For parts 11(a) to 11(h), assume A and B are **specific** decision problems and that f is a polynomial-time function for reducing A to B.

11(a) [2] if $A \rightarrow_f B$ and $A \in P$, then $B \in P$

11(b) [2] if $A \rightarrow_f B$ and $B \in NP$, then $A \in P$

11(c) [2] if $A \rightarrow_f B$ and $A \in NPC$, then $B \in NPH$

11(d) [2] if $A \rightarrow_f B$ and $B \in NPC$, then $A \in NPC$

11(e) [2] if $A \rightarrow_f B$ and $B \rightarrow_g A$, then $A, B \in NPC$

11(f) [2] if $A \rightarrow_f B$ and $B \in NPC$, then $B \rightarrow_g A$

11(g) [2] if $A \rightarrow_f B$ and $A \in NPC$, then $B \in NPC$

11(h) [2] if $A \rightarrow_f B$ and $A \in NPH$, then $B \in NPC$

Name: Sherif Ahmed

Below is the template version of breadth-first search. The hook operations are in bold font.

```
Algorithm BFS(G)
  Input graph G
  Output labels the edges of G as discovery edges and cross edges

  initResult(G)
  for all u ∈ G.vertices()
      initVertex(u)
      setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
      initEdge(e)
      setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
      if getLabel(v) = UNEXPLORED
          preComponentVisit(G, v)
          bfsTraversal(G, v)
          postComponentVisit(G, v)
  result(G)
```

```
Algorithm bfsTraversal(G, s)
  Q ← new empty queue
  setLabel(s, VISITED)
  Q.enqueue(s)
  startBFS(G, s)
  while ¬Q.isEmpty() do
      v ← Q.dequeue()
      preVertexVisit(G, v)
      for all e ∈ G.incidentEdges(v) do
          preEdgeVisit(G, v, e)
          if getLabel(e) = UNEXPLORED
              w ← opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  preDiscoveryEdgeVisit(G, v, e, w)
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Q.enqueue(w)
                  postDiscoveryEdgeVisit(G, v, e, w)
              else
                  setLabel(e, CROSS)
                  crossEdgeVisit(G, v, e, w)
          postEdgeVisit(G, v, e, w)
      postVertexVisit(G, v)
  finishBFS(G, s)
```

8. [5] Suppose your boss asks you to design an efficient algorithm to solve an optimization problem. Describe below the strategies you would try first. If you were unsuccessful in designing a polynomial-time algorithm, what would you do?

NP and NP-Complete

Notation: $A \rightarrow_p B$ means instances of problem A can be reduced to instances of problem B by function p in polynomial time.

9. [10] Suppose B is a decision problem. Let b_0 and b_1 be instances of problem B such that the decision algorithm for B always returns no (false) on b_0 and eventually yes (true) on b_1. Reduce the LCS (Longest Common Subsequence) Problem to problem B in polynomial time. LCS can be defined as follows: An instance of LCS is composed of two strings S1 and S2 and a positive integer K. The LCS decision problem asks, is there a common subsequence of S1 and S2 with length at most K? **Hint**: You do not have to remember the LCS algorithm, just call it, i.e., length ← LCS(S1, S2).

   [5] In one sentence describe how LCS computes length and what is its running time.

10. (a) [10 points] Show that LSC∈NP. The LSC (Longest Simple Cycle) decision problem can be stated as follows:

    Given a weighted graph G, does there exist a simple cycle in G with total weight at least *K*? (the total weight of a cycle is the sum of the edge weights in the cycle)

    (b) [10] Reduce the Hamiltonian Cycle (HC) problem to the above LSC problem. HC can be stated as follows:

    HC: **Given a graph *G*, does there exist a cycle in G that visits each vertex exactly once?**

    (c) [5] Since the Hamiltonian Cycle (HC) problem has been proven to be a member of NPC, what, if anything, can we then conclude about the LSC problem based on 10(a) and 10(b)? If there is a conclusion, then state it otherwise explain why nothing can be concluded.

11. Answer **true** or **false** to each of the following questions 11(a) to 11(h). If true, briefly justify your answer (using at most 2 sentences in the space provided below). If false, give a counter example, such as, "A could be MST and B could be halting problem" or "A could be in NPC and B in P", etc. **Zero points without a justification.**

Algorithm Analysis

5. [10] Give a **detailed** analysis of the following algorithm. Give the running time of each line of the algorithm (on this page). What is the total running time of algorithm *Unknown*? **Hint:** do not try to figure out what the algorithm computes; it may or may not do anything useful.

Algorithm *Unknown*(G)
Input: A weighted graph with *n* vertices, and m edges
Output: ?????

```
P ← create array of size n to be referenced by vertex id number (id[u]).
Q ← create array of size n to be referenced by vertex id number.
for each u ∈ G.vertices() do
    Q[id(u)] ← null
for each v ∈ G.vertices() do
    for each u ∈ G.vertices() do
        if G.areAdjacent(u,v) then
            P[id(u)] ← -∞
for each u ∈ G.vertices() do
    for each e ∈ G.incidentEdges(u) do
        z ← G.opposite(u,e)
        if G.valueAt(e) > P[id(z)] then
            P[id(z)] ← valueAt(e)
            Q[id(z)] ← e

return Q
```

Short answer questions:

6. [5] Suppose there are going to be eight nodes in a local area network. If you need to connect those nodes with the least cost (the longer the wire connecting two nodes, the greater the cost), which graph algorithm would you choose to solve your problem? Your choices are: BFS, DFS, Shortest Path, Minimum Spanning Tree, Hamiltonian Path, and Traveling Salesperson (TSP). Briefly justify why your choice is the best and would solve the problem (on this page below).

7. [5] If a graph G has a shortest edge, is there a minimum spanning tree of G containing this edge? Briefly justify your answer (on this page).

Algorithm Design

Sequence ADT:
first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q),
insertBefore(p, o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p),
removeFirst(), size(), isEmpty(), elemAtRank(r), replaceAtRank(r, o),
insertAtRank(r, o), removeAtRank(r), atRank(r), rankOf(p),elements()

BinaryTree ADT:
root(), parent(v), children(v), leftChild(v), rightChild(v), sibling(v),
isInternal(v), isExternal(v), isRoot(v), size(), elements(), positions(),
swapElements(v, w), replaceElement(v, e)

Dictionary ADT
findElement(k), insertItem(k, e), removeElement(k), items()

OrderedDictionary ADT
findElement(k), insertItem(k, e), removeElement(k), closestKeyBefore(k),
closestKeyAfter(k), closestElemBefore(k), closestElemAfter(k)

(General) Graph ADT
numVertices(), numEdges(), vertices(), edges(), aVertex(),
degree(v), adjacentVertices(v), incidentEdges(v),
endVertices(e), opposite(v, e), areAdjacent(v, w), valueAt(v), valueAt(e)
insertVertex(o), removeVertex(v), insertEdge(v, w, o), removeEdge(e),

1. [15] Give pseudo-code for the overriding hook methods that would specialize the BFS template algorithm above so it determines, for each vertex of G, the edge whose weight is less than the weight of any other edge incident on v; the algorithm must return a Sequence of n pairs, (*v*, *MinE*) where *v* is the vertex and *MinE* is the smallest weight edge of the edges incident on *v*. Your solution **must use** the template algorithm above and must calculate *MinE* for each vertex *v* during the traversal, i.e., there must be no loops other than the loops in the BFS algorithm.

   [5] What is the running time of your algorithm? Justify your answer; the running time for each line of your pseudo-code must be shown and for each line of the BFS template algorithm.

2. [15] Define an **efficient** algorithm to compute binomial coefficients. Binomial coefficients are defined as follows:

   $B(n,k) = 1$ if $k=0$ or $k=n$

   $B(n,k) = B(n-1, k-1) + B(n-1, k)$ if $0<k<n$

   E.g., $B(1,1)=1$, $B(1,0)=1$, $B(2,1)=B(1,0)+B(1,1)=2$, $B(2,2)=1$, $B(3,2)=B(2,1)+B(2,2)=3$, $B(2,0)=1$, $B(3,1)=B(2,0)+B(2,1)=3$, $B(3,0)=1$, $B(4,1)=B(3,0)+B(3,1)=4$, $B(4,2)=B(3,1)+B(3,2)=6$, etc.

   [2] What is B(5,3)?
   [5] What is the running time of your algorithm?

3. [15] Given two vertices *u* and *v*, create an algorithm to determine (yes/no) whether or not these two vertices are members of the same **connected component** of G.

4. [5] Define what is meant by a spanning tree of a graph G.

   [15] Given a graph G=(V, E) and a sub-graph T=(W, F) of G. Give an efficient pseudo-code algorithm that determines (yes/no) whether or not T forms a spanning tree of G. **Hint**: use the BFS template method given above.
   [5 points] What is the running time of your algorithm?

components of a forest are trees

Forest

Algorithm HC2LSC(G)

for all edges e in G.edgeSet()
setWeight(e, 1)

n ← G.vertices().size()

return LSC(G, n)

Algorithm LCStoB(S1, S2, k)

~~total~~ =
length ← LCS(S1, S2)
if(length ≥ k)
return 1
else return 0.

Algorithm check(G', k)

n ← nG'
if n < 3 return no
if n ≠ nedges return no
~~DFS~~
ncomp ← DFS(G')

Forest

Depth- and Breadth-First Search 28

28 of 66

StartBFS(G, s)

L ← new Empty Sequence

PrevertexVisit(G, v)
MinE ← null

PreedgeVisit(G, v, e)
if MinE = null then
MinE = e
else if weight(e) < weight(MinE) then
MinE = e

PostVesterVisit(G, v)
L ← insertFirst((v, MinE))

finishBFS(G, s)
return L.

the running time is $O(n+m)$ as we only added $O(1)$ operations in each step.

$B(5,3) = B(4,2) + B(4,3) = 6 + B(3,2) + B(3,3)$
$= 6 + 3 + 1 = 10$

~~Algorithm~~ initialize()

Algorithm Compute BNK(n, k)

B ← new array [n+1][k+1]
B ← initialize(-1)
Compute (n, k)
return B[n][k]

Algorithm compute (n, k)

if k=0 · v k=n then
B[n][k] = 1
else if B[n][k] = -1 then
B[n][k] ← B(compute(n-1, k-1)+Compute(n-1, k)
else return

then
The running time is O(n×k) as in the worst case we may need to amput all the value in the table.

4. → it is a tree (connected + no cycle)
→ it includes all the vertices of G.

15. We will run BFS on T.
checkSP (G, T)
v ← T.aVertex()
nVertices = 0
cycle = false
BFS(T, s)
if (nVertices = G.vertices().size() ∧ ¬cycle)
return yes.
else return no

We need to overridde these methodes

PreVisitVertex (G, v)
nVertices ← nVertices + 1

CrossEdgeVisit(G, v, e, w)
Cycle = true.

The running time is $O(n+m)$ as it is the same running time as BFS(T, s)

- n
- n^2 if areAdjacent(u, v) ?? O(1)
- $O(m)$ incidentEdges O(1) ??

6. As we want the sum of edges to be minimized we have to ~~prove~~ use MST as a solution. We will give as an input to it a dense graph with 8 vertices and $\frac{8 \times 7}{2} = 28$ edges (distances).

7- Yes using Kruskal this edge will be picked in the first step as it we connecting 2 different clouds. (Partition Property)