Solutions for HW1 - CS 6033 Fall 2021

Q5 → Time complexity comparison

5. You have five algorithms, A1 took $O(n^2)$ steps, A2 took $\Theta(n^2)$ steps, and A3 took $\Omega(n^2)$ steps, A4 took $\Omega(n)$ steps, A5 took $o(n^2)$ steps. You had been given the exact number of steps the algorithm took, but unfortunately you lost the paper you wrote it down on. In your messy office you found the following formulas (you are sure the exact formula for each algorithm is one of them):

   (a) $7n \cdot \log_2 n + 12 \log_2 \log_2 n$
   (b) $7 \cdot (2^{2 \log_2 n}) + n/2 + 7$
   (c) $(2^{\log_4 n})/3 + 120n + 7$
   (d) $(\log_2 n)^2 + 75$
   (e) $7n!$
   (f) $2^{2 \log_2 n}$
   (g) $2^{\log_4 n}$

   For each algorithm write down all the possible formulas that could be associated with it.

	A1 $O(n^2)$	**A2** $\theta(n^2)$	**A3** $\Omega(n^2)$	**A4** $\Omega(n)$	**A5** $o(n^2)$
(a)	yes			yes	yes
(b)	yes	yes	yes	yes	
(c)	yes			yes	yes
(d)	yes				yes

	A1 $O(n^2)$	**A2** $\theta(n^2)$	**A3** $\Omega(n^2)$	**A4** $\Omega(n)$	**A5** $o(n^2)$
(e)			yes	yes	
(f)	yes	yes	yes	yes	
(g)	yes				yes

Q6 → Time complexity example

6. Find two functions $f(n)$ and $g(n)$ such that $f(n) \notin o(g(n))$ and $f(n) \in \Omega(g(n))$. If no such functions exist, write "No such functions exist".

One example:

$f(n)=n^2$ $g(n)=n^2$

Q7 → Time complexity magnitudes

7. In your book,[1] complete homework problem 1-1 pg 15 excluding determining the answer for $n!$ and $n\log(n)$. Simplifying assumptions: assume 1 month has 31 days; assume a year has 365 days for a year (and you may assume all years have 365 days even if this is not true).

3. 1 second=10^6 microsecond
1 minute=6*10^7 microsecond
1 hour=3.6*10^9 microsecond
1 day=8.64*10^10 microsecond
1 month=2.68*10^12 microsecond
1 year=3.21*10^13 microsecond

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lg n (quantity level)	10^(3*10^5)	10^(10^7)	10^(10^10)	10^(10^11)	10^(7.7*10^11)	10^(10^13)	10^(9.33*10^14)
sqrt(n)	10^12	3.6*10^15	13*10^18	7.49*10^21	6.71*10^24	9.66*10^26	9.67*10^30
n	10^6	6*10^7	3.6*10^9	8.64*10^10	2.68*10^12	3.21*10^13	3.21*10^15
n^2	10^3	7746	60000	2.93*10^5	1.63*10^6	5.67*10^6	5.67*10^7
n^3	100	391	1533	4420	13890	31781	147514
2^n	19	25	31	35	41	45	51

1 century= 3.21*10^15 microsecond

Q8 → Algorithm correctness for printing duplicates

8. Does the following algorithm correctly print out all the items that appear more than one time in the array (and prints no other values)?

   - If the code is correct, prove it using a *loop invariant*!
   - If not, prove that it does not work by providing an example on which it fails and show how this example is a counterexample. Then fix the code and prove your code works using a loop invariant.

```
PRINT_DUPLICATES(A)
1) for i = 1 to A.length
2)     for j = i to A.length
3)         if A[i] == A[j]
4)             PRINT(A[i])
```

It is not correct.

For example: A = [1,2,3]

It will print out "123", but there are no duplicates in the array.

We tweak this algorithm by changing the limits for i and j.

```
PRINT_DUPLICATES(A):
for i = 1 to A.length-1
    for j = i+1 to A.length
        if A[i] == A[j]
            print(A[i])
```

Outer loop:

Loop Invariant: Any item in A[1:i-1] which has a duplicate in any location in A has been printed.

Initialization: Before the first iteration, A[1:0] is empty, we don't need to print anything, so it's trivially true.

Maintenance:

From the outer loop invariant, we know that at the start of the loop any item in A[1:i-1] which has a duplicate in A has already been printed. After the inner loop, we know by the inner loop invariant that all duplicates of A[i] that occur in A[i+1:A.length] have been printed.

After loop: if A[i] appears more than once in the array, its duplicate may in A[1:i-1] or A[i+1: A.length]. If its duplicate is in A[1:i-1], A[i] should have been print in the previous loop, and if any item in A[i+1:A.length] equals A[i], it must be printed in this loop. So we can make sure that if A[i] appears more than once in the array, we must print it at least once.

Termination: Loop terminates when i = A.length+1, so for every item in A we have checked and already print out all the duplicate items in A[1:A.length].

Inner Loop:

Loop invariant: At the start of each iteration of the for loop, all items in A[i+1:j-1] which are equal to A[i] have been printed.

Initialization: There are no items in A[i+1:i] which means the loop invariant is trivially true.

Maintenance:

Prior conditions: if A[i] appeared more than once in A[i:j-1] it must be printed out at least once.

During the loop: if A[i] == A[j] we will print out A[i], and for next j' = j+1

After loop: Before the start of the next loop, we can make sure that if A[i] appears more than once in A[i:j'-1] = it must have been printed out.

Termination: The loop terminated when j = A.length + 1, so if A[i] appeared in A[i+1: A.length], it must have been printed out.

P.S: All reasonable answers will get full credits.

Q9 → Time complexity - analyze algorithms

9. What is the run time[2] of the following questions? Expressing your answer using big-Oh, little-oh, Theta, and Omega.

(a)

```
GAUSSES_SUMMATION_SERIES_1(n)
1) s = 0
2) for  i = 1 to n
3)      for j = i to n
4)          s = s + 1
5) return s
```

(b)

```
GAUSSES_SUMMATION_SERIES_2(n)
1) s = 0
2) for  i = 1 to n
3)      s = s + i
4) return s
```

(a) $O(n^2)$, $o(n^3)$, $\Theta(n^2)$, $\Omega(n^2)$

(b) $O(n)$, $o(n^2)$, $\Theta(n)$, $\Omega(n)$

Please notice: Big-Oh, Theta, and Omega notations must be the tightest bound as stated in the footnote.

Q10 → Insertion sort understanding

10. Professor T. Ortis, thought the insertion sort presented in class seems rather inefficient.[3] He thought he could improve the run time so it runs *asymptotically faster* if in the while loop (lines 5-7) instead of using a backward scan through the already sorted items to find where the jth item belongs, he instead used binary search to find where the jth item belongs. Is he right?

No. For the inner loop, although binary search can find the suitable place in O(log n). We still need O(n) to move those items. So the asymptotic time-bound for the sort algorithm hasn't been improved.

Q11 → Recursion understanding

11. For the following algorithm: Show what is printed by the following algorithm[4] when called with MAXIMUM$(A, 1, 5)$ where $A = [1, 2, 3, 4, 5]$? Where the function PRINT simple prints its arguments in some appropriate manner.

```
MAXIMUM(A, l, r)
1) if (r − l == 0)
2)     return A[r]
3)
4) lmax = MAXIMUM(A, l, ⌊(l + r)/2⌋)
5) rmax = MAXIMUM(A, ⌊(l + r)/2⌋ + 1, r)
6) PRINT(rmax, lmax)
7) if rmax < lmax
8)     return lmax
9) else
10)    return rmax
```

Answer: The expected answer is:

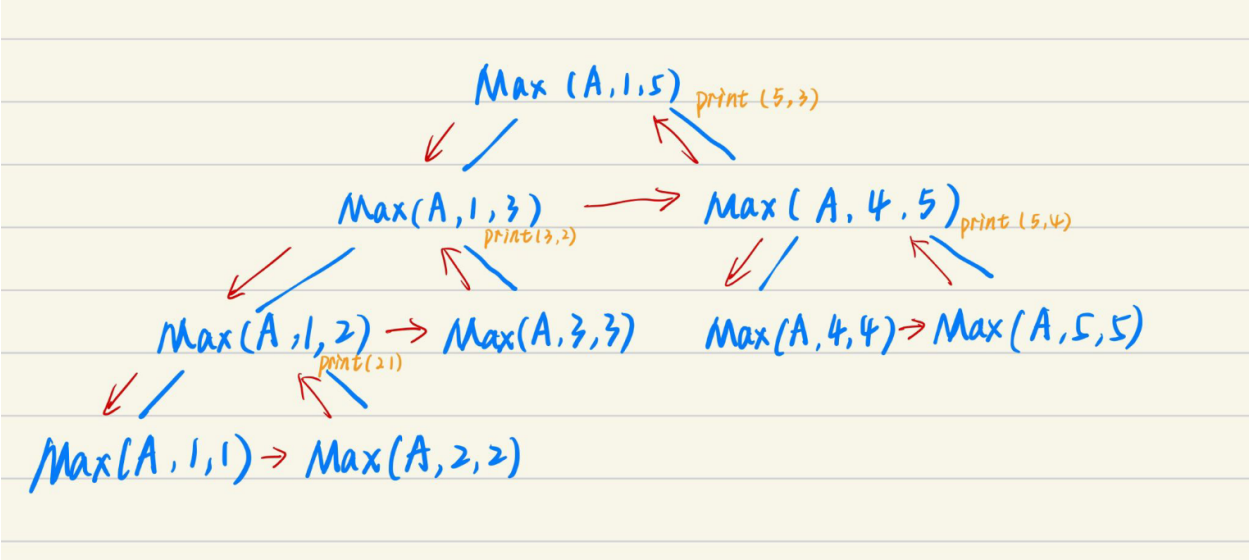
2 1

3 2

5 4

5 3.

Variations in formatting are accepted. Please refer to the tree below for the call stack.



P.S: red arrow means execution order

Q12 → Algorithm application - Buy stocks

12. Trying to finance your education, you decide to invest your D dollars in the stock market. The problem is you don't know which stocks to invest in. You decide ask your friends for stock tips and invest in every stock any of your friends suggests. If you receive n unique stock names you will invest D/n in each of the n stocks. Since some of your friends have given you the same stock name, you need to find a way to remove the duplicates.

    Design an algorithm to perform this task. You may call any algorithm as a subroutine that was presented in lecture 1. Your algorithm must run in $O(n^2)$ time where n is the number of names given to you by your friends.

    State what the worst case running time is of your algorithm in big-Oh, Theta, and Omega notation.[5]

Answer: Let us assume we have the list of all stocks suggested in A. A simple algorithm that satisfies the constraints is:

```
result_array = [ ]                          //Initialize an empty array
result_array.append(A[1])                   //Add first element to result
for i = 2 to A.length                       //Start from second element
      is_duplicate = false                  //Initialize a flag to find if
element is a duplicate
      for j = 1 to result_array.length      //Loop through all elements of
result_array
            if result_array[j] == A[i]      //Compare with each element of
result_array
                  is_duplicate = true       //Set flag to true if it is a
duplicate element
      if not is_duplicate                   //After exiting the loop, check if
it was a duplicate
            result_array.append(A[i])       //Add the element to result_array
if
print result_array
```

The above algorithm takes $O(n)$ extra space and runs in $\Theta(n^2)$ time, which means it is also $O(n^2)$ and $\Omega(n^2)$. An improved algorithm would be:

```
A = MERGE_SORT(A)       //Sorting using subroutine from class
j = 1                   //Initialize new index for result subarray
for i = 2 to A.length
      if A[j] != A[i]
            j = j + 1   //Increment index to point to next memory space
            A[j] = A[i]
print A[1 ... j]        //Output only the required subarray
```

The code above works because at any point in the traversal, the subarray $A[1 \cdots j]$always has the unique elements in order from the subarray $A[1 \cdots i - 1]$.

The runtime for the algorithm is $\Theta(n \log n)$ time, which means it is also $O(n \log n)$ and $\Omega(n \log n)$ and it uses $O(1)$extra space.