Probabilistic Algorithms

At times we come across problems which may be hard problems (NP hard, NP complete), or problems which become infeasible to compute for a given input size due to the algorithm that is used. For example an $O(n^2)$ algorithm might be pretty good for smaller input sizes, but practically useless if we need to solve the problem for input sizes $\geq 10^6$ regularly.

However, for some problems, we can come up with algorithms which may not guarantee the correct answer, but can guarantee a very high probability of giving the right answer. These types of algorithms can provide a huge improvement over the deterministic algorithms for the problem, and can have huge applications provided the probability for failure is sufficiently low.

However, since there is a small probability of the algorithm failing, people with a malicious intent could take a look at the algorithm, and construct a case for which the algorithm would fail, and the malicious user could take advantage of this failure. This is when randomization comes into play

Randomization

By introducing randomization into a probabilistic algorithm, it becomes significantly more tough to break an algorithm, as it's hard to construct a case while predicting what the random number would generate next.

Example of the usefulness of Randomization

Say we have a list of size n where half of them are $1s$ and the other half of them to be $0s$.

If the problem we have at hand is to find the index of any 0 that occurs in the list, then we could simply try solving this problem by going through the list.

```
for i in binary_string:
    if i == 0:
        return current_index
    else
        continue
```

This algorithm resembles the Geometric distribution, number of failures till a success. Since the probability of getting a 0 is 50%, we can say that the expected number of iterations, and the variance will be

$$E(iterations) = 1/p = 1/0.5 = 2$$

$$Variance = \frac{1-p}{p^2} = \frac{0.5}{0.5^2} = 2$$

Hence, we can see that the algorithm we used should terminate quite quickly. However, we're making one very important assumption while calculating these values, which is that we are assuming that the input is truly random.

If we were told that the strings given will be truly random, then we can safely say that our algorithm should terminate quickly.

However, if we aren't given this guarantee, there's nothing to stop the user from entering malicious input to *break* our algorithm.

For example, the user may enter a list of size n, with the first $n/2$ bits as 1, and the next $n/2$ bits as 0. This would cause our algorithm to iterate over half of the entire list before terminating.

Even if we tried to modify the algorithm, it could still be broken based on the modifications we make. Checking the even indexes first, then the odd indexes, iterating from both the first element and the last element, all these algorithms can be broken, as the behavior of the algorithm is highly predictable given an input.

Hence, if we aren't given the guarantee that the input is random, we could make use of randomization to end up using the same properties as if the input was random.

Therefore, instead of iterating from the start and going through the entire list with some predetermined order, we could pick random indexes, making it nearly impossible to construct a malicious input to break the algorithm.

```
while true:
    pick a random element in list, call it i
    if i == 0:
        return curr_index
    else
        continue
```

By introducing randomness, we've made the probability of the algorithm running too long very less, even in the instance of a malicious user attempting to break the algorithm.

Running Time

We have an estimate on the average number of times that the algorithm would run. However, it would be more descriptive and informational to give the probability of the algorithm running in some specified time.

If we say that finding the first 0 took x attempts, we can say that the answers to our queries would've looked something like this $1111\ldots0$. Hence, the probability of getting this exact configuration is $1/2^x$.

We can also define the probability of not getting a single 0 in y attempts to be $1/2^y$. Using this, we can define the probability of finding a 0 in y tries to be

$1 - 1/2^y$.

Hence if we only have 3 attempts to find a 0, then the probability of succeeding will be $7/8 = 0.875$.

The probability of finding a 0 skyrockets to ≈ 0.996 when we increase the number of attempts to 8. Hence we can see that although the algorithm doesn't have a fixed running time, it has a very high probability of running extremely quickly.

Is the random we use truly random?

While the random values generated by library functions in various languages seem to be quite random, computers practically cannot generate truly random values. Most languages implement their random value generating functions using a *Linear Congruential Generator,* which uses an initial seed. It can be observed that if we tried running the `rand()` function in C++, we end up getting the same values every time we execute it. Hence, it's recommended to provide the current time in seconds as the seed, by doing `srand(time(NULL))` hence, making it tougher to predict the values, and this is usually sufficient to get "random" values.

However, based on the determination and passion of the malicious user, even this may not be sufficient, and may still be breakable. Since the seed only changes every second, the malicious user may attempt to run the algorithm at a certain time at which he may have calculated the values that would be generated with such a seed.

Obviously this may not work every time as the user may not be able to get the algorithm to run at the exact time he pleases, but if the user wishes to, he could definitely break it within a few attempts, as he's bound to get an execution at the second he wishes eventually

Therefore, the most safe option for generating random numbers is using the number of milliseconds as the seed for the random number generator which can be found using `std::chrono::system_clock::now().time_since_epoch() / std::chrono::milliseconds(1);`, as predicting the millisecond count at the time of execution is nearly impossible.