CS32 Homework 4:

Problem 2:

a.

The time complexity for this program is O(N^3):

Because there a total of three loops and each loops would run exactly N times (each variable goes from 0 to N-1) and total number of runs is N\*N\*N=N^3.

Therefore the time complexity if O(N^3).

b.

The bottom loop (most inner loop) would run exactly N times. The middle loop would run from 0 to i. Therefore, if we exclude the outer loop (assume the syntax is correct) and it would run of a complexity of O(N\*(N-1)/2) times, which is about O(N^2/2)because it’s a sum of consecutive integers.

The outer loop runs N times as well. So the total complexity is O(N^3/2). Thence, the total effective complexity is still O(N^3). But in reality it is faster than a .

Problem 3:

a.

int func(const int a[], int n)

{

int counter=0;

for (int i=0; i<n; i++)

{

for (int k=i; k<n; k++)

{

if (a[i]>a[k] && i<k)

counter++;

}

}

return counter;

}

Here’s a function that takes about O(n^2/2) time to complete the inversion check, since we practically ignore the 0.5 multiplier in the complexity calculation, so it’s O(n^2). (or one can change i to 0 in the second loop as well, it’s still O(n^2)).

b.

When we use merge sort , we basically recursively call the merge function which calls the mergesort function. Mergesort function is the one that divides the the array into two and sort each one.

So the algorithm would go something like this:

In mergesort:

Divide the two arrays, set each starts and ends postion, create a new list for copying, create a index counter for the new array.

While for the two arrays the start have not reach the end

If (array1[start1]<=array2[start2])

Newarr[index]=array1[start1];

Increase the index for the new array and old array if used.

else

newarr[index]=array2[start2];

Increase the index for the new array and old array if used.

Increase counter by mid index-start1

Add the rest of elements to the new arr

Add the elements in the new array into the original array.

Return counter.

In merge,

If start haven’t reached to the end yet

Call merge(parameter for the first half)

Call merge (parameter for the second half)

Call mergesort.

Return counter.

Above is the algorithm, partially taken from classwork.

The reason we use mid index-start1, because when we call this function, array1 should be already sorted and therefore all the unchecked elements in array1 should satisfy inversion.

Since it’s basically same to the mergesort function except that I added a little feature, the complexity of this program would be O(nlogn), the same as merge sort.

Problem 4:

Create a node A that contains following information:

A boolean variable lets call it isExist.

A integer called m\_times counting how many times it shows up.

A index number called n.

The value of the member called m data.

The left child called m\_left

The right child called m\_right

The parent called m\_parent.

Another node B that have:

A key and a map.

Now we use a binary search tree (sort of since there’s addition frequency value in each node).

Create N number of B nodes.

Create N number of A nodes

Set all things NULL for now.

Set one of the first node into the first value.

Get data’s value.

Put the value as the key, change the corresponding ‘s map to 1.

Use node A to put it as the parent and first element

Now we add some more:

While (we can still get data member)

Get data’s value.

If we use the value there doesn’t exist a key

Put the value as the key, change the corresponding ‘s map to 1.

Put the map’s value to m\_times in the corresponding A node.

Compare with the first value:

if larger than the value

if the children pointer is null

put it as the children. Set m\_right, Set it’s parent pointer, m\_parent.

Else

Use pointer m\_right to go to it’s right child and recursively call this algorithm.

If smaller than the value

if the children pointer is null

Put it as the left child, set m\_left and set m\_parent.

Else

Use pointer m\_left to recursively call this algorithm.

Now it’s sorted binary tree.

Comment: We basically use the hash table to make sure the value haven’t been inserted yet which save a lot of time. Hash table’s complexity is O(1) which is nice.

Since there are only log(N) distince value, we are basically inserting log(N) values into this binary tree. A binary tree has a complexity of N\*log(N) since we are still check N times in the beigning but only inserting log(N) values into the tree, we basically get O(N\*log(log(N))) complexity.