Problem Set 1

1. Prove that $\log n! = \theta(n \log n)$. Use this to find out if $\lceil \log n \rceil!$ and $\lceil \log \log n \rceil!$ are polynomially bounded. A function $f(n)$ is said to be polynomially bounded if there exists an integer k such that $f(n) = O(n^k)$.

2. Iterated logarithmic function is defined as $\log^*(n) = min\{i \geq 0 : \log^i n \leq 1\}$. For example, $\log^* 16 = 3$ where log is taken w.r.t base 2. Which is aymptotically larger: $\log(\log^* n)$ or $\log^*(\log n)$?

3. Write an efficient algorithm that checks whether a given singly linked list contains a loop. A loop is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that $v_1 \to v_2 \to \ldots \to v_k \to v_1$.

4. Consider a stack with an additional operation, $MULTIPOP(S, k)$ which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. The cost of the opearation $MULTI - POP(S, k)$ is k, while that of $PUSH(S, x)$ and $POP(S)$ is 1. Now consider a sequence of n stack operations on an initially empty stack, where each operation is either $PUSH, POP$ or $MULTIPOP$. Prove that the total cost of these n operations is $\theta(n)$.

5. Design a data structure *SpecialStack* that supports all the stack operations like $push(), pop(), isEmpty(), top()$ and an additional operation $getMin()$ which should return minimum element from the *SpecialStack*. All these operations of *SpecialStack* must be $O(1)$. To design *SpecialStack*, you should only use standard Stack data structure and no other data structure like arrays, list, etc.

6. Describe a $O(n)$-time algorithm that, given a set S of n integers in sorted order and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

7. Given an unsorted array A of size n that may contain duplicates and a number $k < n$, design an $O(n)$ algorithm that returns *true* if array contains duplicates within a distance of k, ie. there exists $i, j \in [0, n-1]$ such that $A[i] = A[j]$ and $|i - j| < k$.

1. To prove: $\log n! = \Theta(n \log n)$

Proof: $\log n! = \log 1 + \log 2 + \log 3 + \log 4 + \cdots + \log n$

$\leq \log n + \log n + \log n + \cdots + \log n$

$\leq n \log n$

Thus, $\log n! \leq n \log n$

Also, $\log n! \geq \frac{n}{2} \log \frac{n}{2}$ $\left(\log \frac{n}{2} + \log \frac{n}{2} + \log \frac{n}{2} \cdots\right)$ (n/2 times (second half of the sum))

$\Rightarrow \log n! \geq \frac{n}{2} \log \frac{n}{2}$

$\therefore \frac{n}{2} \log \frac{n}{2} \leq \log n! \leq n \log n$

$\Rightarrow \log n! = \Theta(n)$ Proved.

2. 2. Iterated logarithmic function is defined as $\log^*(n) = min\{i \geq 0 : \log^i n \leq 1\}$. For example, $\log^* 16 = 3$ where log is taken w.r.t base 2. Which is aymptotically larger: $\log(\log^* n)$ or $\log^*(\log n)$?

$\log^*(n) = \min\{i \geq 0 : \log^i n \leq 1\} = 1 + \log^*(\log n)$

$\log_2^*(16) = \min\{i \geq 0 : \log^i 16 \leq 1\}$

$\log^*(16) = 1 + \log^*(4) = 2 + \log^*(2) = 3 + \log^*(1)$

$= 3$

We know that $\log^* n < \log n$

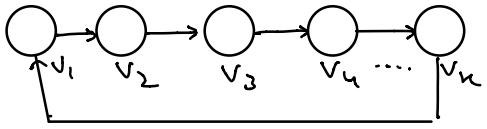
$\Rightarrow \log(\log^* n) < \log(\log n)$ (Taking log on both sides)

& $\log^*(\log n) < \log(\log n)$ (Replacing n by $\log n$)

$\log^*(n)$ is a very slow growing function.
For instance, $\log^*(2^{65536}) = 5$.
Taking log of this value will only compress it down further.

Thus, $\log(\log^* n) < \log^*(\log n)$

3.



Algorithm findLoop(Node head):
Input: Head of the linked list.
Output: return True if loop detected.

```
Node current ← head
while (current != null) do:
    if (current.visited = 1):
        return True
    current.visited ← 1
    current ← current.next
return False
```

Method 2: Floyd's Cycle Detection Algorithm (Hare–Tortoise Algorithm)

Algorithm findLoop(Node head):
Input: Head of the linked list.
Output: return True if loop detected.

```
Node slow ← head
Node fast ← head
while ((slow != null) and (fast != null) and (fast.next != null)):
        slow ← slow.next
        fast ← fast.next.next
        if (fast = slow):
            return True
return False
```

5. SpecialStack → push(), pop(), isEmpty(), top(), getMin()

class SpecialStack:

```
currentMin ← inf
Stack mainStack; Stack minStack;

Algorithm push(element):          ⟶ this is the builtin stack push method
    mainStack.push(element);
    if (element < currentMin):
        minStack.push(element)
        currentMin ← element
```

```
Algorithm pop():
    if (mainStack.isEmpty()):
        return error
    if (mainStack.top() = currentMin):
        minStack.pop()
        if (not minStack.isEmpty()):
            currentMin ← minStack.top()
        else:
            currentMin ← inf
    mainStack.pop()        ← builtin pop method
```

```
Algorithm isEmpty():
    if (mainStack.isEmpty()):        → builtin method
        return True
    return False
```

```
Algorithm top():
    if (mainStack.isEmpty()):
        return error
    return mainStack.top()
```

```
Algorithm getMin():
    return currentMin
```

6:

```
Algorithm checkSum(S[], n, x):
Input: Sorted set of n integers S[0....n-1] and target value x
Output: True if ∃ i,j s.t. S[i] + S[j] = x, else False

    i ← 0 ;  j ← n-1.
    while (i<j):
        if (S[i] + S[j] = x):
            return True
        else if (S[i] + S[j] > x):
            j ← j-1
        else:
            i ← i+1
    return False
```

7: Ex: $[1, 2, 7, 9, 9, 4, 5, 9]$, $k=2$

map:
1 - [0]
2 - [1]
7 - [2]
9 - [3,4,7]

Algorithm detectDuplicates(int A[], int n, int k):
Input: Unsorted array A[0...n-1] of size n, target distance k.
Output: return True if $\exists\, i,j \in [0, n-1]$ s.t. $A[i] = A[j]$ & $|i-j| < k$.

```
HashMap map;      //Map from Integer -> Vector of Integers

for i ← 0 to (n - 1) do :
    map[A[i]].add(i)

for key in map:
    for i ← 0 to (key.value.size-2):
        if(key.value.get(i+1) - key.value.get(i) < k):
            return True
return false
```

Problem Set 2

1. Let A be an array of n elements with the following property: there exists an $i \in [0, n-1]$ such that $A[0] < A[1] < \ldots A[i-1] < A[i] > A[i+1] > \ldots A[n]$. Show how to find such an i in $O(\log n)$ time.

2. Given a balanced binary search tree on n nodes and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is $O(n)$ and only $O(\log n)$ extra space can be used. Any modification to binary search tree is not allowed.

3. You are given two binary search trees (not necessarily balanced). Design an algorithm that merges the two given trees into a balanced binary search tree in linear time.

4. Prove that the result of inserting any increasing sequence of $2^k - 1$ numbers into an initially empty AVL tree results in a perfectly balanced tree of height $k - 1$.

5. Call a family of trees balanced if every tree in the family has height $O(\log n)$, where n is the number of nodes in the tree. For each property below, determine whether the family of binary trees satisfying that property is balanced. If you answer is "no", provide a counterexample. If your answer is "yes", give a proof.

   - Every node of the tree is either a leaf or it has two children.
   - The size of each subtree can be written as $2^k - 1$, where k is an integer (k is not the same for each subtree).
   - There is a constant $c > 0$ such that, for each node of the tree, the size of the smaller child subtree of this node is at least c times the size of the larger child subtree.
   - There is a constant c such that, for each node of the tree, the heights of its children subtrees differ by at most c.
   - The average depth of a node is $O(\log n)$. (Recall that the depth of a node x is the number of edges along the path from the root of the tree to x.)

6. Given an array of n elements, where each element is at most k away from its target position (its position in sorted array), design an algorithm that sorts the array in $O(n \log k)$ time.

1. $A[0 \ldots n-1]$. $\exists\, i \in [0, n-1]$ s.t. $A[0] < A[1] < A[2] \ldots A[i-1] < A[i] > A[i+1] \ldots > A[n-1]$

Algorithm binarySearch (int A[ ], int n):

Input: $A[0 \ldots n-1]$. $\exists\, i \in [0, n-1]$ s.t. $A[0] < A[1] < A[2] \ldots A[i-1] < A[i] > A[i+1] \ldots > A[n-1]$

Output: index i which satisfies the above condition.

```
// Also handle corner cases (i = 0 and i = n-1)
low ← 0;  high ← n-1
while (low <= high):
    mid ← (low + high) / 2
    if ((A[mid-1] < A[mid]) and (A[mid] > A[mid+1])):
        return mid
    else if ((A[mid-1] < A[mid]) and (A[mid] < A[mid+1])):
        low ← mid+1
    else if ((A[mid-1] > A[mid]) and (A[mid] > A[mid+1])):
        high ← mid-1

return -1
```

3. You are given two binary search trees (not necessarily balanced). Design an algorithm that merges the two given trees into a balanced binary search tree in linear time.

```
Algorithm mergeBSTs (Node head1, Node head2):
    // first generate the sorted vectors of BST1 & BST2.
       Then merge these two vectors.
       Then make balanced BST using median method.
    vector1 ← inorder(head1)
    vector2 ← inorder(head2)
    size1 ← vector1.size();  size2 ← vector2.size()
    mergedVector ← merge(vector1, vector2)
    root ← makeBalancedBST(mergedVector, 0, size1 + size2 - 1)
    return root

Algorithm inorder (Node root):
// Returns a vector containing nodes of the tree in sorted order.
    Vector v;
    if (root = null):
        return
    inorder(root.left)
    v.add(root.data)
    inorder(root.right)
    return v;
```

```
Algorithm merge (Vector v1, Vector v2):
// Returns a sorted vector generated by merging v1 & v2
    Vector v3;
    i ← 0;   j ← 0;
    while((i < v1.size()) and (j < v2.size()):

        if(v1.get(i) < v2.get(i)):
            v3.add(v1.get(i))
            i ← i+1
        else:
            v3.add(v2.get(j))
            j ← j+1
    while(i < v1.size()):
        v3.add(v1.get(i))
    while(j < v2.size()):
        v3.add(v2.get(j))
    return v3
```

```
Algorithm makeBalancedBST(Vector v, int start, int end):
    n ← v.size()
    Node root;
    if(start > end):
        return
    mid ← (start + end)/2
    root.data ← v.get(mid)
    root.left ← makeBalancedBST(v, start, mid-1)
    root.right ← makeBalancedBST(v, mid+1, end)

    return root
```

COL106: Solutions for Quiz-2

(October 8, 2020)

Question 1. What is the worst case time complexity of the modified version of insertion sort algorithm where we use binary search for finding the correct position for inserting the next number?

(A) $O(n)$

(B) $O(n \log n)$

(C) $O(n^2)$

(D) $O(n^2/\log n)$

Answer: (C). Irrespective of the time taken for searching for finding the correct position, it will take $j-1$ swaps in the worst case to insert the j^{th} element in the sorted list. Hence the total time will be $\sum_{j=2}^{n}(j-1)$ in the worst case, which is $O(n^2)$.

Question 2. A stack sortable permutation is defined as a permutation whose elements may be sorted by an algorithm whose internal storage is limited to a single stack data structure. For example, the permutation (2,1,3) can be sorted as follows:

```
Push 2 from input to stack
Push 1 from input to stack
Pop 1 from stack to output
Pop 2 from stack to output
Push 3 from input to stack
Pop 3 from stack to output.
```

However, the permutation (2,3,1) cannot be sorted in this manner. Which of the following is not a stack sortable permutation?

(A) (5,1,4,2,3)

(B) (3,4,5,1,2)

(C) (5,4,1,2,3)

(D) (1,2,3,5,4)

Answer: (B). Try out the examples yourself. It is interesting to note that a stack sortable permutation always takes the following form: If we consider the sequence in sorted non-decreasing order: let this sequence be $<s_1, s_2, s_3, \ldots, s_n>$. Then the permutation is stack

sortable if and only if there exists an integer i: $1 \leq i \leq n$, such that, it can be obtained by any interleaving of the two sequences: $<s_1, s_2, \ldots, s_i>$ and $<s_n, s_{n-1}, \ldots, s_{i+1}>$.

Question 3. Compute the time complexity of the following code :

```
int sum = 0, j = 1;
int n = <some value>

for (int i =0; i<n; i++)
{
    while (j<i)
    {
        sum += 1;
        j *= 2;
    }
}
```

(A) $\Theta(n \log n)$

(B) $\Theta(n^2)$

(C) $\Theta(\log n)$

(D) $\Theta(n)$

Answer: (D). The for loop is executed $\Theta(n)$ times. Thus, the while loop condition will be checked $\Theta(n)$ times. Note that j is only initialized once in the beginning. It is not reinitialized every time in the i loop. The while loop will only be entered when $j = i - 1$ and it is doubled in the loop. Thus the loop statements will never execute more than once in any given iteration of the i loop. Hence the while loop will be executed $O(n)$ times (actually one can argue that they will only be executed $O(\log n)$ times, but that does not change the overall complexity).

Question 4. What exactly is average case time complexity?

(A) The average of best case and worst case time complexity.

(B) The time complexity averaged over all possible input instances.

(C) The worst case time complexity averaged over all possible values of n, the input size

(D) It is worst case time complexity itself

Answer: (B). By definition.

Question 5. Which of the following statements is/are correct?

1. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$, then $f(n) + d(n) = O(min(g(n), e(n))$
2. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$, then $f(n) \cdot d(n) = O(g(n) \cdot e(n))$

3. If $f(n)$ is $O(g(n))$, then $g(n)$ is $\Omega(f(n))$

4. If $f(n) = \Theta(g(n))$, then $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$

(A) 1,3,4

(B) 1,2,4

(C) 1,2,3

(D) 2,3,4

Answer: (D). It is easy to see that Statement 1 is not true by taking $f(n) = n$ and $g(n) = n^2$. The other statements can be verified by applying the definitions. A formal proof is illustrated here for Statement 4. Try to prove the other statements formally yourself.

Proof. $f(n) = \Theta(g(n)) \implies$ there exist constants c_0, c_1, n_0, n_1 such that

$$f(n) \leq c_0 \cdot g(n) \text{ for all } n \geq n_0 \text{ and}$$

$$f(n) \geq c_1 \cdot g(n) \text{ for all } n \geq n_1.$$

Let $n_2 = \max\{n_0, n_1\}$. Then

$$c_1 \cdot g(n) \leq f(n) \leq c_0 \cdot g(n) \text{ for all } n \geq n_2.$$

In other words,

$$\frac{1}{c_0} \cdot f(n) \leq g(n) \leq \frac{1}{c_1} \cdot f(n) \text{ for all } n \geq n_2.$$

This means that $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$. □

Question 6. Following is an incorrect pseudocode for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

```
declare a character stack
while ( more input is available )
{
    read a character
    if ( the character is a '(' )
        push it on the stack
    else if ( the character is a ')' and the stack is not empty )
        pop a character off the stack
    else
        print "unbalanced" and exit
}
print "balanced"
```

Which of these unbalanced sequences does the above code think is balanced?

(A) ((())

(B) ())(()

(C) (()()))

(D) (()))()

Answer: (A). Try all the inputs. It is interesting to note that the program accepts all inputs that are prefix of a sequence of parentheses that are balanced. A correct implementation would just need to check at the end that the stack is empty.

Question 7. In the lecture, we have been taught two strategies for increasing Stack size, namely, Growth Strategy and Tight Strategy. Now, we want to analyze what-if we set the size of the array to the square of the current Phase (as in lecture) every time we reach the end. This way, we can start with an array size of 1, then square 2 in phase 2 to get an array of size 4, then 9, then 16 and so on.

What will be the complexity, in this case, to fill the stack up to n ?

(A) $O(n^2)$

(B) $O(n \log n)$

(C) $O(n^{3/2})$

(D) $O(n^2 \log n)$

Answer: (C). In phase i, $(i-1)^2$ elements need to be copied and then the stack can be filled with another $i^2 - (i-1)^2$ elements. Thus the complexity of the i^{th} phase is $\Theta(i^2)$. In all to fill n elements there will be $\sqrt{n}$ phases. Hence the total complexity will be:

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

Hence the best answer is $O(n^{3/2})$.

Question 8. How many swaps are required to sort the following array using insertion sort?

{ 15, 1, 10, 2, 5, 4, 8 }

(A) 13

(B) 15

(C) 11

(D) 18

Answer: (C). You should be able to check yourself. The number of swaps in iterations 1,2,3,4,5,6 are 1,1,2,2,3,2 respectively for a total of 11.

Question 9. The characters of a string are pushed into a stack. Consider an optimal algorithm to check whether the string is palindrome using stacks only. What is the minimum number of **EXTRA** stacks that will be needed? Recall that a palindrome is a string that reads the same backwards, e.g. "racecar".

(A) 0

(B) 1

(C) 2

(D) Palindrome cannot be checked using Stacks only.

Answer: (B). Suppose the size of the string is n. There is a simple algorithm to check for a palindrome using one extra stack. We pop out $\lfloor n/2 \rfloor$ elements and push them into the extra stack one by one. We then repeatedly pop out 1 element from both the stacks and check if they are the same till no more elements are let in the stacks. Note that there will be one extra element in case the total size of the string is an odd number; this can easily be handled by discarding the extra (middle) element once $\lfloor n/2 \rfloor$ elements have been moved to the extra stack as this element does not need to be matched.

Question 10. Let F be the set of all functions from $N \leftarrow \Re^+$ and $f(n), g(n) \in F$. Consider the statements:

1. $f(n) = O(g(n))$
2. $f(n) = \Theta(g(n))$
3. $f(n) = \Omega(g(n)$

What can you say about these statements ?

(A) At least one of the above statements must be true for any given pair of functions $f(n)$ and $g(n)$

(B) For some functions $f(n)$ and $g(n)$, none of the statements need to be true

(C) More than one of the statements is true for any given pair of functions $f(n)$ and $g(n)$

(D) All statements are true for any given pair of functions $f(n)$ and $g(n)$

Answer: (B). Consider the functions:

$$f(n) = \begin{cases} n & \text{if } n \text{ is prime} \\ n^3 & \text{if } n \text{ is not prime} \end{cases}$$

and $\quad g(n) = n^2.$

Question 11. Let F be the set of all functions from $N \leftarrow \Re^+$ and $f(n), g(n) \in F$. Then which of the statements is true?

(A) $f(n) + g(n) = \Theta((max(f(n), g(n)))$

(B) $f(n) + o(f(n)) = \Theta(f(n))$

(C) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$

(D) $f(n) = \Theta(f(n/2))$

Answer: (A). Statements (A) and (B) are easy to check. Let us see why (C) and (D) are not true.

(C) Consider the functions $f(n) = 2n$ and $g(n) = n$. Then $2n$ is $O(n)$, but 2^{2n} is not $O(2^n)$.

(D) Consider the function $f(n) = 2^n$. Then 2^n is not $\Theta(2^{n/2})$.

Question 12. let $W(n)$ be worst case running time of an algorithm, $A(n)$ be it's average case running time and $B(n)$ be best case running time of algorithm. Then which of the following is true

(A) $A(n) + B(n)$ is $O(W(n))$

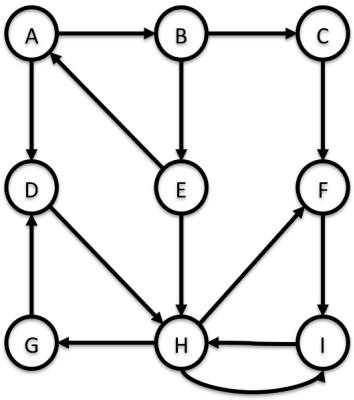
(B) $A(n) + B(n)$ is $\Theta(W(n))$

(C) $B(n) + W(n)$ is $\Theta(A(n))$

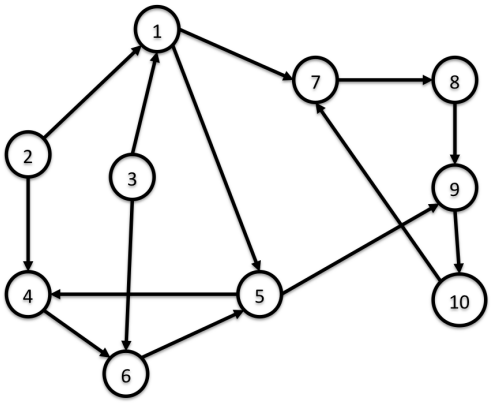
(D) $B(n) + W(n)$ is $O(A(n))$

Answer: (A). Clearly $B(n) \leq A(n) \leq W(n)$. Thus (A) is true. To see that the other options are not true, consider a program that takes n binary inputs. Thus there are 2^n input instances. Suppose that the algorithm runs in time $\Theta(1)$ for $2^n - 1$ of these input instances and takes time $\Theta(2^n)$ for one instance. Then $B(n) = \Theta(1)$, $A(n) = \Theta(1)$ and $W(n) = \Theta(2^n)$.

1. We know that the strongly connect components in any directed graph form a partition of vertices in the graph. So, the strongly connected components in a given graph can be represented as a partition of vertices. Consider the directed graphs G_1 and G_2 below and answer the questions that follow:



(a) Graph G_1



(b) Graph G_2

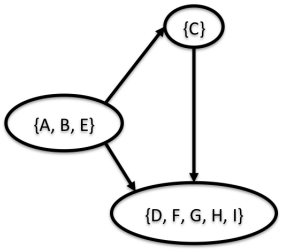
(a) (½ point) Give the strongly connected components of graph G_1.

(a) $\{A, B, E\}, \{C\}, \{D, G, H, F, I\}$

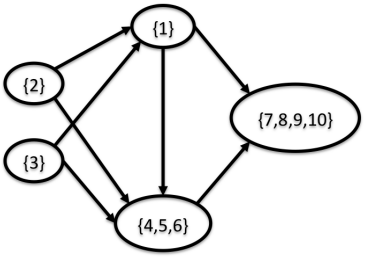
(b) (½ point) Give the strongly connected components of graph G_2.

(b) $\{1\}, \{2\}, \{3\}, \{4, 5, 6\}, \{7, 8, 9, 10\}$

(c) (½ point) Draw the graph G_1^{scc} corresponding to the graph G_1 (as discussed in the lectures).



(d) (½ point) Draw the graph G_2^{scc} corresponding to the graph G_2 (as discussed in the lectures).



2. (3 points) Design an algorithm that takes as input a directed acyclic graph G and determines if there is a directed path that visits every vertex exactly once. Give pseudocode and discuss correctness and running time.

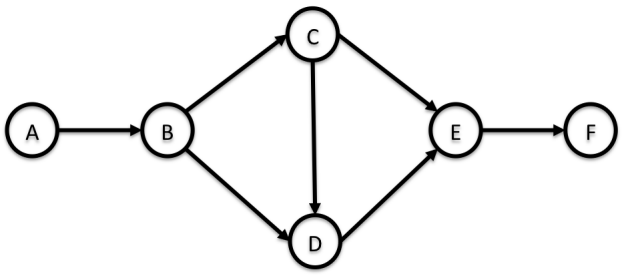


Figure 1: Example of a DAG that has a path that visits all vertices. This path is $A \to B \to C \to D \to E \to F$.

Solution: Consider the following algorithm for the problem:

```
VisitAllVertices(G)
  - For i = 1 to n
      - If there are more than one vertex with in-degree 0 in G,
        then return("No such path exists")
      - Let v be the unique vertex with in-degree 0 in G
      - Let G' be the graph obtained from G by removing v and its outgoing edges
      - G ← G'
  - return("such a path exists")
```

Proof of correctness: The proof of correctness for the above algorithm follows from the next two claims.

Claim 1: If the algorithm outputs "no such path" exists, then there is no path in the graph that visits every vertex once.

Proof. For sake of contradiction, assume that there is a path P in the graph that visits every vertex once. The algorithm outputs "no such path exists" only when there is an iteration i such that the number of in-degree 0 vertices in that iteration is at least 2. Suppose u and v are two vertices in iteration i that have in-degree 0 in iteration i. Without loss of generality, let u be the vertex that is visited before v as per the path P. This means that there is a path from u to v. This means that in the beginning of the i^{th} iteration the in-degree of v cannot be 0. This is a contradiction. □

Claim 2: If the algorithm outputs "such a path exists", then there is a path in the graph that visits every vertex.

Proof. The algorithm outputs "such a path exists" only when in every iteration of the for loop, it finds a unique vertex with 0 in-degree. Let the unique vertex in the i^{th} iteration be v_i. Then for all i, there is a directed edge from vertex v_i to vertex v_{i+1} in the graph (otherwise v_i will not be the unique vertex in iteration i with in-degree 0). This means that there is a path $v_1 \to v_2 \to ... \to v_n$ in the graph. □

Implementation and running time: In every iteration, we need to find vertices with in-degree 0. Suppose we maintain a set S that contains all the 0 in-degree vertices at the beginning of iteration i for all i. We will also maintain the in-degree of all the vertices in graph G at the beginning of iteration i for all i. Initially, we run the algorithm for finding the in-degrees for one of the previous homework problems to compute the in-degrees of all vertices of G and pick the ones with in-degree 0 to be put in set S.

Picking a vertex from the set S is an $O(1)$ operation. However, we will have to update this set S and in-degree information the end of every iteration. How much does that cost? Suppose in the i^{th} iteration, we pick a vertex v. For the next iteration, we remove this vertex and all the directed edges from v to vertices in G. This changes the in-degree of vertices in G'. So, if v has an edge to $u_1, ..., u_l$, then we decrease the in-degree of these vertices and if after decreasing the in-degree of a vertex, the in-degree becomes 0, then we add this vertex in the set S. The running time is proportional to the number of vertices plus the number of edges which is $O(n + m)$.

3. (3 points) Given a directed graph, design an algorithm that determines if there is a vertex from which all other vertices in the graph are reachable. Your algorithm should also output one such vertex in case such a vertex exists. Give pseudocode and discuss correctness and running time.

Solution: Here is the algorithm for this problem:

`VertexReachable(`G`)`

- Use the algorithm discussed in class for finding the strongly connected components of G
- Construct the graph G^{scc}
- If there is a unique vertex i in G^{scc} with in-degree 0, then output *any* vertex in the strongly connected component V_i corresponding to the vertex i
- Else output("No such vertex exists")

Proof of correctness: Let $V_1, V_2, ..., V_k$ denote the vertex sets of the strongly connected components of G and let the vertices of G^{scc} be denoted by $1, ..., k$. Note that G^{scc} is a DAG. The correctness of our algorithm follows from the next two claims.

Claim 1: If there is a *unique* source vertex i in G^{scc}, then all other vertices in G are reachable from any vertex $v \in V_i$.

Proof. For the sake of contradiction, let us assume that there is a vertex u that is not reachable from $v \in V_i$. Let u belong to the set V_j. First note that $j \neq i$ (otherwise u and v are in the same strongly connected component and hence reachable from one another). Let us start from j and travel backwards in G^{scc} along its in-coming edge to another vertex j_1 (such an edge exists otherwise j is also a source node). Now, we repeat the previous step and traverse backwards along the in-coming edge of j_1 and so on. Eventually, we will reach a vertex that we had already seen before. This gives a contradiction since this means that G^{scc} is not a DAG. □

Claim 2: If there are more than one source nodes in G^{scc}, then there does not exist a node from which all other nodes are reachable.

Proof. For any non-source node i in G^{scc}, V_i does not contain a node from which all other vertices are reachable because vertices in V_j corresponding to a source node j are not reachable from any vertex $v \in V_i$ in G. Now, consider any source node j in G^{scc}. This cannot be a node from which all nodes are reachable since there is another source node i in G^{scc} and we know that there cannot be a path from i to j in G^{scc} which implies that there is no path from any vertex $u \in V_i$ to any vertex in $v \in V_j$ in G. □

Running time: Finding the strongly connected components in a strongly connect graph takes $O(n+m)$ time. Constructing G^{scc} also takes time proportional to the size of the graph G which is $O(n+m)$. This is because all we need to do is to go through all edges (u, v) of G and then add edge from the strongly connected component containing u to the strongly connected component containing v. The size of G^{scc} is at most the size of G and finding whether there is a unique vertex with in-degree 0 takes $O(n + m)$ time. So, the total time for the algorithm is $O(n + m)$.

4. (1 point) Consider the Interval Scheduling problem discussed in the lecture. Given the following intervals as input, give the solution picked by `GreedySchedule` algorithm discussed in class:

$$I_1 = (11, 13), I_2 = (2, 7), I_3 = (6, 8), I_4 = (3, 5), I_5 = (4, 10),$$
$$I_6 = (9, 14), I_7 = (8, 12), I_8 = (1, 2), I_9 = (12, 16), I_{10} = (15, 17).$$

Solution: I_8, I_4, I_3, I_7, I_9.

5. (2 points) Consider the following different greedy algorithm for the Interval Scheduling algorithm:

```
DifferentGreedySchedule
  - Initialize R to contain all intervals
  - While R is not empty
    - Choose an interval (S(i), F(i)) from R that has the largest value of S(i)
    - Delete all intervals in R that overlaps with (S(i), F(i))
```

Prove or disprove: The above greedy algorithm always produces an optimal solution.

Solution: We will prove that the above algorithm also returns an optimal solution. Let F denote the largest finishing time of any interval. Consider the intervals $(F - F(1), F - S(1)), (F - F(2), F - S(2)), ..., (F - F(n), F - S(n))$. This is also an instance of the Interval Scheduling problem. Note that any optimal solution for this problem instance is an optimal solution to the original problem instance. The greedy algorithm that we discussed in class, when run on this new instance picks intervals based on the earliest finishing time which corresponds to the above greedy algorithm picking intervals based on largest start time when run on the original instance. The sequence of intervals picked by these algorithms are the same. This means that the above greedy algorithm outputs an optimal solution.

6. (2 points) Consider the Job Scheduling problem discussed in class. Given the following (duration, deadline) pairs:

$$J_1 = (2, 10), J_2 = (3, 5), J_3 = (4, 8), J_4 = (5, 7), J_5 = (6, 6)$$

calculate the maximum lateness for each of the schedules below:

(a) J_1, J_2, J_3, J_4, J_5

	J_1	J_2	J_3	J_4	J_5
Finish time	2	5	9	14	20
Deadline	10	5	8	7	6
Lateness	0	0	1	7	14

From the above the maximum lateness of this schedule is 14.

(b) J_2, J_1, J_5, J_3, J_4

	J_2	J_1	J_5	J_3	J_4
Finish time	3	5	11	15	20
Deadline	5	10	6	8	7
Lateness	0	0	5	7	13

From the above the maximum lateness of this schedule is 13.

(c) J_5, J_3, J_1, J_2, J_4

	J_5	J_3	J_1	J_2	J_4
Finish time	6	10	12	15	20
Deadline	6	8	10	5	7
Lateness	0	2	2	10	13

From the above the maximum lateness of this schedule is 13.

(d) J_5, J_4, J_3, J_2, J_1

	J_5	J_4	J_3	J_2	J_1
Finish time	6	11	15	18	20
Deadline	6	7	8	5	10
Lateness	0	4	7	13	10

From the above the maximum lateness of this schedule is 13.

7. (2 points) You are driving on a straight road from point s to point t in a car that can run for D miles on a full tank. There are n gas stations on the road and their locations with respect to the starting location s are given to you. You want minimize the number of stops that you need to make (for filling gas) for reaching from s to t. Assume that you have a full tank at start s. Design an algorithm to determine which gas stations to stop to minimize the total number of stops. Give pseudocode. Prove that your algorithm returns an optimal solution and discuss running time of your algorithm.

Figure 2: An example scenario.

Solution: Here is a greedy algorithm for this problem:

```
DrivingStrategy
  - Set current location to s
  - While t is not reachable from current location:
      - Let G be the farthest gas station that is reachable from current location.
      - Drive to G, fill up the tank, and update the current location to the location of G
  - Drive from current location to t
```

We will now prove that the greedy algorithm makes the minimum number of stops. We will do this using "greedy stays ahead" idea. Let $g_1, g_2, ..., g_k$ be the sequence of stops made by the greedy algorithm and $s_1, s_2, ..., s_l$ be the optimal sequence of stops. We will show by induction that for all $1 \leq i \leq k$, $g_i \geq s_i$:

Base case: $g_1 \geq s_1$ is true since the greedy algorithm picks the farthest gas station that is reachable from s.

Inductive step: We assume that $g_1 \geq s_1, ..., g_j \geq s_j$ and show that $g_{j+1} \geq s_{j+1}$. We know that $dist(s_j, s_{j+1}) \leq D$ which implies that $dist(g_j, s_{j+1}) \leq D$ (using the induction hypothesis that $g_j \geq s_j$). Since g_{j+1} is the farthest gas station with respect to g_j and $dist(g_j, s_{j+1}) \leq D$, we have that $g_{j+1} \geq s_{j+1}$.

So, we know that for all $1 \leq i \leq k$, $g_i \geq s_i$. This means that $l = k$, since if $l < k$, then that means that greedy algorithm makes more stops after stopping at g_l even though t is reachable from g_l (since t is reachable from s_l) which is a contradiction.

The algorithm needs to sort the gas stations with respect to their distance from s. This takes $O(n \log n)$ time. After this, the algorithm makes a linear scan, which takes a total of $O(n)$ time.

8. (3 points) Consider a hypothetical country *Binary Land* where they only use coins. There are n types of coins of denominations $1, 2^1, 2^2, 2^3, ..., 2^{n-1}$. Suppose you are a bank teller in *Binary Land* and your job is to give money to customers for any requested value V that is an integer. You should minimize the number of coins while paying V units of money. Design an algorithm that takes V as input and outputs the payment method that minimizes the total number of coins. Give pseudocode. Prove that your algorithm returns an optimal solution and discuss running time of your algorithm.

(The output is supposed to be an n-tuple of non-negative integers $(s_1, s_2, ..., s_n)$ s.t.: $\sum_{i=1}^{n} s_i 2^{i-1} = V$.)

Solution: Here is a greedy algorithm that we will analyze:

```
GreedyCoinSelect(V)
  - For i = n to 1
      - m ← ⌊V / 2^(i-1)⌋
      - g[i] ← m
      - V ← V − m · 2^(i-1)
  - return(g[1], g[2], ..., g[n])
```

Correctness proof: Let a solution be represented as an n-tuple $(m_1, ..., m_n)$ which denotes that the number of coins of value v_1 is m_1, coins of value v_2 is m_2 and so on. Let $(g_1, ..., g_n)$ be a greedy solution and $(o_1, ..., o_n)$ be any optimal solution. We first show that for all $1 \leq i < n, o_i \leq 1$.

Claim 3.1: For all $1 \leq i < n$, $o_i \leq 1$.

Proof. For the sake of contradiction, assume that there is an index $1 \leq i < n$ such that $o_i \geq 2$. In this case, the solution $(o_1, ..., o_{i-1}, o_i - 2, o_{i+1} + 1, ..., o_n)$ is also a valid solution but with smaller number of coins which is a contradiction. □

We are now ready to prove that $(o_1, ..., o_n) = (g_1, ..., g_n)$.

Claim 3.2 $(o_1, ..., o_n) = (g_1, ..., g_n)$.

Proof. For the sake of contradiction, assume that there is an index j such that $g_j \neq o_j$. Let j be the largest such index. That is, $g_{j+1} = o_{j+1}, ..., g_n = o_n$. This means that $o_j < g_j$ (since the greedy algorithm always picks the maximum number of coins of largest denomination). Now, we show a

contradiction from the following sequence of equations.

$$\begin{aligned}
\sum_{i=1}^{n} o_i \cdot 2^{i-1} &= \sum_{i=1}^{j-1} o_i \cdot 2^{i-1} + \sum_{i=j}^{n} o_i \cdot 2^{i-1} \\
&\leq \sum_{i=1}^{j-1} 1 \cdot 2^{i-1} + \sum_{i=j}^{n} o_i \cdot 2^{i-1} \quad \text{(using Claim 3.1)} \\
&= 2^{j-1} - 1 + \sum_{i=j}^{n} o_i \cdot 2^{i-1} \\
&< (o_j + 1) \cdot 2^{j-1} + \sum_{i=j+1}^{n} o_i \cdot 2^{i-1} \\
&\leq g_j \cdot 2^{j-1} + \sum_{i=j+1}^{n} o_i \cdot 2^{i-1} \quad \text{(since } g_j > o_j\text{)} \\
&= g_j \cdot 2^{j-1} + \sum_{i=j+1}^{n} g_i \cdot 2^{i-1} \quad \text{(since } g_{j+1} = o_{j+1}, ..., g_n = o_n\text{)} \\
&\leq \sum_{i=1}^{n} g_i \cdot 2^{i-1}
\end{aligned}$$

This is a contradiction since the value of both solutions should be the same (equal to V). □

Running time: The number of arithmetic operations involved in the algorithm is $O(n)$. However, if n is large, then we cannot regard division/multiplication as a unit operation. In that case, raising 2 to powers of $1, 2, 3, ..., n-1$ takes $O(n)$ time. Doing the initial operations on V would cost $O(n \cdot \log V)$ time. After the first iteration, the operations are performed on $n-1, n-2, ..., 1$ bit numbers. The time for the iteration in which $i = j$ will be $O(j^2)$. So, the total running time will be $O(n^3 + n \cdot \log V)$.

We can get a better running time by exploiting the fact that the binary representation of V is available to us as input. Let $v_l v_{l-1} ... v_1$ be the binary representation of V. We note that if $l \leq n$, then $g_1 = v_1$, $g_2 = v_2$, ..., $g_l = v_l$, $g_{l+1} = 0$, ..., $g_n = 0$. In case $l > n$, then we have $g_1 = v_1$, $g_2 = v_2$, ..., $g_{n-1} = v_{n-1}$ and g_n = Decimal value of $(v_l v_{l-1} ... v_n \underbrace{00...0}_{n-1 \text{ terms}})$. So, the running time for this algorithm would be $O(n + \log V)$.

1. Consider the weighted graph G_1 below and answer the questions that follow:

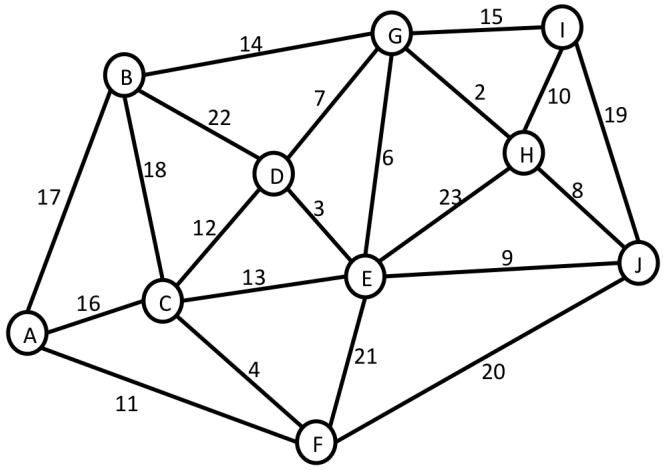


Figure 1: Graph G_1

(a) Give the weight of the minimum spanning tree of G_1.

(a) **70**

(b) Give the edges in the minimum spanning tree of G_1 *(Edges may be specified by a pair of vertices, e.g., (A, B))*

(b) **(A, F), (C, F), (C, D), (D, E), (E, G), (B, G), (G, H), (H, J), (H, I)**

2. What is the sequence of edges picked by:

(a) the Prim's algorithm when executed on the above graph G_1 with starting vertex A?

(a) **(A, F), (F, C), (C, D), (D, E), (E, G), (G, H), (H, J), (H, I), (G, B)**

(b) the Kruskal's algorithm when executed on the above graph G_1?

(b) **(G, H), (D, E), (C, F), (G, E), (H, J), (H, I), (A, F), (C, D), (B, G)**

3. Let G be any undirected, weighted graph with distinct edge weights. Let G' be a graph obtained from G by increasing the weight of each of the edges of G by 10 *(i.e., if the weight of an edge e in G is 5, then the weight of this edge in G' is 15)*. If T is a minimum spanning tree of G, then prove that T is also a minimum spanning tree of G'.

Solution: From the previous discussion section, we know that G and G' have unique MSTs. Let T be the unique MST of G and T' be the unique MST of G'. Assume for the sake of contradiction that $T \neq T'$. This means that:

$$(\text{weight of } T \text{ in } G) + (n-1) \cdot 10 \neq (\text{weight of } T' \text{ in } G').$$

If LHS $<$ RHS, then the weight of T is G' is smaller than the weight of T' in G' contradicting with the fact that T' is the MST of G'. On the other hand, if LHS $>$ RHS, then this means that the weight of T' in G is smaller than the weight of T in G which contradicts with the fact that T is the MST of G.

4. Given a weighted, undirected graph G, the *product-weight* of a spanning tree T of G is the product of the weights of edges in the tree T.

   (For example, for the graph shown below on the left, the product-weight of the spanning tree shown on the right is $2 \cdot 3 \cdot 4 = 24$.)

   Design an algorithm that takes as input any weighted, undirected graph G such that edge weights are positive and outputs a spanning tree of G with maximum product-weight.

   (a) Give pseudocode for your algorithm.

   (b) Prove that your algorithm always outputs a spanning tree with maximum product-weight.

Solution: Consider the following algorithm that outputs a spanning tree of a given graph with maximum weight.

```
ModifiedPrimAlgorithm(G)
  - S ← {u} //u is an arbitrary vertex in the graph
  - T ← {}
  - While S does not contain all vertices
      - Let e = (v, w) be the maximum weight edge between
        S and V \ S
      - T ← T ∪ {e}
      - S ← S ∪ {w}
```

We now prove that the above algorithm produces a maximum spanning tree of any given graph.

Proof of correctness: Let $e_1, ..., e_{n-1}$ denote the sequence of edges picked by the Prim's algorithm and let T be any Maximum Spanning Tree (henceforth, MaxST) of G. Let e_i be the first edge in the sequence $e_1, ..., e_{n-1}$ that is not present in T. Let $(S, V - S)$ denote the cut (that includes the starting vertex) just before picking e_i during the execution of the Prim's algorithm. Suppose we add the edge e_i in T. This creates a cycle. Consider the edges in this cycle that go across the cut $(S, V - S)$. All these edges have weights equal to the weight of e_i. Otherwise, we can exchange the smaller weight edge with e_i to get a tree with more weight than T. Consider any one such edge e_j and let $T' = T - \{e_j\} \cup \{e_i\}$. Note that T' is a spanning tree with the same cost as T. Now, if T' contains all edges $e_1, ..., e_{n-1}$, then we are done. Otherwise, we repeat the argument to construct a spanning tree of same cost as T but that includes all edges $e_1, ..., e_{n-1}$.

Running time: The running time will be the same as the running time of the Prim's algorithm which is $O(|E| \cdot \log |V|)$.

Given the graph G where the weight of edge e is given by $w(e)$, consider the graph G', which is the same as G except the weight of edge e in G' is $w'(e) = \log(w(e))$. We give the following claim with respect to G and G'.

Claim 1: Any spanning tree of G' with maximum weight is a spanning tree of G with maximum product-weight.

Proof. Let T be a spanning tree of G' with maximum weight. For the sake of contradiction, assume that there is a spanning tree T' such that the product-weight of T' in G is more than the product-weight of T in G. However, this means that the weight of T' in G' is larger than the weight of T in G (take logarithm of product-weights) which is a contradiction. □

The above claim suggests that all we need to do is to find a maximum spanning tree of G' which we have already done in the previous problem. So, the algorithm for this problem is the following:

`MaxProduct(`G`)`

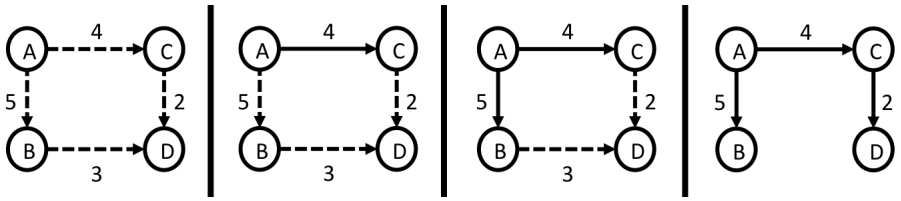
- Construct G' from G by taking logarithm of the edge weights
- return the tree obtained by executing `ModifiedPrimAlgorithm(`G'`)`

Running time: (*You were not expected to write this.*) The running time is the time to construct G' (which is $O(|V| + |E|)$) plus the time to run the algorithm in the previous problem (which is $O(|E| \cdot \log |V|)$). So, the total running time is $O(|E| \cdot \log |V|)$.

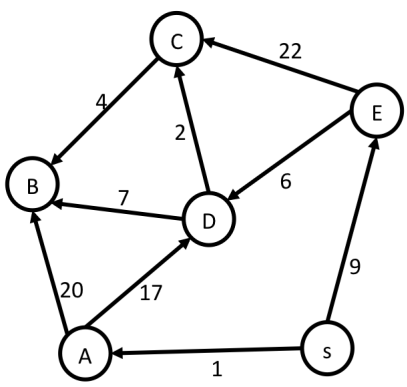
Note that there are multiple ways of approaching this problem. The solution above is just one of the ways of solving this problem. The goal of the above solution was to communicate a number of different ideas regarding Spanning Trees. You might be able to give much shorter solution.

5. Given any directed, weighted graph $G = (V, E)$, consider executing the Dijkstra's algorithm on G. Note that the edges "picked" by the algorithm *(i.e., the edge (u, v) in Claim 2 in the slides on Shortest Path topic)* form a rooted tree with root s. This rooted tree is a *Shortest Path Tree* of G. What this means is that the path from s to any vertex v in T is a shortest path from s to v in G.

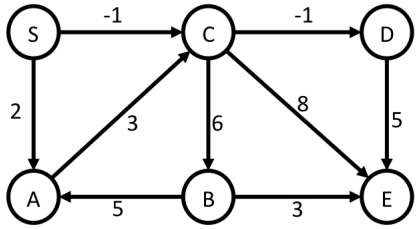
   (For example, the figure below shows the sequence of edges "picked" by the Dijkstra's algorithm when executed on the graph on the left. The shortest part tree is shown on the right.)



Consider graphs G_2 and G_3 below and answer the questions that follow.

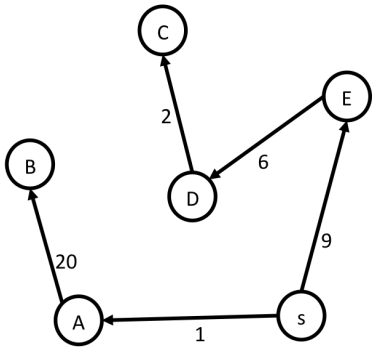


Graph G2

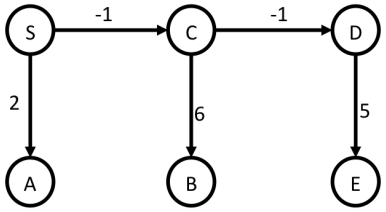


Graph G3

(a) Draw the rooted tree obtained when the Dijkstra's algorithm is executed on the graph G_2 above with s as the source vertex.



(b) (½ point) Draw the rooted tree obtained when the Dijkstra's algorithm is executed on the graph G_3 above with s as the source vertex.



(c) Is the rooted tree in part (b), a shortest path tree for G_3? That is, is it true that for all vertices v, the path from s to v in the tree is a shortest length path from s to v in G_3?
(Note that from the discussion in the lecture, when the edge weights are not all positive, then the Dijkstra's algorithm is NOT guaranteed to produce a shortest path tree.)

(c) **Yes**

6. Argue that the Prim's and Kruskal's algorithm produce a minimum spanning tree for **any** graph *(note that in the lecture we only proved that they output the MST for graphs with distinct edge weights)*

Solution: Let $e_1, ..., e_{n-1}$ denote the sequence of edges picked by the Prim's algorithm and let T be any MST of G. Let e_i be the first edge in the sequence $e_1, ..., e_{n-1}$ that is not present in T. Let $(S, V - S)$ denote the cut (that includes the starting vertex) just before picking e_i during the execution of the Prim's algorithm. Suppose we add the edge e_i in T. This creates a cycle. Consider the edges in this cycle that go across the cut $(S, V - S)$. All these edges have weights equal to the weight of e_i (otherwise there is a spanning tree of smaller weight as per class discussion). Consider any one such edge e_j and let $T' = T - \{e_j\} \cup \{e_i\}$. Note that T' is a spanning tree with the same cost as T. Now, if T' contains all edges $e_1, ..., e_{n-1}$, then we are done. Otherwise, we repeat the argument to construct a spanning tree of same cost as T but that includes all edges $e_1, ..., e_{n-1}$.

The argument for Kruskal's algorithm will be similar.

1. Suppose you use the algorithm `Length-LCS` discussed in class (slide #11, Feb. 16) to find the length of the longest common sequence between strings $S =$ "$BCADB$" and $T =$ "$BDCBA$". Recall that the algorithm, fills a 5×5 table L. Show this table L.

Solution:

1	1	1	1	1
1	1	2	2	2
1	1	2	2	3
1	2	2	2	3
1	2	2	3	3

2. You are given a sequence of numbers in an array $A = [0, 8, 2, 12, 4, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$. Let $L = L[1], ..., L[16]$ be an array of size 16 such that $L[i]$ stores the length of the longest increasing subsequence of A that ends with $A[i]$. Solve the following:

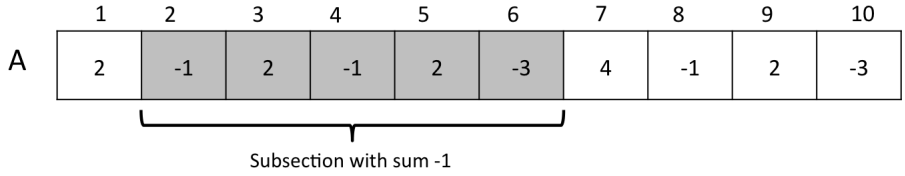
   (a) Fill the entries of L below. *(Use* `Length-LIS` *algorithm discussed in lectures.)*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
L	**1**	**2**	**2**	**3**	**3**	**4**	**4**	**5**	**2**	**5**	**4**	**6**	**3**	**6**	**5**	**7**

   (b) Give a longest increasing subsequence. *(Use* `LIS` *algorithm discussed in lectures.)*

(b) **0, 2, 4, 6, 9, 11, 15**

3. Given a sequence of integers (positive or negative) in an array $A = A[1]A[2]...A[n]$, the goal is to find a *subsection* of this array such that the sum of integers in the subsection is maximized. A subsection is a contiguous sequence of indices in the array. (*For example, consider the array and one of its subsection below. The sum of integers in this subsection is* -1.) Let us call a subsection that maximizes the sum of integers, a *maximum subsection.* Answer the questions that follow:



(a) Design a dynamic programming algorithm to output the sum of integers in a maximum subsection of a given array A. Give pseudocode and discuss running time.

Solution: Let $M(i)$ denote the maximum sum of the subsection of the array $A[1]...A[i]$ that includes $A[i]$. We can write the following recursive formulation for $M(.)$.

$$\forall i > 1, M(i) = \max\{M(i-1) + A[i], A[i]\} \quad \text{and} \quad M(1) = A[1]$$

This follows from the fact that the sum of subsection of $A[1]...A[i]$ that includes $A[i]$ either starts with $A[i]$ or starts at some index $j < i$. In the former case, the sum is $A[i]$ and in the latter case, the maximum sum is the maximum sum of subsection of $A[1]...A[i-1]$ that ends with $A[i-1]$ plus $A[i]$. Taking the maximum of the previous two options gives us the optimal result.

Here is the pseudocode for the algorithm based on the above recursive formulation.

```
MaxSubsectionValue(A, n)
  - M[1] ← A[1]
  - max ← M[1]
  - for i = 2 to n
      - M[i] ← max{M[i − 1] + A[i], A[i]}
      - if (M[i] > max) max ← M[i]
  - return(max)
```

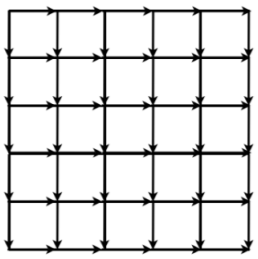
Running time: The algorithm has a for loop that runs for $O(n)$ iterations and costs constant time per iteration. So, the running time of the algorithm is $O(n)$.

(b) Design an algorithm to output a maximum subsection. *A subsection may be defined using the starting and ending index of the subsection.*

Solution: In order to output the subsection that maximizes the sum, we will need to maintain some more information. For every index i, we will store as $P[i]$ the starting index $j < i$ of the subsection of $A[1]...A[i]$ that maximizes the sum and that includes $A[i]$. Here is the pseudocode:

```
MaxSubsection(A, n)
  - M[1] ← A[1]
  - max ← M[1]; maxIndex ← 1
  - P[1] ← 1
  - for i = 2 to n
      - if (A[i] > M[i − 1] + A[i])
          - M[i] ← A[i]
          - P[i] ← i
      - else
          - M[i] ← M[i − 1] + A[i]
          - P[i] ← P[i − 1]
      - if (M[i] > max)
          - max ← M[i]
          - maxIndex ← i
  - return((P[maxIndex], maxIndex))
```

4. There is an $n \times n$ grid of one-way street network. At any intersection, you may either travel from west to east or north to south. You want to compute the number of different ways in which you can travel from the north-west corner to the south-east corner. We will develop a dynamic programming solution for this problem.
(*Below is an example of a* 6×6 *(i.e.,* $n = 6$*). We are interested in finding the number of different ways of going from the top-left corner to the bottom-right corner.*)



Let $W(i, j)$ denote the number of different ways of going from top-left corner to bottom-right corner for a grid of size $i \times j$ (i.e., the grid has i horizontal lines and j vertical lines).

(a) What is the value of $W(1, j)$ for any j and $W(i, 1)$ for any i?

> **Solution:** Let the intersection of the i^{th} horizontal line and the j^{th} vertical line be denoted by the tuple (i, j). There is only one way of going from $(1, 1)$ to $(1, j)$ and any j. Similarly, there is only one way of going from $(1, 1)$ to $(i, 1)$ for any i. So, we have for any $i \geq 1, W(i, 1) = 1$ and for any $j \geq 1, W(1, j) = 1$.

(b) Write $W(i, j)$ in terms of $W(i - 1, j)$ and $W(i, j - 1)$ when $i, j > 1$. Give brief explanation for the relationship that your give.

> **Solution:** For all $i, j > 1$, we have $W(i, j) = W(i - 1, j) + W(i, j - 1)$. This is because the only way of getting from $(1, 1)$ to (i, j) is either via $(i - 1, j)$ or via $(i, j - 1)$. Moreover, the paths via $(i - 1, j)$ to (i, j) are different from the paths via $(i, j - 1)$ to (i, j).

(c) Use the recursive formulation developed in (b) to design an algorithm that outputs the number of different ways of going from top-left corner to bottom-right corner of an $n \times n$ grid. Give pseudocode for your algorithm.

> **Solution:** Here is a "table-filling" algorithm for this problem:
>
> ```
> NumWays(n)
>   - For j = 1 to n
>       - W[1, j] ← 1
>   - For i = 1 to n
>       - W[i, 1] ← 1
>   For i = 2 to n
>       For j = 2 to n
>           - W[i, j] ← W[i − 1, j] + W[i, j − 1]
>   - return(W[n, n])
> ```

(d) Discuss running time of your algorithm.

Solution: The algorithm fills a table of size n^2. Filling each table entry takes $O(1)$ time. So, the running time of the above algorithm is $O(n^2)$.

5. You are given n types of coin denominations of values $v_1 < v_2 < ... < v_n$ (all integers). Assume $v_1 = 1$, so you can always make change for any integer amount of money C. You want to make change for C amount of money with as few coins as possible.

   We will solve this using Dynamic Programming. Let $M(i, c)$ denote the minimum number of coins needed to make a change for c when we are allowed only coins of values $v_1, v_2, ..., v_i$.

   (a) What is the value of $M(1, c)$ for any positive integer c?

   Solution: $M(1, c) = c$, since we are allowed only coins of value $v_1 = 1$.

   (b) Give a recursive formulation for $M(i, c)$. Give brief explanation as to why the recursive relationship that you give should hold.
   (Hint: Try writing $M(i, c)$ in terms of $M(i-1, c)$ and $M(i, c')$ for some appropriate $c' < c$)

   Solution:

$$M(i,c) = \begin{cases} M(i-1,c) & \text{if } v_i > c \\ \min\{M(i-1,c), M(i,c-v_i)+1\} & \text{Otherwise.} \end{cases}$$

   If $v_i > c$, then the no coins of value v_i can be used. Otherwise, we consider these cases:

   1. Number of coins used to construct the change for c using coins with value $v_1, ..., v_{i-1}$, and
   2. Number of coins used to construct the change for $(c - v_i)$ using coins of value $v_1, ..., v_i$ and plus one (coin of value v_i).

   We pick the case which minimizes the number of coins.

   (c) Use the recursive formulation developed in (b) to design an algorithm that outputs the minimum number of coins needed to make a change for C. Give pseudocode.

   Solution: Here is the "table-filling" algorithm for finding the minimum number of coins:

```
CoinChange(C)
  - For i = 1 to n
      - M[i, 0] ← 0
  - For c = 1 to C
      - M[1, c] ← c
  - For i = 2 to n
      - For c = 1 to C
          - If (v_i > c)then M[i, c] ← M[i − 1, c]
          - else M[i, c] ← min {M[i − 1, c], M[i, c − v_i] + 1}
  - return(M[n, C])
```

   (d) Modify your algorithm in (c) so that it outputs the number of coins of each value that is needed to make change for C using the minimum number of coins. Give pseudocode

   Solution: Here is the algorithm that outputs the number of coins of each value:

```
CoinChangeNum(C)
  - For i = 1 to n
      - M[i, 0] ← 0
      - P[i, 0] ← −1
  - For c = 1 to C
      - M[1, c] ← c
      - P[1, c] ← c − 1
  - For i = 2 to n
      - For c = 1 to C
          - If (v_i > c)
              - M[i, c] ← M[i − 1, c]
              - P[i, c] ← −1
          - else
              - If (M[i − 1, c] < M[i, c − v_i] + 1)
                  - M[i, c] ← M[i − 1, c]
                  - P[i, c] ← −1
              - else
                  - M[i, c] ← M[i, c − v_i] + 1
                  - P[i, c] ← c − v_i
  - Coins ← FindCoins(P, C)
  - return(Coins)
```

```
FindCoins(P, C)
  - For i = 1 to n
      A[i] ← 0
  - c ← C; i ← n
  - While(c ≠ 0)
      - While (P[i, c] ≠ −1)
          - A[i] ← A[i] + 1
          - c ← P[i, c]
      - i ← i − 1
  - return(A)
```

(e) In a fictional country there are coins of values $v_1 = 1, v_2 = 3, v_3 = 4, v_4 = 5$. Fill the table representing M below using the recursive formulation that you give in part (b).

M	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	9
2	0	1	2	1	2	3	2	3	4	3
3	0	1	2	1	1	2	2	2	2	3
4	0	1	2	1	1	1	2	2	2	2

6. A string is called *palindromic* if it reads the same whether read left to right or right to left. Given a string $S = s[1]s[2]...s[n]$, you want to find a longest subsequence of S that is palindromic. For example, if S = MAHATMA, then the longest subsequence of S that is palindromic is MAHAM.

   (a) Design an algorithm that outputs the length of the longest subsequence that is a palindrome for a given input string. Give pseudocode.

Solution: Given a string $S = S[1], ..., S[n]$, for any $i < j$ let $L(i, j)$ denote the length of the longest palindromic subsequence of the string $S[i], S[i + 1], ..., S[j]$. First we note that for all $i, L(i, i) = 1$ since a single letter is always a palindrome of length 1. For $i < j$, if $S[i] = T[j]$, then the length of the longest palindromic subsequence of the string $S[i], ..., S[j]$ is equal to the length of the longest palindromic subsequence of $S[i + 1], ..., S[j - 1]$ plus 2. Otherwise, the longest palindromic subsequence will be either a longest palindromic subsequence of string $S[i+1], ..., S[j]$ or that of string $S[i], ..., S[j-1]$. This gives us the following recurrence relation:

$$L(i,j) = \begin{cases} 0 & \text{If } i > j \\ 1 & \text{If } i = j \\ L(i+1, j-1) + 2 & \text{If } S[i] = S[j] \\ \max\{L(i+1, j), L(i, j-1)\} & \text{If } S[i] \neq S[j] \end{cases}$$

Consider the following table-filling algorithm based on the above recursive formulation.

```
LPS(S)
  - n ← |S|
  - For i = 1 to n
      - L[i, i] ← 1; L[i, i − 1] ← 0
  - For s = 1 to n − 1
      - For i = 1 to n − s
          - j ← i + s
          - If (S[i] = S[j])
              - L[i, j] ← L[i + 1, j − 1] + 2
          - else
              - L[i, j] ← max {L[i + 1, j], L[i, j − 1]}
```

   (b) Discuss running time of your algorithm.

Solution: Note that the algorithm fills the upper diagonal of the matrix L and filling each table entry takes $O(1)$ time. So, the running time of the algorithm is $O(n^2)$.

   (c) Design an algorithm that outputs a longest subsequence that is a palindrome for a given input string.

Solution: As usual, we keep extra book-keeping information in the above algorithm to output a longest palindromic subsequence.

```
LPS-subsequence(S)
  - n ← |S|
  - For i = 1 to n
      - L[i,i] ← 1; L[i,i−1] ← 0
  - For s = 1 to n − 1
      - For i = 1 to n − s
          - j ← i + s
          - If (S[i] = S[j])
              - L[i,j] ← L[i+1,j−1] + 2
              - P[i,j] ← " ↙ "
          - else
              - If (L[i+1,j] > L[i,j−1])
                  - L[i,j] ← L[i+1,j]
                  - P[i,j] ← " ↓ "
              - else
                  - L[i,j] ← L[i,j−1]
                  - P[i,j] ← " ← "
  - PrintSubsequence(S, n, P)
```

```
PrintSubsequence(S, n, P)
  - i ← 1; j ← n; k ← 1
  While (i < j)
      - If (P[i,j] = " ↙ ")
          - A[k] ← S[i]; k ← k + 1
          - i ← i + 1
          - j ← j − 1
      - elseif (P[i,j] = " ← ")
          - j ← j − 1
      - elseif (P[i,j] = " ↓ ")
          - i ← i + 1
  - For s = 1 to k − 1
      - Print(A[s])
  - If (i = j)Print(S[i])
  - For s = k − 1 to 1
      - Print(A[s])
```

Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Answer the following:

   (a) (1 point) State true or false Let $f(n), g(n)$ be functions mapping positive integers to positive real numbers in the interval $(1, +\infty)$ such that $f(n) = O(g(n))$. Let $f'(n) = \log_2 f(n)$ and $g'(n) = \log_2 g(n)$. Then $f'(n) = O(g'(n))$.

   (a) **False**

   (b) (3 points) Give reasons for your answer to part (a).

   Solution: Consider $f(n) = 2$ and $g(n) = 1 + 1/n$. We can write $f(n) = O(g(n))$ since for all $n \geq 1, f(n) \leq 2 \cdot g(n)$.
   Now, $f'(n) = \log_2 f(n) = 1$ and $g'(n) = \log_2 g(n) = \log_2 (1 + 1/n)$. Note that for any constant $c > 0$, $1 > c \cdot \log_2 (1 + 1/n)$ given that $n > \frac{1}{2^{1/c}-1}$. This means that for every constant $c > 0$ and $n_0 \geq 1$, there is an integer $n \geq n_0$ such that $f'(n) > c \cdot g(n)$. Such an n may be chosen as $n = \max\{n_0, \lceil \frac{1}{2^{1/c}-1} \rceil + 1\}$. This implies that $f'(n)$ is not $O(g'(n))$.

2. You are given an array A containing n integers from the set $\{1, ..., 100\}$ and you want to find the most frequently occurring element in the array (if there multiple elements with maximum frequency, output any one of them). You are supposed to design an algorithm for this problem.

   (a) (4 points) Write pseudocode for your algorithm.

Solution: Here is the pseudocode for the algorithm:

```
MaxFrequency(A, n)
  - for i = 1 to 100
      - R[i] ← 0
  - for i = 1 to n
      - R[A[i]] + +
  - max ← R[1]; maxnum ← 1
  - for i = 2 to 100
      - if (R[i] > max)
          - max ← R[i]; maxnum ← i
  - return(maxnum)
```

   (b) (2 points) Discuss the running time of your algorithm. Express the running time using Θ notation (that is, give a tight bound).

Solution: The algorithm makes two passes over an array of size 100 and a single pass over an array of size n doing constant number of operations in each pass. The former costs $\Theta(1)$ time and the latter will cost $\Theta(n)$ time. So, the overall running time is $\Theta(n)$.

Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Answer the following:

   (a) (3 points) Use "unrolling of recursion" to solve the following recurrence relation. Show steps and give a simplified expression for $T(n)$.

   $$T(n) = 8T(n/2) + n^3; \quad T(1) = 1 \qquad \textit{(Assume } n \textit{ is a power of } 2.\textit{)}$$

Solution: By unrolling the recursion, we get:

$$\begin{aligned}
T(n) &= 8 \cdot T(n/2) + n^3 \\
&= 8 \cdot (8 \cdot T(n/2^2) + (n/2)^3) + n^3 \\
&= 8^2 \cdot T(n/2^2) + (n^3 + n^3) \\
&= 8^2 \cdot (8 \cdot T(n/2^3) + (n/2^2)^3) + (n^3 + n^3) \\
&= 8^3 \cdot T(n/2^3) + (n^3 + n^3 + n^3) \\
&\vdots \\
&= 8^i \cdot T(n/2^i) + i \cdot n^3 \\
&\vdots \\
&= 8^{\log n} \cdot T(n/2^{\log n}) + \log n \cdot n^3 \\
&= n^3 + n^3 \cdot \log n
\end{aligned}$$

   (b) (1 point) Express $T(n)$ obtained in part (a) using Θ notation:

   (b) $T(n) = \Theta(n^3 \log n)$

2. Given an array A containing n distinct integers in increasing order, you want to determine if there is an index i such that $A[i] = i$. Design an algorithm that outputs such an index if it exists and outputs -1 is no such index exists.

   (a) (4 points) Give pseudocode for your algorithm.

Solution: Here is the pseudocode for the algorithm. The algorithm is called with inputs $(A, 1, n)$.

`FindIndex(`A, i, j`)`
- if $(j < i)$return(-1)
- $mid = \lfloor (i + j)/2 \rfloor$
- if $(A[mid] = mid)$return(mid)
- if $(A[mid] < mid)$return(`FindIndex(`$A, mid + 1, j$`)`)
- else return(`FindIndex(`$A, i, mid - 1$`)`)

(*Please try proving the correctness of the above algorithm as an additional exercise.*)

   (b) (2 points) Do the running time analysis for your algorithm.

Solution: The recurrence relation for the running time $T(n)$ is given as: $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$; $T(1) = \Theta(1)$. If we assume that n is a power of 2, then the recurrence relation can be written as $T(n) = T(n/2) + \Theta(1); T(1) = \Theta(1)$. The solution is the same as that of Binary Search which is $\Theta(\log n)$. For arbitrary n, we can repeat the same argument as in the class. That is, let $2^k \leq n < 2^{k+1}$. Then $ck \leq T(n) < d(k + 1)$ (for some constants c, d) which implies that $T(n) = \Theta(\log n)$.

(*You may also use the informal comment that we discussed in the class that dropping floor does not change the running time.*)

Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. (5 points) Recall that a *proper* binary tree is a binary tree where all the nodes have either 0 children or 2 children. Also recall that node is called *leaf* node iff it does not have any children and a node is called *internal* node iff it is not a leaf node.

   Show that in any proper binary tree, the number of leaf nodes is one more than the number of internal nodes.

Solution: Let $P(n)$ be the statement that "If a proper binary tree with n nodes exists, then in any such n-node tree, the number of leaves is one more than the number of internal nodes". We will argue that $P(n)$ is true for all n using Induction.

Base case: $P(1)$ is true since a proper binary tree with 1 node indeed has only one leaf and 0 internal nodes.

Inductive step: Given that $P(1), P(2), ..., P(k)$ are true consider the statement $P(k+1)$. If there are no proper binary trees with $k+1$ nodes, then $P(k+1)$ is trivially true. Suppose there exists a proper binary tree with $k+1$ nodes. Consider any such tree T. Let v be the node with the maximum depth in T and let u be its parent. Then u's other child is also a leaf. Consider the tree T' obtained by removing the leaves of u. T' has $k-1$ nodes and is also a proper binary tree. So, from our induction assumption, we get that the number of leaves l' in T' is one more than the number of internal nodes i'. That is, $l' = i' + 1$. Let i denote the number of internal nodes of T and l denotes the number of leaves of T. Then from our construction, we have that $l = l' + 1$ and $i = i' + 1$. So, we have $l = l' + 1 = (i' + 1) + 1 = i + 1$. So, $P(k+1)$ holds. This completes the inductive step.

Using the principle of mathematical induction, we conclude that $P(n)$ is true for all n.

2. Answer the following question with respect to the array based heap implementation discussed in class.

   (a) (1 point) State true or false: Given a min-heap containing n entries such that each entry has a unique integer key. The entry with the fifth smallest key may be found in $O(1)$ time.

   (a) **True**

   Reason (you were not expected to write this): Note that you can find the minimum element in $O(1)$ time since it is located at the root or level-0 of the tree. You can find the second minimum element since that you located at level-1 of the tree. Similarly, you can argue that the fifth smallest key cannot be in layer-i for any $i > 4$. Otherwise, there are more than 4 keys that are smaller than this key (*these are all the keys along the path from the node to the root*). So, to find the fifth smallest element, all one needs to do is consider the nodes in levels-0,1,2,3,4 and find the fifth smallest key among these keys. This will cost $O(1)$ time.

   (b) (1 point) State true or false: The array below represents a min-heap (only keys of entries are shown).

   (b) **False**

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key value	3	10	5	13	7	6	10	15	16	12	13	9	8	21	15

   Reason (your were not supposed to write this): The node with key 7 is the child of the node with key 10. This violates the heap-order property.

   (c) (3 points) Consider the array A representing a min-heap below (only keys of entries are shown).

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	3	10	5	13	17	6	11	15	16	21	18	9	8	23	12

   Give the array after performing one `removeMin()` operation.

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Key	5	10	6	13	17	8	11	15	16	21	18	9	12	23

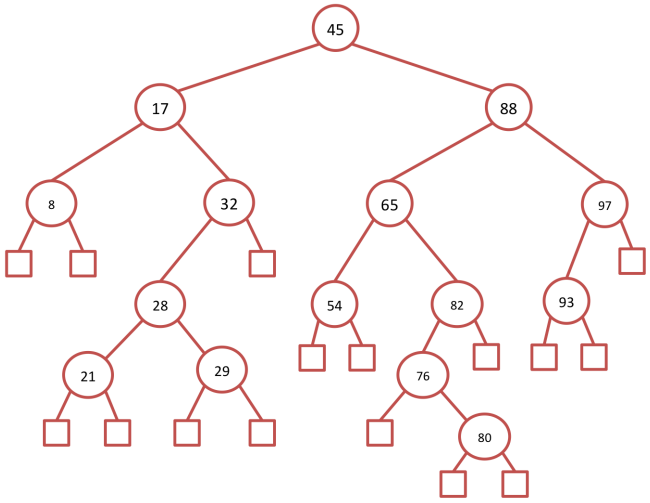
Name: _____

Entry number: _____

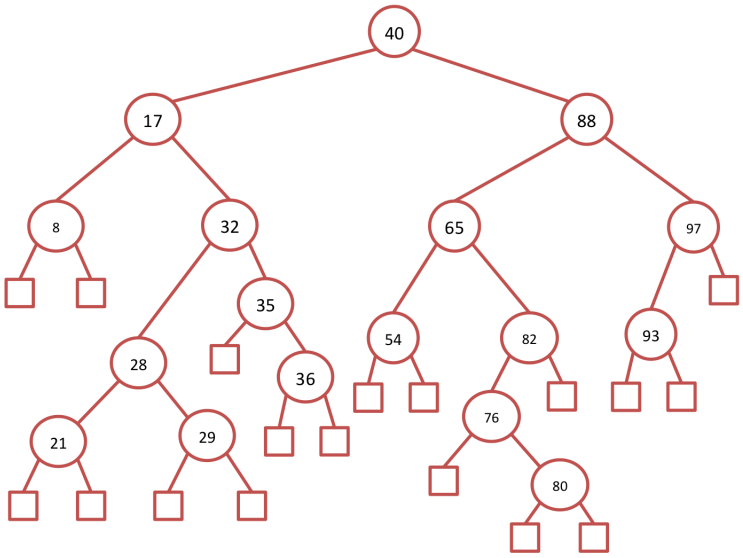
There are 3 questions for a total of 10 points.

---

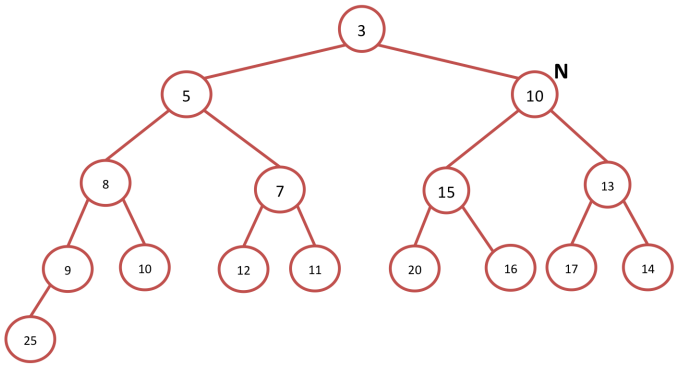
1. Consider the binary search tree below (only keys shown) and answer the question that follows:

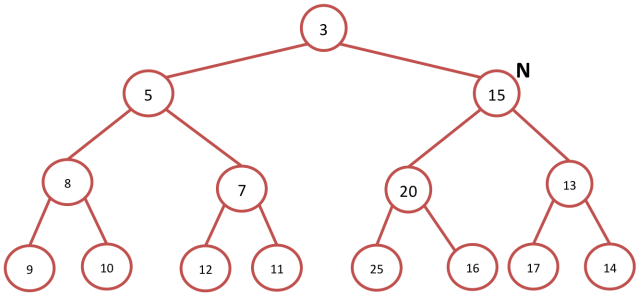


(a) (2 points) Draw the binary search tree obtained after performing `put(40, "A")`, `put(35, "B")`, `put(36, "C")`, `remove(45)` in that order. *You only need to draw the final Binary Search Tree.*



2. (3 points) In Homework-06, you were asked to add a method `deleteNode(Node N)` in the pointer/reference based implementation of Min-Heap. This method should delete an entry stored at a given node N and should run in $O(\log n)$-time. Draw the resulting min-heap when this method is called with node N as shown in figure below. Note that only keys are shown in the heap below.





3. (5 points) Apply the $O(n)$-time bottom-up heap construction algorithm discussed in the class on the array below. Show intermediate arrays and the final array representing the min-heap.

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Key	6	3	16	11	7	17	14	8	5	15	1	2	4	18	13	9	10	12

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Key	6	3	16	5	1	2	13	8	11	15	7	17	4	18	14	9	10	12

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Key	6	1	2	5	3	4	13	8	11	15	7	17	16	18	14	9	10	12

Array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Key	1	3	2	5	6	4	13	8	11	15	7	17	16	18	14	9	10	12

COL106: Data Structures and Algorithms (IIT Delhi, Semester-II-2016-17) Quiz-08

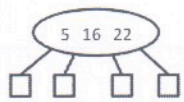
Name: _____

Entry number: _____

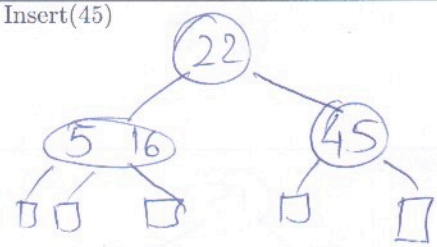
There are 2 questions for a total of 10 points.

---

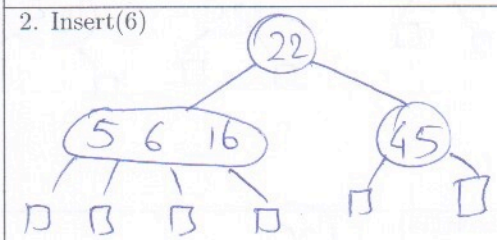
1. (5 points) Perform the sequence of operations on an $(2, 4)$-tree as given below:



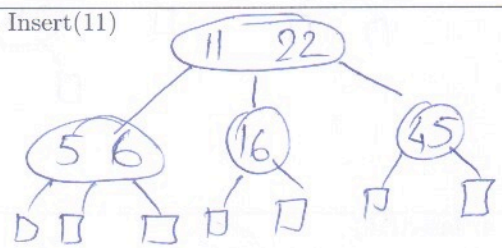
1. Insert(45)



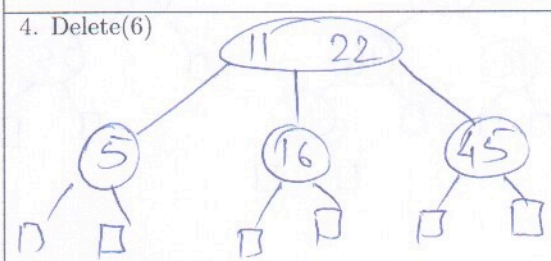
2. Insert(6)



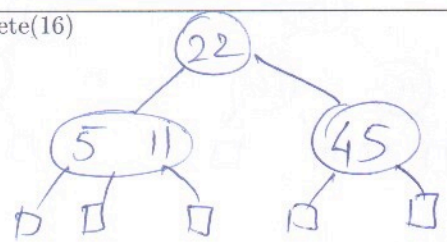
3. Insert(11)



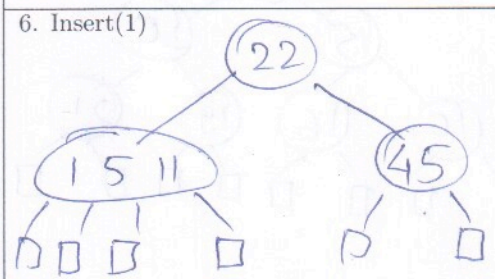
4. Delete(6)



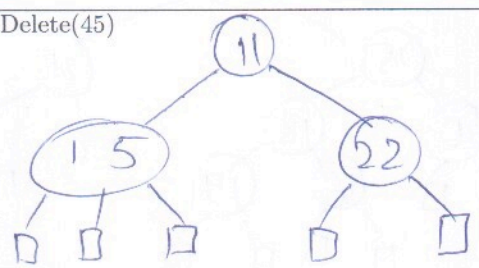
5. Delete(16)



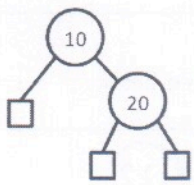
6. Insert(1)



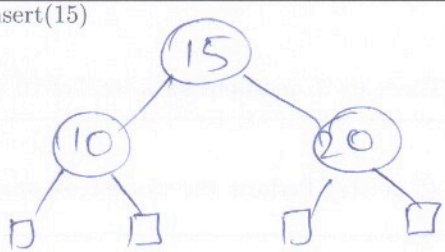
7. Delete(45)



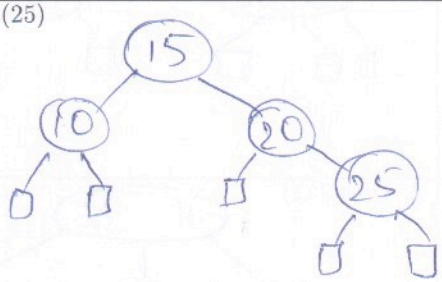
2. (5 points) Perform the sequence of operations on an AVL tree as given below:



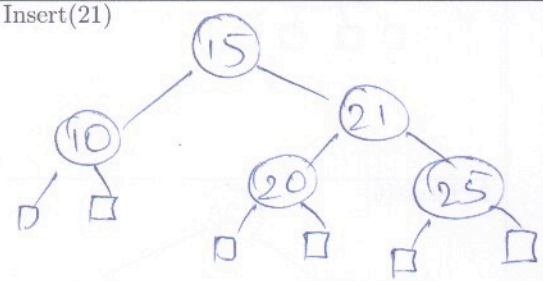
1. Insert(15)



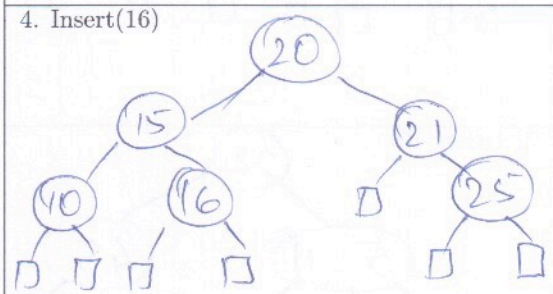
2. Insert(25)



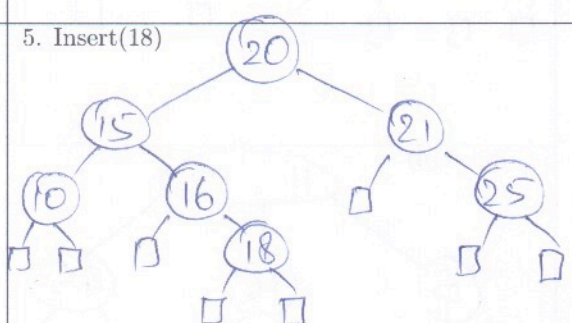
3. Insert(21)



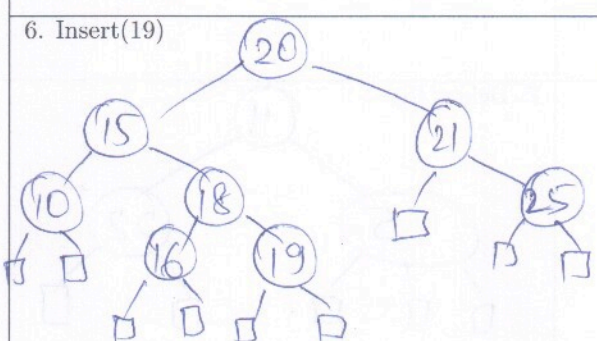
4. Insert(16)



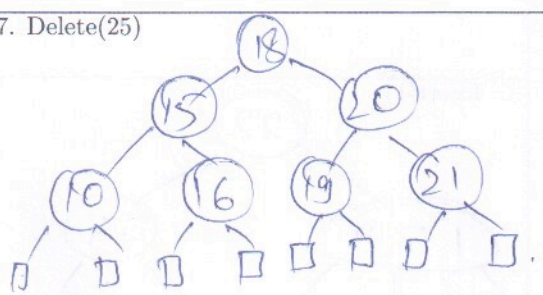
5. Insert(18)



6. Insert(19)



7. Delete(25)



Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Consider a simple hashing example where the universe is $U = \{0, 1, 2, 3, 4, 5\}$ and elements from this universe are supposed to be stored in a table of size 4 with indices $T = \{0, 1, 2, 3\}$. Consider the following hash function family in this context:

$$H = \{h_{a,b} \mid a \in \{1, 2, 3, 4, 5\} \text{ and } b \in \{0, 1, 2, 3, 4, 5\}\} \quad \text{where} \quad h_{a,b}(x) = ((a \cdot x + b) \ mod \ 6) \ mod \ 4$$

As discussed in class, a random hash function from this family $h_{a,b}$ is picked by choosing a from set $\{1, 2, 3, 4, 5\}$ and b from set $\{0, 1, 2, 3, 4, 5\}$ uniformly at random. Answer the following:

(a) (1 point) State true or false: H is a 2-universal hash function family.

(a) **False**

(b) (4 points) Give reason for your answer to part (a).

Solution: Table 1 gives the value of $h_{a,b}(0)$ for various values of a, b. Table 2 gives the value of $h_{a,b}(2)$ for various values of a, b.

$a \backslash b$	0	1	2	3	4	5
1	0	1	2	3	0	1
2	0	1	2	3	0	1
3	0	1	2	3	0	1
4	0	1	2	3	0	1
5	0	1	2	3	0	1

Table 1: Table for $h_{a,b}(0)$

$a \backslash b$	0	1	2	3	4	5
1	2	3	0	1	0	1
2	0	1	0	1	2	3
3	0	1	2	3	0	1
4	2	3	0	1	0	1
5	0	1	0	1	2	3

Table 2: Table for $h_{a,b}(2)$

From these values, we can see that:

$$\mathbf{Pr}_{h_{a,b} \leftarrow H}[h_{a,b}(0) = h_{a,b}(2)] = \frac{14}{30} > \frac{1}{4}.$$

Hence H is not a 2-universal hash function family.

2. We know that a B-tree is a $(d, 2d)$-tree for an appropriately chosen d. Assume that $d = 3$ for all the subparts below:

   (a) (1 point) State true or false: Starting from an empty B-tree, the B-tree obtained after inserting keys from any set S is the same irrespective of the order in which the keys are inserted.

   (a) **False**

   (b) (2 points) Give reason for your answer to part (a).

   Solution: Consider $S = \{1, 2, 3, 4, 5, 6, 7\}$. The B-tree obtained for sequences $\{1, 2, 3, 4, 5, 6, 7\}$ (see fig on left) and $\{2, 3, 4, 5, 6, 7, 1\}$ (see fig on right) are different and are given below.

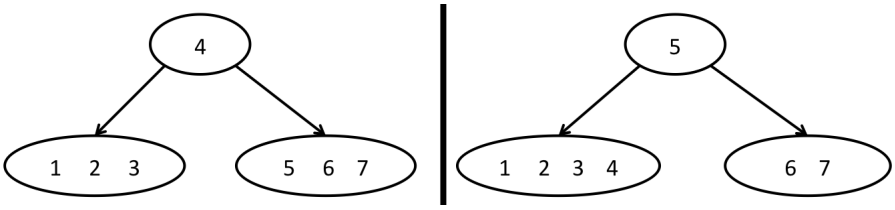


Figure 1: (Leaves have not been shown for this $(3, 6) - tree$)

   (c) (1 point) Starting from an empty B-tree, draw the B-tree after inserting keys $10, 3, 8, 21, 15, 4, 6, 19, 28, 31, 32$ in that order.

   Solution: See top figure below.

   (d) (1 point) Starting from the B-tree obtained in part (c), draw the B-Tree after deleting $32, 15$ (in that order).

   Solution: See bottom figure below.

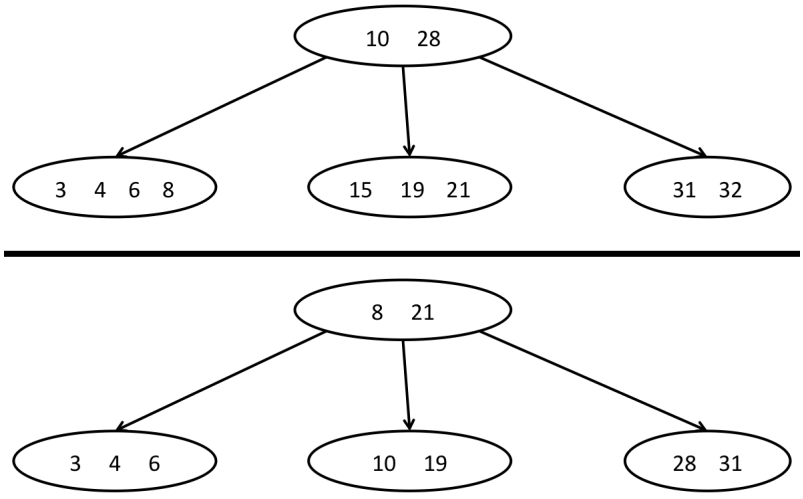


Figure 2: (Leaves have not been shown for this $(3, 6) - tree$)

Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Answer the following questions:

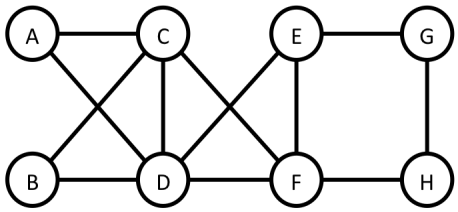
   (a) (1 point) State true or false: There exists an undirected graph G with 10 vertices and 37 edges such that there are exactly two strongly connected components in G.

   (a) **False**

   (b) (2 points) Give reason for your answer to part (a).

   Solution: (You were not expected to write this. This is just for explanation of the answer) Let $n_1 \geq n_2$ denote the number of vertices in the two strongly connected components. Let $L(n_1, n_2)$ be the maximum number of edges that such a graph could have. We have $L(5,5) = \binom{5}{2} + \binom{5}{2} = 20, L(6,4) = \binom{6}{2} + \binom{4}{2} = 21, L(7,3) = \binom{7}{2} + \binom{3}{2} = 24, L(8,2) = \binom{8}{2} + \binom{2}{2} = 30, L(9,1) = \binom{9}{2} = 36$. In none of the cases, the graph could have more than 36 edges. So, such a graph does not exist.

   (c) (2 points) Show the layers obtained when doing a BFS starting with the vertex A on the undirected graph below.



   (c) $Layer(0) = \{A\}, Layer(1) = \{C, D\}, Layer(2) = \{B, F, E\}, Layer(3) = \{G, H\}$

2. (a) (1 point) State true or false: Any strongly connected undirected graph with n vertices and $(n-1)$ edges is a tree.

(a) **True**

(b) (4 points) Give reasons for your answer in part (a).

Solution: (You were not expected to write this. This is just for explanation of the answer)
Let $P(n)$ denote the proposition "any strongly connected undirected graph with n vertices and $(n-1)$ edges is a tree". We will prove $\forall n, P(n)$ using induction.
Base case: $P(1)$ is true since a graph with 1 vertex and 0 edges is indeed a tree.
Inductive step: Assume that $P(1), P(2), ..., P(k)$ are true for an arbitrary k. We will show that $P(k+1)$ is true. Consider any strongly connected graph G with $k+1$ vertices and k edges. Then there is a vertex v with degree exactly 1. Otherwise the sum of degrees will be $\geq 2(k+1)$ but this is not possible since we know that sum of degrees is equal to $2|E|$ which in this case is $2k$. Consider the graph G' obtained by removing the vertex v and its connecting edge. Note that G' is still strongly connected and it has k vertices and $k-1$ edges. Using the induction hypothesis, we get that G' is a tree. This implies that G is a tree.

COL106: Data Structures and Algorithms (IIT Delhi, Semester-II-2016-17) Quiz-11

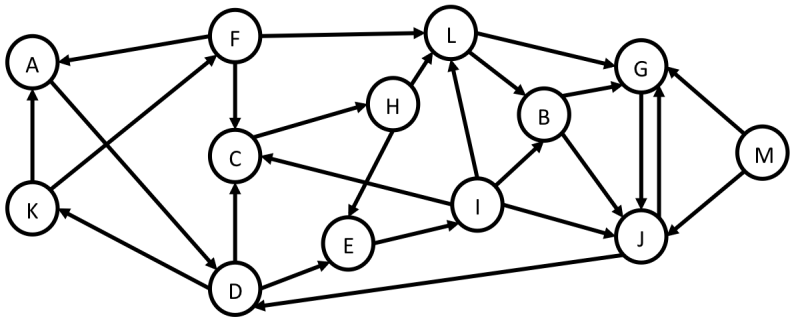
Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Consider the following directed graph below and answer the questions:



(a) (1 point) How many strongly connected components does the graph have?

(a) **2**

(b) (1 point) Give all the vertices that are in the strongly connected component that contains the vertex H.

(b) **{A, B, C, D, E, F, G, H, I, J, K, L}**

(c) (1 point) Give a vertex from which all other vertices are reachable. If no such vertex exists, write "no such vertex".

(c) **M**

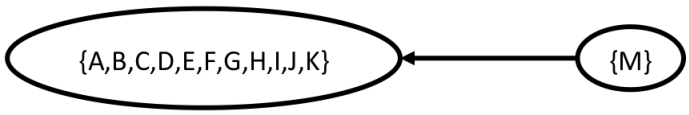
(d) (1 point) Give a pair of vertices (u, v) such that there is a path from u to v and there is a path from v to u. If no such pair exists, write "no such pair".

(d) **(A, F)**

(e) (1 point) Run `DFS(`A`)` (that is, DFS with starting vertex A) and output the vertices in the order they are explored. That is, if a vertex u is marked "explored" before vertex v, then u should be before v in your sequence.

(e) **A, D, K, F, C, H, E, I, L, B, G, J**

(f) (1 point) Draw the graph G^{SCC} corresponding to the above graph G.



2. You are supposed to design an algorithm to minimize the number of coins needed to make change when given as input:

   - The integer value W for which the change is required, and
   - The currency system that has coins of integer values V_1, V_2, V_3, V_4, V_5. These coin values satisfy the following properties:
     1. $V_1 = 1$
     2. For all $1 \leq i < 5$, $\frac{V_{i+1}}{V_i} \geq 2$.

   Consider the following greedy algorithm for this problem:

```
GreedyCoinSelect(V_1, V_2, V_3, V_4, V_5, W)
  - For i = 5 to 1
     - m ← ⌊W / V_i⌋
     - g[i] ← m
     - W ← W − m · V_i
  - return(g[1], g[2], ..., g[5])
```

   (a) (4 points) Show that the above greedy algorithm does not always return an optimal solution. (*That is, you have to give an input on which the algorithm does not return an optimal solution.*)

Solution: Consider coin of values: $v_1 = 1, v_2 = 5, v_3 = 11, v_4 = 22, v_5 = 44$. Note that for all $1 \leq i < 5$, $\frac{v_{i+1}}{v_i} \geq 2$. Let $V = 15$. The greedy algorithm on input $(1, 5, 11, 22, 44, 15)$ outputs $(4, 0, 1, 0, 0)$ that uses 5 coins. The optimal solution is $(0, 3, 0, 0, 0)$ and uses only 3 coins.

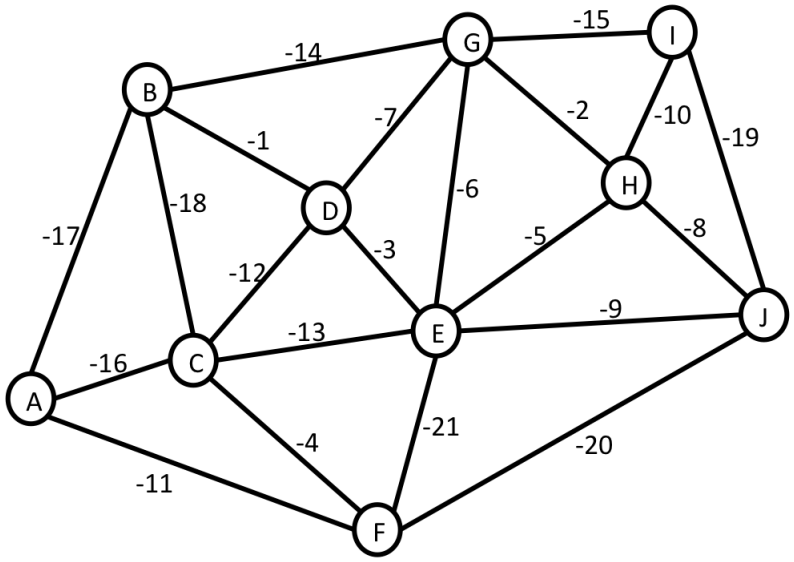
Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Consider the weighted graph (edges have negative weights) below and answer the questions that follow:



(a) (1 point) Give the weight of the minimum spanning tree of the graph.

(a) **-146**

(b) (1 point) Give the edges in the minimum spanning tree of the graph *(Edges may be specified by a pair of vertices, e.g., (A, B))*

(b) $(E,F),(F,J),(J,I),(B,C),(A,B),(G,I),(B,G),(C,D),(H,I)$

(c) (1½ points) What is the sequence of edges picked by the Prim's algorithm when executed on the above graph with starting vertex A?

(c) $(A,B),(B,C),(B,G),(G,I),(I,J),(J,F),(F,E),(C,D),(I,H)$

(d) (1½ points) What is the sequence of edges picked by the Kruskal's algorithm when executed on the above graph?

(d) $(E,F),(F,J),(J,I),(B,C),(A,B),(G,I),(B,G),(C,D),(H,I)$

2. (5 points) There are n men with heights $m_1, m_2, ..., m_n$ and n women with heights $w_1, w_2, ..., w_n$. You have to match men to women for a dance such that the sum of absolute value of difference in height of each pair, is minimized.
(*For example, consider $n = 2$ and $m_1 = 5, m_2 = 7, w_1 = 6.5, w_2 = 6$. In this case, if we match first man with second woman and second man with first woman, then the sum of absolute value of difference is* $|5 - 6| + |7 - 6.5| = 1.5$)

Design an algorithm to solve this problem. Give pseudocode and discuss running time.
(*You may assume that the heights of men and women are given in arrays $M[1...n]$ and $W[1...n]$ and your output should consist of n pairs of the form (i, j) indicating that the i^{th} man is matched with the j^{th} woman.Furthermore, you may assume that $M[1] \leq M[2] \leq ... \leq M[n]$ and $W[1] \leq W[2] \leq ... \leq W[n]$.*)

Solution: Here is the pseudocode for a greedy algorithm that works for this problem.

```
DancePairs(M, W, n)
  - S ← {}
  - for i = 1 to n
      - S ← S ∪ (i, i)
  - return(S)
```

Running time: The running time of the algorithm is $O(n)$ since it just runs a loop with n iterations.

Sketch of proof of correctness: (*you were not expected to write this*)

We will show that the following greedy algorithm minimizes the sum of absolute difference in height:

> Let the men and women be sorted in increasing order of heights. Match the first man in the sequence to the first woman, second man in the sequence to the second woman, and so on.

We will now prove optimality. As per our assumption, $m_1 \leq m_2 \leq ... \leq m_n$ and $w_1 \leq w_2 \leq ... \leq w_n$. Consider any optimal solution σ. This solution is just a permutation of $1...n$. This means that as per the optimal solution, M_1 is matched with $W_{\sigma(1)}$, M_2 is matched with $W_{\sigma(2)}$, and so on. Let i be the smallest index such that $\sigma(i) \neq i$. Let j be the index such that $\sigma(j) = i$. Clearly, $j > i$. Consider another solution σ_1 that is the same as σ, except $\sigma'(i) = i$ and $\sigma'(j) = \sigma(i)$. We will show that the average height difference as per σ' is less than equal to the average height difference as per σ. Let the average height difference as per σ and σ' be denoted by $H(\sigma)$ and $H(\sigma')$ respectively. We have:

$$H(\sigma) - H(\sigma') = \frac{1}{n} \cdot (|m_i - w_{\sigma(i)}| + |m_j - w_i| - |m_i - w_i| - |m_j - w_{\sigma(i)}|)$$

We consider the following 6 cases:

1. $w_i \leq w_{\sigma(i)} \leq m_i \leq m_j$: In this case, $H(\sigma) - H(\sigma') = 0$.
2. $w_i \leq m_i \leq w_{\sigma(i)} \leq m_j$: In this case, $H(\sigma) - H(\sigma') = (1/n) \cdot 2 \cdot (w_{\sigma(i)} - m_i) \geq 0$.
3. $m_i \leq w_i \leq w_{\sigma(i)} \leq m_j$: In this case, $H(\sigma) - H(\sigma') = (1/n) \cdot 2 \cdot (w_{\sigma(i)} - w_i) \geq 0$.
4. $w_i \leq m_i \leq m_j \leq w_{\sigma(i)}$: In this case, $H(\sigma) - H(\sigma') = (1/n) \cdot 2 \cdot (m_j - m_i) \geq 0$.
5. $m_i \leq w_i \leq m_j \leq w_{\sigma(i)}$: In this case, $H(\sigma) - H(\sigma') = (1/n) \cdot 2 \cdot (m_j - w_i) \geq 0$.

6. $m_i \leq m_j \leq w_i \leq w_{\sigma(i)}$: In this case, $H(\sigma) - H(\sigma') = 0$.

So, in all the cases $H(\sigma') \leq H(\sigma)$. We continue with this exchange operation and at the end we obtain our greedy solution.

Name: _____

Entry number: _____

There are 2 questions for a total of 10 points.

---

1. Recall the Longest Increasing Subsequence problem discussed in class. Consider the sequence of numbers in array $A = [14, 8, 2, 7, 4, 10, 6, 0, 1, 9, 5, 13, 3, 11, 12, 15]$. As in the class discussion, let $L(i)$ denote the length of the longest increasing subsequence of $A[1...n]$ that ends with $A[i]$.

   (a) (2 points) Fill the table for L below.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$L[i]$	1	1	1	2	2	3	3	1	2	4	3	5	3	5	6	7

   (b) (1 point) Give a longest increasing subsequence.

   (b) **2, 4, 6, 9, 11, 12, 15**

2. You are given n types of coin denominations of values $v_1 < v_2 < ... < v_n$ (all integers). Assume $v_1 = 1$, so you can always make change for any integer amount of money C. You want to make change for C amount of money with as few coins as possible.

   We will solve this using a Dynamic Programming different than the one we gave in the homework. Let $T(c)$ denote the minimum number of coins needed to make a change for an integer amount c. Note that $T(1) = 1$ since the minimum number of coins needed to make a change for 1 is 1. Answer the questions that follow:

   (a) (2 points) Give a recursive formulation for $T(c)$. Give brief explanation as to why the recursive relationship that you give should hold.
   (Hint: Try writing $T(c)$ in terms of $T(c')$'s for $c' < c$)

   Solution: For all $c > 1$,

$$T(c) = \min_{i \text{ such that } v_i \leq c} \{T(c - v_i)\} + 1.$$

   Given that the optimal solution for making change for c uses at least one copy of the i^{th} coin, the minimum number of coins will be $T(c - v_i) + 1$. The minimum number of coins needed to make a change for c is the minimum over all possibilities for i.

   (b) (3 points) Use the recursive formulation developed above to design an algorithm that outputs the minimum number of coins needed to make a change for an integer amount C. Give pseudocode and discuss running time.

   Solution: Here is the pseudocode for the "table-filling" algorithm:

```
MinimumCoins(v_1, ..., v_n, C)
  - T[1] ← 1
  - for c = 2 to C
      - minCoins ← c
      - for j = 1 to n
          - if (v_j ≤ c)
              - if (T[c − v_j] + 1 < minCoins) minCoins ← T[c − v_j] + 1
      - T[c] ← minCoins
  - return(T[C])
```

Running time: The algorithm fills a table of size C and filling each table entry takes $O(n)$ time. So, the running time of this algorithm is $O(n \cdot C)$. (*Note that this is the same running time as the algorithm given in the homework.*)

(c) (2 points) In a fictional country there are coins of values $v_1 = 1, v_2 = 3, v_3 = 5, v_4 = 7$. Fill the table representing T below using your recursive formulation.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	1	2	1	2	1	2	3	2	3	2	3	2	3	4	3