CS111-Design and Analysis of Algorithms

Bicol University

Legazpi City, Albay

Programming Project

Members:

Renmar Balana

Raphael James Madrid

Joshua Navia

BSCS - II C

I. Structure of the Code

line (001 - 004)	header files
line (006 - 007)	constants definition
line (010 - 028)	function prototype declaration
line (031 - 226)	main function
(033 - 042)	variable declarations, initialize array to the heap
(046 - 051)	prompt user input for N
(052 - 054)	Generate random array, then print on screen
(057 - 062)	prompt user input for X
(063 - 065)	Generate sorted array, then print on screen
(071 - 080)	make 4 copies of the Random array
(083 - 084)	run and clock the algorithm for random input, then print the time: insertion sort;
(087 - 091)	merge sort;
(094 - 098)	quicksort ;
(101 - 106)	heap sort;
(111 - 120)	make 4 copies of the Sorted array
(123 - 124)	run and clock the algorithm for sorted input, then print the time: insertion sort;
(127 - 131)	merge sort;
(134 - 138)	quicksort ;
(141 - 145)	heap sort;
(150 - 172)	print to a text file named "Array_Values+Processor_Time", the time taken the algorithms, original random array, random array after sorting, and the array with already sorted values.
(174-217)	Test case to see if the respective arrays are actually sorted inside and Generate output file for insertion,merge,quick,and heap sort to double check if it is truly sorted. (Optional)
(220 - 223)	freeing of all the allocated memory used earlier.
(225 - 226)	end of main
line (230 - 483)	Function Definitions
(230 - 249)	Function to generate random array
(250 - 256)	Function to generate an array sorted in ascending order.
(258 - 277)	Function for insertion sort
(279 - 289)	Function for merge sort
(290 - 335)	Function to merge subarrays of mergesort
(337 - 349)	Function for Quicksort
(350 - 373)	Function to find partition position for Quicksort
(375 - 386)	Function for Heapsort
(387 - 403)	Function to heapify
(406 - 410)	function to swap the value of two integers
(412 - 437)	function to print an array to a screen
(438 - 463)	function to print an array to a text file
(465 - 472)	median function, finds median between 3 integers (Unused code)
(474 - 483)	delay function, delays clock by the entered parameter in milliseconds

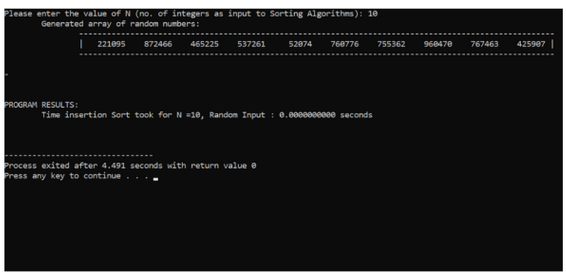
II. Screenshots of actual execution

1. Compute the average (over five different runs) of the actual running time T(N) for five different N values, N=10 , 100 ,1000 ,10000,100000 .

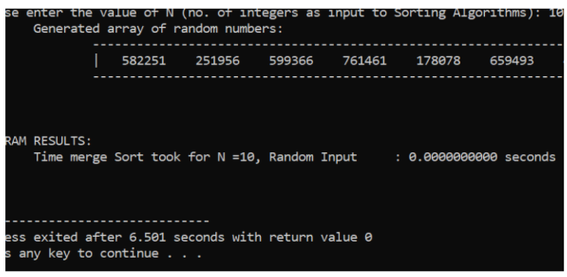
The pictures below show the results of our last attempt running the program with five different N values.

Random N=10

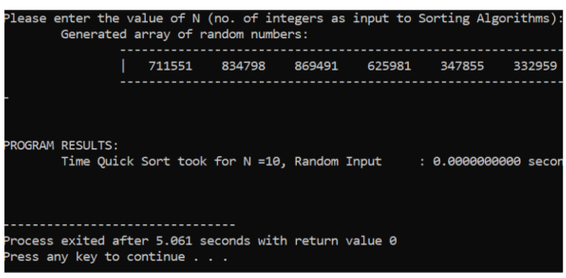
Insertion Sort



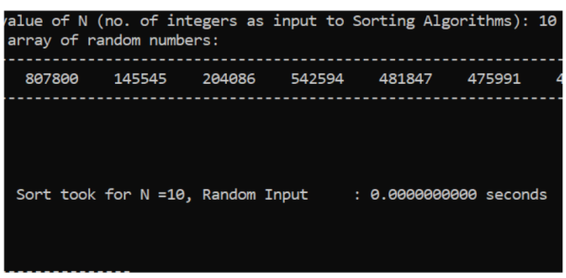
Merge Sort



Quick Sort

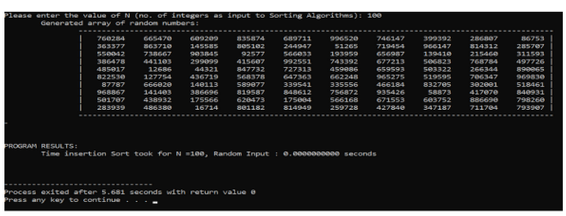


Heap Sort

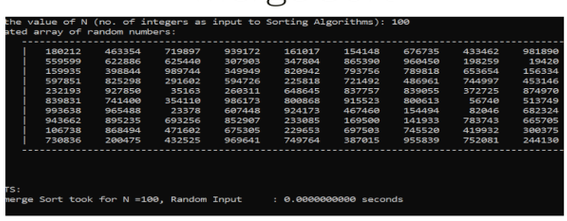


Random N=100

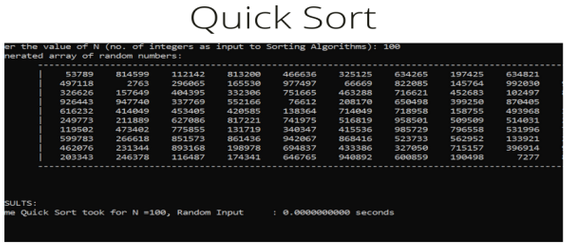
Insertion Sort



Merge Sort



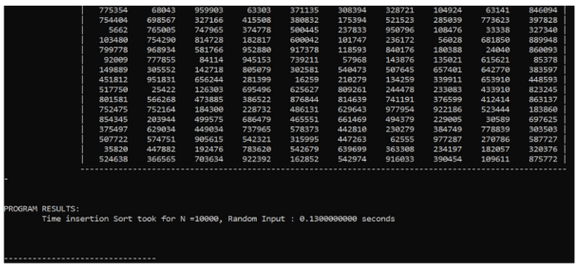
Quick Sort



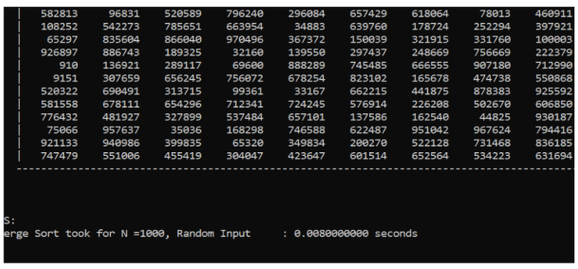
Heap Sort

Random N=1000

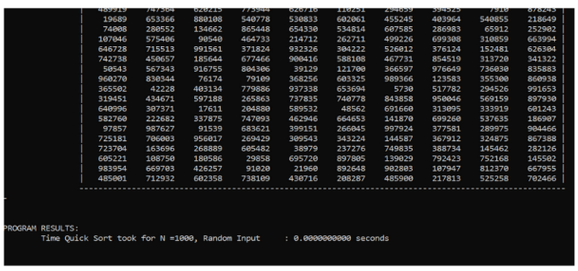
Insertion Sort



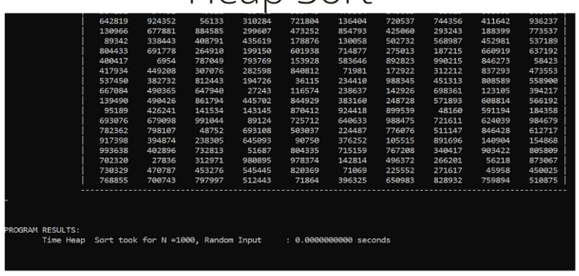
Merge Sort



Quick Sort

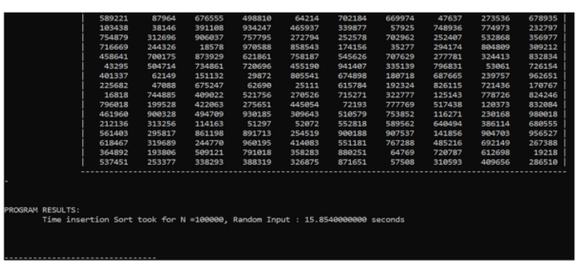


Heap Sort

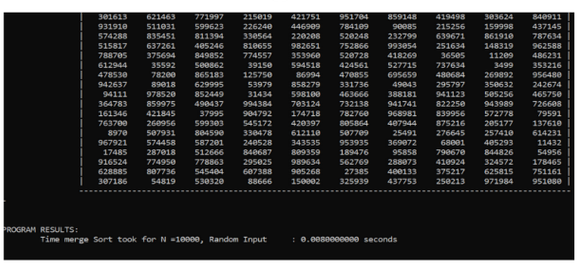


Random N=10000

Insertion Sort



Merge Sort

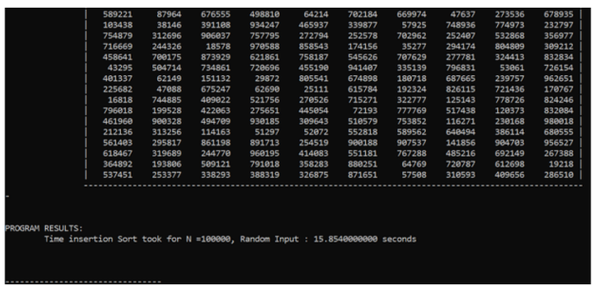


Quick Sort

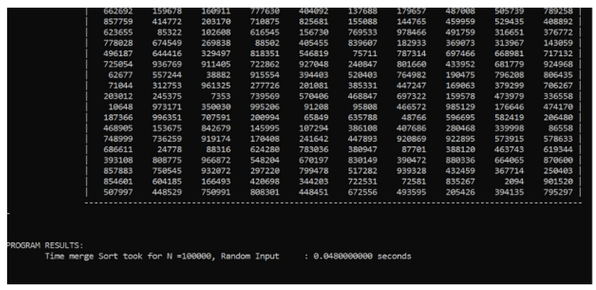
Heap Sort

Random N=100000

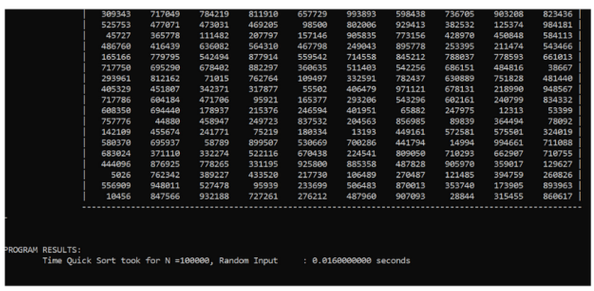
Insertion Sort



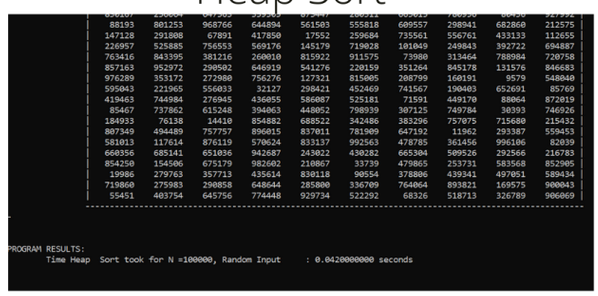
Merge Sort



Quick Sort



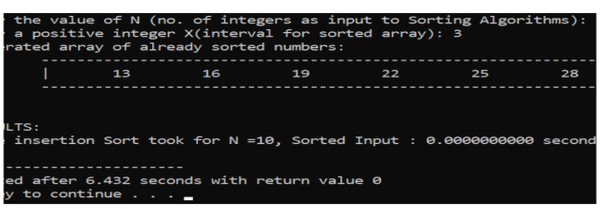
Heap Sort



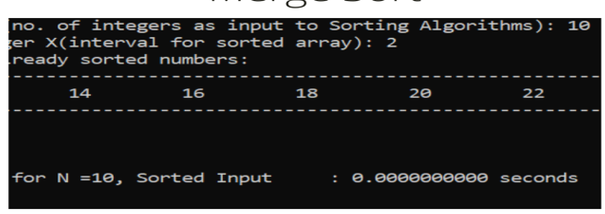
2.Repeat (1) using Part A.2.b as above (i.e., input array is sorted) to compute corresponding running time T '(N) for N=10 ,100 ,1000 , 10000,100000 .

Sorted N=10

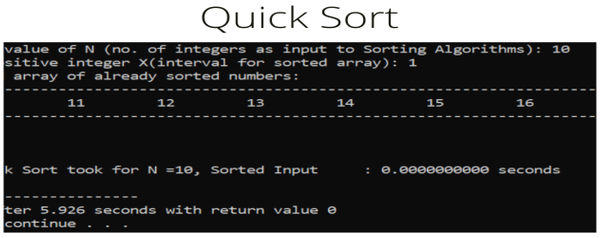
Insertion Sort



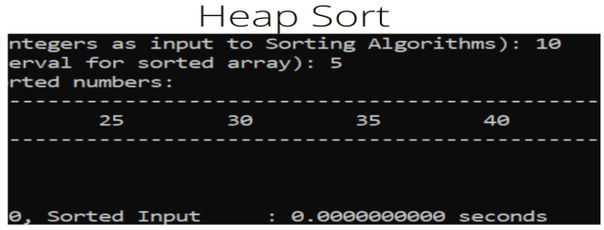
Merge Sort



Quick Sort

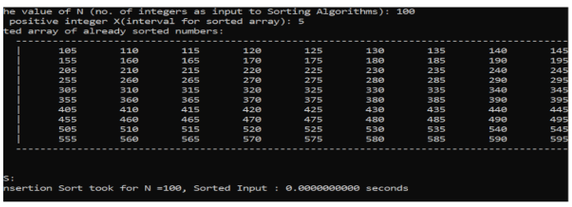


Heap Sort

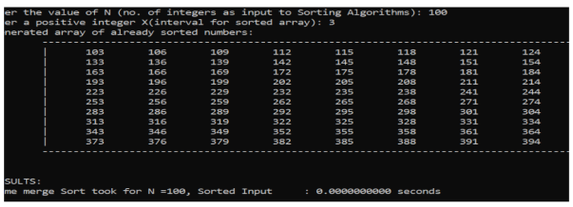


Sorted N=100

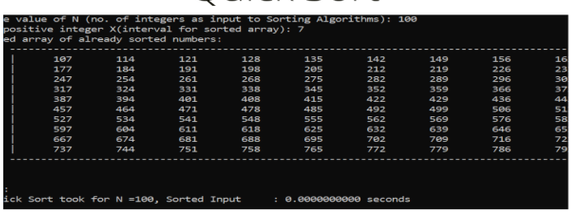
Insertion Sort



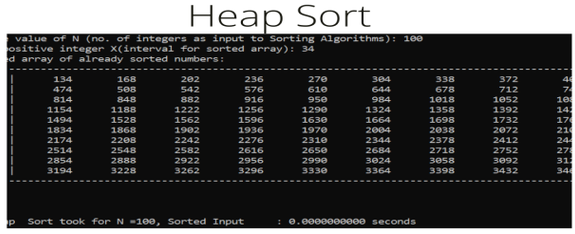
Merge Sort



Quick Sort

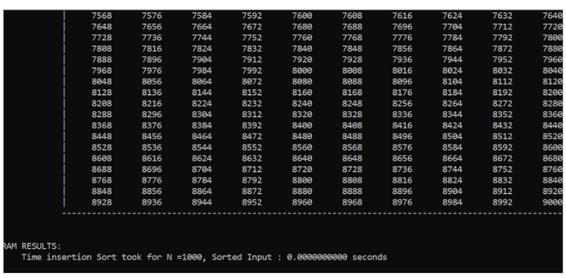


Heap Sort

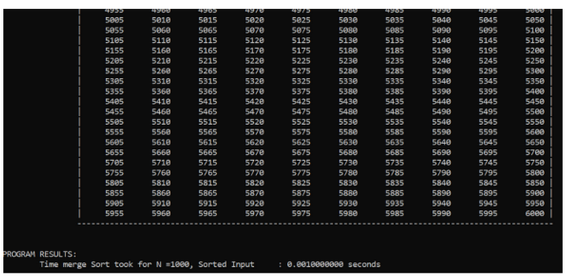


Sorted N=1000

Insertion Sort



Merge Sort

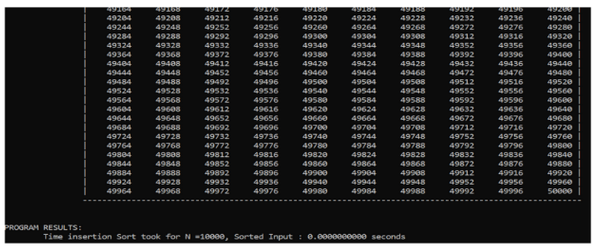


Quick Sort

Heap Sort

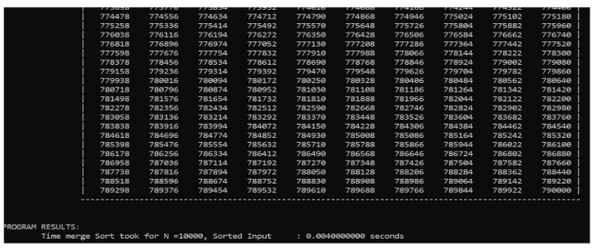
Sorted N=10000

Insertion Sort



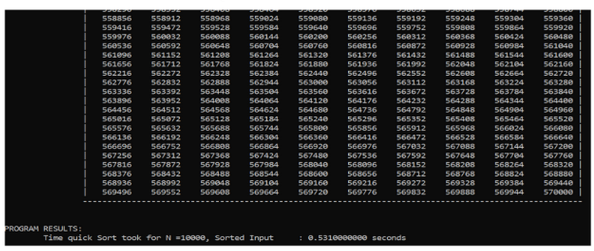
```
PROGRAM RESULTS:
      Time insertion Sort took for N =10000, Sorted Input : 0.0000000000 seconds
```

Merge Sort



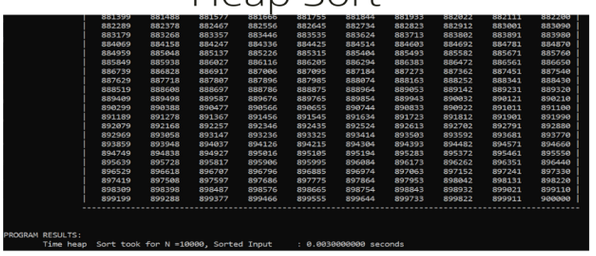
```
PROGRAM RESULTS:
      Time merge Sort took for N =10000, Sorted Input     : 0.0040000000 seconds
```

Quick Sort



```
PROGRAM RESULTS:
      Time quick Sort took for N =10000, Sorted Input     : 0.5310000000 seconds
```

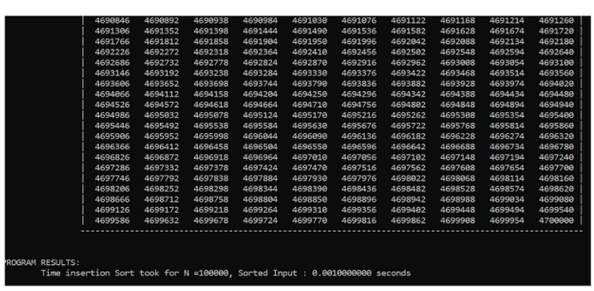
Heap Sort



```
PROGRAM RESULTS:
      Time heap  Sort took for N =10000, Sorted Input     : 0.0030000000 seconds
```

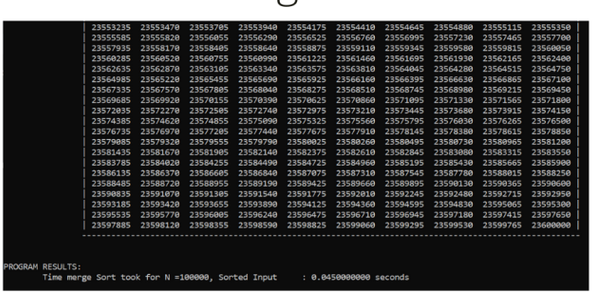
Sorted N=100000

Insertion Sort



```
PROGRAM RESULTS:
      Time insertion Sort took for N =100000, Sorted Input : 0.0010000000 seconds
```

Merge Sort



```
PROGRAM RESULTS:
      Time merge Sort took for N =100000, Sorted Input     : 0.0450000000 seconds
```

Quick Sort

```
PROGRAM RESULTS:
      Time quick Sort took for N =100000, Sorted Input     : 65.7490000000 seconds
```

Heap Sort

```
PROGRAM RESULTS:
      Time heap  Sort took for N =100000, Sorted Input     : 0.0410000000 seconds
```

III. Analysis of the outputs

3. Present the results in Table format, where rows of the tables will indicate different N values and columns will indicate the T(N) for the different sorting algorithm.

Average Running Time for an Input Array that is Random

N	Insertionsort	Mergesort	Quicksort	Heapsort
10	0	0	0	0
100	0	0	0	0
1000	0	0	0	0
10000	0.1472	0.0082	0	0.0032
100000	13.5048	0.0532	0.0192	0.0412

Average Running Time for an Input Array that is Sorted

N	Insertionsort	Mergesort	QuickSort	Heapsort
10	0	0	0	0
100	0	0	0	0
1000	0	0.0024	0.0112	0.0002
10000	0	0.0042	0.5360	0.0030
100000	0.0008	0.044	61.7186	0.0346

for n = 10 and 100, the time is too fast to be clocked, so data is only zero and inconclusive.

- ➔ Insertion sort progressively gets slower as the input size gets larger for random input. It's also the slowest algorithm for a random array. In the worst case, this algorithm would perform O(n^2) time. For an input array that is already sorted, insertion sort is always fast no matter the size of N. This is the best case for insertion sort as it takes only O(n) time.

- ➔ Time for Merge sort is consistently fast, and grows longer as N gets larger. And it doesn't matter if the array is random or already sorted. This is because the time complexity of merge sort for all cases, is always O(n*logn). Its downside is that it takes more memory space, because we need to also declare temporary arrays that will store the divided subarrays.

- ➔ Quicksort is the fastest performing algorithm for a randomized array, so we can say its name is well deserved. Like other divide-and-conquer algorithms, its average time is O(n*log n). However, the case is the complete opposite if it analyzes an array that is already sorted.
  Quicksort is notably the slowest across all values of N, if the input array is already sorted. This is because the chosen pivot for quicksort is always the last element, which will always be the largest because the array is sorted in ascending order. Therefore, partitioning will take O(n) time (no swapping will be done anyways), for all the n-number of elements. So for worst case, time will be O(n^2)

- ➔ Heapsort, although not a divide-and-conquer algo, is similar to how merge sort behaves, where it doesn't matter very much if the input array is random or already sorted.

Conclusion:

As expected, the time it takes for all the sorting algorithms to process the data also increases directly proportional as N grows larger.

Quick sort is the best sorting algorithm to use if the array to be sorted is randomly spread. However, we need to be careful in using it, because feeding an already sorted array into quicksort will take a lot of time to process.

Inversely, Insertion sort works best in an array that is already sorted, but struggles in practical use, because we usually arrange unsorted data in real world scenarios.

When all of the facts and experiments are taken into consideration, it becomes clear that each sorting algorithm has its own set of strengths and weaknesses when dealing with the values of arrays. This is strongly supported by pictures of our executions per input of N and the presentation of the results in table format, which enable the user to easily see the differences between each sorting algorithm and conclude what is efficient and less efficient in producing an output,taking in consideration what the end user is asking,for example randomized or Sorted output of arrays.

IV. Challenges encountered

- Figuring out a function to print the generated array into a neat and organized table, so we can confirm the functions that generate the random and sorted array are behaving as expected.

  We chose 10 columns, as the base of the grid, the 11^{th} element would be printed on the next row and so on…

- Naïve implementation of array, ex: int array[SIZE]; can only store a limited number of elements. We later found out this is because this kind of array is stored on the computer's stack memory, which is quite limited and not designed to handle very large amounts of data.

  A better implementation would be to use malloc to allocate an adequate amount of memory, to exactly fit the size of the array we will need. By using malloc, tha array is stored in the heap; we were even able to reach 1 million integers as input.

- Quicksort gave us a problem, because testing an already sorted array through a quicksort would result in the worst possible case, that is T(n) = n^2. This is because our initial trials in partitioning used the last element as its pivot. Since the last element will always be the largest for an already sorted array, this makes the recursion depth for large amounts of input to crash the program by stack overflow. We were only able to reach 25,000 as the value of N, before the program would crash.

  1st attempt to fix: pivot was chosen randomly. reached 55,000 as the value of N. Still not enough.

  2nd attempt to fix: pivot chosen was the median between first, second and last elements. Only 65,000 was feasible.

  3rd attempt to fix: Quicksort function was changed into a half-recursive algorithm. Instead of recursing both sub partitions, only the smaller side used recursion, while the larger side was iterated. This way, the recursion depth won't reach dangerously high levels that would crash the program. Instead of having one very deep recursion, we end up with multiple "shallow" recursions.

  After having solved the stack overflow problem, we later changed back the program to pick the last element as the pivot, as we believe this would best simulate the worst case for quicksort given an array with already sorted elements.

- rand() function can only generate integers from 0 until 32767, which is equivalent to the binary value 111111111111111: 15 bits that are all flipped. Because the MAXRANGE to be used is 1 million, with binary value 11110100001001000000, we need a minimum of 20 bits to be represented.

Decimal Value of X	Binary Value	Operation
32,767	111111111111111	Shift to the left: **x <<= 15**
1,073,709,056	11111111111111100000000000000	XOR x with another random number (10,922) **x ^= rand()** X 11111111111111100000000000000 rand 010101010101010 Result: 11111111111111010101010101010
1,073,719,978	11111111111111010101010101010	Modulo by 1M+1 **x %= MAXRANGE +1** 1,073,719,978 % 1,000,001 718,905
718,905	10101111100000111001	

However, this solution is not perfect. Because of the process we used, shifting the random number then performing XOR with another random number can yield 1,073,741,823 at most (XOR of two 32,767).

111111111111111111111111111111 (30 1's will yield:) 1,073,741,823 as maximum possible value generated, before modulo operation.

Because the actual sample size of 1 billion+ is not perfectly divisible by 1 million and 1,

range (0 – 740,749) appears 1074 times, when 1,073,741,823 % MAXRANGE+1. (740,750)

range (740,750 – 1,000,000) appears 1073 times only. (259,251)

740,750 * (1074) + 259,251 * (1073) = 1,073,741,823

1,073,741,823 % 1,000,001 = 740,750

1,073,741,823 / 1,000,001 = 1073.74074926

Probability = (number of possible outcomes)/(total number of outcomes)

ideal probability: (1/1,000,001) =0.000000999999

probability (0-740,749): 1074/1,073,741,823 = 0.00000100024
probability (740,750 – 1,000,000): 1073/1,073,741,823 = 0.00000099930

Sample set is very, very, very slightly more biased in the first 74% integers of 1 million.
But this bias is so small that it is practically negligible.