N. Onur Vural, 21902330, CS 201 SEC1

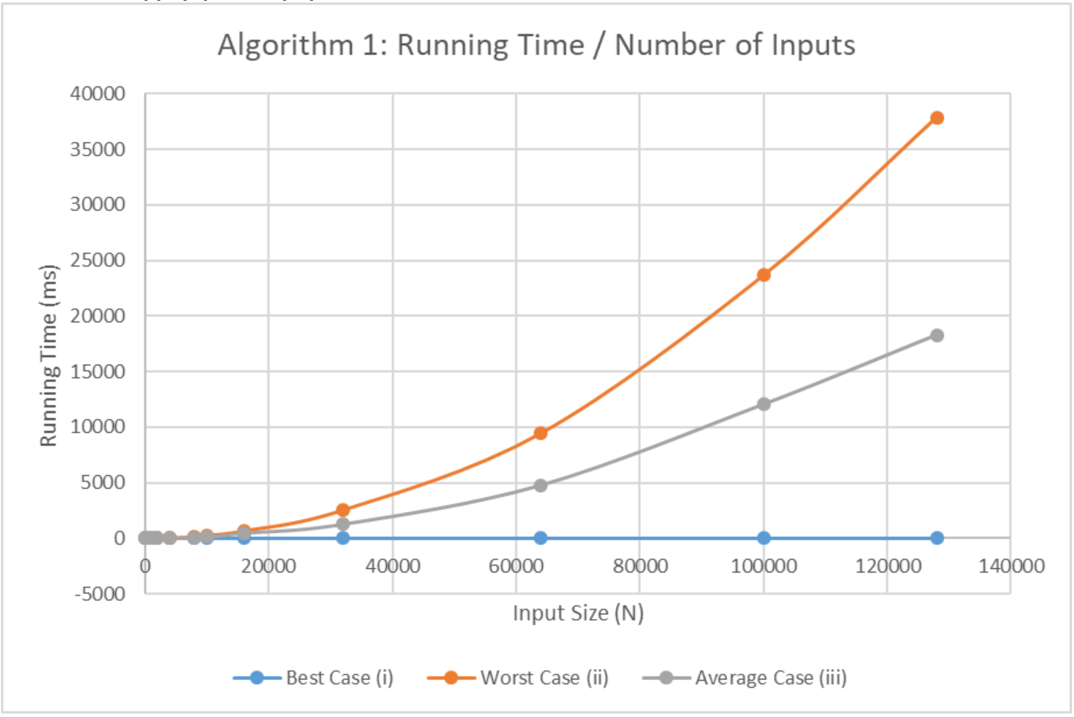
Homework Assignment 2

```
int* sortAlgorithm1(const int* arr1, const int* arr2, const int N ){
   int* arr3 = new int[2*N];
   int counter = 0;
   int holder = N;

   for ( int i = 0; i < N; i++) //=> O(N)
   {
      arr3[i] = arr1[i];
   }

   for ( int i = 0; i < N; i++ ) //=> O(N)
  {
        if ( arr3[i] > arr2[counter])
        {
         for ( int k = holder; k > i; k--) //=> O(N) O(N) = O(N^2)
           arr3[k] = arr3[k-1];
         arr3[i] = arr2[counter];
         holder++;
         counter++;
        }
   }
   while (counter < N) //=> O(N)
  {
         arr3[holder++] = arr2[counter++];
   }
   return arr3;
}

int* sortAlgorithm2(const int* arr1, const int* arr2, const int N ){
  int count1 = 0, count2 = 0, index = 0;
  int* arr3 = new int[2*N];

  while ( count1 < N && count2 < N) //=> O(N)
  {
   if (arr1[count1]< arr2[count2]){
      arr3[index] = arr1[count1];
      index++;
      count1++;
   }
   else{
      arr3[index] = arr2[count2];
      index++;
      count2++;
   }
  }

  //Remaining items will be copied
  while ( count1 < N) //=> O(N)
  {
      arr3[index] = arr1[count1];
      index++;
      count1++;
  }

  while ( count2 < N) //=> O(N)
  {
      arr3[index] = arr2[count2];
      index++;
      count2++;
  }
  return arr3;
}
```

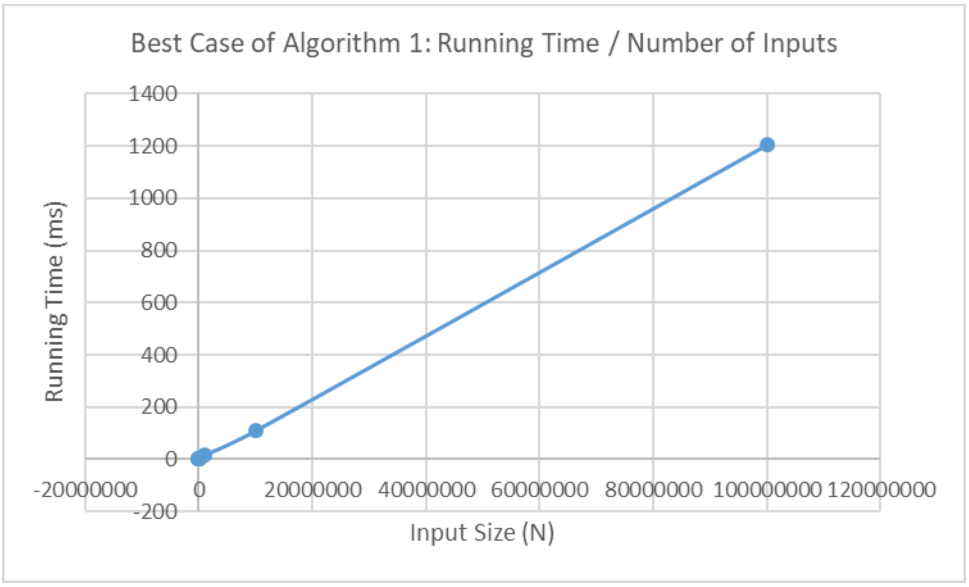
ALG1	**Running Time (ms)**			**ALG2**	**Running Time(ms)**		
N	**i**	**ii**	**iii**	**N**	**i**	**ii**	**iii**
100	0	0	0	100	0	0	0
1000	0	2	1	1000	0	0	0
2000	0	13	6	10000	0	0	0
4000	0	36	18	100000	1	1	1
8000	0	149	70	1000000	11	10	13
10000	0	261	109	10000000	115	113	124
16000	0	681	476	100000000	815	794	963
32000	1	2549	1275				
64000	1	9449	4791				
100000	1	23655	12071				
128000	2	37843	18303				
1000000	16						
10000000	109						
100000000	1204						

Table 1: Raw data table of first and second algorithm to plot the graphs

- **Use your results to generate a plot of running time (y-axis) versus the input size N (x-axis), per ordering case (i.e., for (i), (ii), and (iii)**

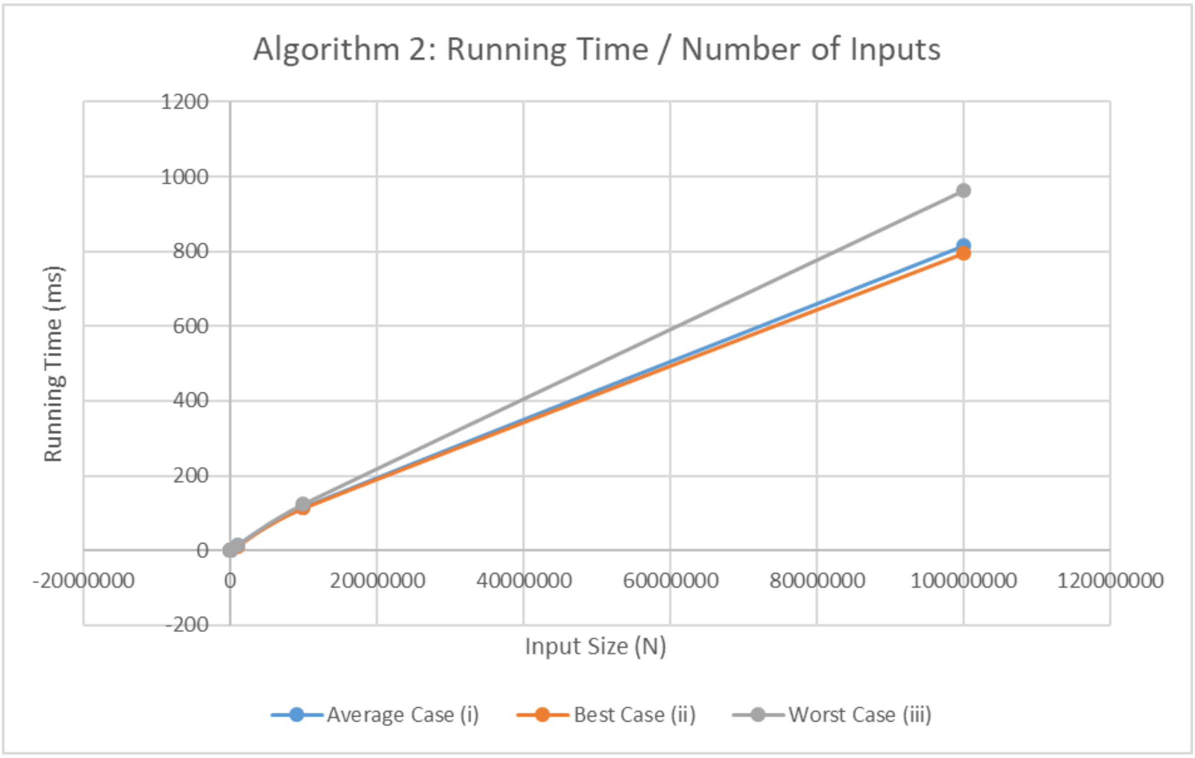


Graph 1: Running Time versus Input Size plots for each case of Algorithm 1.



Graph 1.b: Running Time versus Input Size plot for best case of Algorithm 1.

The best case is included separately as linear increase, O(N), is much less when compared to quadratic increase, O(N^2). Because of that, the best case almost looks like a constant operation in *Graph 1,* however that is not true as the input size becomes larger and larger and approaches to infinity, it turns out to be O(N) as expected. That is because in Algorithm 1, whatever the case is the first loop is always executed for N items indeed.



Graph 2: Running Time versus Input Size plots for each case of Algorithm 2.

- **Based on your plots indicate the best, average and worst cases for each algorithm and the corresponding worst case time complexity**

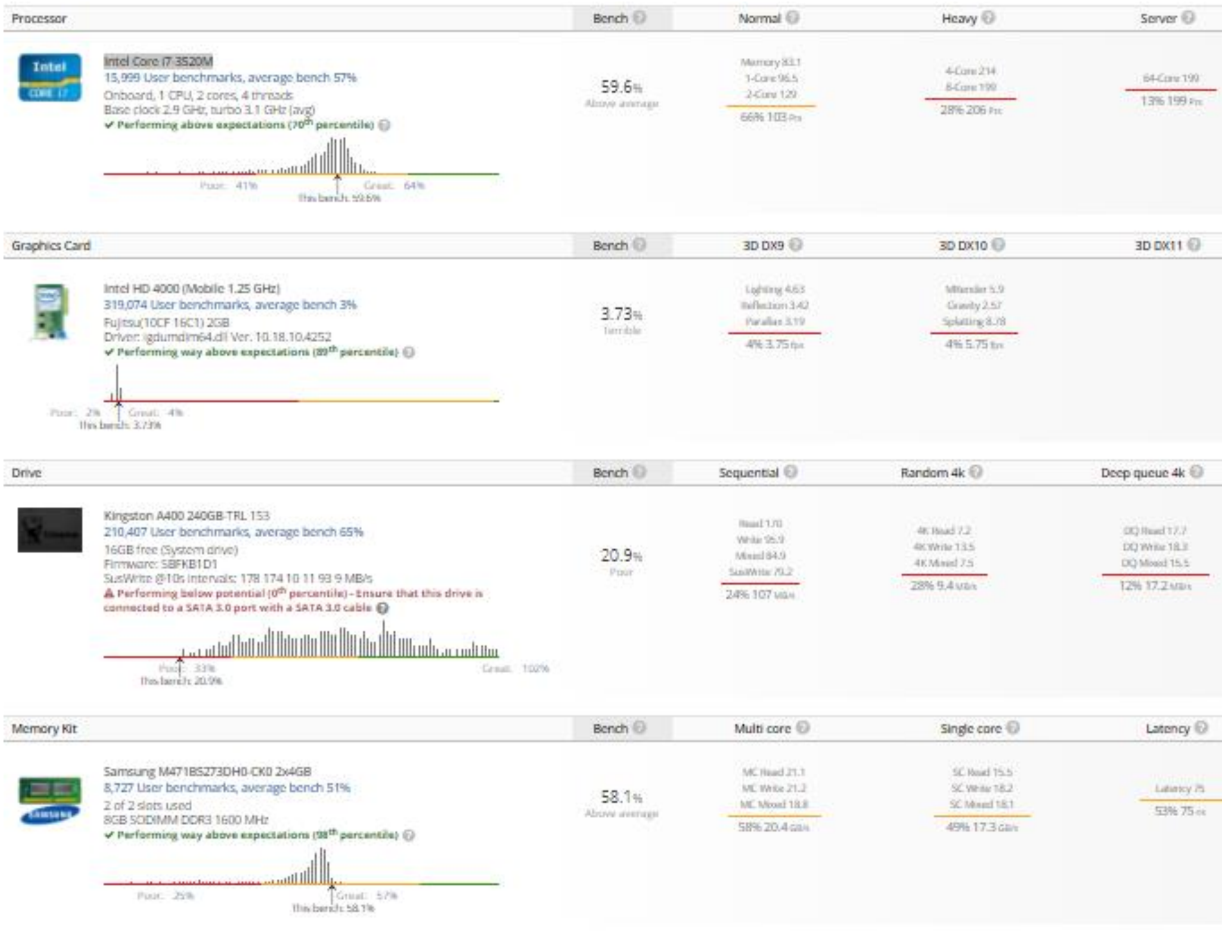
For Algorithm 1:
Best Case = all numbers in arr1 are smaller than arr2: O(N) (never enters to second loop in nested loop structure basically)
Average Case = there is no such ordering between these arrays: O(N^2) (enters into second loop in nested loop structure)
Worst Case = all numbers in arr2 are smaller than arr1: O(N^2) (enters into second loop in nested loop structure / N number of shifts will happen)

For Algorithm 2:
Best Case = all numbers in arr2 are smaller than arr1: O(N) (enters into one loop no matter what)
Average Case = all numbers in arr1 are smaller than arr2: O(N) (enters into one loop no matter what)
Worst Case = there is no such ordering between these arrays: O(N) (enters into one loop no matter what)

It must be addressed that average and best case totally depend on the rvalue and lvalues of the specific implementation, in other words whether the items in arr1 is compared with arr2 or the items in arr2 is compared with arr1 in terms of being smaller (by <) and which one is copied first and thus can interchange in some trials. But this difference is negligible as they both approach to N for large input sizes.

- **Provide the specifications (processor, RAM, operating system etc.) of the computer you used to obtain these execution times. You can use any computer with any operating system for this assignment, but make sure your code compiles and runs on the dijkstra server.**

System: Fujitsu LIFEBOOK S792
Processor: Intel Core i7-3520M
RAM: 8GB
OS: Windows 10
https://www.userbenchmark.com/UserRun/33441262



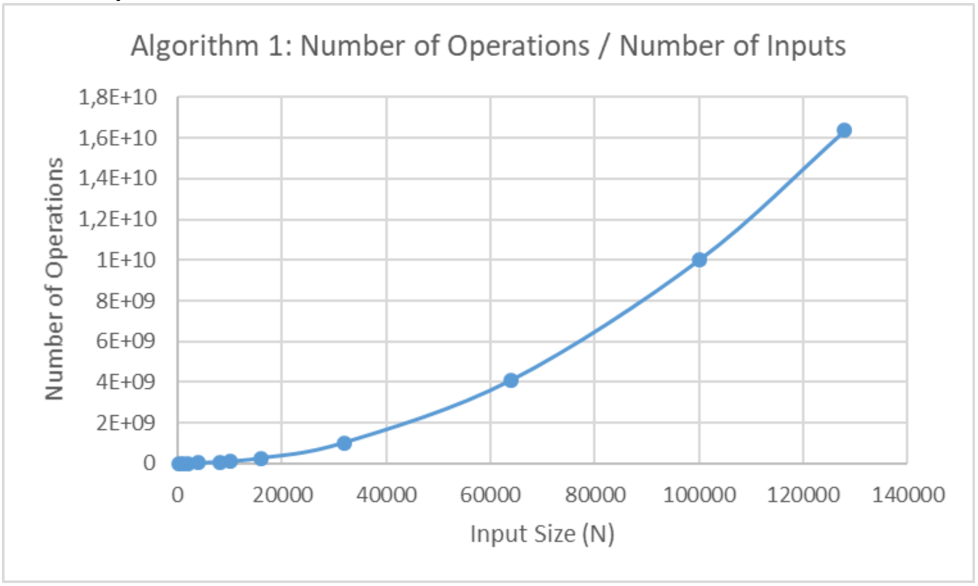
Processor	Bench	Normal	Heavy	Server
Intel Core i7 3520M 15,999 User benchmarks, average bench 57% Onboard, 1 CPU, 2 cores, 4 threads Base clock 2.9 GHz, turbo 3.1 GHz (avg) ✓ Performing above expectations (70th percentile) Poor: 41% Great: 64% This bench: 59.6%	59.6% Above average	Memory 83.1 1-Core 96.5 2-Core 129 66% 103 Pts	4-Core 214 8-Core 199 28% 206 Pts	64-Core 199 13% 199 Pts

Graphics Card	Bench	3D DX9	3D DX10	3D DX11
Intel HD 4000 (Mobile 1.25 GHz) 319,074 User benchmarks, average bench 3% Fujitsu(10CF 16C1) 2GB Driver: igdumdim64.dll Ver. 10.18.10.4252 ✓ Performing way above expectations (89th percentile) Poor: 2% Great: 4% This bench: 3.73%	3.73% Terrible	Lighting 4.63 Reflection 3.42 Parallax 3.19 4% 3.75 fps	MRender 5.9 Gravity 2.57 Splatting 8.78 4% 5.75 fps	

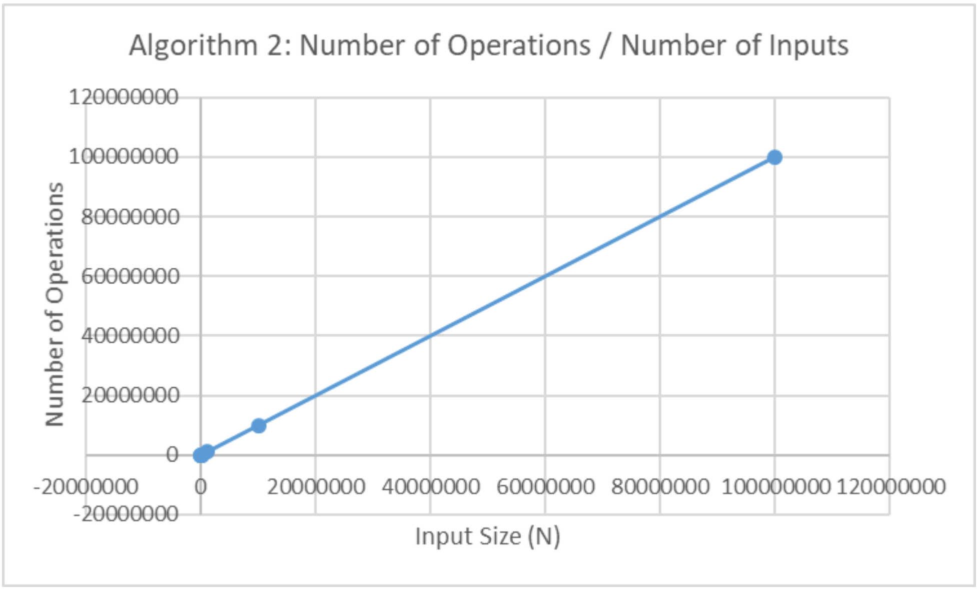
Drive	Bench	Sequential	Random 4k	Deep queue 4k
Kingston A400 240GB-TRL 153 210,407 User benchmarks, average bench 65% 16GB free (System drive) Firmware: SBFKB1D1 SusWrite @10s intervals: 178 174 10 11 93 9 MB/s ⚠ Performing below potential (0th percentile) - Ensure that this drive is connected to a SATA 3.0 port with a SATA 3.0 cable Poor: 33% Great: 102% This bench: 20.9%	20.9% Poor	Read 170 Write 95.9 Mixed 84.9 SusWrite 70.2 24% 107 MB/s	4K Read 7.2 4K Write 13.5 4K Mixed 7.5 28% 9.4 MB/s	DQ Read 17.7 DQ Write 18.3 DQ Mixed 15.5 12% 17.2 MB/s

Memory Kit	Bench	Multi core	Single core	Latency
Samsung M471B5273DH0-CK0 2x4GB 8,727 User benchmarks, average bench 51% 2 of 2 slots used 8GB SODIMM DDR3 1600 MHz ✓ Performing way above expectations (98th percentile) Poor: 25% Great: 57% This bench: 58.1%	58.1% Above average	MC Read 21.1 MC Write 21.2 MC Mixed 18.8 58% 20.4 GB/s	SC Read 15.5 SC Write 18.2 SC Mixed 18.1 49% 17.3 GB/s	Latency 75 53% 75 ns

- **Plot the expected worst case growth rates obtained from the theoretical analysis by using the same `N` values that you used in your simulations. That is, using your theoretical analysis, for each `N` value plot the expected number of operations.**



Graph 3: Expected Operation Number versus Input Size plot for worst case of Algorithm 1.



Graph 4: Expected Operation Number versus Input Size plot for worst case of Algorithm 2.

- **Compare the expected growth rates obtained in step 5 and the worst case results obtained in step 3, and discuss your observations in a paragraph.**

First conclusion that can be drawn by comparing running time for worst cases of *Graph 1* and *Graph 2* with expected number of operations in *Graph 3* and *Graph 4* is that; these two properties are proportional to each other. In other words, running time of the algorithms for the worst case show consistency with their corresponding graphs of number of operations and therefore the running time faithfully represents the number of operations required. As in Big-O notation the efficiency of an algorithm is decided by focusing on the worst case scenario, the worst cases were determined by checking which one of the cases (i)(ii)(iii) was taking more time overall. In Algorithm 1, the best case was all of items in arr1 being smaller than arr2. This is mostly because the sorting technique of Algorithm 1 was that: The first array items were inserted directly and the items of the second array were inserted by checking those items of arr1 and when needed, shifting the items which requires to re-visit the items and thus making the time complexity O(N^2). In the best case, as all items of arr1 are smaller, there is no need to shift any item and therefore the second loop is never executed which reduces the complexity to O(N). The average and worst cases were both O(N^2) because of no other reason than revisiting items while shifting. Although the average case takes lower running time, it still shows quadratic growth and that is because even if the loop is exited without exactly re-visiting each element, those numbers are neglected (like when N^2-X, X is always neglected if it is a constant number). In Algorithm 2, all cases showed the growth rate of O(N) no matter what as the algorithm proceeds by checking pairs of arr1 and arr2 and thus entering a loop in any way, checking every item only once. In this regard, it can be stated that overall, Algorithm 2 is a more efficient algorithm to solve the problem which runs much faster for very large numbers of inputs in the perspective of time complexity and Big-O principles. Therefore, the experimental results that are obtained by running both algorithms several times with a number of different input sizes show parallelism with the theoretical hypothesis that can be reached by simply analyzing the algorithms and checking how the loops inside are working when the worst case is considered. In the light of this observation, it is safe to state that the experimental findings justify the theoretical expected growth rates represented with the Big-O notation.