Homework 1

Comp 3270

Haden Stuart / has0027

Problem 1.)

Given each of the values: A1 = O(n), A2 = O(nlogn), A3 = O(n^2). We must figure out what the time complexity is for each using the assumptions provided in the lecture slide. For this problem we are told that we will be using a machine running at 4 GHz meaning $4 * 10^9$ clock cycles per second. Since a machine typically requires about 200 clock cycles to execute one computation step we know that the computer can execute $2 * 10^7$ steps per second.

- For A1:

  A1 = O(n) where n = 200 million steps

  $n = 200{,}000{,}000 = 200 * 10^6$ steps

  Now to cancel units we divide steps by steps/sec:

  $(200 * 10^6) / (2 * 10^7) = \boxed{10 \text{ Seconds}}$

- For A2

  A2 = O(nlogn) where n = 200 million steps and log is base 2

  $n\log n = (200 * 10^6) * \log(200 * 10^6) = (200 * 10^6) * (27.5754)$

  = 5,515,080,000 steps

  Now to cancel units we divide steps by steps/sec:

  $(5{,}515{,}080{,}000) / (2 * 10^7) = \boxed{275.75 \text{ Seconds}}$

- For A3

  A3 = O(n^2) where n = 200 million steps

  $n^2 = (200 * 10^6)^2 = 400 * 10^{14}$ steps

  Now to cancel units we divide steps by steps/sec:

  $(400 * 10^{14}) / (2 * 10^7) = \boxed{200 * 10^7 \text{ Seconds}}$

Problem 2.)

1. Connect a driver with empty seats to people traveling to a specific target location
2. Query a remote service to determine specific requests submitted by customers
3. Each request is defined by the position of the person making the ride sharing request
4. Location of a person is specified as pair (latitude, longitude)
5. Generates the shortest possible route that visits each location exactly once (to pick up a customer) and return to the target location

- Inputs
  1.) Driver with empty seats location
  2.) Customer location
  3.) Customer desired location

- Data Representation
  1.) We can use a node structure to connect a driver to the node of the desired location for current customers.
  2.) Using this we can set the driver to a node with data for the current latitude and longitude, the customer to a node with their respective latitude and longitude, and a final node for the desired location of the customer with that latitude and longitude.

- Outputs
  1.) With the inputs provided, the driver will be given the shortest path to the customer's location then the driver will be given the shortest path to the customer's desired location.

Problem 3.)

Given *n* numbers, we are requested with finding the *i* largest numbers in sorted order. From what we are given we will assume that the numbers are not already sorted. So, for our first task we must sort the given set of numbers using the most efficient method. From our lecture 4 slides we talked about insertion sort and how "Before the start of jth iteration, 2≤j≤n, of the for loop of lines 1-8, the <u>subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order</u>". Using this statement, we can use insertion sort to our advantage in this problem by placing the values in descending order without needing to sort the entire set.

Step 1.) Set the first value as the max for the set

Step 2.) Iterate through the rest of the set and compare each value to the max

Step 3.) If another value is larger place that value in front and move the previous value over

Step 4.) Once sorted through, set the new max to the next value in the set

Step 5.) Repeat the previous steps *i* times to find the *i* largest numbers in the set

Problem 4.)

- Output when input is array (a) above: 55
- Output when input is array (b) above: 0
- Output when input is array (c) above: 0
- Output when input is array (d) above: 14
- When the input array contains all negative integers, the output will be 0 since the sum will never be greater than the initialized value of max which was 0.
- When the input array contains all non-negative integers, the output will be the sum of the entire array.

Problem 5.)

Step	Big-Oh Complexity
1	O(1)
2	O(1)
3	O(n)
4	O(1)
5	O(n)
6	O(1)
7	O(1)
8	O(1)
9	O(1)
Complexity of the algorithm	$O(n^2)$

Problem 6.)

Step	Cost of each execution	Total # of times executed
1	1	1
2	1	1
3	1	n+1
4	1	n
5	1	$\sum^{n}_{i=1} (i+1)$
6	6	$\sum^{n}_{i=1} (i)$
7	3	$\sum^{n}_{i=1} (i)$
8	2	$\sum^{n}_{i=1} (i)$
9	1	1

$T(n) = 1 + 1 + (n+1) + n + \sum^{n}_{i=1} (i+1) + (6)\sum^{n}_{i=1} (i) + (3)\sum^{n}_{i=1} (i) + (2)\sum^{n}_{i=1} (i) + 1$

⇨ $4 + 2n + (11)\sum^{n}_{i=1} (i) + \sum^{n}_{i=1} (i+1)$

⇨ $4 + 2n + (11)[n(n+1)/2] + [n(n+1)/2] + n$

⇨ $4 + 3n + (12)[n(n+1)/2]$

⇨ $6n^2 + 6n + 3n + 4$ $\underline{T(n) = 6n^2 + 9n + 4}$

Problem 7.)

This algorithm is Incorrect. Using the example file stated, "1 2 3 3 3 4 3 5 6 6 7 8 8 8 8", the algorithm should return the value 9 since there are 9 consecutive duplicate numbers. However, once the algorithm is run it returns the value 3 instead. The problem with this algorithm is that once a number has been read in for the values *i* and *j*, the algorithm compares the two, then moves on to the next set of values. This causes the algorithm to skip certain value comparisons. The correct method for stepping through this algorithm to follow would be to:

1.) Set the *i* value to the first value in the file
2.) Set the *j* value to the next value in the file
3.) Compare the two
4.) Set the *i* value to the *j* value
5.) Set the *j* value to the next value in the file
6.) Repeat steps 3 – 5 until the end of the file

Problem 8.)

Let's assume for a contradiction that A does not collect enough information to partition around the kth largest value without further comparisons. We know that to find the kth largest in the set, the algorithm A must sort the set of values such that all values greater than the kth largest are separate from all values that are less than the kth largest. Once this set has been sorted we no longer need to compare values since we only must find the kth largest value. However, this statement contradicts our starting statement saying that "we do not have enough information to partition the set without further comparisons. Thus, proving our theory.

Problem 9.)

Using this algorithm, let's test the values 0 and 1 for the base cases since n <= 1. The return values for 0 and 1 are 0 and 1 respectively. Now if we plug in the values 0 and 1 into the equation $3^n - 2^n$ we get:

- $3^0 - 2^0 = 1 - 1 = 0$
- $3^1 - 2^1 = 3 - 2 = 1$

Both answers match the return values from the algorithm, thus proving that the base cases work for this algorithm. Now let's assume that for all n = m where m > 1, $g(m) = 3^m - 2^m$. To prove this, we must test that the input (m+1) will still be solved in the form $\underline{3^{m+1} - 2^{m+1}}$. Plugging (m+1) into the algorithm, we see that (m+1) > 1 so it begins the recursive call. This gives us:

⇨ $5*g(m + 1 - 1) - 6*g(m + 1 - 2)$
⇨ $5*g(m) - 6*g(m - 1)$
⇨ $5*(3^m - 2^m) - 6*(3^{m-1} - 2^{m-1})$ (Using the fact: $g(m) = 3^m - 2^m$)
⇨ $5*3^m - 5*2^m - 2*3*3^{m-1} + 2*3*2^{m-1}$
⇨ $5*3^m - 5*2^m - 2*3^m + 3*2^m$
⇨ $(5 - 2)*3^m - (5 - 3)*2^m$

- ⇨ $3*3^m – 2*2^m$
- ⇨ $3^{m+1} – 2^{m+1}$

This solution gives us the desired result, thus proving that the algorithm is correct.

Problem 10.)

Initialization

1.) Before the loop, max is set to the value at the beginning of the array.
2.) The start of the loop, i is set to the second value in the array since the first is the max by default.
3.) This holds since this is the same as finding the max of the two values.

Maintenance

1.) During the ith iteration of the loop, the loop compares the value at the current A[i] to the value of max.
2.) If the A[i] value is greater than current max value, max is set to the A[i] value and the loop proceeds until n.
3.) If the A[i] value is not greater than the current max value, max will not be changed.

Termination

1.) At the end of the loop when i = n, the max will be within the values of A[1, … n-1].

Problem 11.)

- ➢ If m is the integer represented by the binary array b[1..k] then $n = t2^k + m$

Initialization

1.) Before the loop, t is set to n and k is set to 0.
2.) Since k is set to 0 we now have b[1..0] which is an empty array.
3.) This results in $n = t2^0 + m$.

Maintenance

1.) Using the loop invariant, we can test for the value (k+1) within the loop.
2.) This gives us $n = (t/2)2^{(k+1)} + m$ => $n = t2^k + m$

Termination

1.) At the end of the loop when t = 0, we get $n = 0*2^{k+1} + m$, resulting in n = m.

Problem 12.)

a.) Using the given swap() function, we can develop an algorithm that reverses a string by continuously swapping the outside values of a string and moving inwards. This will work by swapping the two outside i and j by using swap(A[i], A[j]), then on to swap(A[i+1], A[j+1]) until there are no longer any i and j values to be swapped. We can end the recursion by making the base case check for when i is greater than or equal to j since these two values intersect when all values have been swapped.

```
Algorithm reverseString(A: array[i…j] of characters)
        i,j: Integer
        i = 0
        j = length of array A
        function swap(A[i], A[j])

BEGIN

        if i >= j then return A

        else
          swap(A[i+1], A[j-1])

END
```

b.) Given the string "i<33270!" as an input we get:

```
Swap(A[1], A[8]): Output = !<33270i
                           ||
Swap(A[2], A[7]): Output = !03327<i
                           ||
Swap(A[3], A[6]): Output = !07323<i
                           ||
Swap(A[4], A[5]): Output = !07233<i
                           ||
Hit the base case: Final Output = !07233<i
```