Homework 6 - Brute force, divide and conquer, analysis

Problem 1:

Given an array of size 16, what is the maximum possible *mixed-up-ness* score? Explain why you think this (e.g., give a logical argument or provide an example)

ANS

The maximum possible mixed-up-ness score is 120.

For insertion sort, when we have a reverse array, in the body of the while loop, we need operate the swap statement every time, which meas we will always get a pair of inversion number.
Pseudocode:
INSERTIONSORT(A)

```
1    for i from 1 to size
2    j = i;
3    while j > 0
4        if (array[j] < array[j - 1]) then
5            swap(&array[j], &array[j - 1]);
```

A reverse array. (*n, n-1, ... 1*) .

The number of inversions of the array is $(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=c_n^2$

An Instance array A[6,5,4,3,2,1,0], the number N of array = 7
For A[0]=6, can be consisted of the count inversions with the rest elements of A[5,4,3,2,1,0], we get 6 pairs, which is (n-1)
For A[1]=5, can be consisted of the count inversions with the rest elements of A[4,3,2,1,0], we get 5 pairs,which is (n-2)
For A[2]=4, can be consisted of the count inversions with the rest elements of A[3,2,1,0], we get 4 pairs,which is (n-3)
.....
.....
For A[5]=1, can be consisted of the count inversions with the rest elements of A[4,3,2,1,0], we get 1 pairs,which is (n-6)
For A[6]=0, has no count inversion.
Thus, given an array of size 16, the maximum possible mixed-up-ness score is 120 when array is reverse.

Problem 2:

What is the worst-case runtime of the brute-force algorithm that you implemented? Give a proof (a convincing argument) of this.

ANS

Pseudocode:

SLOW MIXED-UP NESS(A, size)

```
1    count equals to 0;
2    i from 0 to size
3        j from i to size
4            if (array[i] > array[j]) then
5                count++;
6    return count;
```

The worst-case runtime of the brute-force algorithm is that we need to run line 5 for every circulation. That means every element A[i] of array is bigger than A[j], which j > i.

The array is a reverse array. [*n, n-1, ... 1*].
For i = 0, A[0] = n, we run line 4 to compare A[0] and A[j],
A[0] is always smaller than A[j], j∈[1.....n-1], we always need to run line 5.
For i = 1, A[1] = n-1, we run line 4 to compare A[1] and A[j],
A[0] is always smaller than A[j], j∈[2.....n-1], we always need to run line 5,
.....
....
For i = k, A[k] = n-k, we run line 4 to compare A[1] and A[j],
A[0] is always smaller than A[j], j∈[k+1.....n-1], we always need to run line 5.
The big-Oh is $O(n^2)$

Problem 3:

State the recurrence that results from the divide-and-conquer algorithm you implemented in Part 3.

ANS

Pseudocode:

mergeSort(int arr[], int temp[], int left, int right)

```
1    if (left < right)
2        int mid = (left + right) / 2;
3        mergeSort(arr, temp, left, mid)
4        mergeSort(arr, temp, mid + 1, right)
5        merge(arr, temp, l, mid, right)
```

For this part of code, we have two arrays, the left position of the array which equals to zero, and the right position of the array which equals to size of array minus one.

when left is smaller than right, we get the middle position *mid = (left+right)/2*

First Recursion,we operate the mergeSort in line 3from left to middle, we've got the subarray A[left.....middle], which has half number of A[left...right]. The next we will be back to if statement compare left with new right which is the middle. Then we will do this step recursive until the left equals to middle, which means we get a subarray only has one element A[left].

Second Recursion, we operate the mergeSort in line 4 from mid+1 to right. The last time we run line 4, we put the mid(right) = 1, get mid = 0 =left. For now mid +1 = 1,right =1. We get back to if statement *mid+1 = right* =1. Then we operate the line 5 merge function.

Through all the steps above, we get the subarray with only one element. When we done merge function, we will operate the mergeSort again.

Problem 4:

Solve the recurrence for the divide-and-conquer algorithm using the substitution method. For full credit, show your work.

ANS

Assuming that dividing the array and combing the array take in total cn time, and the total time required by merge sort is $T(n)$. For each division, we are actually halving the array, and for each smaller array, it needs $\frac{1}{2}T(\frac{n}{2})$ time.Then we have

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$= 2(2T\left(\frac{n}{4}\right) + \frac{cn}{2}) + cn$$

$$= 2\left[2\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + \frac{cn}{2}\right] + cn$$

$$\ldots \quad = 2^k T\left(\frac{n}{2^k}\right) + kcn$$

Let $n = 2^k, k = \ log_2 n$

$$T(n) = nT(1) + cnlog_2 n = n + cnlog_2 n$$

$$T(n) \in O(nlogn)$$

Therefore, we conclude that both the worst-case and best-case big-Oh complexity of merge sort are $O(nlogn)$.

Problem 5:

Confirm that your solution to #4 is correct by solving the recurrence for the divide-and-conquer algorithm using the master theorem. For full credit, clearly define the values of a, b, and d.

ANS

The definition of the master theorem is:

If $T(n) = a * T(\frac{n}{b}) + n^d$ *for a>1, b>1, d≥0*

There are three cases:

if $d > log_b a \quad T(n) = O(n^d)$

else if $d = log_b a \quad T(n) = O(n^d log n)$

else if $d < log_b a \quad T(n) = O(n^{log_b a})$

According to #4, we have the running time of merge sort is $T(n) = 2^n * T(\frac{n}{n^n}) + O(n^1) + O(1)$

which means *a* = 2^n, *b* = 2^n, *d = 1*

According to the Master Theorem, it's the second case,

d = $log_b a = 1,\ T(n) = O(n^d log n) = O(n log n)$

The big-Oh is the same as the complexity of divide-and-conquer algorithm with recursion, thus it's definitely correct.