

Data Structures 2

Assignment (1)

Sorting Techniques and Time Complexity

Name: Nouran Hisham ID: 6532

Name: Nourhan Waleed ID: 6609

Name: Fatema Moharam ID: 6655

Description of the program

Our program uses 6 different sorting techniques to sort randomly generated arrays of different sizes (**1000, 10000, 100000, 200000, 300000, 400000, 500000**) and then compare the time (milliseconds) taken by each sorting technique to sort these arrays.

The random arrays of different sizes are generated by a function called ***arraygenerator(int n)**, where **n** is the required number of elements in the array, and then a pointer to the first address of this generated array, to make sure that we always sort the original array not the one that has already been sorted by previous sorting methods, is sent to the time functions which call the corresponding sorting technique function to sort the array and calculate how much time it took each sorting algorithm to sort the given arrays in ascending order.

For example, the ***arraygenerator(int n)** generates an array of 100 elements then a pointer to the first address of this array is sent to ***timequicksort(const int original[],int n)** which from within calls **quicksort(int* arr,int low, int high)** that actually sorts the array and then ***timequicksort(const int original[],int n)** calculates the time taken by **quicksort(int* arr,int low, int high)** to sort the array and returns this value.

Our sorting techniques are:

- 3 $O(n^2)$ sorting algorithms which are: Selection Sort, Bubble Sort and Insertion Sort.
- 3 $O(n \log n)$ sorting algorithms which are: Merge Sort, Heap Sort and Quick Sort.

Selection Sort

Description:

The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning. The algorithm maintains two sub-arrays in a given array.

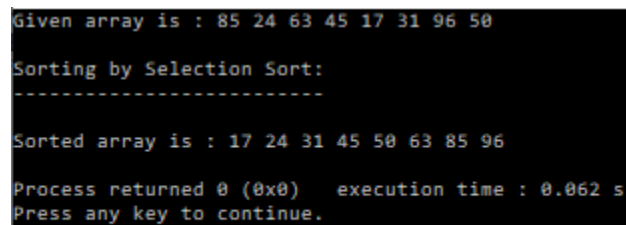
- 1) The sub-array which is already sorted.
- 2) Remaining sub-array which is unsorted.

In every iteration of selection sort, the minimum element from the unsorted sub-array is picked and moved to the sorted sub-array.

Pseudo code:

```
procedure selection sort
    array : array of items
    n     : size of list
    for i = 1 to n - 1
        /* set current element as minimum*/
        position = i
        /* check the element to be minimum */
        for j = i+1 to n
            if array[j] < array[position] then
                position = j;
            end if
        end for
        /* swap the minimum element with the current element*/
        if position != i then
            swap array[position] and array[i]
        end if
    end for
end procedure
```

Sample run from the program:

A screenshot of a terminal window with a black background and white text. It shows the execution of a selection sort program. The output includes the initial array, the sorting process, the sorted array, and system information like execution time and a prompt to press a key to continue.

```
Given array is : 85 24 63 45 17 31 96 50
Sorting by Selection Sort:
-----
Sorted array is : 17 24 31 45 50 63 85 96

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Bubble Sort

Description:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$.

Pseudo code:

```
Bubble_sort(list, n)
    for c=0 to n-c-1
        for d=0 to n-c-2
            if list[d]>list[d+1]
                Swap list[d] and list[d+1]
```

Sample run from the program:

```
Given array is : 85 24 63 45 17 31 96 50
Sorting by Bubble Sort:
-----
Sorted array is : 17 24 31 45 50 63 85 96
Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

Insertion Sort

Description:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

To sort an array of size n in ascending order:

- 1: Iterate from $arr[1]$ to $arr[n]$ over the array.
- 2: Compare the current element (key) to its predecessor.
- 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Pseudo Code:

```
procedure insertion_Sort( array, n )
    int j
    int element
    for i = 1 to n do:
        /* select value to be inserted */
        element = array[i]
        j = i-1
        /*locate j for the element to be inserted */
        while j > 0 and array[j-1] > element do:
            array[j+1] = array[j]
            j = j -1
        end while
        /* insert the number at j */
        array[j] = element
    end for
end procedure
```

Sample run from the program:

```
Given array is : 85 24 63 45 17 31 96 50
Sorting by Insertion Sort:
-----
Sorted array is : 17 24 31 45 50 63 85 96

Process returned 0 (0x0)   execution time : 0.250 s
Press any key to continue.
```

Merge Sort

Description:

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

Pseudo Code:

```
mergeSort(arr, l, r):  
    if l > r  
        return  
    m = (l+r)/2  
    mergeSort(arr, l, m)  
    mergeSort(arr, m+1, r)  
    merge(arr, l, m, r)  
end
```

Sample run from the program:

```
Given array is : 85 24 63 45 17 31 96 50  
Sorting by Merge Sort:  
-----  
Sorted array is : 17 24 31 45 50 63 85 96  
Process returned 0 (0x0)   execution time : 0.078 s  
Press any key to continue.
```

Heap Sort

Description:

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

Pseudo Code:

```
heapSort(array, n) {  
  for i <- n downto 2 {  
    exchange array[1] <-> array[i]  
    heapsize <- heapsize -1  
    heapify(array, n, i) }  
Heapify(array, n ,i) {  
  max <- i  
  leftChild <- 2*i+1  
  rightChild <- 2*i+2  
  if (leftChild<=n) and  
  (array[leftChild]>array[max])  
    max <- leftChild  
  if (rightChild<=n) and  
  (array[rightChild]>array[max])  
    max <- rightChild  
  if (max != i) { exchange array[i] <->  
array[max]  
    heapify(array, n, max) }}
```

Sample run from the program:

```
Given array is : 85 24 63 45 17 31 96 50
Sorting by Heap Sort:
-----
Sorted array is : 17 24 31 45 50 63 85 96
Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

Quick Sort

Description:

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quick sort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code:

```
quicksort(arr, low, high)
    if (low < high)
        pivotlocation = partition(arr, low, high)
        quicksort(arr, low, pivotlocation-1)
        quicksort(arr, pivotlocation + 1, high)

partition(arr, low, high)
    set high as pivot
    leftwall = low - 1
    for i = low to high
        if arr[i] < pivot
            swap arr[i] and arr[leftwall]
            leftwall++
    swap pivot and arr[leftwall+1]
    return leftwall + 1
```

Sample run from the program:

```
Given array is : 85 24 63 45 17 31 96 50
Sorting by Quick Sort:
-----
Sorted array is : 17 24 31 45 50 63 85 96

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

Comparison between the times taken by each sorting technique

According to the following sample run, table and graph, we concluded that the sorting techniques that have a time complexity of $O(n^2)$ are not suitable or practical to use for large arrays because they take a significantly long time to sort them. However, sorting techniques that have a time complexity of $O(n \log n)$ were so quick to sort large arrays of 100,000 elements.

Sample runs:

1) Array of 1000 elements

```
Number of array elements = 1000
time taken by quick sort = 0.000000 milliseconds
time taken by selection sort = 0.000000 milliseconds
time taken by heap sort = 0.000000 milliseconds
time taken by insertion sort = 0.000000 milliseconds
time taken by bubble sort = 0.000000 milliseconds
time taken by merge sort = 0.000000 milliseconds
```

2) Array of 10000 elements

```
Number of array elements = 10000
time taken by quick sort = 0.000000 milliseconds
time taken by selection sort = 124.000000 milliseconds
time taken by heap sort = 0.000000 milliseconds
time taken by insertion sort = 79.000000 milliseconds
time taken by bubble sort = 296.000000 milliseconds
time taken by merge sort = 0.000000 milliseconds
```

3) Array of 100000 elements

```
Number of array elements = 100000
time taken by quick sort = 16.000000 milliseconds
time taken by selection sort = 11765.000000 milliseconds
time taken by heap sort = 32.000000 milliseconds
time taken by insertion sort = 7281.000000 milliseconds
time taken by bubble sort = 29030.000000 milliseconds
time taken by merge sort = 31.000000 milliseconds
```

4) Array of 200000 elements

```
Number of array elements = 200000
time taken by quick sort = 32.000000 milliseconds
time taken by selection sort = 46139.000000 milliseconds
time taken by heap sort = 62.000000 milliseconds
time taken by insertion sort = 25874.000000 milliseconds
time taken by bubble sort = 116341.000000 milliseconds
time taken by merge sort = 31.000000 milliseconds
```

5) Array of 300000 elements

```
Number of array elements = 300000
time taken by quick sort = 46.000000 milliseconds
time taken by selection sort = 103997.000000 milliseconds
time taken by heap sort = 94.000000 milliseconds
time taken by insertion sort = 58061.000000 milliseconds
time taken by bubble sort = 261414.000000 milliseconds
time taken by merge sort = 62.000000 milliseconds
```

6) Array of 400000 elements

```
Number of array elements = 400000
time taken by quick sort = 78.000000 milliseconds
time taken by selection sort = 184589.000000 milliseconds
time taken by heap sort = 125.000000 milliseconds
time taken by insertion sort = 104247.000000 milliseconds
time taken by bubble sort = 465205.000000 milliseconds
time taken by merge sort = 78.000000 milliseconds
```

7) Array of 500000 elements

```
Number of array elements = 500000
time taken by quick sort = 93.000000 milliseconds
time taken by selection sort = 288679.000000 milliseconds
time taken by heap sort = 172.000000 milliseconds
time taken by insertion sort = 162323.000000 milliseconds
time taken by bubble sort = 726525.000000 milliseconds
time taken by merge sort = 94.000000 milliseconds
-----
Process returned 0 (0x0)   execution time : 2594.703 s
Press any key to continue.
```

Excel Sheet

Excel Table:

Number of elements	Quick Sort	Bubble Sort	Merge Sort	Selection Sort	Insertion Sort	Heap Sort
1000	0	0	0	0	0	0
10000	0	296	0	124	79	0
100000	16	29030	31	11765	7281	32
200000	32	116341	31	46139	25874	62
300000	46	261414	62	103997	58061	94
400000	78	465205	78	184589	104247	125
500000	93	726525	94	288679	162323	172

Excel Graph:

