Assignment 1

John Abo

30088517

1. .
2. Since the output must be a permutation of the input, we also need to prove that contains the same element as . (We already know that it is in ascending order)
3. Loop invariant:

For each iteration, the smallest element in cannot be at a position greater than .

Initialization:

Before entering the inner for loop, , where is the last position in the array. Therefore, the loop invariant holds prior to the first iteration of the loop.

Maintenance:

The only change occurring in the loop is that the smallest element is being moved towards the position , meaning it is either decrementing or staying the same position. The position of the smallest element can then never be greater than . Decrementing preserves the loop invariant, as the lowest element should at least be in prior to decrementing.

Termination:

We see that the condition for the for loop is that , so we must have after the loop terminates. Substituting that into the loop invariant we get:

The smallest element in subarray cannot be in a position be in a position greater than . Since is the first position, we can conclude that the that the first element of the array is now the smallest as well.

1. Loop invariant:

The subarray should be sorted and contain the smallest elements of .

Initialization:

When before entering the subarray is empty and is trivially sorted. Therefore, the loop invariant holds prior to the first iteration of the loop.

Maintenance:

The loop ensures the smallest element that isn’t already an element of is moved to , where is in the next value that should be added to . Therefore when is incremented, will be trivially sorted and contain the smallest elements of , thus the loop invariant holds.

Termination:

We see the condition for the for loop to end is that , so we must have after the loop terminates. Substituting that into the loop invariant we get:

The subarray should be sorted and contain the smallest elements of . The last element should then be the largest element. We can then conclude that the entire array sorted.

1. The outer loop is run at times, while the inner loop is run times, meaning the worst case of this implementation of bubble sort is . This is the same as insertion sort.
2. *Highest* Growth Rate

for this function

Same reasoning as above

In this case

In this case

These functions are equal

These functions are equal

Because

These functions are equal

In this case

for this function

These functions are equal

These functions are equal, constant growth rate is the lowest

*Lowest* Growth Rate

1. A function that is neither or for all functions in part a) would be, where it would be when and when . This prevents the function from being either greater or less than for all , and is thus neither nor
2. We’re given:
3. We see that:

The polynomial is:

Observe that and the polynomial have the same order of growth, so we get that:

1. Conjecture:

Guess the solution will be in the form:

Meaning there exists and such that for all we have:

Basis:

We know for some constant . We can consider

So long as we choose and

Induction Step:

Supposed for every , we have:

We then see that:

This holds for all

From the basis, we chose and . The induction step works for all , so we choose and . This gives us

For all and . We then get

1. Code is included with the submission.
2. and are algorithms given in class to get position of right and left child. The singly list is a dynamic set and adds element to the head of the linked list while receives a specified value then removes that value from the list. and is assumed to take constant time.

:

Input: Where and is a max heap

Output: An Array containing the largest elements of

//This will hold the positions of nodes being compared which can vary in number

Create singly linked list

//If is or negative, then it should return the empty array as there is no value to add

If then return ;

//Adds nodes to be searched to list

;

;

While do

//The following line removes the next maximum from max-heap because it is searching for the maximum among all the children nodes in . This takes linear time, as the list must be traversed, so get for the next step.

position from with largest key and set to

;

;

;

Return ;

Loop Invariant: contains the children of the keys that are an element of and , so that contains the largest elements of array .

Initialization: is inserted with the children of which is . Thus, they are elements of both arrays that would satisfy the loop invariant, as the is the maximum of array .

Maintenance: The next largest element that should follow should be at a position that is part of the list , which will be found and set to . will be set to that next largest element, so when is incremented, the loop invariant holds.

Termination: We see the condition for the loop is so when the loop ends. Substituting this back into the loop invariant we get:

contains the children of the keys that are an element of and , so that contains the largest elements of array . Thus the array contains the largest elements of .

Worst case scenario:

The most elements that will contain is when nodes have been searched and both of the children are added to the list. Since this search must be done times, we can conclude that .