EECE7205 – Fundamental of Computer Engineering

Name – Ankesh Kumar
NU ID - 002208893

Homework 2

Problem 1 :

Approach for solving problem :

Here, I have implemented the mentioned the Pseudo code in Question part and included the <random> class for generating the random in range 1 to 100.

As per the execution Screenshot, firstly we use MERGE-SORT to sort the 30 elements.

Afterwards, we add the logic to check if the array provided in merge function is already sorted, then it skips the whole merge algorithm part and makes program efficient.

Running Screenshot

```
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ g++ HW2_Problem1.cpp -o HW2_Problem1
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ ./HW2_Problem1
Generated random array : 45 71 89 96 39 38 96 78 54 77 54 19 25 87 85 12 66 7 58 89 59 16 51 57 24 51 45 39 41 21
Sorted elements of array : 7 12 16 19 21 24 25 38 39 39 41 45 45 51 51 54 54 57 58 59 66 71 77 78 85 87 89 89 96 96
Already sorted array before sorting:
1 2 3 4 5 6 7 8 9 10
Array after sorting:
1 2 3 4 5 6 7 8 9 10
```

Problem 2: In the above MERGE algorithm and to avoid having to check whether either list is empty in each basic step, a sentinel value of ∞ is placed at the end of each list. Rewrite the algorithm so that it does not use sentinels, instead it stops once either array L or R has had all its elements copied back to A and then copies the remainder of the other array back into A.

No need to submit any C++ program code for this problem (only the updated algorithm in pseudo code similar to what is presented in the previous problem)

Pseudo code –

MERGE(A,p,q,r)

1. n1 = q – p + 1
2. n2 = r – q
3. *let L[1….n1] and R[1…n2] be the new arrays*
4. **for** i = 1 to n1
5. L[i] = A[p + i]
6. **for** j = 1 to n2
7. R[j] = A[q + 1 + j]
8. i = 0
9. j = 0
10. k = p
11. **while** i<n1 and j<n2
12. **if** L[i] <= R[j]
13. A[k] = L[i]
14. i = i+1;
15. **else** A[k] = R[j]
16. j = j+1
17. k = k+1

Name – Ankesh Kumar
NU ID - 002208893

```
18. while i < n1
19.     A[k] = L[i]
20.     i = i+1
21.     k = k+1
22. while j < n2
23.     A[k] = R[j]
24.     j = j+1
25.     k = k+1
```

Summary :

In this The Merge operation takes three indices p, q, and r as parameters to merge two sorted subarrays within the array A. It involves creating two temporary subarrays L and R from the original array and then merging them back into the original array.

- Calculate the sizes of the two subarrays, n1 and n2.
- Create two new arrays, L and R, to store the subarrays.
- Populate L and R with the respective elements from A.
- Initialize pointers i, j, and k for the two subarrays and the original array.
- Compare elements from L and R, selecting the smaller one to be placed in the correct position in A.
- Continue this process until either L or R is exhausted.
- If there are remaining elements in L or R, copy them to the original array A.

Problem 3 :

Write a C++ program to test the following two functions. Call the functions with values for n between 1 and 10.

```
int F1(int n)
{
if (n == 0) return 1;
return F1(n - 1) + F1(n - 1);
}


int F2(int n)
{
if (n == 0) return 1;
if (n % 2 == 0) {
int result = F2(n / 2);
return result * result;
}
else
return 2 * F2(n - 1);
}
```

Submit your test program and in your report answer the following questions:
a. What does each function do?
b. Which function is faster (hint: test them with n = 30)?
c. Guess the running time growth of each function and hence explain why one function is faster than the other.
Approach for solving problem:

What does each function do?

The function is used to calculate the 2^n of the number(n) passed into the function.
The difference lies in the way which the function is called.

EECE7205 – Fundamental of Computer Engineering

Name – Ankesh Kumar
NU ID - 002208893

First one uses the recursive method whereas 2nd method, utilises the method of divide and conquer method.
At running the C++ application, we can see that the F2 gives faster result compare to time taken by F1.
The second function (F2) is faster. As we also learnt from lectures that the divide and conquer method have **O(logn)** as n/2 + n/4 + n/8 …… for solving the method where the calculation halves in each calculation whereas
In F1, the multiplication keeps on increasing in each step.
F1(1) = F1(0) + F1(0) = 1+1 = 2
F1(2) = F1(1) + F1(1)
= (F1(0) + F1(0)) + (F1(0) + F1(0))
= (1 + 1) + (1 + 1)
= 4
So, the **order of growth is O(2^n)** as in first step it's 2, 2*2, 2*2*2…..
Execution Screenshot –

```
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ g++ HW2_Problem3.cpp -o HW2_Problem3
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$ ./HW2_Problem3
F1(1): 2
F2(1): 2

F1(2): 4
F2(2): 4

F1(3): 8
F2(3): 8

F1(4): 16
F2(4): 16

F1(5): 32
F2(5): 32

F1(6): 64
F2(6): 64

F1(7): 128
F2(7): 128

F1(8): 256
F2(8): 256

F1(9): 512
F2(9): 512

F1(10): 1024
F2(10): 1024

F1(30): 1073741824
F2(30): 1073741824
akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram$
```

EECE7205 – Fundamental of Computer Engineering

Name – Ankesh Kumar
NU ID - 002208893

Problem 4:

Approach for solving problem :

The objective of the ProcedureX is making an ascending/ sorted array.
The Step3 and 4, shifts the larger value to right index position if the A[j] value is less than A[j-1].
The Step 2, it transverses the list starting from the end.
Step 1, Running of the outer loop signifies for running it the number of times as it would transverse enough so that the loop iterates through each element.
In the end, we get a sorted list.

n = A.length → worst case running time

		cost	times
	ProcedureX (A)		
1.	for i = 1 to A.length - 1	c_1	n
2.	for j = A.length down to i + 1	c_2	$n \times (n-1)$
3.	if A[j] < A[j-1]	c_3	$n(n-1)$
4.	exchange A[j] with A[j-1]	c_4	$n(n-1)$

$$T(n) = c_1 n + c_2[n(n-1)] + c_3[n(n-1)] + c_4[n(n-1)]$$
$$= c_1 n + [n^2 - n][c_2 + c_3 + c_4]$$
$$= n^2[c_2 + c_3 + c_4] + n[c_1 - c_2 - c_3 - c_4]$$

$$T(n) = an^2 + bn$$

Hence Asymptotic notation for worst case is
$O(n^2)$

Summary :

The objective of the ProcedureX is making an ascending/ sorted array.
As, there is loop inside loop, while trying to find the Time complexity equation, the was to the power of 2. So, for the worst case scenario, the Asymptotic equation came out to be $O(n^2)$.

EECE7205 – Fundamental of Computer Engineering

Name – Ankesh Kumar
NU ID - 002208893

Problem 5 :

Approach for solving problem :

```
1. RecursiveInsertionSort(A, n)
2. if n > 1
3.         InsertionSortRecursion(A, n - 1)
4.         Insert(A,n)
```

The recurrence question would be given by
T(n) = T(n-1) + Time equation by Insert.
According to the lecture, the Insert function complexity would be

```
key = A[i]        ---- O(1)
j = i – 1         ---- O(1)
while j > 0 and A[j] > key:  --------O(n)
        A[j + 1] = A[j]
        j = j - 1
// Insert the key into its correct position
A[j + 1] = key    ------ O(1)
```

Hence, it would be – O(n) as we are not calculating the constants.
The Recurrence equation would become –
T(n) = T(n-1) + O(n)

$T(n) = T(n-1) + n$

$T(n-1) = T(n-2) + n-1$

$T(n-2) = T(n-3) + n-2$

$\vdots$

$T(2) = T(1) + 2$

On addition →

$T(n) = T(n-1) + n$

$T(n) = [T(n-2) + n-1] + n = T(n-2) + 2n - 1$

$T(n) = (T(n-3) + n-2) + 2n-1 = T(n-3) + 3n - 3$

$\vdots$

$T(n) = T(1) + n(n-1) = O(n^2)$

According to Recursion Tree Algorithm.

Hence for base case, for one element in array, it would by $T(1)$
and for worst case → $O(n^2)$.

Summary :

We used the recursion tree method to solve the recurrence equation.

Its asymptotic notation comes down to $O(n^2)$ according to this method.