choose an element y uniformly at random from S: every element in S has equal probability of being chosen.
2. By comparing each element of S with y, determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y.
3. Recursively sort S_1 and S_2. Output the sorted version of S_1, followed by y, and then the sorted version of S_2.

We analyse the *expected* number of comparisons in an execution of **Randomised Quicksort**. Note that all the comparisons are performed in Step 2, in which we compare a randomly chosen partitioning element to the remaining elements. For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of *rank* i (the ith smallest element) in the set S. Thus, $S_{(1)}$ denotes the smallest element of S, and $S_{(n)}$ the largest. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution, and the value 0 otherwise. Thus X_{ij} is a count of comparisons between $S_{(i)}$ and $S_{(j)}$, and so the total number of comparisons is $\sum_{i=1}^{n}\sum_{j>i} X_{ij}$. We are interested in the expected number of comparisons, which is clearly

$$\mathbf{E}[\sum_{i=1}^{n}\sum_{j>i} X_{ij}] = \sum_{i=1}^{n}\sum_{j>i} \mathbf{E}[X_{ij}]. \tag{2}$$

This equation uses an important property of expectations called *linearity of expectation*.

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared in an execution. Since X_{ij} only assumes the values 0 and 1,

$$\mathbf{E}[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij}. \tag{3}$$

To facilitate the determination of p_{ij}, we view the execution of **Randomised Quicksort** as a binary tree T each node of which is labeled with a distinct element of S. The root of the tree is labeled with the element y chosen in Step 1, the left sub-tree of y contains the elements in S_1 and the right sub-tree of y contains the elements in S_2. The structures of the two sub-trees are determined recursively by the executions of **Randomised Quicksort** on S_1 and S_2. The root y is compared to the elements in the two sub-trees, but no comparison is performed between an element of the left sub-tree and an element of the right sub-tree. Thus, there is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

The in-order traversal of T will visit the elements of S in a sorted order and this is precisely what the algorithm outputs; in fact, T is a (random) binary search tree. However, for the analysis we are interested in the level-order traversal of the nodes. This is the permutation π obtained by visiting the nodes of T in increasing order of the level numbers, and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at distance exactly i from the root.

To compute p_{ij}, we make two observations.

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i,j\}$, then $S_{(i)}$ will belong to the left sub-tree of $S_{(k)}$ while $S_{(j)}$ belong to the right sub-tree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, when $k \in \{i,j\}$, there is an ancestor-descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by **Randomised Quicksort**.
2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element. Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j-i+1)$.

We have thus established that $p_{ij} = 2/(j-i+1)$. By (2) and (3), the expected number of comparisons is given by

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j>i} p_{ij} &= \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1}\\ &\le \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k}\\ &\le 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k}.\end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the *nth Harmonic number*, defined by $H_n = \sum_{k=1}^{n} 1/k$.

Theorem 2.1 *The expected number of comparisons in an execution of* **Randomised Quicksort** *is at most* $2nH_n$.

Since $H_n \sim \ln n + \Theta(1)$, the expected running time of **Randomised Quicksort** is $\mathrm{O}(n \log n)$.

3 Random walks and a 2-SAT algorithm

We now consider a *random walk* on a connected, undirected graph G with vertex set V. A random walk is a discrete-time stochastic process which has many algorithmic applications. We will study one such simple application here.

In a random walk on a graph G, we envision a particle residing at some vertex of G. At a typical step the particle proceeds from its current position v (a vertex of G) to a neighbour of v chosen uniformly at random. (This is known as the *simple random walk*; more generally, one may consider random walks in which the various neighbours of v are chosen with differing probabilities.) We are interested in questions such as: starting at a vertex u, what is the expected number of steps before the walk first arrives at v? Classical tools from the theory of *Markov chains* enable us to answer such questions. For instance, it is known that when G is a chain linking n vertices, the expected number of steps to walk from one end to the other is $O(n^2)$. It is also known that for any connected graph G, the expected number of steps from any vertex to any other is always $O(n^3)$; this bound cannot be improved in general.

Consider the *satisfiability* problem, in which an instance consists of a set of clauses in conjunctive normal form (CNF). The boolean inputs are called *variables*, which may appear in either uncomplemented or complemented form in a clause. The uncomplemented or complemented variables in a clause are known as *literals* (respectively, *unnegated* and *negated* literals). A clause is said to be satisfied if at least one of the literals in it is TRUE. A solution consists either of an assignment of boolean values to the variables that ensures that every clause is satisfied (such an assignment is known as a *truth assignment*), or a negative answer that it is not possible to assign inputs so as to satisfy all the clauses simultaneously.

The *k-SAT* problem is the special case of the SAT problem in which each clause in the input formula contains exactly k literals. We seek an assignment of (boolean) values to the variables such that all the clauses are satisfied, or an assurance that no such assignment exists. While the k-SAT problem is $\mathcal{NP}$-hard for $k \geq 3$, it is solvable in polynomial time for $k = 1$ or $k = 2$. In this section we present a simple polynomial-time randomised algorithm for solving the 2-SAT problem.

Suppose we start with an arbitrary assignment of values to the literals. As long as there is a clause that is unsatisfied, we modify the current assignment as follows: we choose an arbitrary unsatisfied clause, and pick one of the (two) literals in it uniformly at random; the new assignment is obtained by complementing the value of the chosen literal. After each such step, we check to see if there exists an unsatisfied clause under the current assignment; if not, the algorithm terminates successfully with a satisfying assignment. If there is a satisfying assignment for this instance, how long does it take for this process to discover it?

Given an instance with a satisfying assignment, let us fix our attention on a particular satisfying assignment A, and refer to the values assigned by A to the literals as the "correct values." Let n be the number of variables in an instance. The progress of this algorithm can be represented by a particle moving between the integers $\{0, 1, \ldots, n\}$ on the real line. The position of the particle indicates how many variables in the current solution have the correct values. At each iteration, we complement the current value of one of the literals of some unsatisfied clause, so that the particle's position changes by 1 at each step. In particular, a particle currently at position i, for $0 < i < n$, can only move to positions $i-1$ or $i+1$. A particle at location 0 can only move to 1, and the process terminates either when the particle reaches position n, or it may terminate at some other position with a satisfying assignment other than A.

The crucial observation is the following: in an unsatisfied clause, at least one of the two literals has an incorrect value. With probability at least 1/2 we increase (by one) the number of variables having their correct values. The motion of the particle thus resembles a random walk on the line; we have noted above that the expected number of steps in the random walk from one end of the line to the other is $O(n^2)$.

Theorem 3.1 *The expected number of steps for the above 2-SAT algorithm to find a satisfying assignment is* $O(n^2)$.

4 Tail bounds for randomised algorithms

In the examples above we have bounded the expected running times of two randomised algorithms. Very often we seek stronger performance guarantees; for instance, we might ask whether there is a significant probability that the running time of randomised quicksort exceeds $4n \ln n$ steps. We now consider tools from probability theory that enable to answer such questions, allowing us to assert that the probability that the running time of randomised quicksort exceeds $4n \ln n$ is at most $1/n^2$. We then show how these tools can be used in the design and analysis of randomised algorithms, using as an example a selection algorithm. We begin with the Markov inequality, a fundamental tool we will invoke repeatedly when we develop more sophisticated bounding techniques. Let X be a discrete random variable, and $f(x)$ be any real-valued function. Then the expectation of $f(X)$ is given by

$$\mathbf{E}[f(X)] = \sum_x f(x)\mathbf{Pr}[X = x].$$

Theorem 4.1 (Markov Inequality) *Let Y be a random variable assuming only non-negative values. Then for all $t \in \mathcal{R}^+$,*

$$\mathbf{Pr}[Y \geq t] \leq \frac{\mathbf{E}[Y]}{t}.$$

Equivalently,

$$\mathbf{Pr}[Y \geq k\mathbf{E}[Y]] \leq \frac{1}{k}.$$

Proof: Define a function $f(y)$ by $f(y) = 1$ if $y \geq t$, and 0 otherwise. Then $\mathbf{Pr}[Y \geq t] = \mathbf{E}[f(Y)]$. Since $f(y) \leq y/t$ for all t,

$$\mathbf{E}[f(Y)] \leq \mathbf{E}\left[\frac{Y}{t}\right] = \frac{\mathbf{E}[Y]}{t},$$

and the theorem follows. □

The following generalisation of Markov's inequality underlies its usefulness in deriving stronger bounds.

Lemma 4.2 *Let Y be any random variable, and h any non-negative real function. Show that for all $t \in \mathcal{R}^+$,*

$$\mathbf{Pr}[h(Y) \geq t] \leq \frac{\mathbf{E}[h(Y)]}{t}.$$

The first of these is the Chebyshev bound; we will apply this to the analysis of a simple randomised selection algorithm.

For a random variable X with expectation μ_X, its *variance* σ_X^2 is defined to be $\mathbf{E}[(X - \mu_X)^2]$. The *standard deviation* of X, denoted σ_X, is the positive square root of σ_X^2.

Theorem 4.3 (Chebyshev's Inequality) *Let X be a random variable with expectation μ_X and standard deviation σ_X. Then for any $t \in \mathcal{R}^+$,*

$$\mathbf{Pr}[|X - \mu_X| \geq t\sigma_X] \leq \frac{1}{t^2}.$$

Proof: First, note that

$$\mathbf{Pr}[|X - \mu_X| > t\sigma_X] = \mathbf{Pr}[(X - \mu_X)^2 > t^2\sigma_X^2].$$

The random variable $Y = (X - \mu_X)^2$ has expectation σ_X^2, and applying the Markov inequality to Y bounds this probability from above by $1/t^2$. □

5 Randomised selection

Consider finding the kth smallest number in a given set S of n given numbers. It is known that any deterministic algorithm requires at least $2n$ comparisons to find the median of n given numbers. We now describe a simple algorithm that, on any input, will find the kth smallest element in $n + k + o(n)$ steps with probability $1 - o(n)$.

We assume that the elements of S are all distinct, although it is not very hard to modify the following analysis to allow for multi-sets. Let $r_S(t)$ denote the rank of an element t (the kth smallest element has rank k) and let $S_{(i)}$ denote the ith smallest element of S. We extend the use of this notation to subsets of S as well. Thus we seek to identify $S_{(k)}$.

Algorithm LazySelect:

Input: A set S of n elements from a totally ordered universe, and an integer k in $[1, n]$.

Output: The kth smallest element of S, $S_{(k)}$.

```
1. Pick $n^{3/4}$ elements from $S$, chosen independently and uniformly at random with replacement; call this multi-set of elements $R$.
2. Sort $R$ in $O(n^{3/4} \log n)$ steps using any optimal sorting algorithm.
3. Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$,
   let $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing $a$ and $b$ to every element of $S$, determine $r_S(a)$ and $r_S(b)$.
4. if $k < n^{1/4}$, let $P = \{y \in S \mid y \leq b\}$;
   else if $k > n - n^{1/4}$, let $P = \{y \in S \mid y \geq a\}$;
   else if $k \in [n^{1/4}, n - n^{1/4}]$, let $P = \{y \in S \mid a \leq y \leq b\}$;
   Check whether $S_{(k)} \in P$ and $|P| \leq 4n^{3/4} + 2$. If not, repeat Steps 1-3
   until such a set $P$ is found.
5. By sorting $P$ in $O(|P| \log |P|)$ steps, identify $P_{(k - r_S(a) + 1)}$, which is $S_{(k)}$.
```

In Step 1 we sample with replacement: for instance, if an element s of S is chosen to be in R on the first of our $n^{3/4}$ drawings, the remaining $n^{3/4} - 1$ drawings are all as likely to pick s again as any other element in S. This style of sampling appears to be wasteful, but we employ it here because it keeps our analysis clean. Sampling without replacement would result in a marginally sharper analysis, but in practice this may be slightly harder to implement: throughout the sampling process, we would have to keep track of the elements chosen so far.

Figure 1 illustrates Step 3, where small elements are at the left end of the picture and large ones to the right. Determining (in Step 4) whether $S_{(k)} \in P$ is easy since we know the ranks $r_S(a)$ and $r_S(b)$ and we compare either or both of these to k, depending on which of the three **if** statements in Step 4 we execute. The sorting in Step 5 can be performed in $O(n^{3/4} \log n)$ steps.

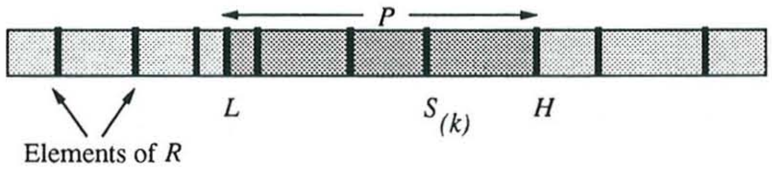

Figure 1: *The* **LazySelect** *algorithm.*

Thus the idea of the algorithm is to identify two elements a and b in S such that both of the following statements hold with high probability:

1. the element $S_{(k)}$ that we seek is in P;
2. the set P of elements between a and b is not very large, so that we can sort P inexpensively in Step 5.

We examine how either of these requirements could fail. We focus on the most interesting case when $k \in [n^{1/4}, n - n^{1/4}]$, so that $P = \{y \in S \mid a \leq y \leq b\}$; the analysis for the other two cases of Step 4 is similar and in fact somewhat simpler.

If the element a is greater than $S_{(k)}$ (or if b is smaller than $S_{(k)}$), we fail because P does not contain $S_{(k)}$. For this to happen, fewer than ℓ of the samples in R should be smaller than $S_{(k)}$ (respectively, at least h of the random samples should be smaller than $S_{(k)}$). We will bound the probability that this happens using the Chebyshev bound.

The second type of failure occurs when P is too big. To study this, we define $k_\ell = \max\{1, k - 2n^{3/4}\}$ and $k_h = \min\{k + 2n^{3/4}, n\}$. To obtain an upper bound on the probability of this kind of failure, we will be pessimistic and say that failure occurs if either $a < S_{(k_\ell)}$ or $b > S_{(k_h)}$. We prove that this is also unlikely, again using the Chebyshev bound. Before we perform this analysis, we establish an important property of independent random variables. Recall the definition of a joint density function $p(x, y)$ for random variables X and Y.

Definition 5.1 *Let X and Y be random variables, and $f(x, y)$ be a function of two real variables. Then,*

$$\mathbf{E}[f(X,Y)] = \sum_{x,y} f(x,y)p(x,y).$$

For independent random variables X and Y

$$\mathbf{E}[XY] = \mathbf{E}[X]\mathbf{E}[Y]. \tag{4}$$

Lemma 5.1 *Let $X_1, X_2, \ldots, X_m$ be independent random variables. Let $X = \sum_{i=1}^{m} X_i$. Then $\sigma_X^2 = \sum_{i=1}^{m} \sigma_{X_i}^2$.*

Proof: Let μ_i denote $\mathbf{E}[X_i]$, and $\mu = \sum_{i=1}^{m} \mu_i$. The variance of X is given by

$$\mathbf{E}[(X-\mu)^2] = \mathbf{E}[(\sum_{i=1}^{m}(X_i - \mu_i))^2].$$

Expanding the latter and using linearity of expectations, we obtain

$$\mathbf{E}[(X-\mu)^2] = \sum_{i=1}^{m} \mathbf{E}[(X_i-\mu_i)^2] + 2\sum_{i<j} \mathbf{E}[(X_i-\mu_i)(X_j-\mu_j)].$$

Since all pairs X_i, X_j are independent, so are the pairs $(X_i - \mu_i), (X_j - \mu_j)$. By (4), each term in the latter summation can be replaced by $\mathbf{E}[(X_i-\mu_i)]\mathbf{E}[(X_j-\mu_j)]$. Since $\mathbf{E}[(X_i-\mu_i)] = \mathbf{E}[X_i] - \mu_i = 0$, the latter summation vanishes. It follows that

$$\mathbf{E}[(X-\mu)^2] = \sum_{i=1}^{m} \mathbf{E}[(X_i-\mu_i)^2] = \sum_{i=1}^{m} \sigma_{X_i}^2.$$

□

We measure the running time of **LazySelect** in terms of the number of comparisons performed by it.

Theorem 5.2 *With probability $1 - O(n^{-1/4})$, **LazySelect** finds $S_{(k)}$ on the first pass through Steps 1-5, and thus performs only $2n + o(n)$ comparisons.*

Proof: The time bound is easily established by examining the algorithm; Step 3 requires $2n$ comparisons, and all other steps perform $o(n)$ comparisons, provided the algorithm finds $S_{(k)}$ on the first pass through Steps 1–5. We now consider the first mode of failure listed above: $a > S_{(k)}$ because fewer than ℓ of the samples in R are less than or equal to $S_{(k)}$ (so that $S_{(k)} \notin P$). Let $X_i = 1$ if the ith random sample is at most $S_{(k)}$, and 0 otherwise; thus $\mathbf{Pr}[X_i = 1] = k/n$, and $\mathbf{Pr}[X_i = 0] = 1 - k/n$. Let $X = \sum_{i=1}^{n^{3/4}} X_i$ be the number of samples of R that are at most $S_{(k)}$. Note that we really do mean the number of samples, and not the number of distinct elements. The random variables X_i are *Bernoulli trials*: each may be thought of as the outcome of a coin toss. Then, using Lemma 5.1 and the variance of a Bernoulli trial with success probability p

$$\mu_X = \frac{kn^{3/4}}{n} = kn^{-1/4},$$

and

$$\sigma_X^2 = n^{3/4}\left(\frac{k}{n}\right)\left(1 - \frac{k}{n}\right) \leq \frac{n^{3/4}}{4}.$$

This implies that $\sigma_X \leq n^{3/8}/2$. Applying the Chebyshev bound to X,

$$\mathbf{Pr}[|X - \mu_X| \geq \sqrt{n}] = \mathbf{Pr}[|X - \mu_X| \geq 2n^{1/8}\sigma_X] = \mathrm{O}\left(n^{-1/4}\right).$$

An essentially identical argument shows that

$$\mathbf{Pr}[b < S_{(k)}] = \mathrm{O}\left(n^{-1/4}\right).$$

Since the probability of the union of events is at most the sum of their probabilities, the probability that either of these events occurs (causing $S_{(k)}$ to lie outside P) is $\mathrm{O}(n^{-1/4})$.

Now for the second mode of failure — that P contains more than $4n^{3/4} + 2$ elements. For this, the analysis is very similar to that above in studying the first mode of failure, with k_ℓ and k_h playing the role of k. The analysis shows that $\mathbf{Pr}[a < S_{(k_\ell)}]$ and $\mathbf{Pr}[b > S_{(k_h)}]$ are both $\mathrm{O}(n^{-1/4})$ (the reader should verify these details). Adding up the probabilities of all of these failure modes, we find that the probability that Steps 1-3 fail to find a suitable set P is $\mathrm{O}(n^{-1/4})$. □

Exercise 5.1 *Theorem 5.2 tells us that the probability that* **LazySelect** *terminates in* $2n + \mathrm{o}(n)$ *steps goes to 1 as* $n \to \infty$. *Suggest a modification in the algorithm that brings the constant in the linear term down to* 1.5 *from* 2.

This adds to the significance of **LazySelect**: the best known deterministic selection algorithms use $3n$ comparisons in the worst case, and are quite complicated to implement. Further, it is known that any deterministic algorithm for finding the median requires at least $2n$ comparisons, so we have a randomised algorithm that is both fast and has an expected number of comparisons that is provably smaller than that of any deterministic algorithm.

Reference

R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

DISCUSSION

Rapporteur: Richard Achmatowicz

Lecture One

A number of questions were raised during the talk. To the question of whether the presented algorithm for randomized quicksort would fail if equal keys were present, Dr Raghavan replied that it was necessary to make some arrangement to deal with them, but that quicksort doesn't change the relative order of equal keys. Dr Raghavan was also asked to whom the presented analysis of the randomized quicksort algorithm was due. He replied that it is very similar to the technique used by Clarkeson and Shore in the analysis of geometric algorithms. He first encountered the analysis in 1984 from Karp. Professor Benson asked whether, given that randomized algorithms are good, was there any advantage in taking 3 random numbers instead of one? Dr Raghavan noted that it was a good point and that he would be talking about that subject later in his talk.

After the talk, the main discussion began.

Dr Andersson asked whether randomized algorithms weren't the opposite of adaptive algorithms, where advantage is taken of the structure of the data? Dr Raghavan answered that there is a trade off between simplicity and adaptability in deciding which approach to use. It also depends on how much analysis is performed on the data.

Professor Randell asked how much of the field has been driven by the cryptography community? How much of the work is public? Dr Raghavan estimated that roughly 1/10 of the published work is cryptography-related.

Professor Randell then asked if Dr Raghavan envisaged the field [of randomized algorithms] growing? He replied in the affirmative, stating that the main attractions are simplicity and/or performance improvements. People often start off using 'quick and dirty' randomized algorithms and then, as they learn more about their data, write a deterministic version. Professor Jones asked if there is empirical evidence to support the conjecture that randomized algorithms are faster than deterministic algorithms? Dr Raghavan stated that there already is a body of experimental evidence showing that they are faster and that as more randomized implementations are created, this assertion gets stronger.

Lecture Two

During the presentation of Yao's MinMax theorem, it was asked whether it is allowed to use algorithms which are incorrect but correct for the given inputs? Dr Raghavan replied that algorithms used could be correct for all inputs except those with zero probability.

After the talk, the main discussion began.

A discussion arose as to why running time of randomized algorithms should be expressed in terms of problem size, as in 'the probability that quicksort's running time exceeds n*n is n*log(n)', and not in absolute terms. As an example, Dr Andersson pointed out that if we said quicksort performed O(n squared) every tenth time, it would be a very bad algorithm.

Professor Nievergelt asked if when comparing the running time of deterministic algorithms to that of probabilistic algorithms, is it fair to compare worst case running time for deterministic algorithms to expected running time of probabilistic algorithms? Wouldn't it be fairer to, for example, use the average running time of deterministic algorithms in comparison? Dr Raghavan stated that the analysis of randomized algorithms always takes into account the worst possible input. To compare with the average running time of deterministic algorithms would favour these. Dr Andersson supported this view.