

Analysis of Computer Algorithms

Assignment 02:

Outputs screenshots file and Explanation for algorithms

Ganesh Gundekarla

11700551

We had implemented four algorithms:

Process on how implementation is done for MacOS:

Required apps:

- a. FileZilla.
- b. Cisco any connect client. (vpn).
- c. Terminal to access cell machine.

Step one: Using vpn to connect to the unt server:

Here, we had used vpn to connect to the vpn.unt.edu server, to access the cell machine which is the primary step in order to access the cell machine in ubuntu.

Step two: Connection to a cell machine in the unt server .

In terminal we gotta type :

```
ssh gg0640@ CELL06-CSE.ENG.UNT.EDU
```

This will allow us to access a cell machine in the unt server by using our Euid login credentials. After some time, the server gets initiated.

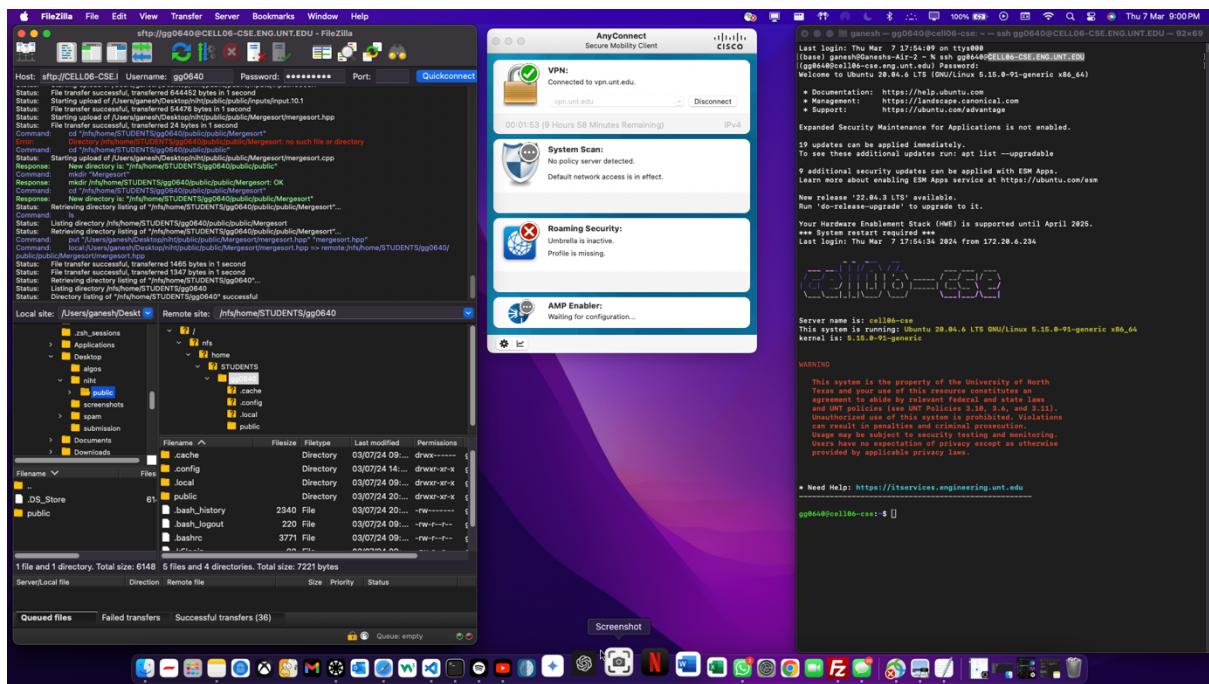
Step 03: Connection of a ftp file transfer protocol with the server .

This step is important in order to upload the algorithm files we had prepared to the cell machine to set up and try out with multiple inputs.

- a. Here, in the File zilla app we need to set the HOST =" CELL06-CSE.ENG.UNT.EDU"
- b. Username: gg0640
- c. Password: my credentials

With this the connection to the server will be complete. With this we will be able to set up a file sharing connection with the server.

The below screenshot shows the successful connection.



We had implemented four algorithms:

Also mentioned hpp files just for reference

Insertion Sort Implementation:

Here, insertion sort is being implemented in the given missing code:

Then, I compiled it in the server using the command :

g++ insertionSort.cpp




```

5
6 int main(int argc, char *argv[])
7 {
8
9     // i numbers in a file
10    vector<int> A;
11    if (argc != 2)
12    {
13        cout << "Provide an input file as argument";
14    }
15    else
16    {
17        FILE *file = fopen(argv[1], "r");
18        if (file == 0)
19        {
20            cout << "ERROR: file does not exist" << endl;
21            return -1;
22        }
23        else
24        {
25            int x;
26            fscanf(file, "%d", &x);
27            while (!feof(file))
28            {
29                A.push_back(x);
30                fscanf(file, "%d", &x);
31            }
32            fclose(file);
33        }
34    }
35

```

Implementation:

```

// Implementation here
int n = A.size();
for (int i = 1; i < n; ++i)
{
    int key = A[i];
    int j = i - 1;
    while (j >= 0 && A[j] > key)
    {
        A[j + 1] = A[j];
        j = j - 1;
    }
    A[j + 1] = key;
}

if (n <= 10)
{
    cout << "And this is the sorted output:" << endl;
    for (int i = 0; i < n; ++i)
    {
        cout << A[i] << " ";
    }
    cout << endl;
}
else
{
    cout << "Sorted output is too large to display" << endl;
}

cout << "Checking if the output is actually sorted ..." << endl;
bool sorted = true;
for (int i = 1; i < n; ++i)
{
    if (A[i - 1] > A[i])
    {
        cout << "Output is NOT SORTED: " << A[i - 1] << " is greater than " << A[i] << "(index " << i << ")" << endl;
        sorted = false;
    }
}
if (sorted)
{
    cout << "\tThe output is sorted" << endl;
}
cout << "=====" << endl;
<< endl;
<< endl;
<< endl;
return 0;

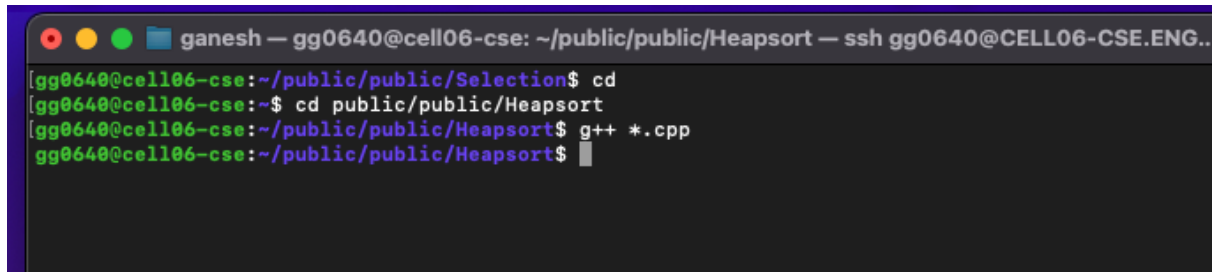
```

Screenshot

Heapsort Implementation:

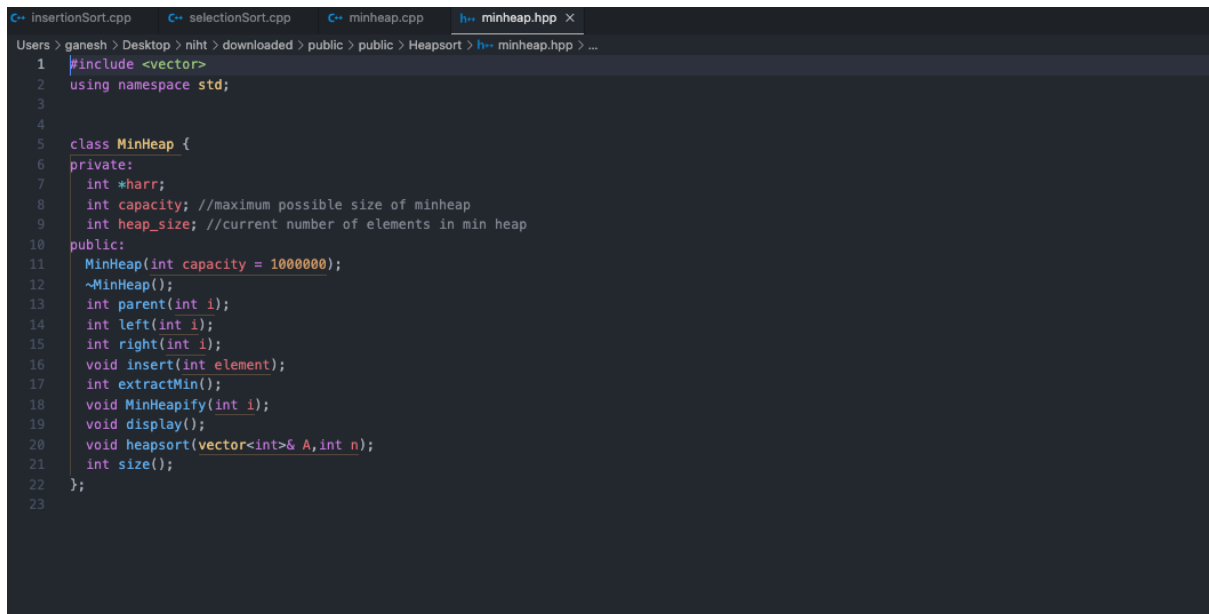
The code has been placed the implemented are and then compiled it using:

g++ *.cpp



```
ganesh — gg0640@cell06-cse: ~/public/public/Heapsort — ssh gg0640@CELL06-CSE.ENG..
[gg0640@cell06-cse:~/public/public/Selection$ cd
[gg0640@cell06-cse:~$ cd public/public/Heapsort
[gg0640@cell06-cse:~/public/public/Heapsort$ g++ *.cpp
[gg0640@cell06-cse:~/public/public/Heapsort$
```

Implementation:



```
InsertionSort.cpp selectionSort.cpp minheap.cpp minheap.hpp x
Users > ganesh > Desktop > niht > downloaded > public > public > Heapsort > h++ minheap.hpp > ...
1  #include <vector>
2  using namespace std;
3
4
5  class MinHeap {
6  private:
7      int *harr;
8      int capacity; //maximum possible size of minheap
9      int heap_size; //current number of elements in min heap
10 public:
11     MinHeap(int capacity = 1000000);
12     ~MinHeap();
13     int parent(int i);
14     int left(int i);
15     int right(int i);
16     void insert(int element);
17     int extractMin();
18     void MinHeapify(int i);
19     void display();
20     void heapsort(vector<int>& A,int n);
21     int size();
22 };
23
```

.cpp files

```

1  #include "minHeap.hpp"
2  #include <climits>
3  #include <stdio.h>
4  #include <iostream>
5
6  void swap(int *x, int *y){
7      int temp = *x;
8      *x = *y;
9      *y = temp;
10 }
11
12 MinHeap::MinHeap(int size) {
13     heap_size = 0;
14     capacity = size;
15     harr = new int[size];
16 }
17 MinHeap::~MinHeap() {}
18
19 int MinHeap::left(int parent) {
20     int i = parent*2 + 1;
21     return (i < heap_size) ? i: -1;
22 }
23
24 int MinHeap::right(int parent) {
25     int i = parent*2 + 2;
26     return (i < heap_size) ? i:-1;
27 }
28
29 int MinHeap::parent(int child) {
30     if (child != 0) {
31         int i = (child - 1) >> 1;
32         return i;
33     }
34     return -1;
35 }
36

```

```

36
37 int MinHeap::size() { return heap_size; }
38
39 void MinHeap::insert(int element) {
40     if (heap_size == capacity) {
41         cout << "Cannot insert key" << endl;
42         return;
43     }
44
45     // Insert the new key at the end
46     int i = heap_size;
47     harr[i] = element;
48     heap_size++;
49
50     // Fix the min heap property if it is violated
51     while (i != 0 && harr[parent(i)] > harr[i]) {
52         swap(&harr[i], &harr[parent(i)]);
53         i = parent(i);
54     }
55 }
56
57 int MinHeap::extractMin() {
58     if (heap_size <= 0)
59     {
60         cout<<"There are no elements in the heap\n"<<endl;
61         return INT_MAX;
62     }
63
64     if (heap_size == 1)
65     {
66         heap_size--;
67         return harr[0];
68     }
69
70     int root = harr[0];
71     harr[0] = harr[--heap_size];
72     MinHeapify(0);
73
74     return root;
75 }
76
77 }
78

```

```

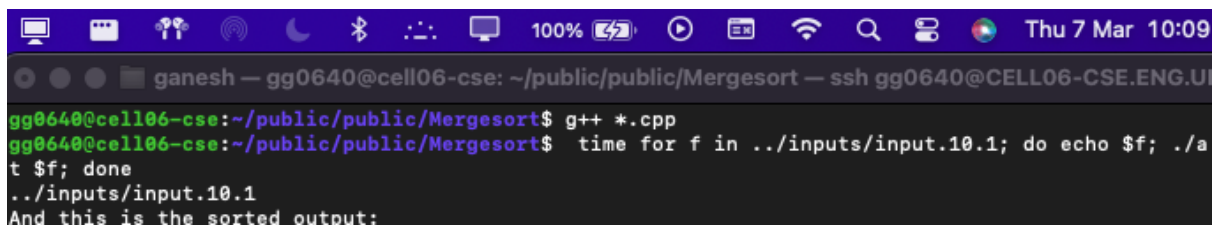
67 // int MinHeap::extractMin() {
68 // }
69
70 void MinHeap::MinHeapify(int i){
71     int l = left(i);
72     int r = right(i);
73     int smallest = i;
74     if (l < heap_size && l>0 && harr[l] < harr[i])
75         smallest = l;
76     if (r < heap_size && r>0 && harr[r] < harr[smallest])
77         smallest = r;
78     if (smallest != i)
79     {
80         swap(&harr[i], &harr[smallest]);
81         MinHeapify(smallest);
82     }
83 }
84
85 void MinHeap::display() {
86     cout<<"MinHeap:- ";
87     cout << heap_size << endl;
88     for(int i = 0; i < heap_size; i++){
89         cout << harr[i] << " ";
90     }
91     cout << endl;
92 }
93
94 void MinHeap::heapsort(vector<int>& A,int n) {
95     MinHeap *hp= new MinHeap(n);
96
97     // Inserting elements from vector to heap
98     for(int i=0;i<n;i++)
99         hp->insert(A[i]);
100
101     for(int i=hp->heap_size-1;i>=0;i--)
102     {
103         A[i]=hp->extractMin();
104     }
105
106     n--;
107
108     delete hp;
109 }

```

Merge sort implementation here:

Code has been written in the implementation part and compiled using:

g ++ *.cpp



```

ganesh — gg0640@cell06-cse: ~/public/public/Mergesort — ssh gg0640@CELL06-CSE.ENG.U
gg0640@cell06-cse:~/public/public/Mergesort$ g++ *.cpp
gg0640@cell06-cse:~/public/public/Mergesort$ time for f in ../inputs/input.10.1; do echo $f; ./a
t $f; done
../inputs/input.10.1
And this is the sorted output:

```

Implementation: .cpp file:

```

    }
}

int n = A.size();
Mergesort(A, 0, n - 1);

if (n <= 10)
{
    cout << "And this is the sorted output:" << endl;
    for (int i = 0; i < n; ++i)
    {
        cout << A[i] << " ";
    }
    cout << endl;
}
else
{
    cout << "Sorted output is too large to display" << endl;
}

cout << "Checking if the output is actually sorted ..." << endl;
bool sorted = true;
for (int i = 1; i < n; ++i)
{
    if (A[i - 1] > A[i])
    {
        cout << "Output is NOT SORTED: " << A[i - 1] << " is greater than " << A[i] << "(index " << i << ")" << endl;
        sorted = false;
    }
}
if (sorted)
{
    cout << "\tThe output is sorted" << endl;
}
cout << "=====" << endl;
    << endl
    << endl
    << endl;
return 0;
}

```

Screenshot

.hpp code:

mersort.cpp x minheap.cpp minheap.hpp
eers > ganesh > Desktop > nint > downloaded > public > public > Mergesort > h++ mergesort.hpp > ...

```
1 #include <iostream>
2 #include <vector>
3 #include <climits>
4 #include <cmath>
5
6 using namespace std;
7
8 // Implement mergesort algorithm here.
9 // Merge two sorted subarrays A[l..m] and A[m+1..r]
10 void Merge(vector<int> &A, int l, int m, int r)
11 {
12     int n1 = m - l + 1;
13     int n2 = r - m;
14
15     // Create temporary arrays
16     vector<int> L(n1 + 1);
17     vector<int> R(n2 + 1);
18
19     // Copy data to temporary arrays L[] and R[]
20     for (int i = 0; i < n1; i++)
21         L[i] = A[l + i];
22     for (int j = 0; j < n2; j++)
23         R[j] = A[m + 1 + j];
24
25     // Merge the temporary arrays back into A[l..r]
26     int i = 0, j = 0, k = l;
27     while (i < n1 && j < n2)
28     {
29         if (L[i] <= R[j])
30         {
31             A[k] = L[i];
32             i++;
33         }
34         else
35         {
36             A[k] = R[j];
37             j++;
38         }
39         k++;
40     }
41
42     // Copy the remaining elements of L[], if there are any
43     while (i < n1)
44     {
45         A[k] = L[i];
46         i++;
47         k++;
48     }
49 }
```

```
58
59 // Main function to implement merge sort
60 void Mergesort(vector<int> &A, int l, int r)
61 {
62     if (l < r)
63     {
64         // Find the middle point
65         int m = l + (r - l) / 2;
66
67         // Sort first and second halves
68         Mergesort(A, l, m);
69         Mergesort(A, m + 1, r);
70
71         // Merge the sorted halves
72         Merge(A, l, m, r);
73     }
74 }
```

Explanation for four sort algorithms:

a. Insertion sort:

This code will be basically sorting a list of integers that are being read from a file.

1. The first step is reading input from the inputs folder and then are being stored in vector 'A'.

2. A loop is being executed and the algo is being implemented in that loop , here , for(int j=1;j<n;j++) , the each iteration will be done and every time the current element which is basically A[j] that is the one that is selected for insertion into the array which is a sorted one , which spans for A[0,.....j-1]
3. Here, then the element A[j] is being compared with elements in the sorted one which is a subarray compared from left to right which is done until the correct position for the index A[j] is being found.
4. Then the value A[j] is placed for the correct position in the sorted place in the array.
5. At the output line, after the sorting is being done, the program checks whether the output is really being sorted or not which results at the output.
6. This is basically done by iterating through each element if the element is greater than the next element, then it just simply prints at the output that the array is not sorted.

Time complexity in this case:

- a. The worst time complexity would be for this insertion sort is $O(n^2)$. the best scenario would be if the inputs given are already sorted which results in $O(n)$, we consider the worst case scenario but It can perform if the input is partially sorted or for a smaller inputs.
- b. Even the inputs get higher each time ,I think inputs doesn't have any significant impact if its based on a certain order of inputs .
- c. Space complexity would be $O(n)$.

b. Selection sort:

1. Here, the code will be reading the inputs from a file which is specified as to read from a specific inputs folder.
2. Then as usual, these are being stored in a vector A .
3. The selection sort starts with algorithm which is being implemented in a loop.
4. After each iteration, the algo will find the minimum element which is lying in the unsorted portion of the array and then starting from index I, which will be swapped with the element at index i .
5. Ultimately, the process will be building an effective sorted portion of array which is being done from left to right.

6. But the process for displaying output is a bit different, here , if the number of elements is less than or that equal to 10 , the sorted output will be displayed or else ,the number of elements greater than 10 will lead to displaying of 'sorter output is too larger to display ' in the command prompt .
7. This sorting algorithm also checks if the output array is being properly sorted or not.
8. By iteration through each element, if any condition , it finds any element is greater than the next one , it displays as a output of 'the array is not sorted'.
9. **Time complexity:**
 - a. The worst case would be $O(n^2)$ for selection sort, as it will be roughly swapping n swaps for $n-1$ comparisons and hence then overall time complexity is $O(n^2)$.
 - b. Space complexity would be same for the above as $O(n)$.

This sort is not efficient for large inputs unlike the merge sort or the quick sort.

Heap sort:

The code is implements a minheap along with addition of a heap sort algorithm using the minheap.

- a. The class minheap will be representing the data structure for the min heap data.
- b. Here, the class contains private variables harr which is a array that is used to store the elements of the heap . Here the capacity is defined as the maximum size of the heap and the 'heap_size' will be the currently stored elements in the heap.
- c. Here, it is having a both a constructor and destructor and also the parent , left , right functions to basically retrieve the indices of the parent , child respectively .
- d. The function insert is done to insert an. Element in the heap and extractmin is used to extract the minimum element from the heap.
- e. The func 'minheapify()' is used to maintain the heap which starts from the beginning of the index.
- f. The display function is done to display the elements of the heap .
- g. The 'heapsort' function is perform a heap sort on the vector which stores the elements.
- h. Here, the source file , will be containing the definitions of fucntions of the minheap that are being implemented.

- i. This basically works by sort the input vector by constructing a heap which then repeatedly extracts the min element from it every time .
- j. For functions the run time will be different which is in logarithmic values.
- k. Complexity:
 - a. Space complexity of minheap is $O(n)$ when n = maximum capacity .
 - b. So this code basically implements a minheap class which is done along together with a heap sort algorithm using the minheap.
 - c. Here, these have a logarithmic complexity which has a time complexity of $O(n \log n)$.
 - d. Very efficient for sorting large inputs.

Merge sort :

- a. Here the 'merge' function's job is to merge the two sorted arrays into a single one . This will be taking two parameters 'A' input one and the 'l' which is the left index of a first subarray and the middle index 'm ' and the index which is present to the right of subarray .
- b. Here , the 'merge sort' function goal is to divide the array recursively into half parts and this goes on until each of the element has only one element .
- c. This takes parameters 'a' with the left index of the sub array part and the r which is the right index of the present subarray.
- d. Here , the goal of 'main' function is to then read the file (input file) and store them into a vector termed as 'A'.
- e. The display of output goes the same as , if the number of elements is less than or equal to 10 , the sorted output is displayed , or if the inputs are greater it simply displays ' the sorted output cannot be displayed '
- f. This then comes to end by checking if the sorted output is actually sorted or not by iterating it again .

Time complexity :

Worst case time complexity: the worst is $O(n \log n)$

Since it recursively divides the array into half parts until each of the sub array is being left out with only one element , then merges it .hence it has $\log n$ levels done for n times.

Space complexity is $O(n)$.

Run times are given below , this are also included in the pdf
 “RUNTIMES.pdf”.

these times are provided below.

	10.1	10.2	10.3	10.4	10.5	Average
Insertion sort	0m0.004s	0m0.003s	0m0.003s	0m0.003s	0m0.004s	0m0.0034s
Selection sort	0m0.005s	0m0.004s	0m0.004s	0m0.004s	0m0.004s	0m0.0042s
Heapsort	0m0.004s	0m0.003s	0m0.003s	0m0.003s	0m0.003s	0m0.0032s
Mergesort	0m0.003s	0m0.004s	0m0.003s	0m0.003s	0m0.003s	0m0.0032s

This is input size 10

	100.1	100.2	100.3	100.4	100.5	Average
Insertion sort	0m0.003s	0m0.003s	0m0.004s	0m0.003s	0m0.003s	0m0.0032s
Selection sort	0m0.004s	0m0.003s	0m0.003s	0m0.004s	0m0.003s	0m0.0034s
Heapsort	0m0.003s	0m0.003s	0m0.003s	0m0.003s	0m0.003s	0m0.003s
Mergesort	0m0.003s	0m0.003s	0m0.003s	0m0.003s	0m0.003s	0m0.003s

This is input size 100

	1000.1	1000.2	1000.3	1000.4	1000.5	Average
Insertion sort	0m0.005s	0m0.005s	0m0.006s	0m0.006s	0m0.007s	0m0.0058s
Selection sort	0m0.005s	0m0.009s	0m0.006s	0m0.005s	0m0.005s	0m0.006s
Heapsort	0m0.004s	0m0.003s	0m0.005s	0m0.004s	0m0.004s	0m0.004s
Mergesort	0m0.004s	0m0.004s	0m0.004s	0m0.004s	0m0.004s	0m0.004s

This is input size 1000

	10000.1	10000.2	10000.3	10000.4	10000.5	Average
Insertion sort	0m0.232s	0m0.237s	0m0.184s	0m0.185s	0m0.276s	0m0.2228
Selection sort	0m0.186s	0m0.187s	0m0.198s	0m0.186s	0m0.186s	0m0.1886s
Heapsort	0m0.018s	0m0.018s	0m0.016s	0m0.008s	0m0.010s	0m0.014s
Mergesort	0m0.018s	0m0.024s	0m0.027s	0m0.025s	0m0.019s	0m0.0226s

This is input size 10000

	100000.1	100000.2	100000.3	100000.4	100000.5	Average
Insertion sort	0m18.277s	0m18.375s	0m18.157s	0m23.955s	0m20.999s	0m19.952s
Selection sort	0m18.217s	0m18.265s	0m18.101s	0m18.233s	0m18.098s	0m18.382s
Heapsort	0m0.132s	0m0.121s	0m0.161s	0m0.171s	0m0.167s	0m0.150s
Mergesort	0m0.201s	0m0.185s	0m0.176s	0m0.179s	0m0.176s	0m0.183s

This is input size 10,0000

	1000000.1
Insertion sort	40m4.539s
Selection sort	30m36.622s
Heapsort	0m0.729s
Mergesort	0m1.464s

This is input size 100,0000

SUMMARY:

Based on the run-times the heapsort and the merge sort are the quickest
 Afterall , with the insertion sort and the selection sort being the slowest.

Insertion sort has an run time of 40minutes having time complexity of

$O(n^2)$ which makes it inefficient for large inputs. This is because this involves shifting the element one after one till its alright and this makes it slow time in sorting. This is not recommended for large inputs for poor performance. **Selection sort** has a time complexity of $O(n^2)$ which also due to it repeatedly select the minimum element from the unsorted array and then swaps it with the sorted one making it also inefficient. Merge sort has a time complexity of $O(n \log n)$ which basically divides the array into a half part each time done recursively. sorts the half parts and finally merges them into a final one making it efficient for large inputs. Heapsort has a time complexity of $O(n \log n)$ which does it by building a heap structure and extracting the min from it and resulting in a sorted one.

Code output can found in each sort output files

Run time results can be found out in RUNTIMES pdf , (the given pdf for assignment and I filled the values in the same table)