CSPC42 - Design and Analysis of Algorithms

ALGOS ASSIGNMENT - 1

Name: Rajneesh Pandey

Roll Number: 106119100

Branch: CSE(B)

Semester: 4

Randomized Algorithms

What is a Randomized Algorithm?

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). And in Karger's algorithm, we randomly pick an edge.

How to analyse Randomized Algorithms?

Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity is O(E). Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case.

On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst-case time complexity. Below are the Two generally helpful aspects in analysis of such algorithms:

- ⇨ Linearity of Expectation
- ⇨ Expected Number of Trials until Success.

For example, Consider below a randomized version of Quicksort:

A Central Pivot is a pivot that divides the array in such a way that one side has at-least 1/4 elements.

```
------------------------------------------------------------
-----
// Sorts an array arr[low..high]
randQuickSort(arr[], low, high)


1. If low >= high, then EXIT.
```

```
2. While pivot 'x' is not a Central Pivot.
  (i)   Choose uniformly at random a number from
[low..high].
        Let the randomly picked number number be x.
  (ii)  Count elements in arr[low..high] that are
smaller
        than arr[x]. Let this count be sc.
  (iii) Count elements in arr[low..high] that are
greater
        than arr[x]. Let this count be gc.
  (iv)  Let n = (high-low+1). If sc >= n/4 and
        gc >= n/4, then x is a central pivot.
3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements
   randQuickSort(arr, low, sc-1)

5. // Recur for greater elements
   randQuickSort(arr, high-gc+1, high)
-----------------------------------------------------
----
```

The important thing in our analysis is, Time Taken by step 2 is O(n).

How many times while loop runs before finding a central pivot?

The probability that the randomly chosen element is central pivot is 1/n.

Therefore, expected number of times the while loop runs is n (See this for details)

Thus, the expected time complexity of step 2 is O(n).

What is overall Time Complexity in Worst Case?

In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst-case height of recursion tree is Log 3/4 n which is O(Log n).

```
-------------------------------------------------------------
----
T(n) < T(n/4) + T(3n/4) + O(n)
T(n) < 2T(3n/4) + O(n)

Solution of above recurrence is O(n Log n)
-------------------------------------------------------------
----
```

Note that the above randomized algorithm is not the best way to implement randomized Quick Sort. The idea here is to simplify the analysis as it is simple to analyse.

Typically, randomized Quick Sort is implemented by randomly picking a pivot (no loop). Or by shuffling array elements. Expected worst case time complexity of this algorithm is also O(N*Log(N)).

Classification

Randomized algorithms are classified in two categories:

1) ***Las Vegas***: These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value. For example, Randomized Quicksort always sorts an input array and expected worst case time complexity of Quicksort is O(N*Log(N)).

2) ***Monte Carlo***: Produce correct or optimum result with some probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity. For example, this implementation of Karger's Algorithm produces minimum cut with probability greater than or equal to 1/(n^2) (n is number of vertices) and has worst case time complexity as O(E).

Another example is Fermat's Method for Primality Testing.

Example to Understand Classification:

```
Consider a binary array where exactly half elements
are 0
and half are 1. The task is to find index of any 1.
```

A Las Vegas algorithm for this task is to keep picking a random element until we find a 1. A Monte Carlo algorithm for the same is to keep picking a random element until we either find 1 or we have tried maximum allowed times say k. The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expect value. The Expected Number of Trials before Success is 2, therefore expected time complexity is O(1).
The Monte Carlo Algorithm finds a 1 with probability [1 – $(1/2)^k$]. Time complexity of Monte Carlo is O(k) which is deterministic.

Applications and Scope:

- Consider a tool that basically does sorting. Let the tool be used by many users and there are few users who always use tool for already sorted array. If the tool uses simple always going to face worst case situation. On the other hand if the tool uses Randomized QuickSort, then there is no user that always gets worst case. Everybody gets expected O(n Log n) time.

- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.
- Number-Theoretic Applications: Primality Testing
- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: Matrix permanent Counting combinatorial structures.
- Parallel and distributed computing: Deadlock avoidance distributed consensus.
- Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
- Derandomization: First devise a randomized algorithm then argue that it can be derandomized to yield a deterministic algorithm.