

CSE 101 Spring 2025 ,

Homework 1 is due: Tuesday, April 8 at 11:59pm

### Instructions

The following are general rules for homework problems.

Homework should be done in groups of one to five people. You are free to change group members at any time throughout the quarter. Problems should be solved together, not divided up between partners. A single representative of your group should submit your work through Gradescope. Submissions must be received by 11:59pm on the due date, and there are no exceptions to this rule.

Homework solutions should be turned in through Gradescope by 11:59pm on the due date. No late homeworks will be accepted for any reason. Please ensure that your submission is legible, with the bulk word-processed but with possibly hand drawn images or diagrams included.

Students should consult their textbook, class notes, lecture slides, instructors, TAs, and tutors when they need help with homework. Students should not look for answers to homework problems in other texts or sources, including the internet or by using AI. You can post clarification questions about graded homework problems on Piazza viewable by all, but questions about your solutions should be only viewable by instructors.

Your assignments in this class will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should always explain how you arrived at your conclusions, using mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported.

Your goal should be to convince the reader that your results and methods are sound.

For questions that require pseudocode, you can follow the same format as the textbook, or you can write pseudocode in your own style, as long as you specify what your notation means. For example, are you using “=” to mean assignment or to check equality? You are welcome to use any algorithm from class as a subroutine in your pseudocode. For example, if you want to sort list A using InsertionSort, you can call InsertionSort(A) instead of writing out the pseudocode for InsertionSort.

### Graded problems

1. True or False? (you must justify your claim. 2 points each for correct answer, 2 points each for short explanation.)

a.  $2^{n/2} \in \Theta(2^n)$

This claim is False.

$$2^{n/2} = (2^n)^{1/2}$$

$\sqrt{2^n}$  grows significantly slower than  $2^n$ , therefore,

$c * 2^n \leq \sqrt{2^n}$  is not true for any constant  $c$  as  $n$  goes to  $\infty$  so we cannot say that  $2^{n/2}$  may act

as a lower bound for  $2^{n/2}$  regardless of a constant multiple, and thus  $2^{n/2} \notin \Theta(2^n)$

b.  $(n^3 + 2n + 1)^4 \in \Theta((n^4 + 4n^2)^3)$

This claim is True.

One can see that the order of each polynomial will be 12, as both the left and right side have a leading  $n^{12}$  term when simplified. Because they have the same order,

$$(n^3 + 2n + 1)^4 \in \Theta((n^4 + 4n^2)^3)$$

c.  $\log(n^{20}) \in \Theta(\log n)$

This claim is True.

$\log(n^{20}) = 20 * \log(n)$ , which is equivalent to  $c * \log(n)$  where  $c$  is 20.

Thus, the equation  $c * \log(n) = \log(n^{20})$  is satisfied. Furthermore, we can see that

$c_1 * \log(n) \leq \log(n^{20}) \leq c_2 * \log(n)$ , indicating that  $\log(n^{20})$  is in fact bounded by a constant, so  $\log(n^{20}) \in \Theta(\log(n))$

d.  $\sum_{i=1}^n i^2 \in \Theta(n^3)$

This claim is True.

$$\sum_{i=1}^n i^2 = 1 + 4 + 9 + \dots + n^2$$

We know that  $\sum_{i=1}^n i = n(n + 1)/2$

We may find a closed form to determine the result of the summation at any given  $n$ . Our summation follows the pattern:

1, 5, 14, 30, 55, 91, 140...

This seems to be some form of cubic polynomial, so we will solve for polynomial coefficients.

$$\sum_{i=1}^n i^2 = An^3 + Bn^2 + Cn + D, D = 0 = An^3 + Bn^2 + Cn$$

Forming a system of 3 equations for 3 variables

When  $n = 1, A + B + C = 1$

When  $n = 2, 8A + 4B + 2C = 5$

When  $n = 3, 27A + 9B + 3C = 14$

Solve for A, B, C

$$6A + 2B = 3$$

$$24A + 6B = 11$$

$$6A = 2$$

$$A = 1/3, B = 1/2$$

$$C = 1 - 1/2 - 1/3 = 1/6$$

So the closed form can be expressed as  $1/3(n^3) + 1/2(n^2) + 1/6(n)$ ,  
 $= ((1/3n + 1/2) * n) + 1/6 * n$  (Horner's method)

We know this is a third order polynomial regardless, which indicates a leading term of  $n^3$ .

Thus,  $\sum_{i=1}^n i^2 \in \Theta(n^3)$

e.  $n! \in \Theta(n^n)$

This claim is False.

$n! = n(n-1)(n-2)\dots(1) \leq n * n * \dots * n$ ,  $n$  times. Also,

$n(n-1)\dots(1) * c \leq n * n * \dots * n = n^n$  for all  $c$ , as a constant  $c$  will not allow the left hand side to scale faster than the right for any  $c$  as  $n$  goes to  $\infty$ .

Therefore,  $n!$  can never act as an upper bound for  $n^n$ , thus,  $n! \notin \Theta(n^n)$ .

2. The Fibonacci numbers  $F_0, F_1, \dots$ , are defined by  $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$  for  $n \geq 2$ . Prove that for all fixed integers  $k \geq 0, F_{n+k} \in \Theta(F_n)$ . (20 points.)

Claim: For all fixed integers  $k \geq 0, F_{n+k} \in \Theta(F_n)$

Base case: When  $k = 0, F_{n+0} = F_n \in \Theta(F_n)$

Induction Hypothesis:

Assume that  $F_0 \dots F_i$  terms, where  $i \geq 0, \in \Theta(F_n)$

Inductive Step:

$$F_n \leq F_i + F_{i-1} \leq 2F_n \in \Theta(F_n)$$

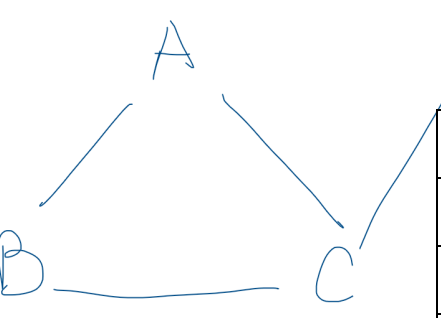
$$F_n \leq F_{i+1} \leq 2F_n \in \Theta(F_n)$$

Conclusion: By the induction principle, for all fixed integers  $k \geq 0, F_{n+k} \in \Theta(F_n)$



Susan's:	2	4	5	8	10	12							
Sum:	1	3	4	8	9	15	16	24	25	35	36	48	49

Example on adjacency matrix



	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	1	1	0	1
D	0	0	1	0

(b) Correctness proof:

Necessity proof ( if  $G$  contains Triangle then return True)

Suppose there exist an triangle in  $G$ , composed by three different vertices  $u, v, w$ , i.e.  $A[u][v] = 1$ ,  $A[u][w] = 1$ ,  $A[v][w] = 1$ .

When the algorithm runs, after iteration of  $v$  and  $w$ ,  $A[v][w] = 1$  will be detected. Then when iterating over all vertices  $u \neq w$  and  $u \neq v$ , there must have  $A[u][v] = 1$  and  $A[u][w] = 1$ , so the algorithm returns True.

Sufficiency proof ( if algorithm returns True then  $G$  contains Triangle)

If the algorithm returns true, then we know there exist a pair of vertex  $v, w$  satisfy that  $A[v][w] = 1$ , and there is  $u$  i.e.  $A[u][v] = 1$  and  $A[u][w] = 1$ . So the three vertices are pairwise connected by edges, forming a triangle.

(c) Time analysis and efficiency

Considering the worst situation, in the first loop we iterate all the vertex pairs  $(v, w)$ , resulting in a time complexity of  $O(|V|^2)$ .

Then, for each valid edge, found with  $A[v][w] == 1$ , we perform a loop across all vertices once again to determine if there is an edge  $\{u, v\}$  and  $\{u, w\}$ . This means the time complexity will increase by the number of valid edges times the number of vertices we must also check, or  $|E||V|$ .

Adding above contributions, we get  $O(|V||E| + |V|^2)$  time algorithm.

5. Let  $A[1..n]$  be an unsorted array of distinct integers. We wish, for each position  $1 \leq i \leq n$ , to find  $NextLarger[i]$ , the smallest index  $j$  with  $j > i$  and  $A[i] < A[j]$ ; we define  $NextLarger[i] = n + 1$  if no such  $j$  exists. Give an  $O(n)$  time algorithm to compute the array  $NextLarger[1..n]$ . Hint: Go through the elements from 1 to  $n$ , and use a data structure to keep track of those positions where  $NextLarger$  has not yet been defined. (20 points: 5 points for clear description of algorithm, 5 points for short correctness proof, and 10 points for time analysis and efficiency.)

To solve this problem, we can scan through the array while using a stack to keep track of the indices where we don't know what the next larger indices of those indices yet.

A stack is the most effective data structure for this problem because any indices at the bottom of the stack have the  $arr[index]$  larger than any of the  $arr[index]$  indices that come into the stack later. Otherwise, we would have recorded the next larger index and popped the earlier index from the stack. Moreover, when we compare and the  $arr[current\ index] < arr[index\_top\_of\_stack]$ , then there is no further need to compare with any other indices lower down in the stack since the elements on top of the stack has to be smaller than the ones in the bottom. When we finish the scan, anything left in the stack is the indices of element where we couldn't find the next larger, so we assign  $n+1$ .

Algorithm:

```
def nextLargerArray(arr: List)->list:
    n=len(arr)
    stack=[]
    nextLarger= [0]*(n+1)
    For i in range (1,n+1):
        While not stack.isEmpty():
            undefined_pos_index = stack.peek()
            If arr[undefined_pos_index] < arr[i]:
                stack.pop()
                nextLarger[undefined_pos_index]=i
            Else:
                break
        stack.append(i)
    While not stack.isEmpty():
        undefined_pos_index =stack.peek()
        stack.pop()
        nextLarger[undefined_pos_index]=n+1
    Return nextLarger
```

Proof:

Loop invariant: At the start of each iteration, 2 properties are true:

1. The indices inside the stack have not found their next larger index
2. The stack is strictly decreasing-ordered. The indices on top of the stack have element value  $<$  the element value of indices lower in the stack. Because otherwise the lower index would have been popped because the new incoming index was the lower index's next larger index.

Base case:

- Before the for loop starts, the nextLarger array is empty, and the stack is empty, so it is vacuously true that nextLarger contains the next larger indices of 0 elements, and the stack contains 0 indices of elements that haven't found the next larger index. The invariant holds

Assume after  $t$  iterations: if  $\text{arr}[t]$  is added to the stack, then either the next larger index of  $\text{arr}[t]$  has not been found yet. And the value of  $\text{arr}[t]$  follows the order of the stack. The loop invariant holds

After  $t+1$  iterations:

- We peek the stack to compare, if  $\text{arr}[t+1] > \text{arr}[\text{undefined\_pos\_index}]$  then we pop the  $\text{undefined\_pos\_index}$  from the top of the stack and update the nextLarger array at  $\text{undefined\_pos\_index} = i$ .
- We repeatedly check the next  $\text{undefined\_pos\_index}$  of the stack until either the stack is empty or  $\text{arr}[t+1] < \text{arr}[\text{undefined\_pos\_index}]$ . We can stop checking because the element of the index lower in the stack is always larger than those upper of the stack.
- When we finish checking, we append the index  $t+1$  to the stack, the invariant still holds because the  $\text{arr}[t+1]$  has a value that is smaller than the indices at a lower position in the stack. The invariant is maintained

Termination:

When we have iterated over  $n$  elements, any elements that remain in the stack must have no larger element in the array, so we can pop them and assign them with  $n+1$  in their corresponding nextLarger array

## II. Time complexity analysis

def nextLargerArray(arr: List)->list:

$n = \text{len}(\text{arr}) \Rightarrow O(1)$

$\text{stack} = [] \Rightarrow O(1)$

$\text{nextLarger} = [0] * (n+1) \Rightarrow O(1)$



```

For i in range (1,n+1): => O( n + 8 ) => O(n)
    While not stack.isEmpty(): => O(7)
        undefined_pos_index = stack.peek() => O(1)
        If arr[undefined_pos_index] < arr[i]: => O(1)
            stack.pop() => O(1)
            nextLarger[undefined_pos_index]=i => O(1)
        Else: => O(1)
            break => O(1)
    stack.append(i) => O(1)

While not stack.isEmpty(): => O(n)
    undefined_pos_index =stack.peek() => O(1)
    stack.pop() => O(1)
    nextLarger[undefined_pos_index]=n+1 => O(1)

Return nextLarger => O(1)

```

$$T(n) = O(n+8) + O(n) + c = O(n)$$

### Ungraded problems

1. True or False? (you must justify your claim.)
  - a.  $2^{2^n} \in \Theta(2^n)$
  - b.  $(n^2 + 2n + 1)^3 \in \Omega((3n^3 + 4n^2)^2)$
  - c.  $(\log(n))^{10} \in \Theta(\log(n))$
  - d. For constant  $k > 1$ ,  $\prod_{i=1}^n k^i \in \Theta(k^{n+1})$
  - e.  $n! \in O(n^n)$
2. Assume that  $f$  is a non-decreasing function from integers to positive integers. Assume  $f(2n) \in O(f(n))$ . Prove that, for all integers  $c > 0$ ,  $f(cn) \in \Theta(f(n))$ .
3. The Fibonacci numbers  $F_0, F_1, \dots$ , are defined by  $F_0 = 0$ ,  $F_1 = 1$ ,  $F_n = F_{n-1} + F_{n-2}$  for  $n \geq 2$ 
  - a. Prove that  $F_n + F_{n-1} + F_{n-2} + \dots + F_0 = F_{n+2} - 1$
  - b. Using the above claim, prove that  $F_n + F_{n-1} + \dots + F_0 \in \Theta(F_n)$
4. The *reverse* of a directed graph  $G = (V, E)$  is another directed graph  $G^R = (V, E^R)$  on the same vertex set, but with all edges reversed; that is,  $E^R = (u, v) : (v, u) \in E$ . Give a  $O(|V|^2)$  time algorithm that takes a directed graph  $G$  in adjacency matrix format and outputs an adjacency matrix for  $G^R$ .
5. A *tournament* is a directed graph where, for every pair of distinct vertices  $u, v$ , exactly one of the edges  $(u, v)$  and  $(v, u)$  are in the graph.
  - a. A Hamiltonian path is a directed path that goes through every vertex exactly once. Prove that every tournament has a Hamiltonian path.
  - b. Give an efficient algorithm that, given a tournament, finds a Hamiltonian path in the tournament, using your proof above. (Remember, for algorithm design problems, you always need to clearly describe the algorithm, prove its correctness, and justify its runtime.)
6. Let  $A[1..n]$  and  $B[1..n]$  be sorted arrays of distinct integers. We wish, for each position  $1 \leq i \leq n$ , to find  $\text{FirstLarger}[i]$ , the smallest index  $j$  with  $A[i] \leq B[j]$ ; we define  $\text{FirstLarger}[i] = n + 1$  if no such  $j$  exists. Give an  $O(n)$  time algorithm to compute the array  $\text{FirstLarger}[1..n]$ .

