

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.

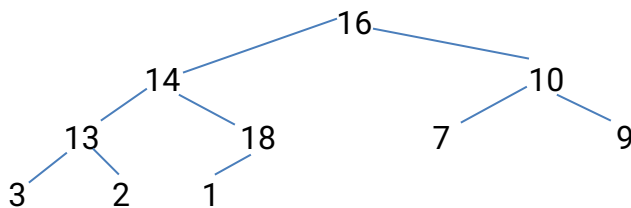
```

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

```

Since the heapify assumes that the root of the tree is already max-heap so first for loop is starting from index $i = \text{array}/2 - 1$. $\text{array}/2 - 1$ 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 $i = \text{array}/2 - 1$ gives us $i = 4$.



$n=10$

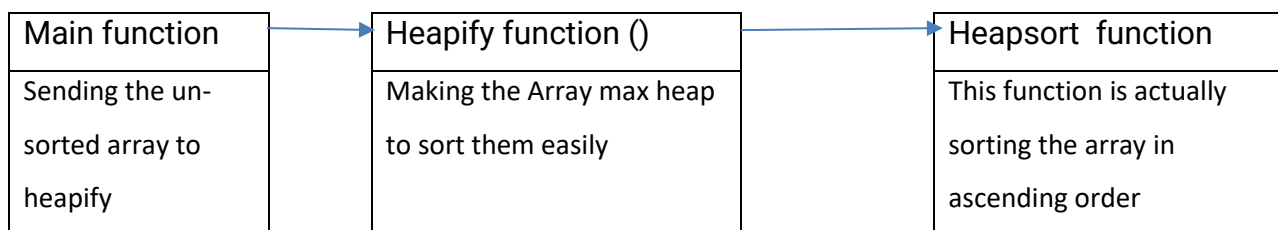
$i = n/2 - 1 = 4$.

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 leaf 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 :



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

```

F:\University Work\5th semester\Analysis of Algorithm\HEAP SORTING.exe
Initial Array:
41 67 34 0 69 24 78 58 62 64

Sorted array is
0 24 34 41 58 62 64 67 69 78

-----
Process exited after 0.1568 seconds with return value 0
Press any key to continue . . .
  
```

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
Initial 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
Initial 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
Initial 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 than lower, than it will return. If not then it will call the partition function and give it the value of the starting 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

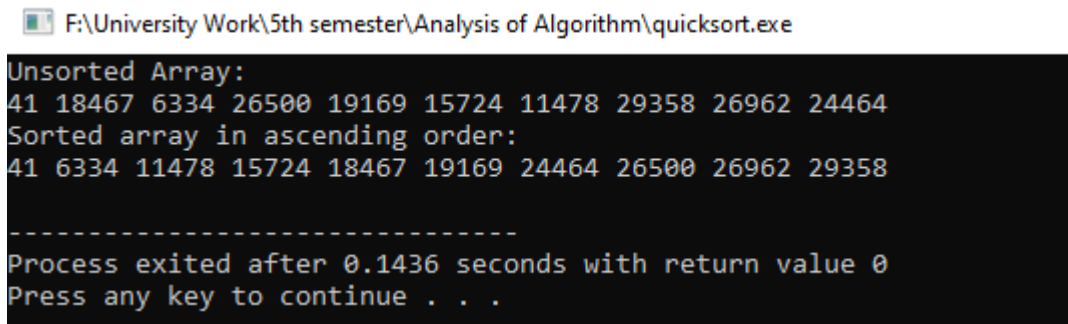
1. Test Case 1 for 10 elements:

- Unsorted Array:

41 18467 6334 26500 19169 15724 11478 29358 26962 24464

- Sorted array in ascending order:

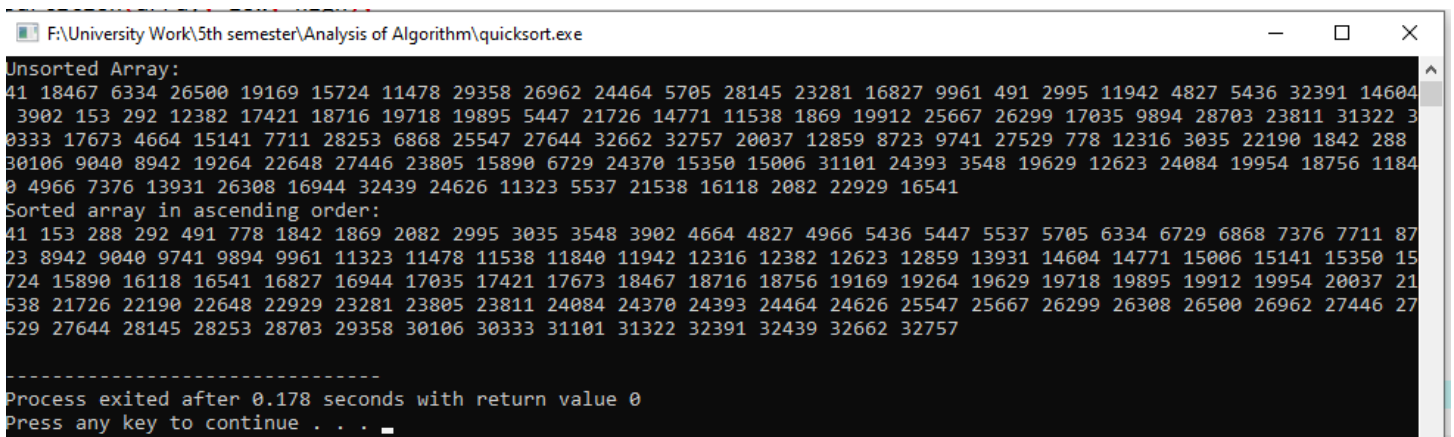
41 6334 11478 15724 18467 19169 24464 26500 26962 29358



```
F:\University Work\5th semester\Analysis of Algorithm\quicksort.exe
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
-----
Process exited after 0.1436 seconds with return value 0
Press any key to continue . . .
```

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.



```
F:\University Work\5th semester\Analysis of Algorithm\quicksort.exe
Unsorted 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 3
0333 17673 4664 15141 7711 28253 6868 25547 27644 32662 32757 20037 12859 8723 9741 27529 778 12316 3035 22190 1842 288
30106 9040 8942 19264 22648 27446 23805 15890 6729 24370 15350 15006 31101 24393 3548 19629 12623 24084 19954 18756 1184
0 4966 7376 13931 26308 16944 32439 24626 11323 5537 21538 16118 2082 22929 16541
Sorted array in ascending order:
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 87
23 8942 9040 9741 9894 9961 11323 11478 11538 11840 11942 12316 12382 12623 12859 13931 14604 14771 15006 15141 15350 15
724 15890 16118 16541 16827 16944 17035 17421 17673 18467 18716 18756 19169 19264 19629 19718 19895 19912 19954 20037 21
538 21726 22190 22648 22929 23281 23805 23811 24084 24370 24393 24464 24626 25547 25667 26299 26308 26500 26962 27446 27
529 27644 28145 28253 28703 29358 30106 30333 31101 31322 32391 32439 32662 32757
-----
Process exited after 0.178 seconds with return value 0
Press any key to continue . . .
```

Test Case 3 for 500 elements:

The algorithm correctly sorted an array of the length 500.

F:\University Work\5th semester\Analysis of Algorithm\quicksort.exe

```

Unsorted 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 4833 31115 4639 29658 22704 9930 13977 2306 31673 22386 5021 28745 26924 19072 6270 5829 26777 15573 5097 1651
2 23986 13290 9161 18636 22355 24767 23655 15574 4031 12052 27350 1150 16941 21724 13966 3430 31107 30191 18007 11337 15457 12287 27753 10383 14945 8909 32209 9758 2422
1 18588 6422 24946 27506 13030 16413 29168 900 32591 18762 1655 17410 6359 27624 20537 21548 6483 27595 4041 3602 24350 10291 30836 9374 11020 4596 24021 27348 23199 19
668 24484 8281 4734 53 1999 26418 27938 6900 3788 18127 467 3728 14893 24648 22483 17807 2421 14310 6617 22813 9514 14309 7616 18935 17451 20600 5249 16519 31556 22798
30303 6224 11008 5844 32609 14989 32702 3195 20485 3093 14343 30523 1587 29314 9503 7448 25200 13458 6618 20580 19796 14798 15281 19589 20798 28009 27157 20472 23622 18
538 12292 6038 24179 18190 29657 7958 6191 19815 22888 19156 11511 16202 2634 24272 20055 20328 22646 26362 4886 18875 28433 29869 20142 23844 1416 21881 31998 10322 18
651 10021 5699 3557 28476 27892 24389 5075 10712 2600 2510 21003 26869 17861 14688 13401 9789 15255 16423 5002 10585 24182 10285 27088 31426 28617 23757 9832 30932 4169
2154 25721 17189 19976 31329 2368 28692 21425 10555 3434 16549 7441 9512 30145 18060 21718 3753 16139 12423 16279 25996 16687 12529 22549 17437 19866 12949 193 23105 3
297 20416 28286 16105 24488 16282 12455 25734 18114 11701 31316 20671 5786 12263 4313 24355 31185 20053 912 10808 1832 20945 4313 27756 28321 19558 23646 27982 481 4144
23196 20222 7129 2161 5535 20450 11173 10466 12044 21659 26292 26439 17253 20024 26154 29510 4745 20649 13186 8313 4474 28022 2168 14018 18787 9905 17958 7391 10202 36
25 26477 4414 9314 25824 29334 25874 24372 20159 11833 28070 7487 28297 7518 8177 17773 32270 1763 2668 17192 13985 3102 8480 29213 7627 4802 4099 30527 2625 1543 1924
11023 29972 13061 14181 31003 27432 17505 27593 22725 13031 8492 142 17222 31286 13064 7900 19187 8360 22413 30974 14270 29170 235 30833 19711 25760 18896 4667 7285 125
50 140 13694 2695 21624 28019 2125 26576 21694 22658 26302 17371 22466 4678 22593 23851 25484 1018 28464 21119 23152 2800 18087 31060 1926 9010 4757 32170 20315 9576 30
227 12043 22758 7164 5100 7882 17086 29565 3487 29577 14474 2625 25627 5629 31928 25423 28520 6902 14962 123 24596 3737 13261 10195 32525
Sorted array in ascending order:
41 53 123 140 142 153 193 235 288 292 467 481 491 778 900 912 1018 1150 1416 1543 1587 1655 1763 1832 1842 1869 1924 1926 1999 2082 2125 2154 2161 2168 2306 2368 2421 2
510 2600 2625 2625 2634 2668 2695 2800 2995 3035 3093 3102 3195 3297 3430 3434 3487 3548 3557 3602 3625 3728 3737 3753 3788 3902 4031 4041 4099 4144 4169 4313 4313 4414
4474 4596 4639 4664 4667 4678 4734 4745 4757 4802 4827 4833 4886 4966 5002 5021 5075 5097 5109 5249 5436 5447 5535 5537 5629 5699 5705 5786 5829 5844 6038 6191 6224 62
70 6334 6359 6422 6483 6617 6618 6729 6868 6900 6902 7129 7164 7285 7376 7391 7441 7448 7487 7518 7616 7627 7711 7882 7900 7958 8177 8281 8313 8360 8480 8492 8723 8909
8942 9010 9040 9161 9314 9374 9503 9512 9514 9576 9741 9758 9789 9832 9894 9905 9930 9961 10021 10195 10202 10285 10291 10322 10383 10466 10555 10585 10712 10808 11008
11020 11023 11173 11323 11337 11478 11511 11538 11701 11833 11840 11942 12043 12044 12052 12263 12287 12292 12316 12382 12423 12455 12529 12550 12623 12859 12949 13030
13031 13061 13064 13186 13261 13290 13401 13458 13694 13931 13966 13977 13985 14018 14181 14270 14309 14310 14343 14474 14604 14688 14771 14798 14893 14945 14962 14989
15006 15141 15255 15281 15350 15457 15573 15574 15724 15890 16105 16118 16139 16202 16279 16282 16413 16423 16512 16519 16541 16549 16687 16827 16941 16944 17035 17086
17189 17192 17222 17253 17371 17410 17421 17437 17451 17505 17673 17773 17807 17861 17958 18007 18060 18087 18114 18127 18190 18467 18538 18588 18636 18651 18716 18756
18762 18787 18875 18896 18935 19072 19156 19169 19187 19264 19558 19589 19629 19668 19711 19718 19796 19815 19866 19895 19912 19954 19976 20024 20037 20053 20055 20142
20159 20222 20315 20328 20416 20450 20472 20485 20537 20580 20600 20649 20671 20798 20945 21003 21119 21425 21538 21548 21624 21659 21694 21718 21724 21726 21881 22190
22355 22386 22413 22466 22483 22549 22593 22646 22648 22658 22704 22725 22758 22798 22813 22888 22929 23152 23195 23196 23199 23281 23622 23646 23655 23757 23805 23811
23844 23851 23986 24021 24084 24179 24182 24221 24272 24350 24355 24370 24372 24389 24393 24464 24484 24488 24596 24626 24648 24767 24946 25200 25423 25484 25547 25627
25667 25721 25734 25760 25824 25874 25996 26154 26292 26299 26302 26308 26362 26418 26439 26477 26500 26576 26777 26869 26924 26962 27088 27157 27348 27350 27432 27446
27506 27529 27593 27595 27624 27644 27753 27756 27892 27938 27982 28009 28019 28022 28070 28145 28253 28286 28297 28321 28433 28464 28476 28520 28617 28692 28703 28745
29168 29170 29213 29314 29334 29358 29510 29565 29577 29657 29658 29869 29972 30106 30145 30191 30227 30303 30333 30523 30527 30833 30836 30932 30974 31003 31060 31101
31107 31115 31185 31286 31316 31322 31329 31426 31556 31673 31928 31998 32170 32209 32270 32391 32439 32525 32591 32609 32662 32702 32757

```

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 for 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

```

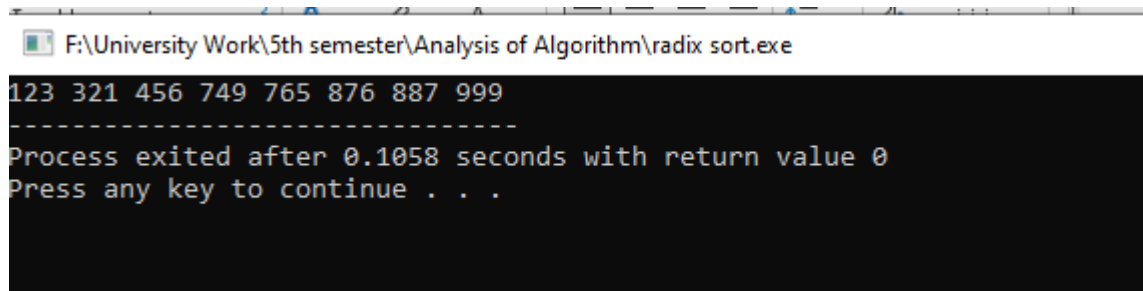
This function is accepting the array and the length of the array as the parameters. This loop iterates through each digit place from the least significant digit to the most significant digit of the numbers in the array. The for loop initializes an exp variable to 1 and continues until the maximum element 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

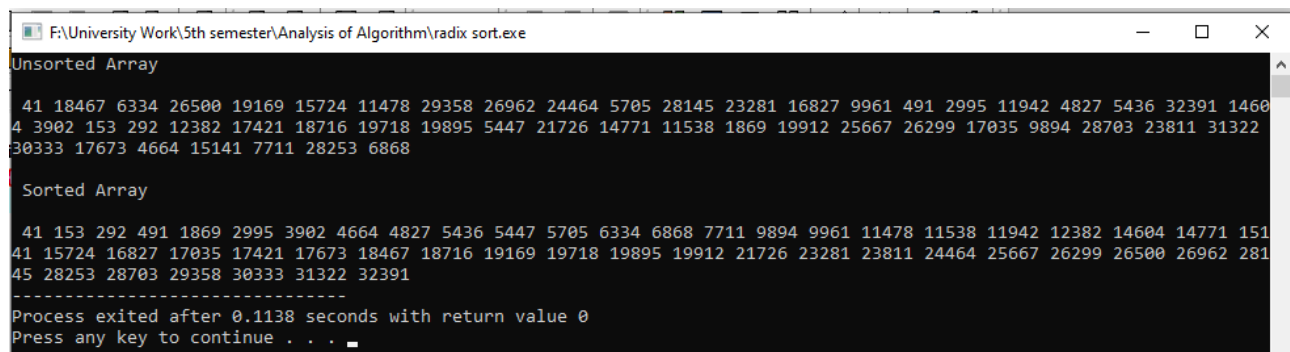
Output Array : 123 321 456 749 765 876 887 999



```
F:\University Work\5th semester\Analysis of Algorithm\radix sort.exe
123 321 456 749 765 876 887 999
-----
Process exited after 0.1058 seconds with return value 0
Press any key to continue . . .
```

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.



```
F:\University Work\5th semester\Analysis of Algorithm\radix sort.exe
Unsorted Array
41 18467 6334 26500 19169 15724 11478 29358 26962 24464 5705 28145 23281 16827 9961 491 2995 11942 4827 5436 32391 14604
4 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
Sorted Array
41 153 292 491 1869 2995 3902 4664 4827 5436 5447 5705 6334 6868 7711 9894 9961 11478 11538 11942 12382 14604 14771 15141
41 15724 16827 17035 17421 17673 18467 18716 19169 19718 19895 19912 21726 23281 23811 24464 25667 26299 26500 26962 28145
45 28253 28703 29358 30333 31322 32391
-----
Process exited after 0.1138 seconds with return value 0
Press any key to continue . . .
```

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.

```

F:\University Work\5th semester\Analysis of Algorithm\radix sort.exe
Unsorted Array
41 18467 6334 26500 19169 15724 11478 29358 26962 24464 5705 28145 23281 16827 9961 491 2995 11942 4827 5436 32391 1460
4 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 12316 3035 22190 1842 288
30106 9040 8942 19264 22648 27446 23805 15890 6729 24370 15350 15006 31101 24393 3548 19629 12623 24084 19954 18756 118
40 4966 7376 13931 26308 16944 32439 24626 11323 5537 21538 16118 2082 22929 16541

Sorted Array
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 8
723 8942 9040 9741 9894 9961 11323 11478 11538 11840 11942 12316 12382 12623 12859 13931 14604 14771 15006 15141 15350 1
5724 15890 16118 16541 16827 16944 17035 17421 17673 18467 18716 18756 19169 19264 19629 19718 19895 19912 19954 20037 2
1538 21726 22190 22648 22929 23281 23805 23811 24084 24370 24393 24464 24626 25547 25667 26299 26308 26500 26962 27446 2
7529 27644 28145 28253 28703 29358 30106 30333 31101 31322 32391 32439 32662 32757

```

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 $O(n \log n)$. 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.