**Aleksandr Itskovich (ai1221)**

**Assignment 1**

Question 1.

a.

|  |  |  |  |
| --- | --- | --- | --- |
| Function |  | Explanation | Type |
|  |  | grows faster than . As n becomes huge, the term can be simplified to just which just becomes | Logarithmic |
|  |  |  | Linear Logarithmic |
|  |  | The leading term doesn’t matter for large n. Simplifies to just | Factorial |
|  |  |  | Fractional Power, Poly Logarithmic |
|  |  | As n becomes huge, the 2 in front doesn’t really matter. | Linear Exponential |
|  |  | simplifies to just because the exponential term grows so much faster than the linear term. Similarly, simplifies to just because the exponential term grows so much more quickly than the polynomial term. As n becomes huge we just have which becomes | Exponential |
|  |  |  | Linear Poly Logarithmic |
|  |  | As n becomes huge, the multiplicative polynomial combined with the term grow much more quickly than the terms in the denominator. So we can simplify it to just the term in the numerator. | Polynomial Logarithmic |
|  |  | As n becomes huge, simplifies to just leaving | Fractional Poly Logarithmic |

Correct Order:

, , *,* , , , , ,

b.

* Execute SwapSort1 on array A = [7, 6, 5, 4, 3, 2, 1] indexed from s = 1 to f = 7.

*First Pass:*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i |  | i + 1 |  | f - 2 |  |  |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i |  | i + 2 | f - 2 |  |  |
| 5 | 6 | 7 | 4 | 3 | 2 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | i |  | f – 2, i + 2 |  |  |
| 5 | 4 | 7 | 6 | 3 | 2 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | i | f – 2 | i + 2 |  |
| 5 | 4 | 3 | 6 | 7 | 2 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | i, f – 2 |  | i + 2 |
| 5 | 4 | 3 | 2 | 7 | 6 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | f – 2 | i |  |
| 5 | 4 | 3 | 2 | 1 | 6 | 7 |

*Second Pass:*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i |  | i + 2 |  | f – 2 |  |  |
| 5 | 4 | 3 | 2 | 1 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i |  | i + 2 | f – 2 |  |  |
| 3 | 4 | 5 | 2 | 1 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | i |  | f – 2, i + 2 |  |  |
| 3 | 2 | 5 | 4 | 1 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | i | f – 2 | i + 2 |  |
| 3 | 2 | 1 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | f – 2, i |  | i + 2 |
| 3 | 2 | 1 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | f – 2 | i |  |
| 3 | 2 | 1 | 4 | 5 | 6 | 7 |

*Third Pass:*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i |  | i + 2 |  | f – 2 |  |  |
| 3 | 2 | 1 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i |  | i + 2 | f – 2 |  |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | i |  | f – 2, i + 2 |  |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | i | f – 2 | i + 2 |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | f – 2, i |  | i + 2 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | f – 2 | i |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

*Fourth Pass:*

On the 4th pass no more swaps are performed because the array is already sorted; after this pass the algorithm ends.

* Which of the above algorithms is correct?

SwapSort1 is incorrect because it does not work on all arrays. For example

|  |  |  |
| --- | --- | --- |
| i, f - 2 |  | i + 2 |
| 1 | 100 | 3 |

According to the way this algorithm is written, it would end after a single pass without actually sorting the array. This makes the algorithm *incorrect* because in order to be correct, an algorithm must work on *all inputs*.

Because it is always swapping i with i + 2, SwapSort1 has a “blind spot” in the i + 1 position, such that two elements might never end up in sorted order for certain array configurations. Here is another example:

First Pass:

|  |  |  |  |
| --- | --- | --- | --- |
| i | f - 2 | i + 1 |  |
| 2 | 3 | 3 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | f – 2, i |  | i + 1 |
| 4 | 3 | 2 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | f – 2 | i |  |
| 4 | 1 | 2 | 3 |

Second Pass:

|  |  |  |  |
| --- | --- | --- | --- |
| i | f – 2 | i + 2 |  |
| 4 | 1 | 2 | 3 |

In the second pass, SwapSort1 will compare 4 with 2 and 1 with 3, and will not perform any swaps in either case. Then the array will end, without returning a sorted result.

Unlike SwapSort1, SwapSort2 *is* correct. In fact, it is just a modification on Bubble Sort. There are two while-loops we need to worry about. The key thing to know is that the first while-loop continues until there are no more swaps; because the swaps only ever move larger values towards the end of the array, we know that we will eventually reach a state where the loop will terminate because there will be no more larger values to swap with smaller values (all the largest values will be in the back).

The second while loop is the same as Bubble Sort, which we know to be correct. Therefore, whatever the state of the array at the end of the first loop, we know that the second *while-loop* will successfully sort it and terminate.

* What is the worst-case number of comparisons made by SwapSort2 when the input array A has length n and is sorted in reverse order?

In this algorithm there are 2 *while* loops, each of which carries out a *for-loop*.

The first for-loop ranges from 1 to n – 2. A comparison is performed at every step of the iteration, therefore each full run of the for-loop performs **n – 2** comparison total.

The question then is, how many times does the first *while-loop* repeat?

In the case when the input array is sorted in reverse order, we have the situation where all of the *largest* values are in the front of the array. With every pass of the *first* while loop, we bubble up the first *two* largest values in the unsorted portion of the array into the *final* two slots of the array. Once the two largest values are occupying positions *n* and *n – 1* in the array, we know that there are no long any larger values that they are going to be swapped with in future passes. In other words, we can think of the last two positions of the array as already being in their *final position*, immune to any more swaps. Therefore, we are still only going to have swaps in the remaining portion of the array, which is now size ­*n – 2*.

Because each pass of the fist *while-loop* reduces the size of the array in which we still have things to swap by 2, we’re going to need to run at least the first *while-loop* at least  **n / 2** times before there are no more swaps. Therefore, the number of comparisons in the first *while-loop* is given by:

In other words, we perform *n – 2* comparisons *n / 2* times.

The second for-loop ranges from 1 to n – 1. Similar to the case above, a comparison is performed at each iteration and so the total number of comparisons performed each time the for-loop runs is (n – 1).

Because we know that the array is already given in reverse sorted order, we know that the worst-case state of the array before the second *while-loop* runs looks something like:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

We know this because of the way the numbers “leap-frog” over each other during the swaps, and that larger pairs traverse the list together towards the end of the array. In other words, after the first *while-loop* completes, we have pairs of values in the array that are either in *sorted* or *reverse order*, with the latter state of course corresponding to the worst-case.

In that worst-case, we would have to run the second *while-loop* two times: once to swap all the reversed pairs, and once to check that the array is sorted, no more swaps are needed, and the loop can end.

Therefore, the worst-case number of comparisons for an array in reverse sorted order for the second *while-loop* is given by:

Putting it all together, the total number of comparisons carried out by SwapSort2 is given by:

* Does your result from above represent the worst-case number of comparisons made by SwapSort2? Justify your answer.

SwapSort2 is essentially just a modification of Bubble Sort. We know from class that the worst-case scenario from bubble sort is when the *smallest* element is last in the array, because then it needs to be swapped all the way to the front of the array in successive passes; i.e. it’s the scenario that requires the greatest number of passes. The case where the array is sorted in reverse-order corresponds to that worst-case scenario, so yes, it does represent the worst-case number of comparisons that need to be made overall.

* Justify that the worst-case runtime of SwapSort2 is of the form for constants a, b, and c.

Generalizing from the equation for the total number of comparisons carried out at each step above, the number of operations performed by SwapSort2 is given by:

Therefore, is of the form where a = ½, b = 1 and c = 2.

c.

The child is carrying out **insertion sort** from class. In insertion sort, we iterate through the array. The portion of the array we’ve already looped through represents the sorted portion. If we encounter a value larger than the rightmost value in the sorted portion, we continuously swap it to the left until it encounters a value smaller than itself, or we run out of values. Then we proceed again to the right. This is what the child is essentially doing.

d.

Note: the dominant term is , therefore:

*Goal: show that*

Therefore, , so is .

Therefore, is .

Note: the dominant term is , therefore:

*Goal: show that that*

Therefore, where , so is .

Therefore, is .

**Question 2: Recurrences**

a.

*Practice1*

Examining *Practice1* first: essentially what the algorithm is doing during each recursive call is running a binary search on half of the array, then passing the other half of the array into a subsequent recursive call. Along the way it performs some constant-time operations such as comparisons, calculations, variable assignments and return statements.

The main functional work of each recursive call is performing the binary search, which we know from class takes work to complete.

We can describe *Practice1* using the recurrence relation:

Because it is in the appropriate form, we can use the Master Method to determine the runtime for the given relation using the constants:

We therefore have . Because is asymptotically larger than , our recurrence relations falls into Case 3 of the Master method, from which is follows that is .

*Practice2*

In this algorithm, we recursively perform binary search on arrays sequences decreasing by 1 with each recursive call.

The worst-case scenario corresponds to the case when all binary searches are unsuccessful. Because we are decreasing the array by one element with each recursive call, we end up making n recursive calls in total, and perform a binary search for (n -1) calls (because the final call hits the base-case and returns immediately. Therefore the total work done is .

The recurrence relation for this algorithm is given by:

We want to show that

Goal: show that

For the sake of induction, let us assume that:

Next, we can use our induction hypothesis to make a substation in the original recurrence relation:

When d is larger than c such that the subtractive term is positive, we are reducing , so that it ends up being asymptotically smaller than .

Therefore,

b.

Because is asymptotically larger than we have Case 3 in the Master Method. is and is .

Because we know that polynomials terms dominate in both and we can compare the functions directly and see that is asymptotically larger than . Therefore we have Case 1 of the Master Method, when is and

We see that is asymptotically larger than because is larger than . Therefore we have Case 1 of the Master Method, when is and .

c.

FindK(A, s, f, k)

if s < f

if f = s + 1

if k = A[s] return s

if k = A[f] return f

**return -1**

else

q1 =

q2 =

**r1 = FindK(A, s, q1, k)**

**r2 = FindK(A, q1 + 1, q2, k)**

**r3 = FindK(A, q2 + 1, f, k)**

**if (r1 > -1) return r1**

**if (r2 > -1) return r2**

**if (r3 > -1) return r3**

**return -1**

else

**if s = f and k = A[s]**

**return s**

**else**

**return -1;**

* Write and justify a recurrence for the runtime of of the above algorithm.

In the algorithm above, we make **3 successive recurrent calls**. Each call operates on roughly **1/3 of the size of the input array, or n/3**. The rest of the operations all take **constant time**, including comparisons, mathematical operations such as addition and division, return statements, etc. So the two components contributing to the runtime of the algorithm are the 3 recursive calls, and the sum of the constant-time operations which can be lumped together in a single constant term.

The final recurrence relation for the algorithm above should be:

The 3 in front of indicates that we are making the recursive call three times, and the indicates that each recursive call operates on about 1/3 of the input array, as discussed.

* Use the recursion tree to show that the algorithm runs in time .

Diagram

Description automatically generated

**Question 4: Hashing**

**a.**

**Final Result**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 22 |  | 12 | 16 | 37 | 18 | 19 | 7 |  | 29 |  | 50 | 38 |

**Steps:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

Insertion Order: 38, 16, 50, 7, 18, 19, 12, 37, 29, 22

Insert 38:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  |  |  |  |  |  |  |  |  |  | 38 |

Insertion Order: 16, 50, 7, 18, 19, 12, 37, 29, 22

Insert 16:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  | 16 |  |  |  |  |  |  |  |  | 38 |

Insertion Order: 50, 7, 18, 19, 12, 37, 29, 22

Insert 50:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  | 16 |  |  |  |  |  |  |  | 50 | 38 |

Insertion Order: 7, 18, 19, 12, 37, 29, 22

Insert 7:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  | 16 |  |  |  | 7 |  |  |  | 50 | 38 |

Insertion Order: 18, 19, 12, 37, 29, 22

Insert 18:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  | 16 |  | 18 |  | 7 |  |  |  | 50 | 38 |

Insertion Order: 19, 12, 37, 29, 22

Insert 19:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  |  | 16 |  | 18 | 19 | 7 |  |  |  | 50 | 38 |

Insertion Order: 12, 37, 29, 22

Insert 12:

Probe Attempt #1:

Probe Attempt #2:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  | 12 | 16 |  | 18 | 19 | 7 |  |  |  | 50 | 38 |

Insertion Order: 37, 29, 22

Insert 37:

Probe Attempt #1:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  | 12 | 16 | 37 | 18 | 19 | 7 |  |  |  | 50 | 38 |

Insertion Order: 29, 22

Insert 29:

Probe Attempt #1:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  |  | 12 | 16 | 37 | 18 | 19 | 7 |  | 29 |  | 50 | 38 |

Insertion Order: 22

Insert 22:

Probe Attempt #1:

Probe Attempt #2:

Probe Attempt #3:

**Final Result**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 22 |  | 12 | 16 | 37 | 18 | 19 | 7 |  | 29 |  | 50 | 38 |

b.

HashInsert(T, k)

m = k *mod* 13

if (T[m] = nil)

T[m] = k

return true

else

for i = 1 to 13

m = (k + (4 \* i) + (2 \* i \* i)) *mod* 13

if (T[m] = nil)

T[m] = k

return true

return false

c.

Assume that the arrays A and B have sizes (i.e. *n*) large enough to contain all possible 3-digit PIN number combinations. In other words, they are large enough to contains values from 000 through 999, with each PIN number occupying one cell of each array.

Assume also that array B uses a simplistic hash implementation where the hash function is the identity function. In other words, pin 555 would be stored at index 555 in the array, 002 would be stored in index 2, etc.

Assume that array A is a regular array where the numbers are entered as they appear (i.e. in no particular order, without gaps).

The algorithm to output all safe passwords from bank A would therefore be:

FindSafePins(A, B, s, f)

for i = s to f (*0 to n*)

pinA = A[i] (*constant*)

if (B[pinA] != nil) (*constant*)

print(pinA) (*constant*)

In the algorithm above we are looping over the entire length of array A once. For each iteration of the loop, we perform a constant amount of work. This is because accessing an element of an array is a constant-time operation, we perform two array accesses: once for A to get the PIN from A, and once in B to see if B contains the same PIN as A. Finally, there is the print operation to output the PIN from A if B does not contain the PIN in A.

In the worst-case array A is completely filled, i.e. it has size *n*. Therefore we can describe the runtime of the algorithm as:

Where *n* represents the number of iterations, and *a* represents the constant work performed per operation, and *b* represents any additional constant work associated with invoking the function.