# [COMPLEXITY & ALGORITHMS 1.1

[Permutation algorithms]

DECEMBER 4, 2019

MICHAEL CORNELISSE (459466) LARS WIEGERS (468477)

# Table of Contents

**Algorithm 1.1** ..... 2
**Analysis Algorithm 1.1** ..... 2
**Implementation** ..... 2
**Achieved results** ..... 2
**Figure 0.1** ..... 2
**Averages algorithm 1.1** ..... 3
**Algorithm 1.2** ..... 4
**Analysis Algorithm 1.2** ..... 4
**Implementation** ..... 4
**Achieved results** ..... 4
**Figure 0.2** ..... 5
**Averages algorithm 1.2** ..... 5
**Algorithm 1.3** ..... 6
**Analysis Algorithm 1.3** ..... 6
**Implementation** ..... 6
**Achieved results** ..... 6
**Figure 0.3** ..... 6
**Averages algorithm 1.3** ..... 7

# Algorithm 1.1

## Analysis Algorithm 1.1

During analysis of the randomPerm algorithm we noticed at least 2 for loops and a random part of the algorithm. Because we could not quantify how many times the random part was gonna run we initially skipped it as part of our Big O notation. Which would make the Big O come out to be O(n2). After discussing this with our teacher (René) we found out that for such a random part of the algorithm it is wise to just add a N for it, which would make the Big O come out to be O(n3).

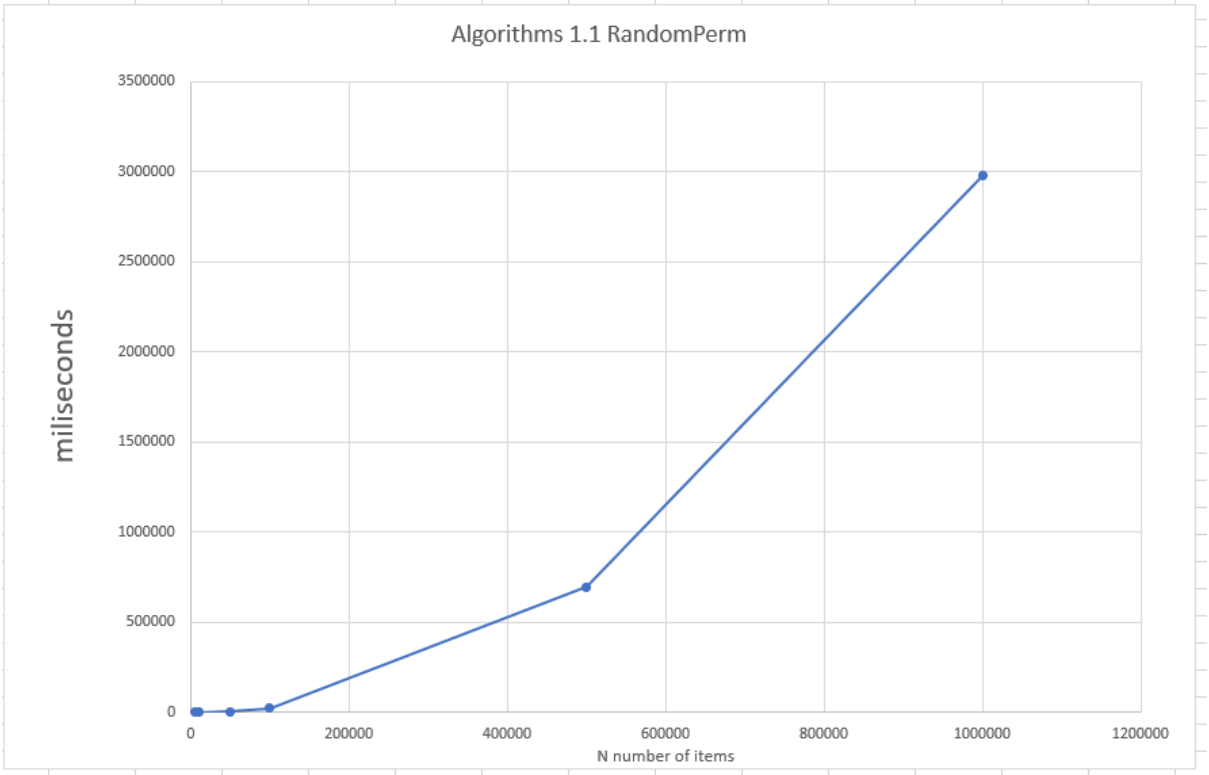
## Implementation

Analysis Algorithm 1 (randomPerm)
During analysis of the randomPerm algorithm we noticed at least 2 for loops and a random part of the algorithm. Because we could not quantify how many times the random part was going to run, we initially skipped it as part of our Big O notation. Which would make the Big O come out to be O(n2). After discussing this with our teacher (René) we found out that for such a random part of the algorithm it is wise to just add a N for it. Which would make the Big O come out to be O(n3).

## Achieved results

In figure 0.1 you see the results after running the randomPerm algorithm, mind that the results stop after the 1000000 N. The reason for this is that we stopped the algorithm after that iteration, is that we believed at that time that with the already achieved results that the algorithm was at least close to O(N^3).

Figure 0.1



## Averages algorithm 1.1

| | Average time algorithm (ms) algorithm 1.1 |
|---|---|
| 5000 | 54 |
| 10000 | 219 |
| 50000 | 5591 |
| 100000 | 25166 |
| 500000 | 695742 |
| 1000000 | 29755283 |
| 5000000 | x |
| 10000000 | x |
| 50000000 | x |
| 100000000 | x |

# Algorithm 1.2

## Analysis Algorithm 1.2

During analysis of the usedPerm algorithm we noticed at least 1 for loop and a random part of the algorithm. We know from the analysis of algorithm 1 that the random part of the algorithm is considered to be n iterations, knowing that can now see that algorithm 2 has a big O notation of O(n2).

## Implementation

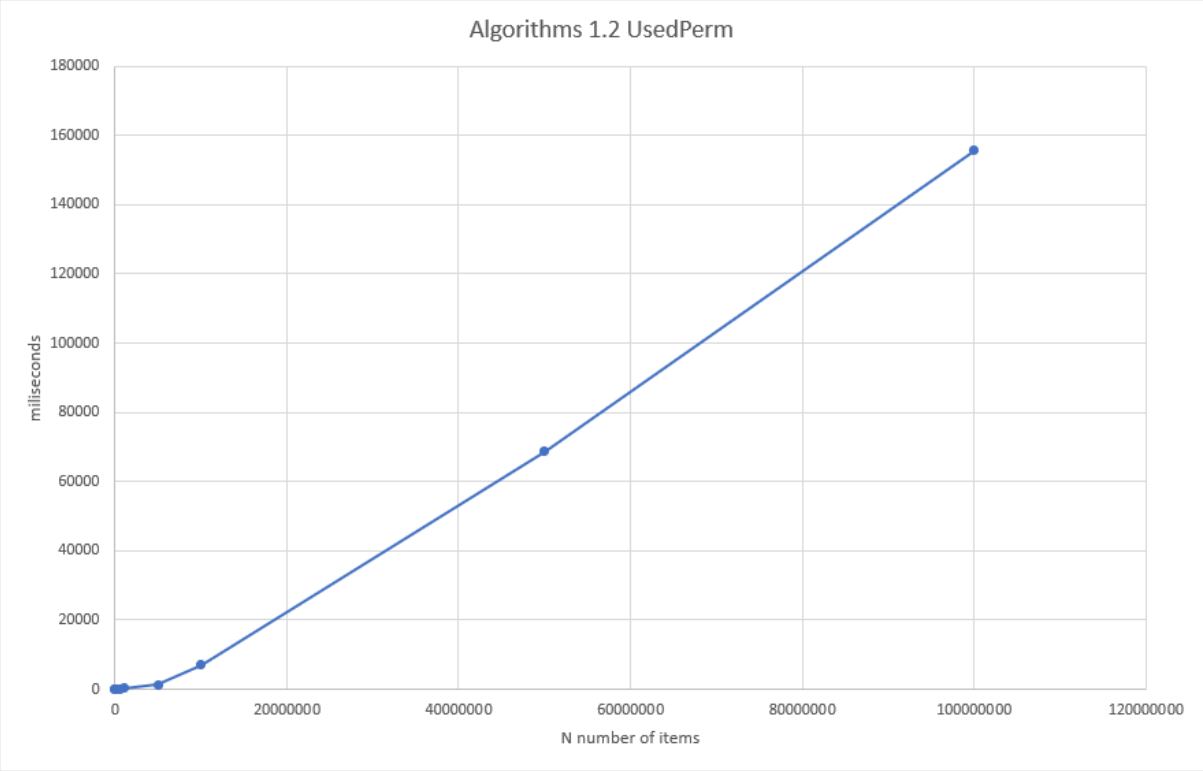
Algorithm 2 (usedPerm)
The design of the second algorithm is quite close to the randomPerm algorithm.
However, in this case we have implemented a second array with Boolean values for comparison whether a number was already present in the generated array.
This allowed us to get rid of the nested for loop inside the while loop and make a comparison for validation. However, this algorithm suffers from the same flaw as algorithm one, the unpredictable nature of the while loop, and the amount of iterations required for finishing the algorithm although we estimated that at worst the algorithm would be O(N^2).

The best case could be O(N) if the correct number is pulled at the first while loop iteration each time. But we deem the chance of this happening highly unlikely so we concluded that the average case generally would be O(N^2).

## Achieved results

In figure 0.2 you see the results after running the usedPerm algorithm, we ran this algorithm for the full range of numbers up until 100000000 N. Overall this algorithm is a lot faster when compared to the randomPerm algorithm. In comparison the second algorithm only makes use of one for loop and a nested while loop, opposed to randomPerms for loop nested while loop with nested for loop structure. Instead we keep a secondary array with Boolean values which we simply flag to true when the element does not exist yet in the array. We estimate that the running time of algorithm 2 is O(N^2).

## Figure 0.2




## Averages algorithm 1.2

| | Average time algorithm (ms) algorithm 1.2 |
|---|---|
| 5000 | 1 |
| 10000 | 1 |
| 50000 | 6 |
| 100000 | 14 |
| 500000 | 104 |
| 1000000 | 216 |
| 5000000 | 1360 |
| 10000000 | 6976 |
| 50000000 | 68692 |
| 100000000 | 155594 |

# Algorithm 1.3

## Analysis Algorithm 1.3

During analysis of the swapPerm algorithm we noticed at least 1 for loop.
Which would make the big O notation O(n). Which is also what we ended up implementing.
Seeing as this is the smallest Big O notation, we also predict that the achieved the lowest results of all the given algorithms.

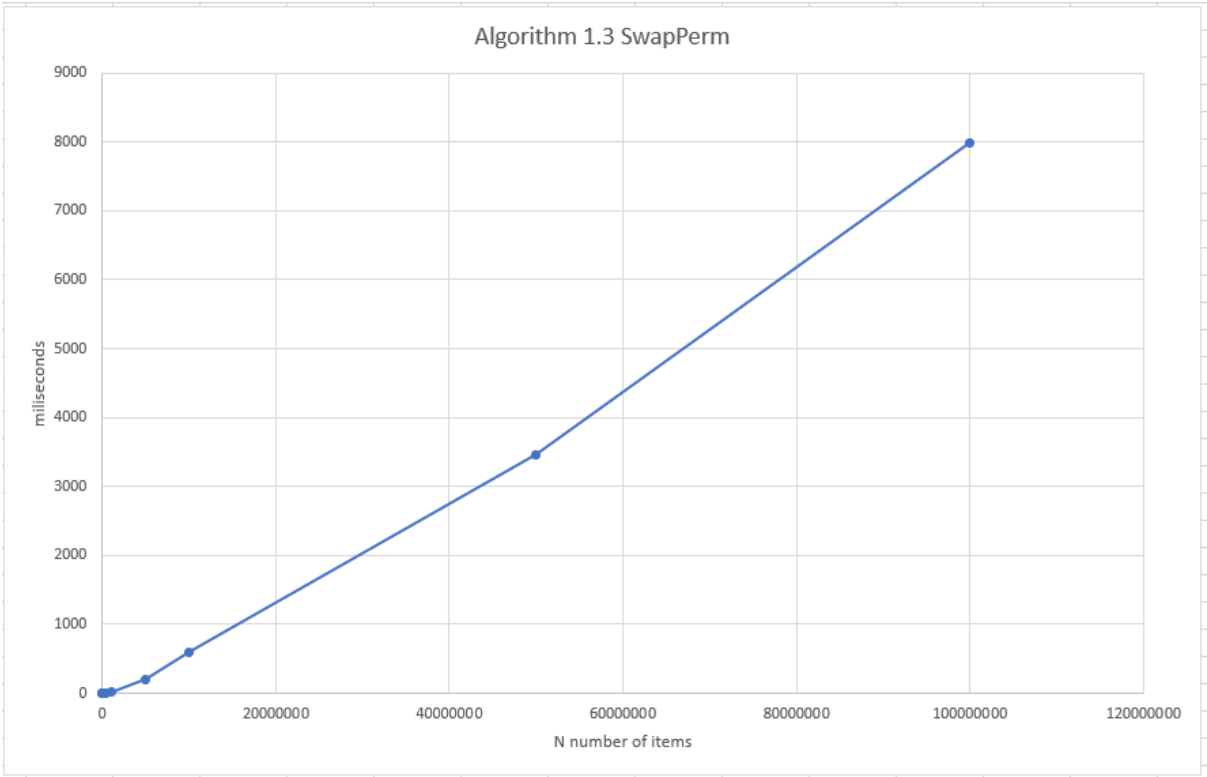
## Implementation

Algorithm 3 (swapPerm)
This algorithm only uses a single for loop to iterate over the array, we make use of a temp container to swap the indexes.

## Achieved results

In figure 0.3 you see the results after running the swapPerm algorithm. We ran this algorithm for the full of numbers up until 100000000 N. As expected this was the fastest algorithm of the three, thanks to the use of a single for loop. The running time is O(N).

### Figure 0.3



## Averages algorithm 1.3

| | Average time algorithm (ms) algorithm 1.3 |
|---|---|
| 5000 | 0 |
| 10000 | 0 |
| 50000 | 0 |
| 100000 | 1 |
| 500000 | 7 |
| 1000000 | 20 |
| 5000000 | 205 |
| 10000000 | 598 |
| 50000000 | 3465 |
| 100000000 | 7989 |