1 Introduction

A Randomized algorithm is an algorithm where certain decisions are made based on the outcomes of coin flips.

There are two kinds of Randomized algorithms.

1. **Monte-Carlo Algorithms**: A Monte Carlo algorithm runs for a pre-specified amount of time but its output may be incorrect occasionally.

2. **Las Vegas Algorithms**: A Las Vegas Algorithm always comes up with the correct answer. The run time of this algorithm is a Random Variable. We might want a *high probability* bound on this random variable.

By *high probability* we mean a probability of at least $1 - n^{-\alpha}$ where $n$ is the input size and $\alpha$ is a constant $\geq 1$. Here $\alpha$ is a probability parameter and is assumed to be specified by the user.

Randomized algorithms have been proven to be useful in theory as well as in practice. The best known algorithms for several fundamental problems happen to be randomized. Also, randomized algorithms for several problems have been demonstrated to perform better than their deterministic counterparts in practice.

Now we consider two simple problems to demonstrate the power of randomized algorithms.

**Problem 1.1**: Input is an array $a[1 : n] = k_1, k_2, \ldots, k_n$ of arbitrary real numbers. The array is such that it has $n/2$ copies of one element and the other elements are distinct. The problem is to identify the repeated element.
**Deterministic Solutions:** There are many ways to solve this problem deterministically. For example we can sort the array and scan through the array to check if two successive elements have the same value. This algorithm will take $O(n \log n)$ time.

We can devise a linear time algorithm as follows: Partition the input into groups of size three each. Note that by pigeon hole principle, there will be at least one group that has at least two copies of the repeated element. We can thus examine one group at a time to check if it has at least two numbers with the same value.

Any deterministic algorithm to solve this problem will need $\Omega(n)$ time. This is because an adversary who is in-charge of selecting the input and who has perfect knowledge about the algorithm can make sure that the first $n/2$ elements examined by the algorithm have distinct values, thereby forcing the algorithm to examine at least $\frac{n}{2} + 1$ elements.

**A Randomized Solution:** A significantly better randomized algorithm can be devised as follows.

```
repeat
    Basic Step: Flip an $n$ sided coin twice. Let $i$ and $j$ be the outcomes.
    if $i \neq j$ and $a[i] = a[j]$ then output $a[i]$ and quit;
forever
```

Note that the above algorithm is Las Vegas. The run time of this algorithm can be analyzed as follows. The probability $p$ that one execution of **Step 1** results in the identification of the repeated element is $\frac{n/2(n/2-1)}{n^2}$. This is because there are $n/2$ ways of picking the value of $i$ in order to be successful, there are $(n/2 - 1$ ways of picking $j$ so as to be successful, and there are $n^2$ ways of picking the values of $i$ and $j$.

The probability $p$ of success in one basic step is $\geq 1/5$ (for all values of $n \geq 10$). Therefore,
probability of failure in one basic step is \( \leq \frac{4}{5} \). Thus the probability of failure in the first \( k \) successive basic steps is \( \leq (\frac{4}{5})^k \). We want this probability to be very low, i.e., \( \leq n^{-\alpha} \).

Equating \( (\frac{4}{5})^k \) and \( n^{-\alpha} \), we get: \( k = \frac{\alpha \log n}{\log(5/4)} \).

In other words, the randomized algorithm runs in \( \leq \frac{\alpha \log n}{\log(5/4)} \) basic steps with probability \( \geq (1 - n^{-\alpha}) \).

**Definition:** We say that the runtime of a Las Vegas Algorithm is \( \tilde{O}(f(n)) \) if there exist constants \( c \) and \( n_0 \) such that the runtime is \( \leq cf(n) \) with probability \( \geq 1 - n^{-\alpha} \).

Using the above definition, we can state that the run time of the randomized algorithm for the identification of the repeated element is \( \tilde{O}(\log n) \).

**Problem 1.2:** Input is an array \( a[1 : n] \) of arbitrary distinct elements. The problem is to output an array element that is at least as large as the median.

A simple linear time deterministic algorithm for this problem is to find and output the maximum of the \( n \) array elements and output that.

Also, it is easy to see that \( \Omega(n) \) time is needed for any deterministic algorithm in the worst case. In contrast we can devise a simple logarithmic time Monte Carlo algorithm for this problem:

Randomly pick \( f(n) \) elements from the array, for some \( f(n) \) to be fixed in the analysis.

This is sampling with replacement.

Find the maximum of these elements and output it.

**Analysis:** Probability that a randomly picked element is less than the median is \( \leq 1/2 \). The above algorithm outputs an incorrect answer only if all the \( f(n) \) elements sampled are less than
the median. The probability of this happening is $\leq (1/2)^{f(n)}$. We want this probability to be very low. Equating $(1/2)^{f(n)}$ with $n^{-\alpha}$ we get: $f(n) = \alpha \log n$.

In summary, the probability that the above algorithm gives an incorrect answer is very low if $\alpha \log n$ elements are sampled. In this case, the run time of the algorithm is $O(\log n)$.

2 Sorting

Sorting is an important problem in computing and has been studied extensively. Several asymptotically optimal sequential algorithms are available. Parallel algorithms on various architectures have also been devised. Randomization has played a vital role in the design of sorting algorithms. A classical example is the quicksort of Hoare.

2.1 Randomized Quicksort

Let the input sequence to be sorted be $X = k_1, k_2, \ldots, k_n$. The quicksort algorithm chooses a partitioning element, partitions $X$ into two, and sorts the two parts recursively. The partitioning element can be picked in many ways. For example, it could be the first element, the last element, the middle element, etc. The original quicksort algorithm proposed by Hoare was deterministic and traditionally the expected run time of the algorithm is calculated assuming that each input permutation is equally likely.

The idea of using randomness in algorithms has been in vogue for a long time. However the initial attempts have been constrained to computing the expected run times of algorithms assuming a probability distribution on all possible inputs. Such analyses will fail if the assumptions made on the input does not hold. In direct contrast, randomized algorithms incorporate randomness as a part of algorithmic steps. No assumptions are made by randomized algorithms on possible inputs. In particular, the analysis done on randomized algorithms will hold on any input (including the worst case input).
The probability space used for the expected run time analysis of deterministic algorithms is the space of all possible inputs. On the other hand, in the analysis of randomized algorithms, the probability space employed is the space of all possible outcomes for coin flips.

Quicksort can be turned into a Las Vegas algorithm if the partitioning element \( k \) is picked randomly. In this case, we partition \( X \) into \( X_1 \) and \( X_2 \) where \( X_1 = \{ q \in X : q < k \} \) and \( X_2 = \{ q \in X : q > k \} \). The algorithm recursively sorts \( X_1 \) and \( X_2 \) and outputs: sorted(\( X_1 \)), \( k \), sorted(\( X_2 \)).

The expected run time of randomized quicksort can be computed as follows. (Here the expectation is done in the space of all possible outcomes for coin flips). Let \( q_1, q_2, \ldots, q_n \) be the sorted order of the input keys. Define the random variables \( X_{ij} \), for \( 1 \leq i, j \leq n \), where \( X_{ij} = 1 \) if the algorithm compares \( q_i \) and \( q_j \) and \( X_{ij} = 0 \) otherwise.

The expected run time \( A(n) \) is nothing but \( E \left[ \sum_{j>i} \sum_{i=1}^{n} X_{ij} \right] \). Let \( p_{ij} \) be the probability that the algorithm compares \( q_i \) and \( q_j \). Then, \( E \left[ \sum_{j>i} \sum_{i=1}^{n} X_{ij} \right] = \sum_{j>i} \sum_{i=1}^{n} E[X_{ij}] \), \( E[X_{ij}] = 1 \times p_{ij} + 0 \times (1 - p_{ij}) = p_{ij} \). Thus, \( A(n) = \sum_{j>i} \sum_{i=1}^{n} p_{ij} \).

To compute the value of \( p_{ij} \) note that every input key will serve as the partitioning key at some point or the other in the algorithm, as the recursion unwinds. Without loss of generality assume that \( i < j \). Consider the keys \( q_i, q_{i+1}, \ldots, q_j \). Before the algorithm starts, all of these keys are in the same part, namely \( X \). As the algorithm proceeds, these keys will continue to be in the same part until some level of recursion. At some point in time some of these keys will go to one part and the others will go to another part. This splitting happens only when one of these keys is picked as the partitioning key. If \( q_i \) is the first among these to be picked as the partitioning key then \( q_i \) will be compared with each of the keys \( q_{i+1}, q_{i+2}, \ldots, q_j \). If \( q_j \) is the first among these keys to be picked as the partitioning key, than \( q_j \) will be compared with each of \( q_i, q_{i+1}, \ldots, q_{j-1} \). On the other hand consider the case when one of the keys \( q_{i+1}, q_{i+2}, \ldots, q_{j-1} \) is picked as the partitioning key before \( q_i \) or \( q_j \). Let \( q \) be this partitioning key. \( q \) will be compared with every other key in this collection. After this partitioning \( q_i \) and \( q_j \) will go to different parts.
and they will never meet again and hence they will not be compared. In summary, \( q_i \) and \( q_j \) will be compared if and only if either \( q_i \) or \( q_j \) is picked as the partitioning key before any of the keys \( q_{i+1}, q_{i+2}, \ldots, q_{j-1} \).

Since the partitioning key is picked randomly, the probability of \( q_i \) and \( q_j \) being compared is \( p_{ij} = \frac{2}{j-i+1} \). As a result, \( A(n) = \sum_{j=i+1}^{n} \sum_{i=1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \). I.e., \( A(n) = \sum_{i=1}^{n} \left( \frac{2}{i} + \frac{2}{i+1} + \frac{2}{i+2} + \cdots + \frac{2}{n} \right) \leq 2 \sum_{i=1}^{n} \left( 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right) \). Since \( 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = O(\log n) \), we get: \( A(n) = \sum_{i=1}^{n} O(\log n) = O(n \log n) \).

### 2.2 Randomized Sorting Algorithm of Frazer and McKellar

Frazer and McKellar have extended the idea of quicksort and have come up with a randomized algorithm for sorting. Their algorithm is one of the earliest randomized algorithms in the literature.

**Basic Idea:** If \( X = k_1, k_2, \ldots, k_n \) is the given input, the idea is to randomly pick \( s \) elements from \( X \), sort this sample, partition \( X \) using the sample keys as the splitters, and sort the resultant parts.

If \( \ell_1, \ell_2, \ldots, \ell_s \) is the sorted order of the sample keys, we partition \( X \) into \( X_i, 1 \leq i \leq (s + 1) \) as follows: \( X_1 = \{ q \in X : q \leq \ell_1 \} \); \( X_i = \{ q \in X : \ell_{i-1} < q \leq \ell_i \} \), for \( 2 \leq i \leq s \); and \( X_{s+1} = \{ q \in X : q > \ell_s \} \). After partitioning \( X \) in this fashion, we sort each \( X_i \) (\( 1 \leq i \leq (s + 1) \)) independently. Finally, we output sorted\((X_1)\), sorted\((X_2)\), \ldots, sorted\((X_{s+1})\).