**Randomized Algorithms**

Table of Contents

[Monte Carlo Algorithms 3](#_Toc63002658)

[Las Vegas Algorithms 4](#_Toc63002659)

[Atlantic City Algorithms 5](#_Toc63002660)

[Frievald’s Algorithm 6](#_Toc63002661)

[Increasing Accuracy 7](#_Toc63002662)

[Process 7](#_Toc63002663)

[Time Complexity 9](#_Toc63002664)

[Correctness Analysis 10](#_Toc63002665)

[Quick Sort 13](#_Toc63002666)

[Basic Quick Sort Algorithm 13](#_Toc63002667)

[Variant Quick Sort Algorithms 14](#_Toc63002668)

[Paranoid Quick Sort 15](#_Toc63002669)

A randomized, or probabilistic, algorithm, is one in which the general process is to select a random number or a set of random numbers, , and make some decision based on the value of . These random numbers can be any real numbers.

For example, we could have a recursive algorithm, and in each level of recursion we could generate a random number, , and perform some activities based on it.

There are three sub-categories of randomized algorithms, Monte Carlo algorithms, Las Vegas algorithms and Atlantic City algorithms.

## Monte Carlo Algorithms

Monte Carlo algorithms, or “Probably Correct” algorithms, are ones that, for the same input, on different executions, give different outputs. This means that sometimes, the outputs will be wrong. We need to be able to handle these incorrect outputs. This is done by executing the algorithm multiple times to reduce the probability of incorrect outputs.

For these scenarios, the runtime is said to be deterministic, meaning they are fixed, even though the outputs may vary.

One example of a Monte Carlo algorithm can be verifying a matrix multiplication. Say we are given three matrices, , and , and we need to check if . We could check for some random elements of and give a decision based on that. Of course, there is a probability that the decision we give is wrong, since we might miss all the incorrect elements.

## Las Vegas Algorithms

For Las Vegas algorithms, or “Probably Fast” algorithms, for the same input, on different executions, the number of steps in the execution vary.

Here, understandably, we cannot get the exact runtime. We can only get the expected runtime. For example, if we have an algorithm that is expected polynomial time, it means that, on average, the algorithm will run in polynomial time.

The good thing about these algorithms is that the output will definitely be correct.

An example of a Las Vegas algorithm is a sorting algorithm. For example, in Quick Sort, we need to pick a partition. Based on which partition we pick, we could end up with more or fewer steps in the algorithm. However, in the end, we will definitely have a sorted algorithm.

## Atlantic City Algorithms

Atlantic City algorithms are both “Probably Correct” and “Probably Fast”, meaning both the output and the number of steps can vary.

## Frievald’s Algorithm

Frievald’s algorithm is a Monte Carlo algorithm to verify matrix multiplications. Say we are given three matrices, , and , and we have to verify if . We also need to ensure that the verification has a certain probability of being correct. For example, we could say that we are certain that the outcome is correct.

The brute force approach to verifying a matrix multiplication is to simply do the multiplication. For matrices, this has a time complexity of . Alternatively, we have previously seen Strassen’s algorithm, which had a time complexity of . There are several other methods as well

* Coppersmith-Winograd -
* Andrew Stothers -
* Le Gall -

However, we do not want any of these methods. This is because we do not actually have to find the result. We already have the result and we just want to verify it. As such, we want a faster method. In fact, the method we will see now has a time complexity of .

### Increasing Accuracy

The algorithm has two possible scenarios in terms of output:

* If , , meaning there are no false negatives. If the matrix multiplication is correct, the algorithm will definitely identify it.
* If , , meaning there are false positives in fifty percent of the cases where the matrix multiplication is incorrect.

This may seem like a bad algorithm, but we will be running it times to ensure the output is correct. Each of the runs will be independent, and we will ensure this by using independent randomizations. As a result of this, the probability of getting a false positive will be multiplied times, meaning it will become .

Say we want the positive outcome to be correct more than of the time. Thus, and , so we need to run the algorithm times. Of course, the negative outcome will always be correct.

### Process

The first step is to choose a binary vector, .

A binary vector is a vector of size in which each of the elements is either a or a . For each element, the probability of it being a or a is each, meaning there is no bias.

The reason we choose a binary vector and not just a vector of random numbers is because analysing the runtime complexity for the latter case will become rather difficult.

The next step is to check if .

Essentially, we are multiplying the matrix with the column vector , and then multiplying the result with the matrix . This will give us a single column. If this column matches with the result of multiplying the matrix with the vector , which produces anther column, then the algorithm gives the output YES. Otherwise, the algorithm gives the output NO.

Example

Say , and . The algorithm will create a random binary vector, . Say .

Thus, our algorithm will say that the multiplication is correct. However, in reality, . Thus, the algorithm was wrong in this case.

However, say we run the algorithm again, this time for .

This time, we managed to detect that the multiplication is wrong.

For both runs, we had to perform the same number of steps, but we got different outputs.

### Time Complexity

The time complexity for the algorithm can be analysed by checking the number of multiplications we are having to perform.

First, we multiple and , an matrix and an matrix. This requires multiplications and the result is an matrix.

Next, we multiply and the previous result, another matrix and an matrix. Again, this required multiplications and the result was an matrix.

Finally, we multiply and . This is the same as the multiplication for and , so this also required multiplications and gives us an matrix.

Thus, the overall time complexity is .

### Correctness Analysis

Our main claim is that, if , then .

Say .

Using associativity, we can say

Since ,

Thus, regardless of the value of , if the multiplication is correct, the result will be . Thus, .

Now to move on to the more complex claim. We said that if , then .

Let , where . Thus, we need to prove that for any .

A bad choice for the value of would fail to detect an incorrect multiplication. This means, even though , will be found. This is a false positive.

For some and , , an element in the matrix , say the value is non-zero. To be able to detect this, we need the -th element of needs to be .

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |
|  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|
|  |  | | |  |  |  |  |  |  |

If the -th element of is , then changing just this one value will turn this into a good choice, since it would be able to detect the error. Even if there are more than errors that the original failed to detect, changing just this one value would allow us to detect one of the errors, which is enough.

This means that if there are choices for that are unable to detect the errors, each of those choices have a corresponding choice of that is different in just the -th element and is thus able to detect the error. Thus, we have at least as many choices for that detect the errors as we have for values of that are unable to detect the errors.

Furthermore, say we change multiple elements from the original bad choice of , including the -th element. Each permutation of changes gives us a different value of , all of which can detect the errors due to the -th element being . Thus, for each bad choice of , we actually have multiple good choices of .

All of this just means that more than half of the choices we make for are good choices. This proves our claim.

Alternatively, the following proof may make sense.

The probability of the single -th element that would have detected a single error being a is . If there are elements that are and fail to detect errors, the probability of all of those entries being is . Thus, the probability of failing to detect errors is at most.

## Quick Sort

There are several sorting algorithms, some of which are:

* Insertion Sort and Selection Sort – Both have a time complexity of
* Radix Sort – This has not been covered. It has a time complexity of , but it only works with numbers, not objects like strings.
* Merge Sort – This has a time complexity of , but it is not in-place. This means it requires an additional array to be created in the process, and it has a space complexity of in the merge step.
* Quick Sort – This is a divide and conquer algorithm and is also an in-place algorithm. It has a time complexity of .

### Basic Quick Sort Algorithm

In quick sort, we pick an element, , from the array, called the pivot element. Then we divide the array into two parts. One part contains every element smaller than and the other part contains every element larger than . Finally, the function is recursively called on both of those parts. The combination step is trivial.

The above algorithm is a basic quick sort. Here, we choose either the first or the last element of the array as the pivot. The partition takes time since we need to check every element in the array to see which subarray to put it in.

For a basic quick sort, in the worst case, the array is sorted is descending order. Thus, if we pick the last element as the pivot, all other elements will be put in the right subarray and the left subarray will be empty. Thus,

Thus, in the worst case, the time complexity for a basic quick sort is . However, for a random set of inputs, the time complexity is . The time complexity only occurs in that specific case.

To get around the worst-case time complexity problem, some may choose to shuffle the input array. Regardless of what the array is, it is shuffled so that the inputs are random and the time complexity can be maintained.

### Variant Quick Sort Algorithms

One variant of the quick sort algorithm picks the median of the array as the pivot. This guarantees a balanced partition, since half the points go to one subarray and half the points go to the other.

It is possible to find the median of an array in time. The way this is done is not being covered here. Thus, the total time complexity becomes

Another variant is to pick the pivot randomly. This also gives a time complexity of , but the proof of this is not being covered here.

### Paranoid Quick Sort

Paranoid quick sort is one variant of the quick sort algorithm. Here, we try to find a balanced partition in a probabilistic manner. There are a few more steps, so the time complexity is affected by a small amount.

The paranoid quick sort algorithm is a Las Vegas algorithm, meaning the number of steps might vary on each run of the algorithm.

#### Algorithm

We pick a random pivot at first and perform the portioning. However, we then check the size of the partitions. Unless and until both partitions are smaller than ¾ths of the original array, i.e. and , we will pick another random pivot and try again. Once this condition is met, only then do we recursively call the function on the two subarrays.

#### Time Complexity Analysis

For analysis purposes, let’s say we have a sorted array. In this case, if we pick any of the elements in the first or last ¼ths of the array as the pivot, we will end up with unbalanced partitions. Let’s classify all such pivots as bad pivots. Instead, if we pick any of the elements from the middle ½ of the array, we will end up with a balanced partition. Let’s call these pivots good pivots. In the best case, we will not pick just a good pivot, but the exact median pivot.

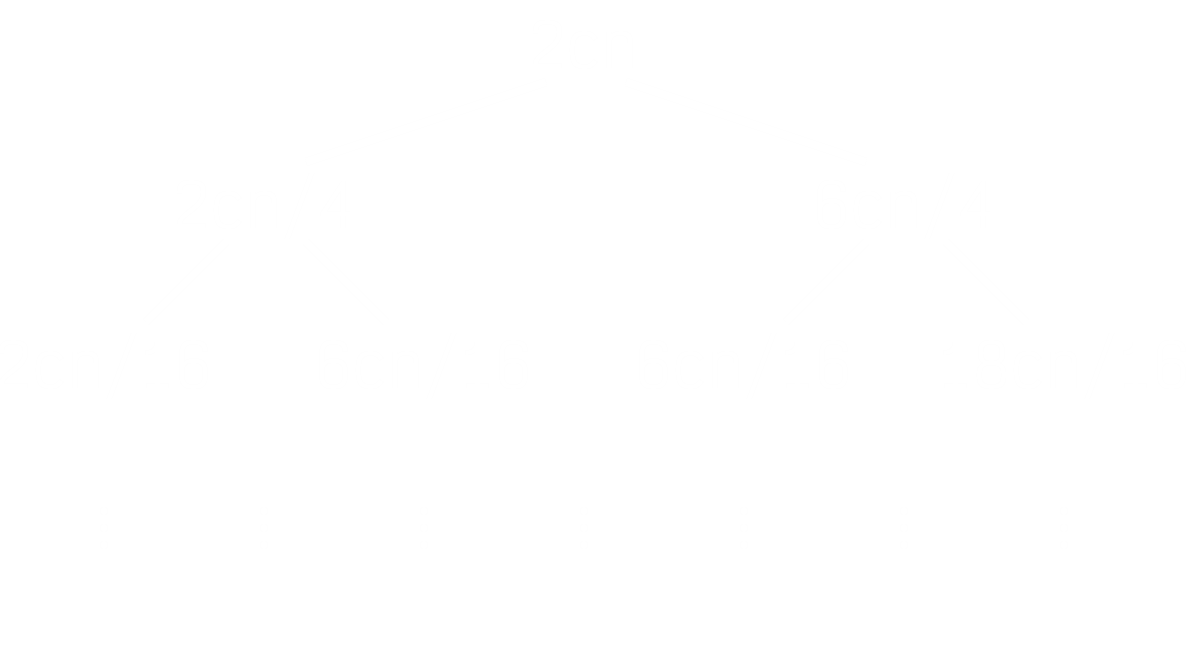
|  |  |  |
| --- | --- | --- |
| Bad | Good | Bad |
|  |  |  |

The probability of picking a good pivot is , since half the elements are good pivots. We will be repeating this step until we get a good pivot. In other words, we will be repeating the step until we get a success. Thus, this is a geometric distribution. The expected value of a geometric distribution is given by

This means that we will only need to repeat the process two times to find a good pivot. Compare this to the time complexity of finding the median element.

In the worst case, the good pivot we picked was just barely good and the two partitions we got are ¾ths and ¼ths the size of the original array respectively. Thus,

Here, the tells us that the time complexity at this level is some constant multiple of . Additionally, we now know the expected repeats to be . Thus, the time complexity at just the first level is . The rest of the time complexity can be calculated using a recursion tree.



For this tree, the left side of every node is being divided by but the right side is being divided by . Because of this difference, the complete tree will be crooked, with more levels on the right side than on the left.

On the left side, the maximum number of times we can divide by before we end up with single elements is , On the right side, this value is .

At each level, the work done is in total. If we think pessimistically and consider the longer, right half of the tree, the time complexity becomes

We can ignore the and the , since we are looking for the asymptotic time complexity, which makes the time complexity .

In reality, the run time will be some constant multiple of , but this constant multiple is probably not that bad, as we saw with the expected value. Thus, this is a probably fast, or Las Vegas, algorithm.