Assignment 2

Pranav Gupta (B15317)

6 March, 17

Contents

1 Problem

Implement the following sorting algorithms using C++ programing language and sort the input sequence in ascending order.

- Insertion sort
- Selection sort
- Rank sort
- Bubble sort
- Merge sort
- Quick sort

2 Insertion Sort

2.1 Pseudo Code

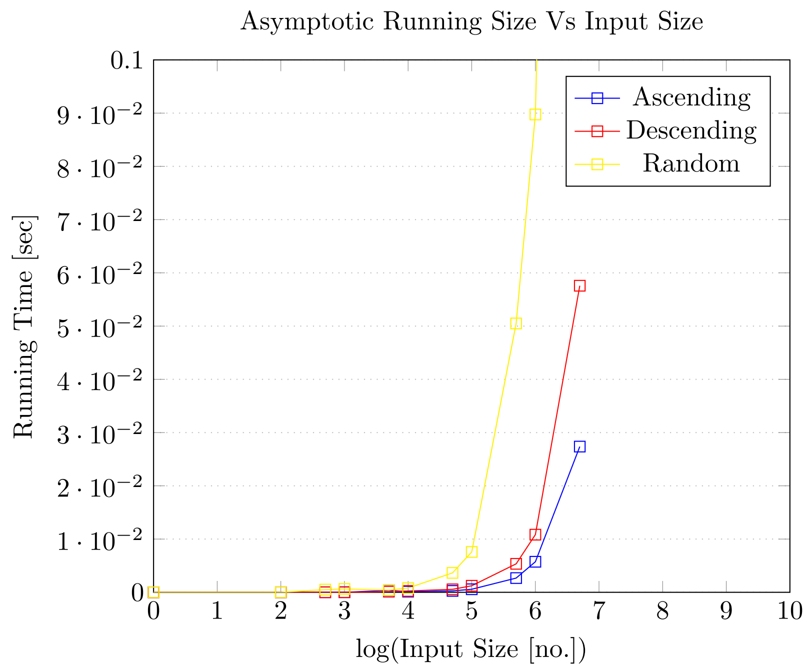
```
INPUT : A[1..n], array of integers
OUTPUT : Rearrangement of A such that A[1] ≤ A[2] ≤ ... ≤ A[n]
for j = 2 to n do
    key = A[j]                ▷ Insert A[j] into the sorted sequence A[1....j − 1]
    i = j − 1
    while i > 0 and A[i] > key do
        A[j + 1] = A[i]
        i = i + 1
    end while
    A[i + 1] = key
end for
```

2.2 Running Time Analysis

Best case : $O(n)$

Worst case : $O(n^2)$

2.3 Plot of running time Vs input size



3 Selection Sort

3.1 Pseudo Code

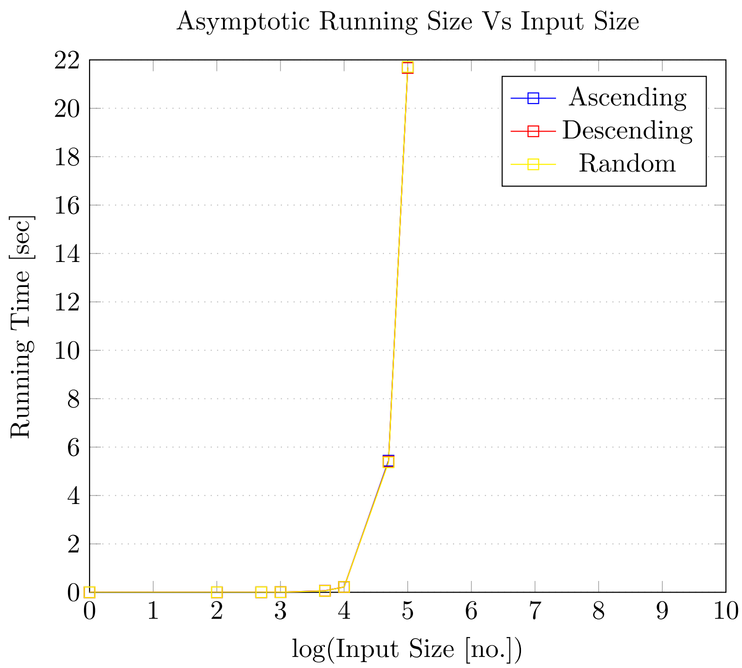
INPUT : $A[1..n]$, array of integers
OUTPUT : Rearrangement of A such that $A[1] \leq A[2] \leq ... \leq A[n]$
$sorted = false$
$j = n$
while $j > 1$ and $sorted = false$ **do**
    $pos = 1$
    $sorted = true$
    ▷ Find the position of the largest element
    for $i = 2$ to j **do**
        if $A[pos] <= A[i]$ **then**
            $pos = i$
        else
            $sorted = false$
        end if
    end for
    ▷ Move $A[j]$ to the position of largest element by swapping
    $temp = A[pos]$
    $A[pos] = A[j]$
    $A[j] = temp$
    $j = j - 1$
end while

3.2 Running Time Analysis

Best case : $O(n)$

Worst case : $O(n^2)$

3.3 Plot of running time Vs input size



4 Rank Sort

4.1 Pseudo Code

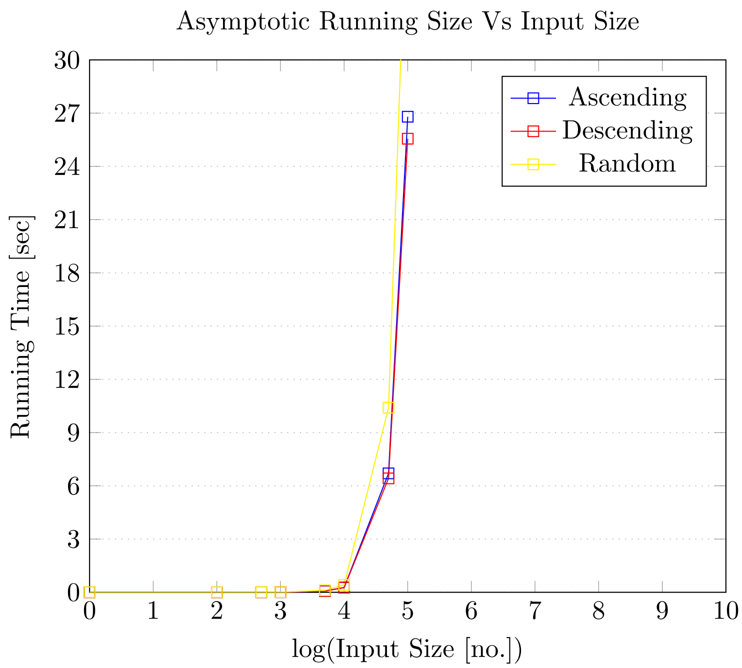
```
INPUT : A[1..n], array of integers
OUTPUT : Rearrangement of A such that A[1] ≤ A[2] ≤ ... ≤ A[n]
for j = 1 to n do
    R[j] = 1
end for
▷ Rank the n elements in A into R
for j = 2 to n do
    for i = 1 to j − 1 do
        if A[i] ≤ A[j] then
            R[j] = R[j] + 1
        else
            R[i] = R[i] + 1
        end if
    end for
end for
▷ Move to correct place in U[1..n]
for j = 1 to n do
    U[R[j]] = A[j]
end for
▷ Move the sorted entries into A
for j = 1 to n do
    A[j] = U[j]
end for=0
```

4.2 Running Time Analysis

Best case : $O(n^2)$

Worst case : $O(n^2)$

4.3 Plot of running time Vs input size



5 Bubble Sort

5.1 Pseudo Code

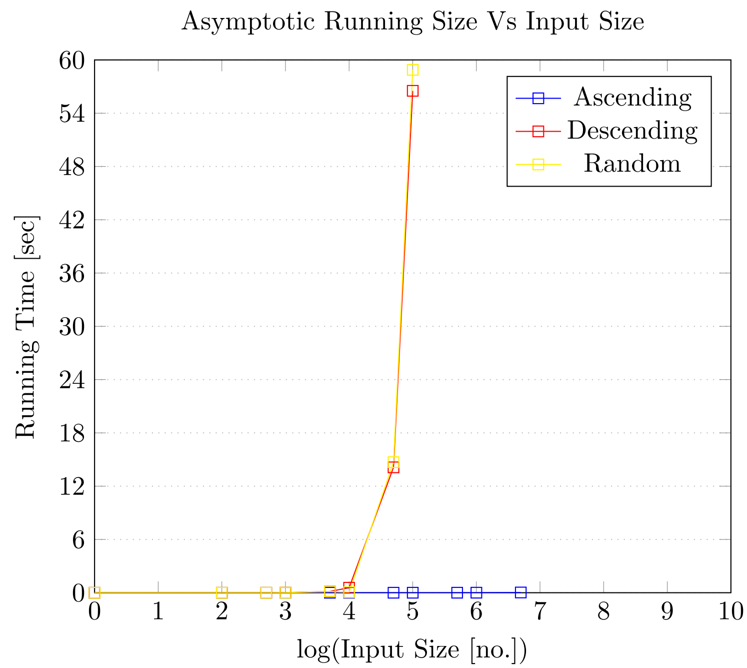
```
INPUT : A[1..n], array of integers
OUTPUT : Rearrangement of A such that A[1] ≤ A[2] ≤ ... ≤ A[n]
j = n
while j ≥ 2 do
▷ Bubble up the smallest element to its correct position
    for i = 1 to j − 1 do
        if thenA[i] > A[i + 1]
            temp = A[i]
            A[i] = A[i + 1]
            A[i + 1] = temp
        end if
        j = j − 1
    end for
end while=0
```

5.2 Running Time Analysis

Best case : $O(n)$

Worst case : $O(n^2)$

5.3 Plot of running time Vs input size



6 Merge Sort

6.1 Pseudo Code

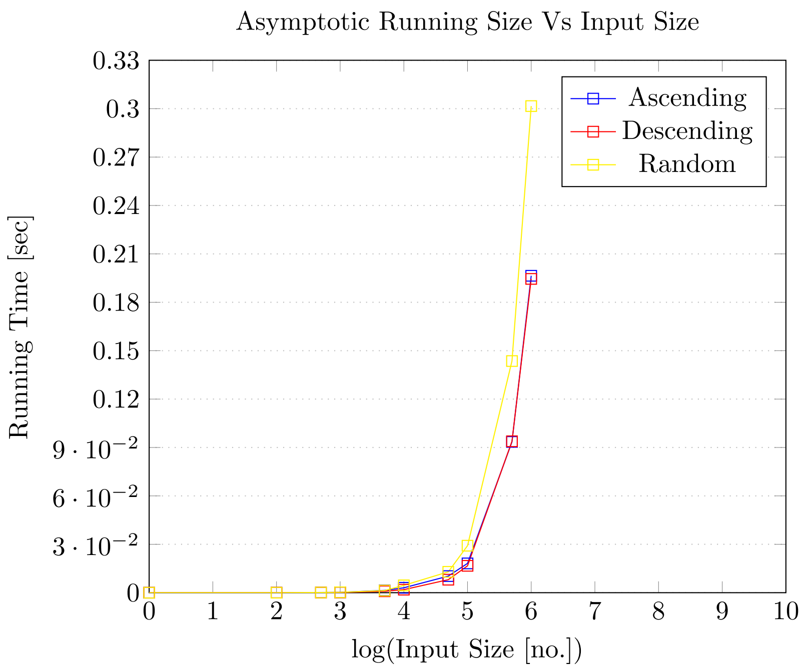
```
INPUT : A[1..n], array of integers
OUTPUT : Rearrangement of A such that A[1] ≤ A[2] ≤ ... ≤ A[n]
procedure MERGE(A, p, q, r)
    n1 = q − p + 1
    n2 = r − q
    A1[n1 + 1] = ∞
    A2[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r do
        if A1[i] ≤ A2[j] then
            A[k] = A1[i]
            i = i + 1
        else
            A[k] = A2[j]
            j = j + 1
        end if
    end for
end procedure
procedure MERGE-SORT(A, p, r)
    if p < r then
        q = (p + r)/2
        MERGE-SORT (A, p, q )
        MERGE-SORT (A, q + 1, r)
        MERGE (A, p, q, r)
    end if
end procedure=0
```

6.2 Running Time Analysis

Best case : $O(n \log n)$

Worst case : $O(n \log n)$

6.3 Plot of running time Vs input size



7 Quick Sort

7.1 Pseudo Code

```
INPUT : A[1..n], array of integers
OUTPUT : Rearrangement of A such that A[1] ≤ A[2] ≤  ≤ A[n]
procedure PARTITION(A, p, r)
    pivot = A[r]                                        ▷ Pivot
    i = p − 1
    j = r + 1
    while  TRUE do
        j = j − 1
        while A[j] > pivot do
            i = i + 1
            while A[i] < pivot do
                if j > i  then
                    exchange A[i]withA[j]
                else if  j = i then
                    return  j − 1
                else
                    return j
                end if
            end while
        end while
    end while
end procedure
procedure QUICK-SORT(A, p, r)
    if  p < r  then
        q = PARTITION (A, p, r)
        QUICK-SORT (A, p, q)
        QUICK-SORT (A, q + 1, r)
```

```
    end if
end procedure
```

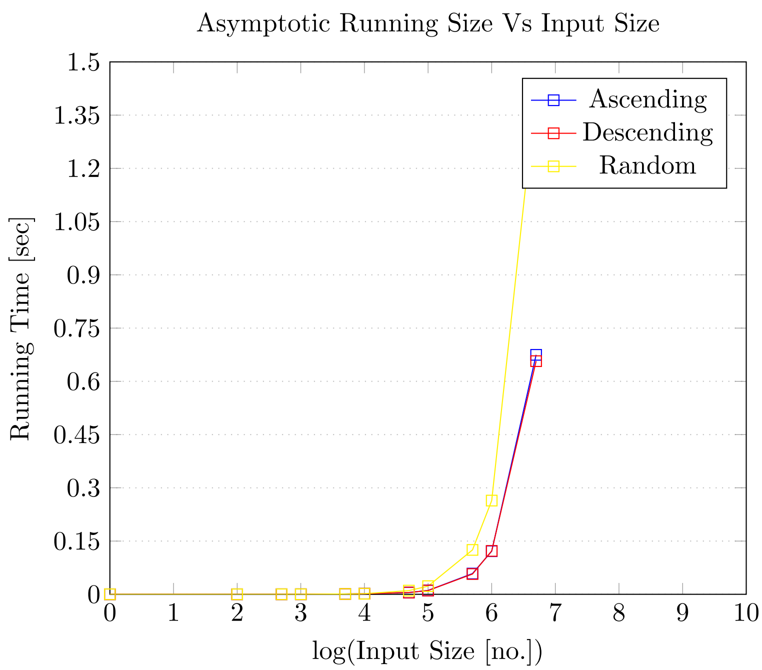
=0

7.2 Running Time Analysis

Best case : $O(n^2)$

Worst case : $O(n \log n)$

7.3 Plot of running time Vs input size



8 Conclusion

All the running time analysis of algorithm are already provided above but some points which are worth mentioning are : 1) Selection Sort,Insertion Sort and Bubble sort are all **in place** sorting algorithms so these can be where input is small(also mostly sorted) and memory is costly. 2)One of the good things about selection sort is that it never makes more than n swaps so can be used quite efficiently where memory write is a costly operation(as selection sort first iterates and then selects the only the required element to be swapped). 3) Insertion sort should not be used where memory write is a costly operation as it makes too much swaps. 4)We also got to know about the stability of sorting algorithms which means that if we have some identical elements in our array then after sorting the indexing of those identical elements should not change. Insertion Sort,Merge Sort,Bubble Sort are some examples of stable sorting algorithms. Some sorting algorithms are not stable, like Heap Sort, Quick Sort, etc. 5)Also the algorithm like merge sort need O(n) extra space so the program reported segmentation fault as our OS keeps bounds on allocation of memory to a certain program and that limit was violated in case of this large input for merge sort and hence program crashed. 6)In computer graphics bubble sort is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n) [Source : GeeksForGeeks].