DEPARTMENT OF CYBER SECURITY

Design and Analysis of Algorithm Project Part II	
SUBMITTED BY	211042 Noman Masood Khan A
SEMESTER	Fifth
SUBJECT	DAA
SUBMITTED TO	Sir Ammar Masood

1. Heap Sort Program

Heap Sort program is developed in C which contains three functions:

1. Max-Heapify Function

The heapify function corrects the single violation of the heap in the subtree with the top-to-bottom approach.

```
5   void heapify(int arr[], int n, int i)
6   {
7       int largest = i; // make the index i element to largest
8       int l = 2 * i + 1; // left node
9       int r = 2 * i + 2; // right node
10
11      // If left child is larger than root
12      if (l < n && arr[l] > arr[largest])
13          largest = l;
14
15      // If right child is larger than the root
16      if (r < n && arr[r] > arr[largest])
17          largest = r;
18
19      // If largest is not root swap the child with the root
20      if (largest != i) {
21          swap(arr[i], arr[largest]);
22
23          // Recursively heapify the affected sub-tree
24          heapify(arr, n, largest);
25      }
26  }
```

Functional parameters includes:

- the value to correct in the sub-tree as an index i
- Array as arr and length of array as n.

Line 12 if condition is checking whether the index value is greater than the left child, if it is then the largest is the left child else the parent node will be the largest node.

Line 16 if condition is also checking whether index value is greater than the right child , if it is greater than the largest will be the right child, else the parent node will be the largest node.

Line 20 is first checking if the largest node is not the parent node itself, then it is swapping the largest node with the parent node, so now the largest node is parent node in the sub-

tree and the previous parent node is in its correct position in the sub-tree.

Line 24 is recalling the max-heapify sort the sub-tree at the largest node which was at the i position.

This recursion will continue till the i index reaches the child node of the tree.

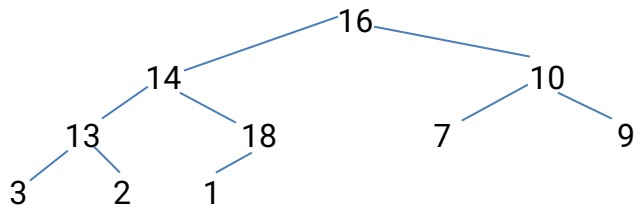
2. HeapSort() Function

This function is sorting the received array in the function parameters. First it is making the array max-heap and then sorting it using the recursion method.

```
29  void heapSort(int arr[], int n)
30  {
31
32      for (int i = n / 2 - 1; i >= 0; i--)
33          heapify(arr, n, i);
34
35      for (int i = n - 1; i >= 0; i--) {
36
37          swap(arr[0], arr[i]);
38
39          heapify(arr, i, 0);
40      }
41  }
```

Since the heapify assumes that the root of the tree is already max-heap so first for loop is starting from index i=array/2 -1. array/2 -1 will give us the index which will be the parent of the first sub-tree.

For example, if we have the array 16 14 10 18 7 9 3 2 8 1, then i = array/2 -1 gives us i = 4.



n=10

i = n/2 -1 = 4.

The for loop will start with the parent node at the 4 positions = 13. Max-heapify function starts its operation from the 4 index position and correct the violation of heaps. Likewise,

for loop will give the next index number = 5 to heapify function to correct this sub-tree and this will continue till the index i becomes 0.

Next for loop is running in the bottom-to-top approach. First it is swapping the node at the root with the last node at the leave of the tree. Then it is running the max-heapify function at the node to correct its position in the sub-tree. With this loop, all the elements will be sorted in the ascending order with the largest element at the root of the node.

The flow of operations in heap array is :

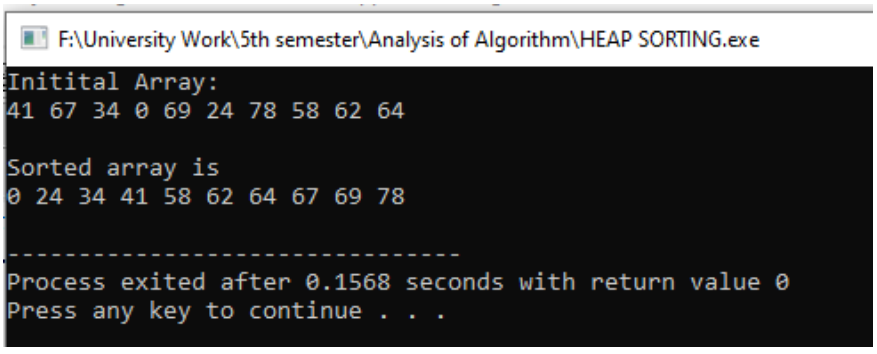
Main function	Heapify function ()	Heapsort function
Sending the un-sorted array to heapify	Making the Array max heap to sort them easily	This function is actually sorting the array in ascending order

1.2 Test Cases Used to Check Code

Test Case 1:

Input Array : 41 67 34 0 69 24 78 58 62 64

Output Array : 0 24 34 41 58 62 64 67 69 78



Test Case 2:

Input Array : 41 467 334 0 169 224 478 358 462 464

Output Array : 0 41 169 224 334 358 462 464 467 478

```
F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
Initital Array:
41 467 334 0 169 224 478 358 462 464

Sorted array is
0 41 169 224 334 358 462 464 467 478

--------------------------------
Process exited after 0.169 seconds with return value 0
Press any key to continue . . .
```

Test Case 3:

Input Array : 41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Output Array : 41 6334 11478 15724 18467 19169 24464 26500 26962 29358

```
F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
Initital Array:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Sorted array is
41 6334 11478 15724 18467 19169 24464 26500 26962 29358

--------------------------------
Process exited after 0.1367 seconds with return value 0
Press any key to continue . . .
```

1.3 Method for Generation of Random Inputs

In the code, I am using the rand() function which is the part of the stdlib library to generate the random whole numbers. I have generated the random number of length 100 and heap sorted it very well.

```
F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
Initital Array:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464 5705 28145 23281 16827 9961 491 2995 11942 4827 5436 32391 14604 3902 153 292 12382 17421 18716 19718 19895 5447
 21726 14771 11538 1869 19912 25667 26299 17035 9894 28703 23811 31322 30333 17673 4664 15141 7711 28253 6868 25547 27644 32662 32757 20037 12859 8723 9741 27529 778 12
316 3035 22190 1842 288 30106 9040 8942 19264 22648 27446 23805 15890 6729 24370 15350 15006 31101 24393 3548 19629 12623 24084 19954 18756 11840 4966 7376 13931 26308
16944 32439 24626 11323 5537 21538 16118 2082 22929 16541

Sorted array is
41 153 288 292 491 778 1842 1869 2082 2995 3035 3548 3902 4664 4827 4966 5436 5447 5537 5705 6334 6729 6868 7376 7711 8723 8942 9040 9741 9894 9961 11323 11478 11538 11
840 11942 12316 12382 12623 12859 13931 14604 14771 15006 15141 15350 15724 15890 16118 16541 16827 16944 17035 17421 17673 18467 18716 18756 19169 19264 19629 19718 19
895 19912 19954 20037 21538 21726 22190 22648 22929 23281 23805 23811 24084 24370 24393 24464 24626 25547 25667 26299 26308 26500 26962 27446 27529 27644 28145 28253 28
703 29358 30106 30333 31101 31322 32391 32439 32662 32757

--------------------------------
Process exited after 0.1995 seconds with return value 0
Press any key to continue . . .
```

1.4 Time and Memory Complexity Comparison

1. When input size is 50, then compilation time is 1.08 seconds and the output size is 3MB.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
- Output Size: 3.07641124725342 MiB
- Compilation Time: 1.08s
```

2. When the Input size is 100, then the compilation time is 1.13 seconds.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
- Output Size: 3.07641124725342 MiB
- Compilation Time: 1.13s
```

3. When the Input size is 500, then compilation time is 1.14 seconds

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
- Output Size: 3.07641124725342 MiB
- Compilation Time: 1.14s
```

2. Quick Sort Program

Quick Sort program in C language and it contains three main functions:

1. Partition Function

Partition function makes partition in the array so the large values are one side of an array and small values are on other side of an array based on the value of pivot.

```
16  //function to partition the array
17  int partition(int array[], int low, int high) {
18
19    // select the rightmost element as pivot
20    int pivot = array[high];
21    // pointer for greater element
22    int i = (low - 1);
23
24    // traverse each element of the array
25    // compare them with the pivot
26    for (int j = low; j < high; j++) {
27      if (array[j] <= pivot) {
28
29        // if element smaller than pivot is found
30        // swap it with the greater element pointed by i
31        i++;
32        // swap element at i with element at j
33        swap(&array[i], &array[j]);
34      }}
35    // swap pivot with the greater element at i
36    swap(&array[i + 1], &array[high]);
37    // return the partition point
38    return (i + 1);
39  }
```

Partition function is accepting the first element, last element and an array as arguments. The last element of an array is being select as pivot and i variable is being assigned the last value of an array which will be pointing to the small elements.

First loop is starting the iteration as j from the first element of an array to the last element of array. If the element at the index j is smaller than the pivot value then the element is swap with the element at index i since i is storing the value of the small part of the array. For loop will continue its iteration in the similar manner and will swap the lower values to the left part of the array and will not swap the larger values so they will remain in the right

part of the array. When this for will end, after it the function will swap the value at the i index with the value at the pivot and i index will be returned.

2. quicksort function

This function is recursively sorting the right and left part of an array through partitioning the array.

```
41  void quickSort(int array[], int low, int high) {
42    if (low < high) {
43
44      int part = partition(array, low, high);
45
46      // recursive call on the left of pivot
47      quickSort(array, low, part - 1);
48
49      // recursive call on the right of pivot
50      quickSort(array, part + 1, high);
51    }
52  }
```

Quicksort function is accepting an array, it's starting and ending index as function parameters. First it is checking whether the value of lower element is greater then lower, than it will return. If not then it will call the partition function and give it the value of the stating index and the last index. This will return the pivot index.

In the next recursive call, the starting index and the pivot index will be sent as parameters to the recursion call. Then again, the quick sort function will run and follow the same steps.

The last recursive call will send next element of pivot element and last element of the array. Then again, the quicksort function will run and find the pivot and continue until the low index is less the high index.

3. Main Function:

Main function is taking the unsorted array as input from the user and passing to the quicksort function which returns the sorted array. The sorted array is being displayed.

2.1 Test Cases

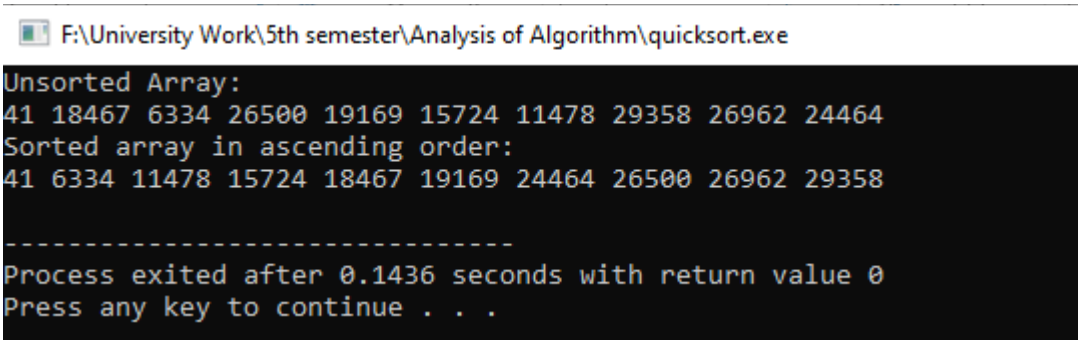
1. Test Case 1 for 10 elements:

- Unsorted Array:

  41 18467 6334 26500 19169 15724 11478 29358 26962 24464

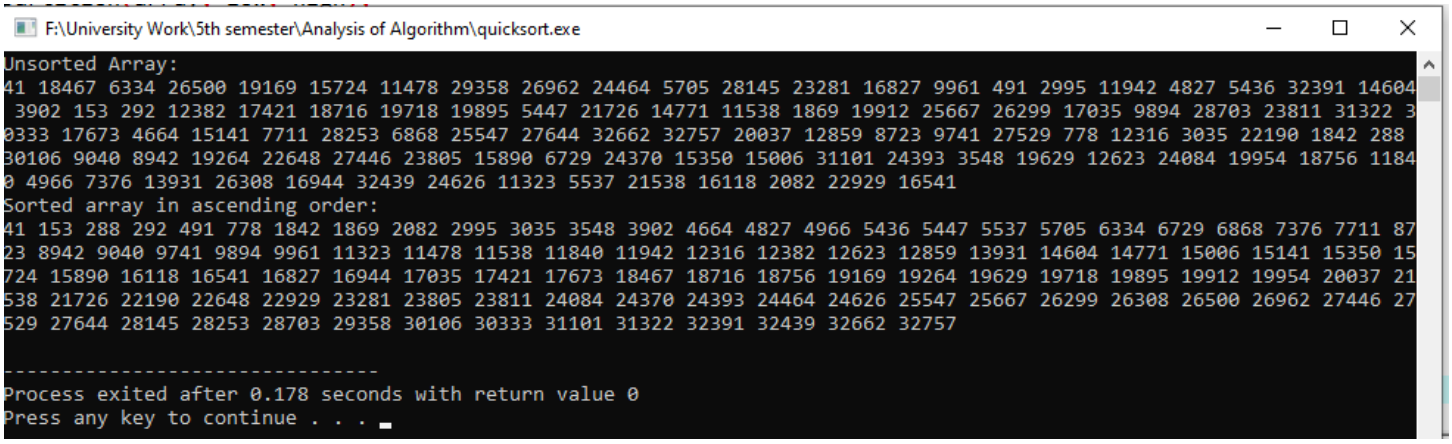
- Sorted array in ascending order:

  41 6334 11478 15724 18467 19169 24464 26500 26962 29358



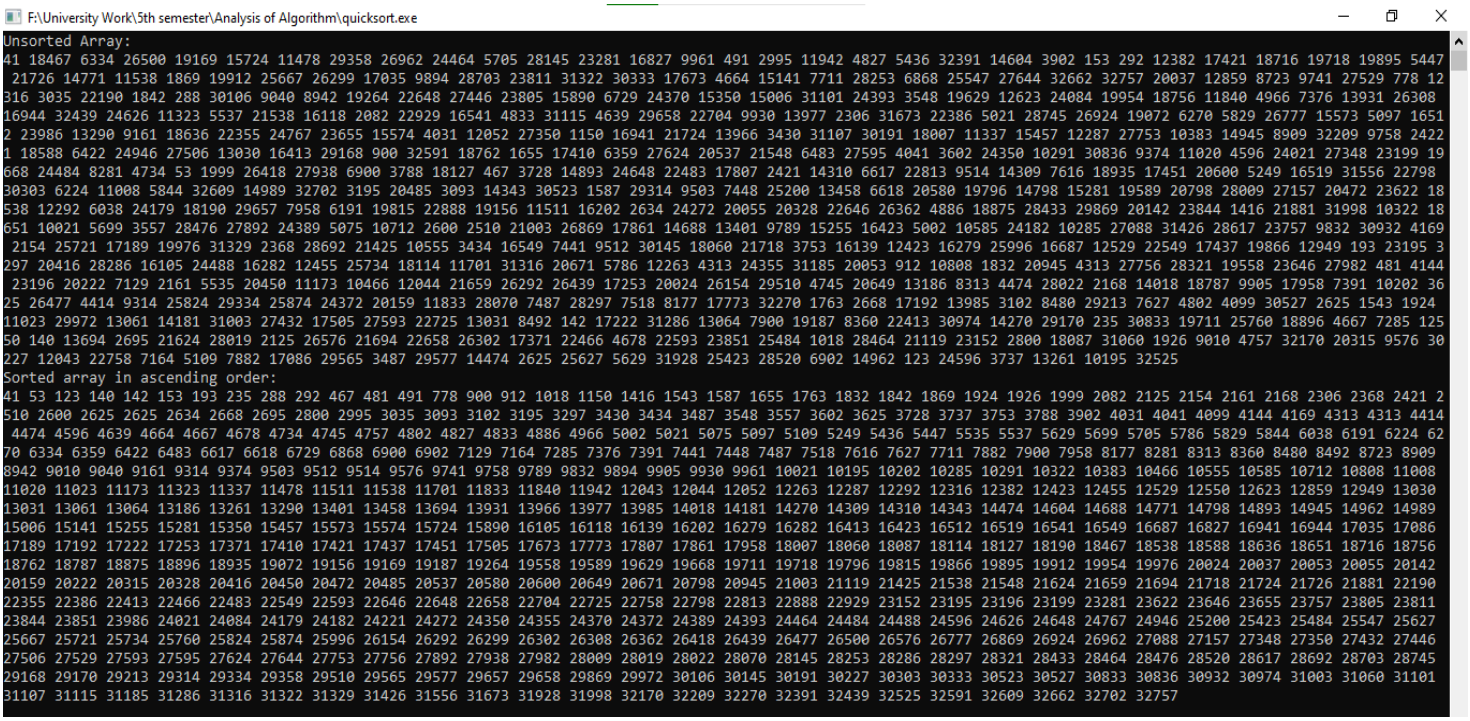
Test Case 2 for 100 elements:

Given elements were 100 that were randomly generated by the function and after the heap sort algorithm runs all the elements were sorted correctly in the order as see in the below figure.



Test Case 3 for 500 elements:

The algorithm correctly sorted an array of the length 500.



2.2 Procedure for the Random Input Generation

I am using the rand() function for generation of all the random inputs which is the part of the stdlib of the c++. Then for loop is being used to iterative over the n times to generate random input over the given range which is being stored in the array. The random inputs are of different length to check the efficiency of the algorithm.

```
57    int n = 100;
58    int data[n];
59
60    for(int i = 0;i<n;i++)
61        data[i] = rand();
62
63    int size = sizeof(data) / sizeof(data[0]);
```

2.3 Comparison of Time and Memory Results

1. When the Input size was 50, then the Time taken for the compilation was 1.08 seconds and size of the exe file is 3 MB.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\quicksort.exe
- Output Size: 3.07474994659424 MiB
- Compilation Time: 1.08s
```

2. When the Input size is 100 then the compilation time has increased to 1.05 seconds and the memory size of the compiled file is same.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\quicksort.exe
- Output Size: 3.07474994659424 MiB
- Compilation Time: 1.05s
```

3. When the input size is 500 elements time taken to compile has increased to 1.09 seconds and memory consumed by the output file is same

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\quicksort.exe
- Output Size: 3.07474994659424 MiB
- Compilation Time: 1.09s
```

The algorithm is taking the constant memory but Time for the compiling is increasing.

3. Radix Sort

Radix sort is used to sort the elements by comparing them digits by digits. Radix sort has the following functions:

1. **getMax() Function**

   This function is used to get the maximum value of the array and return it to the calling function.

```
 4   int getMax(int arr[], int n)
 5   {
 6       int mx = arr[0];
 7       for (int i = 1; i < n; i++)
 8           if (arr[i] > mx)
 9               mx = arr[i];
10       return mx;
11   }
```

2. **CountSort () Function**

   This function is used to sort the elements based on the unit place digits. It is using the counting sort to the digits at each significant place.

```
13   void countSort(int arr[], int n, int exp)
14   {
15       int output[n];
16       int i, count[10] = { 0 };
17
18       for (i = 0; i < n; i++)
19           count[(arr[i] / exp) % 10]++;
20
21       for (i = 1; i < 10; i++)
22           count[i] += count[i - 1];
23
24       for (i = n - 1; i >= 0; i--) {
25           output[count[(arr[i] / exp) % 10] - 1] = arr[i];
26           count[(arr[i] / exp) % 10]--;
27       }
28
29       for (i = 0; i < n; i++)
30           arr[i] = output[i];
31   }
```

The function is taking the array arr, its size n and the element to sort based on its digit place as indicated by the exp. The output array stores the sorted elements and the count array stores the count of occurrences of each digit.

Then the first fort loop is iterating through the array to find an occurrence of digit at the exp position in the arr array and storing it in the count array.

Then in the next for loop, it is adjusting the count array to get the actual position of the elements. It represents the correct position where the element should appear.

In the for loop at the line 24, it is building the output array by placing elements at correct positions based on their digits and at last the result is being stored in the output array.

3. radixsort () Function

```
32
33  void radixsort(int arr[], int n)
34  {
35      int m = getMax(arr, n);
36      for (int exp = 1; m / exp > 0; exp *= 10)
37          countSort(arr, n, exp);
38  }
39
```

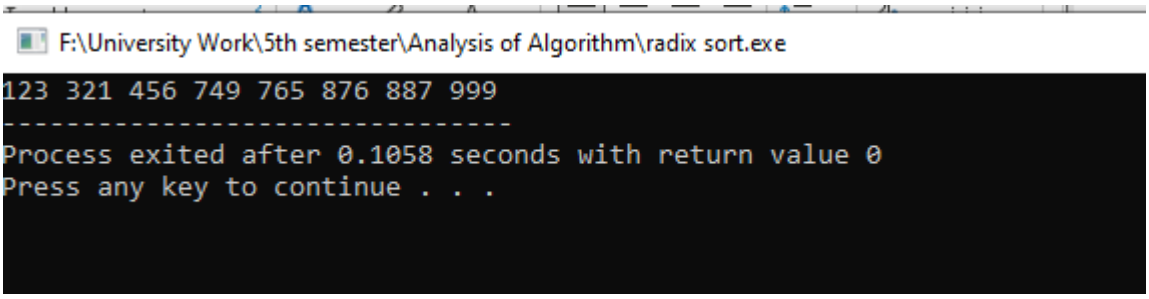
This function is accepting the array and the length of the array as the parameters. This loop iterates through each digit place from the least significant digit to the most significant digit of the numbers in the array. The for loop initializes an exp variable to 1 and continues until the maximum element divided by exp becomes zero. For each iteration, countsort function is being called which is responsible for sorting the elements based on digits at the current exp value.

3.2 Test Cases

Test Case 1:

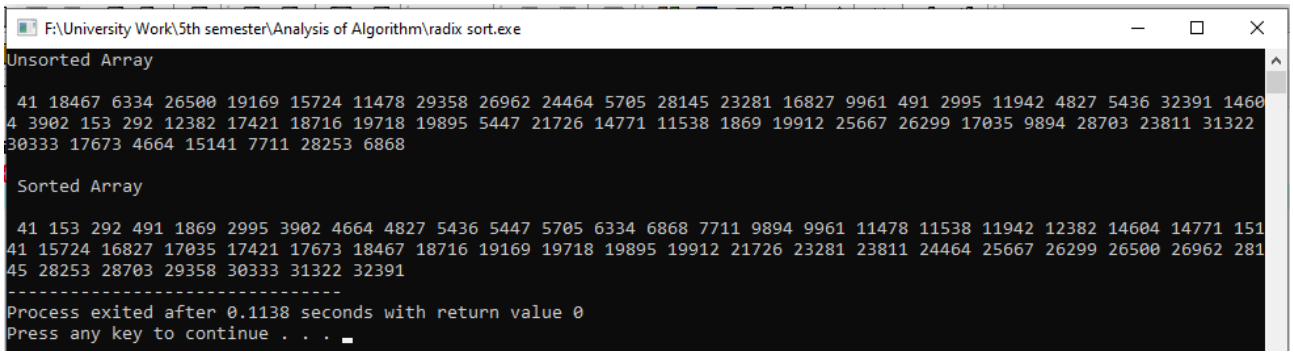
Input Array : 999,765,887,123,876,456,321,749

Output Array : 123 321 456 749 765 876 887 999



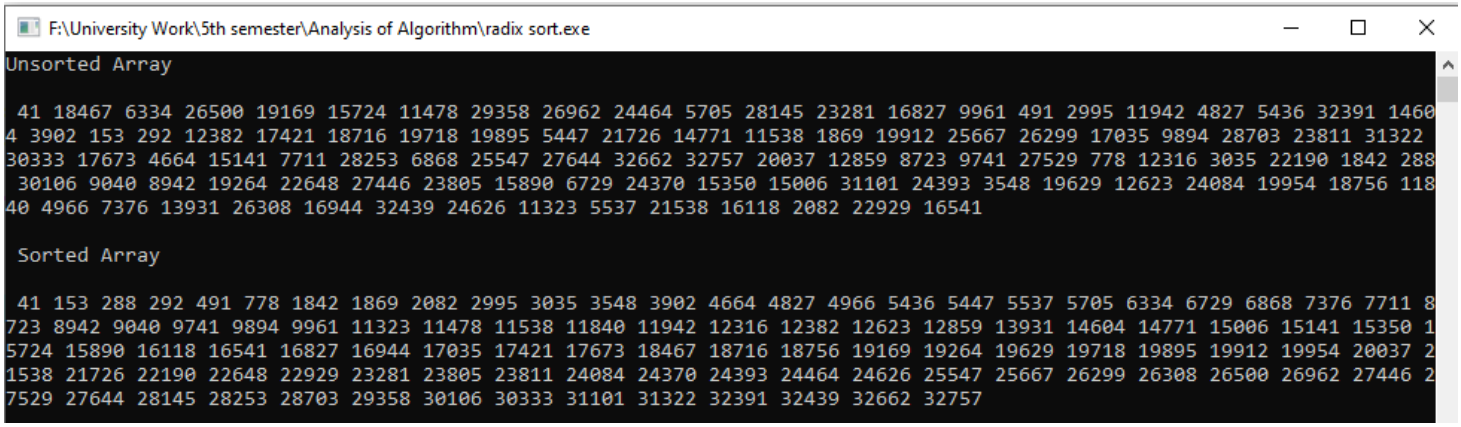
Test Case 2:

Input Array was length of 50. The digits in the input were of different size, but the code sorted it very well.



Test Case 3:

Now the Input Array size is 500. Numbers in the array have number of digits. Radix sort code has sorted all the elements of the array correctly.



3.3 Procedure for Random Input Generation

Radix sort code is using the rand() function of the stdlib library to get the random inputs of any range with random number of digits. The function is generating multiple small and large numbers as seen in the above test cases. All the test cases were generated with this function.

```
46  int main()
47  {
48      int n = 50;
49      int arr[n];
50
51      for (int i = 0;i<n;i++)
52          arr[i] = rand();
53
54      printf("Unsorted Array \n\n ");
55      print(arr, n);
```

3.4 Memory and Time Comparison Results

1. When the size of the inputs is 50 elements then the compiler is taking time of 0.98 seconds and compiles exe size is 3 MB.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\radix sort.exe
- Output Size: 3.1024808883667 MiB
- Compilation Time: 0.98s
```

2. When the size of inputs has increased to 100, then memory space remain same but compilation time has increased to 1.63 seconds.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\radix sort.exe
- Output Size: 3.1024808883667 MiB
- Compilation Time: 1.63s
```

3. Now the size of input is 500. The size of the compiled exe file is same but the compilation time has increased to the 1.02 seconds.

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: F:\University Work\5th semester\Analysis of Algorithm\radix sort.exe
- Output Size: 3.1024808883667 MiB
- Compilation Time: 1.02s
```

4 Conclusion and Recommendation of best Algorithm

Based on the input size of 5000, heap sort was very efficient. Because it offers the consistent average time complexity of O(nlogn). It is proved that the Heap sort performs very well over the large data set. For small data all of them perform very well. And It does not take extra memory, so where we have limited memory, heap sort will be the best choice. As well as, heap sort is stable.