Brandon Vo

1. Have you answered poll question on Brightspace?

Yes

2. Have you read the syllabus?

Yes

3. Have you read the NYU policy on academic integrity (https://www.nyu.edu/about/policies-guidelines compliance/policies-and-guidelines/academic-integrity-for-students-at-nyu.html)?

Yes

4. Have you read the policy on collaboration in the syllabus?

Yes

Brandon Vo

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) $7 \cdot (2^{2 \log 2 n}) + n/2 + 7$ (Equivalent of $7*n^2 + n/2 + 7$)
c) $(2^{\log_4 n})/3 + 120n + 7$ (Equivalent of $1/3*n^{1/2} + 120n + 7$)
d) $(\log_2 n)^2 + 75$
e) $7n!$ (Equivalent of $7n^n$)
f) $2^{2 \log_2 n}$ (Equivalent of $n^2\log^2 2$)
g) $2^{\log_4 n}$ (Equivalent of $\sqrt{x}$)

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

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

∗Many of these questions came from outside sources.

Brandon Vo

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."

$f(n) \notin o(g(n))$ means that the set of f(n) for all n is not the asymptotic upper bound for g(n). This means no values of f(n) are below g(n)'s bounds. In other words, f(n) $\geq$ c*g(n).

$f(n) \in \Omega(g(n))$ means that the set of f(n) is the lower bound of g(n) for all values of n. In other words, f(n) $\geq$ c*g(n)

It would be possible for f(n) and g(n) to fulfill both cases.

$$f(n) = 4n, g(n) = 4n$$

f(n) would be Θ(g(n)) which means that f(n) is also $\Omega(g(n))$. No value of f(n) are below g(n) since f(n) and g(n) are on the same boundary.

f(n) would not be o(g(n)) because no values of f(n) lie under the boundary of g(n). When f(n) and g(n) are equal, then it f(n) $\notin$ o(g(n)).

Brandon Vo

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).

1-1 Comparison of running times

For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

Solving n takes 1 microsecond or 10^6 microseconds.

$$1\ microsecond\ to\ do\ each\ of\ n\ operations\colon\ n = \frac{1\ operation}{1\ microsecond}$$

$$1\ second = 10^6\ microseconds = \frac{n\ operations}{1\ microsecond}$$

$$n = 10^6 operations\ for\ 1\ second$$

For the rest, it will be multiplying by an additional 10^X operations where X is the number of additional seconds needed to convert from seconds to minutes, hours, days, etc.

$$n = 10^6\ for\ 1\ second$$

$$n = 10^6 * 60\ for\ 1\ minute$$

$$n = 10^6 * 60 * 60\ for\ 1\ hour$$

$$n = 10^6 * 60 * 60 * 24\ for\ 1\ day$$

$$n = 10^6 * 60 * 60 * 24 * 31\ for\ 1\ month$$

$$n = 10^6 * 60 * 60 * 24 * 31 * 12\ for\ 1\ year$$

$$n = 10^6 * 60 * 60 * 24 * 31 * 12 * 100\ for\ 1\ century$$

The rest of the asymptotic variants are a manner of having to modify the n value to match the standard n.

$$\lg n : \log_2 2^{10^6} = 10^6\ microseconds = 1\ second$$

$$60 \log_2 2^{10^6} = \log_2 2^{10^6 * 60} = 10^6 microseconds * 60 = 1\ minute$$

$$3600 * \log_2 2^{10^6} = \log_2 2^{10^6 * 60 * 60} = 10^6 microseconds * 3600 = 1\ hour$$

$$\log_2 2^{10^6 * 60 * 60 * 24} = 1\ day$$

$$\log_2 2^{10^6 * 60 * 60 * 24 * 31} = 1\ month$$

$$\log_2 2^{10^6 * 60 * 60 * 24 * 31 * 12} = 1\ year$$

Brandon Vo

$$\log_2 2^{10^6*60*60*24*31*12*100} = 1\ century$$

$To\ translate\ from\ n\ to\ n^2\ and\ n^3, the\ operations\ would\ be\ reduced\ by\ \sqrt{N}\ and\ \sqrt[3]{N}\ respectively$

$$10^6 = n^2$$

$$n = \sqrt{10^6}$$

$$10^6 = n^3$$

$$n = \sqrt[3]{10^6}$$

$$2^n = n$$

$$2^n = 10^6$$

$$n = 6\log_2 10$$

$$n = 6\log_2 10 = 19.93\ for\ 1\ second$$

$$1\ minute\text{:}\ n = \log_2(10^6 * 60) = 25.838$$

$$1\ hour\text{:}\ n = \log_2(10^6 * 60 * 60) = 31.745$$

$$1\ day\text{:}\ n = \log_2(10^6 * 60 * 60 * 24) = 36.3303$$

$$1\ month\text{:}\ n = \log_2(10^6 * 60 * 60 * 24 * 31) = 41.2845$$

$$1\ year\text{:}\ n = \log_2(10^6 * 60 * 60 * 24 * 31 * 12) = 44.869$$

$$1\ century\text{:}\ n = \log_2(10^6 * 60 * 60 * 24 * 31 * 12 * 100) = 51.513$$

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lg n	$2^{10\text{^}6}$	$2^{60*10\text{^}6}$	$2^{10\text{^}6*60*60}$	$2^{10\text{^}6*60*60*24}$	$2^{10\text{^}6*60*60*24*31}$	$2^{10\text{^}6*60*60*24*31*12}$	$2^{10\text{^}6*60*60*24*31*12*100}$
$\sqrt{n}$	10^{12}	10^{12} *60	10^{12}*60*60	10^{12}*3600*24	10^{12}*86400*31	10^{12}*2678400*12	10^{12}*32140800*100
n	10^6	10^6 *60	10^6*60*60	10^6*3600*24	10^6*86400*31	10^6*2678400*12	10^6*32140800*100
N^2	10^3	10^3 *60	10^3*60*60	10^3*3600*24	10^3*86400*31	10^3*2678400*12	10^3*32140800*100
N^3	10^2	10^2 *60	10^2*60*60	10^2*3600*24	10^2*86400*31	10^2*2678400*12	10^2*32140800*100
2^n	19.93	25.838	31.745	36.33	41.284	44.860	51.51

Brandon Vo

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])

This pseudocode runs a double for-loop where the first loop will iterate over the entire array. For each element that the first loop comes across, the second loop will initiate and check the rest of the array for any elements that match the element being checked on.

However, at line 2), the second loop, j, will begin at index I which is the same index as the first for loop. Because this is the same index, it will always fulfill the if condition and print the element. As a result, this will print every element in the array regardless if there is a duplicate or not.

If we use an array [1, 2], then the function would see check A[1] == A[1], A[1] == A[2], and A[2]==A[2]. This would print out 1, 2 despite the only two elements not being duplicates of each other.

The fix for this would be in line 2) to make $j = i + 1$. In line 1), i should go up to A.length-1 to prevent going out of bounds as well.

Loop Invariant proof:

1. **Initialization**: When function initially runs, it checks A[i] == A[j] which can be translated to be A[i]==A[i+1].
    a. For the outer loop invariant, all elements contained in the subset A[1…i] are elements that have already been checked to see if they have or do not have a duplicate in their set.
    b. The inner loop invariant will use the array set of A[i+1…A.length]. At the start of the inner loop iteration, the set A[i+1…A.length] has yet to be proven to be a duplicate element of subset A[1…i].
2. **Maintenance**: The loop invariant will be proven if the loop step proves that A[i] can be checked to see if it does or does not contain a loop from A[i+1…A.length]
    a. Assuming the outer loop will begin on A[i], the inner loop will begin on A[j] which will start as A[i+1].

   b. J will be incremented for each iteration and checked to see if the element is a duplicate of A[i]. The rest of the set A[j+1…A.length] will be the subset yet to be checked to be a duplicate of A[i].
   c. The termination condition is met when A[j] reaches the end of the array at A.length. At this point, all elements from the inner loop A[j…A.length] have been checked to see if they match with A[i]. This means A[i] has been checked and proven if it does or does not have a duplicate value in the set, proving the outer loop invariant property true.
3. **Termination**: The outer loop terminates when j reaches A.length. The inner loop terminates when the loop reaches the end of the array as well.
   a. When the outer loop terminates upon reaching the end, then the entire set of A[1…i] will be the set of elements proven to have or not have a duplicate value somewhere in the array.
   b. When the inner loop terminates upon reaching the end, then the element in A[i] has been proven to either have or not have a duplicate element located in the set A[j…A.length], proving the loop invariant property to be true.

Brandon Vo

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

[1]Make sure you have the third edition.

[2]You do not need to prove our answers. The big-Oh and big-Omega bounds must be tight.

a)

Times Executed	# of Instructions
1	c_1
N	c_2
$\sum_{j=i}^{n} t_j$	c_3
$\sum_{j=i}^{n} (t_j - 1)$	c_4
1	c_5

$$T(n) = c_1 + c_2 n + c_3 \sum_{j=i}^{n} t_j + c_4 \sum_{j=i}^{n} (t_j - 1) + c_5$$

$$T(n) = c_1 + c_2 n + c_3 * \frac{n(n+1)}{2} + c_4 * \frac{n(n-1)}{2} + c_5 \colon\ O\left(\frac{n^2 + n}{2}\right)$$

Brandon Vo

$$o(n^3), \Omega(n^2), \theta(n^2)$$

b)

Times Executed	# of Instructions
1	c_1
n	c_2
n-1	c_3
1	c_4

$$T(n) = c_1 + nc_2 + (n-1)c_3 + c_4 = O(n)$$

$$o(n^3), \Omega(n)$$

$$\theta(n), O(n), \Omega(n), \theta(n)$$

Brandon Vo

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 j^{th} item belongs, he instead used binary search to find where the j^{th} item belongs. Is he right?

function Insertion-Sort(A)

```
1)  for j = 2 to A.length
2)    key = A[j]
3)   // Insert A[j] into the sorted sequence A[1..j-1]
4)    i = j - 1
5)    while i > 0 and A[i] > key
6)        A[i + 1] = A[i]
7)        i = i – 1
8)    A[i + 1] = key
```

While binary search is faster to find where an element needs to be put into, it is not asymptotically fast enough to make a difference on Insertion Sort for a worst-case scenario. When A[j] is found, the function still has to shift the subarray to make room for A[j] due to line 6).

```
6)        A[i + 1] = A[i]
```

All elements from A[i] to A[1] still have to be shifted to the left to make room so that A[j] can be inserted in its sorted position. Even if binary search makes finding the spot faster, it still has to iterate over the pre-sorted array A[1]:A[i], meaning that the lines 5-7 will still share the same asymptotic time complexity of its original counterpart.

Brandon Vo

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, b(l + r)/2c)

5) rmax = maximum(A, b(l + r)/2c + 1, r)

6) print(rmax, lmax)

7) if rmax < lmax

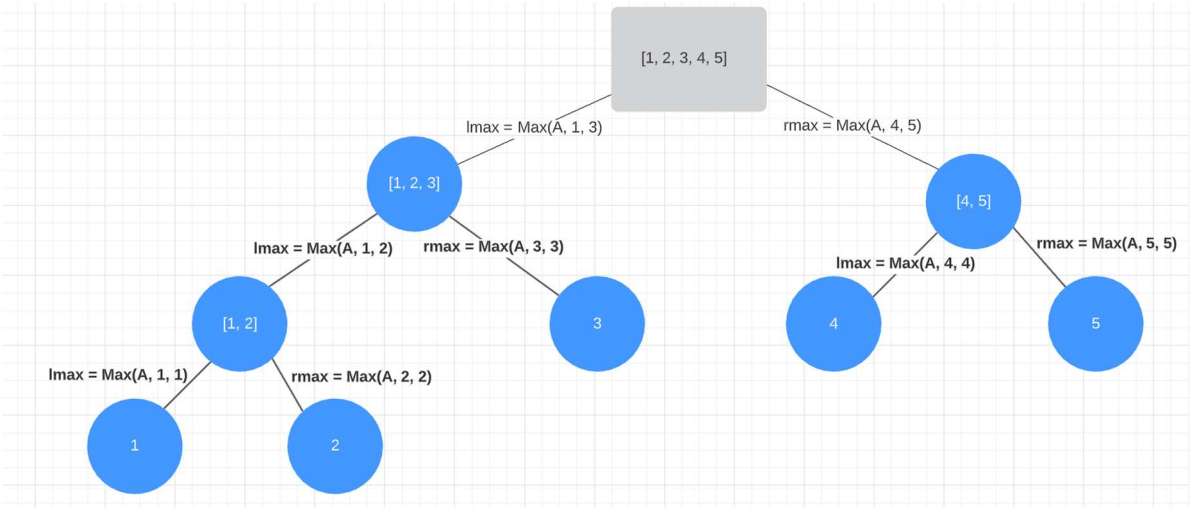
8) return lmax

9) else

10) return rmax

This function works similarly to the divide and conquer algorithm where the function recursively divides itself in half. It prints out the two elements being compared before returning the largest element of the two. If there is only one element left, then that element is returned back.

This is how the Array of [1…5] would be processed by Maximum(A, 1, 5)



Since the maximum function starts with lmax, the print calls will begin from the left side of the decision tree and proceed rightward. The order would go as follows:

1. [1, 2]: Print(2, 1), return 2
2. [1, 2, 3]: Print(3, 1), return 3

3. [4, 5]: Print(5, 4), return 5
4. [1, 2, 3, 4, 5]: Print(5, 3)

This means that we will end up printing (2, 1), (3, 1), (5, 4), (5, 3).

Brandon Vo

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]

Variables: D dollars, n is the number of all stock names, Array STOCK is the array containing the stock names of all the stock options given out by all friends

Invest(D,N)

1) Create array New[n] //Create a new array to hold the set of STOCK

2) Set n = 0 //Will be adjusted to the match the number of all unique elements

3) for(i=1 to STOCK.length) //First index for array New

4) if(STOCK[i] was not marked as a duplicate value) //Ignore duplicates

5) n = n + 1 //A unique element has been found.

6) New[i]=STOCK[i] //Pushback New[i]

7) for(j = i+1 < N.length) //Second index for N

8) if(New[i] == STOCK[j]) //Check for duplicates

9) Mark STOCK[j] as a duplicate value

10) end for

11) end for

12)Set n to be the set of all non-empty elements found in New. //If there were duplicate elements in New, then the array of New shouldn't be full. Adjust the size of n to be the number of elements found in New to get the number of unique stock options.

This leaves us with the array New that contains the set of all stock options provided with its size adjusted to contain only the unique elements taken from N. Because it requires two for-loops that iterate over New and STOCK which start off as size n, then this algorithm would have the time complexity of $O(n^2)$. The worse case scenario would be that there are no duplicates in which case, STOCK and New will have to iterate over the entire array twice. This is because in the worst-time case scenario, the algorithm would essentially have to go over the list of n twice, one to help create array New and one to check array N for duplicates.

Brandon Vo

13. (3 bonus points) Think of a good[6] exam question for the material covered in Lecture 1

Write a version of Merge sort that has the best case time complexity of Ω(n). Why we aren't as concerned with how it could be Ω(n) as opposed to its worst-case scenario O(n log n)?