UNIT - V — ADVANCED ALGORITHMS

PROBABILISTIC AND RANDOMIZED ALGORITHMS

** Deterministic Algorithms - leave nothing to chance.
Running a deterministic algorithm time after time with the same input produces identical results each time.
few of the properties are as follows:

* They are always correct
* They are deterministic - there may be more than one correct output, but the same instance of a problem always produces the same output
* They are always precise - the answer is not given as a range.
* Each of them operates at the same efficiency for the same instance of a problem in different runs.

Ex: Consider the following Example : prime factor of
α4 is 2 - always in deterministic approach.

α4 → [Prime Factor] → 2

** Randomized algorithms (Probabilistic algorithms) fall into four main design categories: randomizations of deterministic algorithms, Monte Carlo algorithms, Las Vegas algorithms & numerical probabilistic algorithms.
Randomization is done to break the connection b/w a particular input & worst-case behavior & thereby homogenize the expected behavior of inputs to the algorithm.

→ These algorithms are Non-deterministic - they can make random but correct decisions: the same algorithm may behave differently when it is applied twice to the same instance of a problem.

→ Not very precise sometimes, overhead due to random generator as we call it many times.

→ Operates at different efficiencies for the same instance of a problem in different runs.

Ex:

24 → Prime factor → 2 or 3

→ Sometimes incorrect: hope we have a high known probability of being correct.

→ Nonterminating sometime: do not produce an answer at all (hope for a low probability for that :)

* All these because randomized algorithms is one that makes random choices during the Execution.

* Why do we need randomized (probabilistic algorithms)

* To help make performance better of algorithms in Worst Cases. & Sometimes we do not have a better
* method than making random Choices.

* RANDOMIZING DETERMINISTIC ALGORITHMS

Randomized algorithms arise from deterministic algorithms by introducing randomness in some of steps of algorithm

Randomized Version of a deterministic algorithm has the most potential of being useful when the average Complexity $A(n)$ of the deterministic algorithm is significantly smaller than the Worst-Case Complexity $W(n)$

Ex: Quicksort - Randomized versions of Quicksort satisfy the ultimate homogenization Condition:

$$B_{exp}(n) = A_{exp}(n) = W_{exp}(n)$$

(Expected Best, Average & Worst Case)

→ Randomizing Quicksort

One way to randomize Quicksort is to make a random choice of a pivot element for the Call to Partition.

Another way to randomize Quicksort is to first randomly permute the elements in a given input list L and then input the permuted list L' to Quicksort.

→ No. of Comparisons by randomized version is same as average number of Comparisons performed by deterministic version of Quicksort.

→ Quicksort Exhibits $O(n^2)$ Worst-Case Complexity, whereas its average Complexity is in $O(n \log n)$. When making a random choice for the first element in RandomizedQuicksort, we assume that each element in the relevant interval has an equal chance of being Chosen.

→ The analysis for the average complexity $A(n)$ of Quicksort given was based on the assumption that each element of the input list $L[0:n-1]$ is equally likely to be chosen as the pivot element. Thus $T_{exp}(L) = A(n)$ for any input list L of size n, so that $W_{exp}(n) = A(n)$.

So consider that we have not eliminated the worst-case $O(n^2)$ behavior in theoretical sense. Indeed, for any input list I of size n, there is a small chance that the random choice of the pivot element is always close to an endpoint of the relevant interval, thereby forcing the algorithm to perform $O(n^2)$ comparisons. However, since Randomized Quicksort makes many random choices of pivot elements, performing $O(n^2)$ comparisons has such a small chance of occuring that for all practical purposes it can be ignored. even for relatively small lists. Thus we can expect number of comparisons performed in any run of Random Quicksort with any given input list L to be very close to $T_{exp}(L)$.

Randomized Quicksort makes $O(n)$ calls to Random, each call assumed to be taking unit time. & overhead due to calling Random is dominated by $\Omega(n \log n)$ comparisons performed by Randomized Quicksort.

→ Randomized Quicksort: based on Stochastic Preconditioning. Input list $I = L[0:n-1]$.

```
procedure Permute(L[0:n-1])
Input: L[0:n-1] (an array of lists elements)
Output: L[0:n-1] (an array of list elements randomly permuted)
for i=0 to n-2 do
    j ← Random(i, n-1)
    interchange (L[i], L[j])
end Permute
```

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called randomized algorithms. Ex: randomized quicksort, we use random numbers to pick next pivot element.

↳ LAS VEGAS - Never outputs incorrect soln, but may not produce soln at all.

↳ MONTE CARLO - produce soln quickly, but may make mistakes (give wrong result)

* LAS VEGAS - Algorithms that use the random input so that they always terminate with the correct answers, but where the Expected running time is finite (not fixed). Unlike a Monte Carlo Algorithm, a Las Vegas algorithm never returns an incorrect answer. But is subject to two types of disasters.

1. It could run arbitrarily long without producing an answer
2. Or it could make a random decision for which there is no recovery (a dead end).

→ The probability that a dead end is reached on any given run is usually quite small, thus, running the algorithm several times significantly increases the probability that the algorithm successfully solves the given problem. (relative to 2nd disaster).

→ Relative to first type of disaster, running beyond a reasonable time usually has such a small probability of occurring that it can safely be ignored in practice.

* Thus, Las Vegas algorithms usually have a high probability of successfully finding the solution to a given problem.

Consider the problem of Searching an Element 'a' in an array of n elements.

Ex: Las-Vegas Search A (A, n)

```
{
    while (true) {
        randomly select an
        Element out of n Elements;
        if (a is found)
            return true;
    }
}
```

Select any Element randomly. Say 'a' if a is found - i.e Compare, then return true.
* Always returns true */

The no. of iterations varies and can be arbitrarily large. Here the iteration is not fixed but always give Correct results.

So, in the above Example, the las Vegas algorithm always outputs the Correct answer, but its running time is a random Variable. (not fixed).

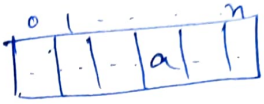
If you want Correct result only and can wait for it then Las Vegas algorithms are preferred.

* MONTE CARLO - Algorithm which have a chance of producing an incorrect result but the running time is fixed. It is a probabilistic algorithm that has a certain probability of returning the Correct answer whatever input is considered. On the other hand, a Las Vegas algorithm never returns an incorrect answer, but it might not return an answer at all. Most useful class of Monte Carlo algorithms are those that have a probability of returning the Correct answer greater than some fixed positive Constant for any input.

Thus, for a fixed real number p, $0 < p < 1$, a p-Correct Monte Carlo Algorithm is a probabilistic algorithm that returns the correct answer with probability not less than p, no matter what input is considered.

Unfortunately, many times there is no efficient method available to test whether an answer returned by a Monte Carlo algorithm for a given input is correct.

Ex: Consider an Searching an Element in an array of n Elements.




```
Monte Carlo SearchA (A, n, x)
{   i=0, tag=false;
    while (i<=x) {
        randomly Select an
        Element out of n Elements;
        i+=1;
        if (a is found)
            tag= true;
    }
}
```

Here we may receive right or wrong answer, but we will finish within a given time.
/* May not always return true */

The Monte Carlo algorithm is guaranteed to Complete in an amount of time that can be bounded by a function the input size and its parameter x, but allows a small probability of Error.

Thus, if you cannot wait and you are okay if the result may be slightly wrong than one can go for Monte Carlo algorithm.