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).

Name: ____________

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)
```

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?

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.**

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 $A \to_f B$ and $A \in P$, then $B \in P$

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

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

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

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

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

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

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

Algorithm Binomial Coeff (n, K)
B ← empty sequence

for i = 0 to n do
for j = 0 to K do
B[i][j] = -1
return BC(B, n, K)

Algorithm BC(B, n, K)
if B[n][K] = -1
if K = 0 or K = n then
B[n][K] = 1
Else
B[n][K] = BC[n-1, K-1] + BC[n-1, K]
return B[n][K]

If $w_K > w$ B[K, w] = B[K-1, w]
Else
B[K, w] = max {B[K-1, w], B[K-1, w - w_K] + b_K }

B[5, 3]
= B[4, 2] + B[4, 3]
= B[3, 1] + B[3, 2] + B[3, 2] + B[3, 3]
= $\underbrace{B[2,0]}_{1}$ + B[2, 1] + B[2, 1] + $\underbrace{[2,2]}_{1}$ + B[2, 1] + $\underbrace{B[2,2]}_{1}$ + 1
=
= 4 + 3 × B[2, 1]
= 4 + 3 × B[1, 0] + B[1, 1]

= 10

a) $A \rightarrow_p B$, $A \in P$ then $B \in P$ False

Let say that A $\notin$ LCS $\in P$

we know that,

LCS $\longrightarrow_p$ subset sum $\in NPC$

$\therefore B \in NPC$

(b) If $A \longrightarrow_f B$ and $B \in NP$ then $A \in P$ - False.

$A \in NP \notin P$

Let say, Hamiltonian cycle $\longrightarrow_f$ Hamiltonian path

A — B

$A \in NPC \notin P$ — $B \in NPC \in NP$

$A \in NP$

c) If $A \longrightarrow_f B$ and $A \in NPC$ then $B \in NPH$ - ~~False~~ True B can either $\in NPC$ & NPH

$A \in NPC$ — $B \in NPC$ or $B \in NPH$ — $B \in NPH$

$NPC, NPH \in NPH$

$\therefore B \in NPH$

d) If $A \rightarrow_f B$ and $B \in NPC$ then $A \in NPC$: False

Let say A is sorting $A \rightarrow_f$ Subsetsum

P — B NPC

Subset sum $\in NPC$

$A \in P$ and $A \notin NPC$

e) If $A \longrightarrow_f B$, $B \rightarrow_f A$ then $A, B \in NPC$: False

$A, B \in P, NP, NPC, NPH$

F

Let A is sorting B is MST

$A \longrightarrow_f B$ $A, B \notin NPC$

f) If $A \rightarrow_f B$ $B \in NPC$ $B \rightarrow_f A$: false.

B is subset problem $\in NPC$
A is sorting $\in P$

$A \rightarrow_f B$ $B \in NPC$ Subset sum cannot be reduced to Sorting
$B \not\rightarrow_f A$

g) If $A \rightarrow_f B$ $A \in NPC$ then $B \in NPC$: false

A is LCS $\in NPC$
$A \rightarrow_f$ Halting   Halting $\in NPH$
LCS   Halting $\notin NPC$

h) If $A \rightarrow_f B$ $A \in NPH$ then $B \in NPC$: False.

Let say, ~~A is Halting problem $\in NPH$~~
~~Halting problem~~

A is subset sum $\in NPC \in NPH$
$\Downarrow$
B $\rightarrow$ Halting problem $\in NPH$ $\notin NPC$

R.13.1 $L \rightarrow_f NPC\ (M)$

L can be solved in Polynomial time
$L \rightarrow M$ (NPC)
$L \rightarrow$ Polynomial
(P)

(P) MST $\rightarrow$ Subset sum (NPC)
~~[illegible]~~
This is not the proof of $P = NP$

R 13.13 N = {23, 59, 17, 47, 14, 40, 22, 8}
sum = 100 ? sum = 130 ? } Subset sum

Randomly pick subset & check whether sum is 100 or not.

C 13.2 $L \xrightarrow{f} M\{5\}$

Binary encoding of input = 5 ??

g B
b_0 b_1 instances of B
return no → b_0
return yes → b_1

Reduce LCS to B in polynomial time
S1, S2, K
LCS(S1, S2) ≤ K

Algorithm LCS2B (S1, S2, K)

length ← LCS (S1, S2) polynomial
if length ≤ K
return b1 {yes}
else
return b0 {No}

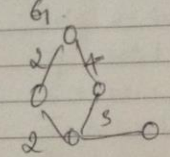
To reduce of any problem from A to B

AtoB(arguments of A)
Calculation of A
Result
if result = true Instance of B
Else
result false Instance of B

(10) LSC ∈ NP   Prove some problem ∈ NP

Longest simple cycle

G



Is there a simple cycle with ~~even~~ atleast weight K.

Guess

Let T be a sub-graph of G with some vertices & edges.

Check

Algorithm LSC(G, K, T)   K = 8

[list of edges] cycle ← cycleBFS(G, T)   {polynomial}

~~if cycle is empty then~~
~~return no~~

Else sum ← 0

for all e ∈ cycle

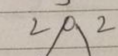
sum ← sum + weight(e)

if sum ≥ K then

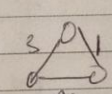
return yes

Else

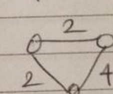
return no



— no

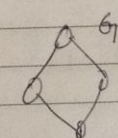


— yes.



— yes

10 b Reduce Hamiltonian cycle to LSC



Algorithms HC2LSC(G)

is_hc ← CheckHC(G) → BFS/DFS check cycle polynomial

Is there a graph G in which all vertices are visited once

~~is_hc~~ G' ← a new graph

G'.addvertex(A)

G'.addvertex(B)

G'.addvertex(C)   G'.addEdge(A, B, 1)

G'.addEdge(B, C, 2)   G'.addEdge(C, A, 4)

if is_hc then

return (G, 5)   yes ≤ 6

Else

return (G, 8)   yes > 6

MST to subset (G, K)

```
T ← MST(G)
total ← sum of T. edges
if total > K then
S ← new sequence
S. insertLast(2)
if sum > k return (S, 1, 1)
else return (S, 2, 2)
```

Shortest Path 2 MST (G, u, v, K)

```
P ← ShortestPath(G, u, v)
sum ← sum of edges in P
M ← new graph
u ← M. insertVertex (u)
v ← insert (v)
e ← M. insertEdge (u, v, 2)
if sum > K then return (M, 1)
else return (M, 2)
```

SAT ∈ NPC

Show SAT belongs to NP

a. Assign true or false to each variable in exp by flipping coin randomly

{ a = T, b = F, c = F d = T, e = T } put in dictionary A

```
Algorithm CheckSAT(Exp, A)
    v ← evaluate(Exp, A)  → post order
    if v then
        return yes
    else return no
```

① CheckMST(G, K) — P

T ← MST(G)

sum ← sum of edges of T

sum ← 0
for all e ∈ G edges
sum ← sum + e

if sum ≥ K
return no
Else
return yes

② Check ShortestPath(G, K) — P
T ← ShortestPath(G)
sum ← sum of edges of T
if sum ≥ K
return no
Else
return yes

③ CheckSubsetSum(S, max, min) — NPC
sum ← sum of all elements of S
if min ≤ sum ≤ max
return yes
return no

④ LCS(S1, S2, K)
C ← LCS(S1, S2)
if C ≥ K return yes
return no

I Whether graph is connected or not.

Alg InitResult (G)
S ← new Empty sequence

For connected graph
InitResult (G)
count ← 0

Alg PreComponentVisit (G, v)
count ← count + 1

Alg result (G)
if count > 1
return notConnected

II. Given v' & v'' present in the same connected graph.

InitResult (G)
count ← 0

PostComponentVisit (G, v)
found ← false

StartVertexVisit (v)
if (v = v') or v = v''
then
count ← count + 1
if count = 2 then
found ← true

Alg result (G)
if found = = true
return sameComponent
Else
return no

III Min or max vertex

visiting vertex
    min ← null

visiting Edge
    preEdgeVertex (G, v, e)
        if minedge = null or weight(e) ≤ minedge
            minedge ← weight (e)
    ~~visitVertex~~
    visitVertex (v)
    S . insert (v, min)

Average weight
    visitingvertex
        count ← G. degree (v)
        total ← 0
    visiting Edge
        total + = weight (e)
        count +1 = 1

PostVertexVisit (v)
    avg ← total / count)
    S. insertLast (avg, v)

IV ~~for checkcycle~~ checks whether T is MST of G

CrossEdgeVisit(G, v, e, w)
    mst ← false

initResult (G)
    T ← subgraph of G
    H ← new Dictionary (Hashtable)
for all v ∈ T
    HT. insertLast(v, v)
    mst ← true

Algorithm preVertex(v)
    if H. findElement(v) = NO_Such_Key
        mst ← false

Algorithm BFS(G, T)
    H1 ← new Dictionary(HashTable)
    for all ...
        return mst

CS435 Algorithms Final Exam

February 2, 2006

Name ______________________

Answer True or False to each question in this section. Please write clearly. [2 points each]

1. __T__ To use the greedy method, a problem should have the property that a series of locally-optimal choices lead to a global optimal configuration.
2. __F__ The greedy approach should never be used on optimization problems.
3. __F__ The *task scheduling* problem is an example of the divide-and-conquer method because tasks are sorted by their running time before assigning them to machines.
4. __T__ The *fractional knapsack* problem can be solved using the greedy method by continued selection of the item with the highest benefit-to-weight ratio.
5. __T__ Recurrence equations are used to evaluate the time-complexity of divide-and-conquer algorithms.
6. __F__ The problem of multiplying big integers of size n has $O(n^{1.585})$ time-complexity. n^2 / $n \log n$
7. __F__ The dynamic programming technique is similar to the divide-and-conquer approach in the way that a problem is divided in smaller, independent sub-problems and the results merged together to form the solution.
8. __T__ When the dynamic programming technique is applied to the *multiplying matrices* problem the time-complexity is reduced from exponential to linear.
9. __F__ Dynamic programming algorithms have a running time that only depends on n.
10. __T__ Two vertices that are adjacent are endpoints of the same edge.
11. __F__ The sum of the degrees of all vertices in a graph G are equal to the number of edges.
12. __F__ A spanning tree of a graph contains only some of the vertices of the graph.
13. __T__ An adjacency list structure has similar performance to the edge list structure but also has performance improvements in methods such as incidentEdges(v) and areAdjacent(u,v).
14. __T__ Depth-first search traversal of an undirected graph uses the backtracking technique to explore the vertices and edges of a graph.
15. __T__ In breadth-first search, the edges are marked as either discovery or cross edges to indicate their role in the spanning tree.
16. __T__ To test whether a graph is connected, a DFS traversal can be performed and if some of the vertices are not marked as discovered, the graph is not connected.
17. __T__ In a computer network, reachability in a directed graph is computed to find out if a message can be routed from node v to node w.

18. ✗ The transitive closure of a graph measures the density of the edges in the graph.

19. ✗ A topological ordering of a digraph is useful in scheduling tasks that have constraints as represented in the graph.

20. T Single-source shortest path algorithms find all the paths between a vertex v and w in a weighted graph.

Multiple choice questions. Pick the best answer. [3 points each]

21. The depth-first search algorithm we studied uses ____.
    a) edge reversal
    b) a min-heap
    c) recursion
    d) transitive closure

22. The ____ problem can be solved optimally with the greedy approach.
    a) sum-of-subsets
    b) traveling salesman
    c) big integer multiplication
    d) fractional knapsack

23. The iterative substitution method is a technique that depends on our ability to ____ that can be converted to the closed-form version of the recurrence equation.
    ✗ a) see a pattern
    b) multiply matrices
    c) draw a tree
    d) apply a formula

24. In the following recurrence relation, the step of merging sub-problem solutions is done ____ times at each level.

$$T(n) = \begin{cases} 5b & \text{if } n < 2 \\ \underbrace{3}_{\text{no of recursive calls}}T(n/2) + \underbrace{4n}_{\text{extra work done}} & \text{if } n \geq 2 \end{cases}$$

    a) 2
    b) 3
    c) 4
    d) 5

List : $6 \times 12 \times (n+m)$

25. The ______ applies to a problem if the best solution always contains optimal solutions of all sub-problems.

a) principle of optimality

b) greedy method

c) iterative method

d) big O notation

$(10)^4$ (10^7)

$\times 72$ $\times 4$

26. Which data structure is preferred when we need space efficiency when representing a graph with 10,000 vertices and 200,000 edges and we also need fast response to the areAdjacent method?

a) adjacency matrix

b) adjacency list

c) edge list

x y

$y = x^2$

edge list

Adjacency List

Adjacency

27. Which of the following problems has not been proven to be intractable, but also does not have a polynomial-time solution?

a) 0-1 knapsack problem NPC

b) minimum spanning tree problem

c) matrix multiplication problem

d) searching problem

28. What is an application of topological ordering?

a) sorting mail by the user's name

b) finding the path between a pair of vertices

c) determining connectivity in a graph

d) showing the inheritance hierarchy in Java interfaces

29. A weighted graph will only have one minimal spanning tree if _____.

a) every edge has a different weight

b) every edge has the same weight

c) every vertex connects to very other vertex

d) every vertex is a separate component

30. A ____ is a sequence of vertices that have an edge between each vertex and its successor.

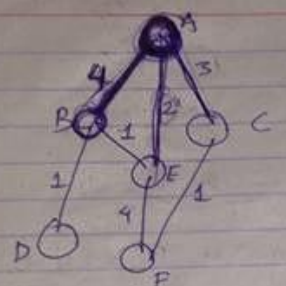
a) cycle

b) map

c) component

d) path

BFS/DFS $O(n+m)$
Shortest $m \log n$

① 

S [(A, 0),
(B, 1),
(C, 1),
(D, 1)
(E, 1),
(F, 1)]

(A, -1)

Algorithm initResult(G)
S ← new Empty sequence $O(1)$

Algorithm preVertexVisit (G, V)
min ← -1 $O(1)$

Algorithm preEdgeVisit (G, v, e)
if min = -1 or weight(e) < min : $O(1)$
min = weight(e) $O(1)$

Algorithm postVertexVisit (G, V) :
S.insertLast ((V, min)) $O(1)$

Algorithm result(G)
return S; $O(1)$

∴ Running : $O(n+m)$

(b,w), (b,w) size()

② 0-1knapsack

B[k, w] = 0 if k=0 w=0

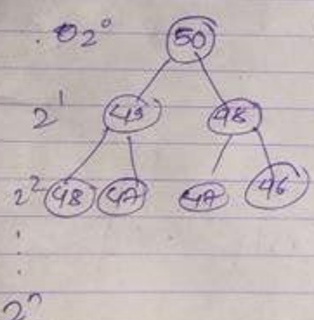
n = B[k-1, w] if wk < w

= max (B[k-1, w-1], B[k-1][w] + bk)

BruteForce

Algorithm 01knapsack (S, k, w)

2^n



if k=0 or w=0
return 0
else
bk, wk ⇐ S.removeFirst ()
if wk < w :
return 01knapsack (S, k-1, w)
else
return max (01knapsack (S, k-1, w-1),
01knapsack (S, k-1, w) + bk)

Efficient

M (n*k)

60

(2)

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

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

Algorithm Binomial (S, n, k)
    if B[n][k] = -1 :
        if k=0 or k=n:
            B[n][k] = 1
        else:
            B[n][k] = Binomial (S,n-1, k-1) +
                      Binomial (S,n-1, k)

    return B[n][k]

S [2|3]

Algorithm Main(S, K)
O(1) n ← S.size()
O(1) B ← new Array [n+1] [k+1]
O(n) for i ← 0 to n
O(n*k)   for j ← 0 to k
O(n*k)   B[i] [j] = -1
O(1) return Binomial (S, n, k)

R.T = O(n*k)

nahi aa... na laga...

2. (b) $B(5,3)$

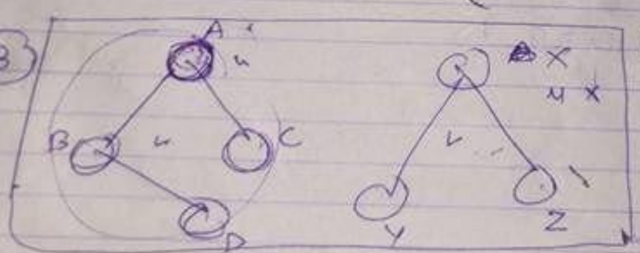
$= B(4,2) + B(4,3)$

$= B(3,1) + B(3,2) + \underbrace{B(3,2)} + \underbrace{B(3,3)}_{1}$

$= \underbrace{B(2,0)}_{1} + \underbrace{B(2,1)}_{2} + \underbrace{B(2,1)}_{2} + \underbrace{\boxed{B(2,2)}}_{1} +$
$\underbrace{B(2,1)}_{2} + \underbrace{\boxed{B(2,2)}}_{1} + 1$

$= 10$

Q. 3



u = A
~~v = D~~
True

u = A
v = Y
NO

Algorithm initResult (G)
result ← no

Algorithm startBFS (G, S)
uFound ← ~~no~~ false
vFound ← ~~no~~ false

Algorithm ~~pre~~VertexVisit (G, V)
if V = u:
uFound ← true
if V = V':
vFound ← true

Algorithm: finishBFS (G, S)
if uFound & vFound:
result ← yes.
~~else if uFound or vFound~~
~~result ← no~~

Algorith result (G)
return result

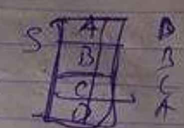
Algorithm: ~~post ComponentVisit()~~
~~if result = no~~
~~break.~~

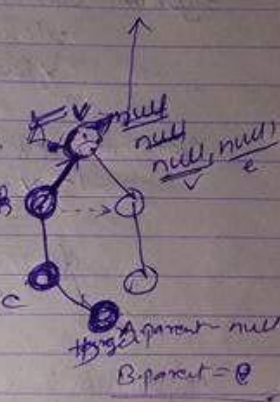
$O(n+m)$

*(13)

① **Path** (x, z)

```
init Result(G)
    S1 ← new stack

startBFS(G, S)
    S.parent ← null

preDiscoveryEdgeVisit(G, V, e, w)
    w.parent ← V

post DiscoveryEdge Visit (G, V, e, w)
    if w = Z :
        while w ¬= null) :
            S1.push(w)
            w ← w.parent

result(G) ~~if S1 is~~
    return S1
```

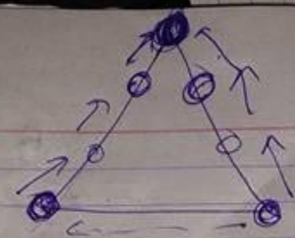


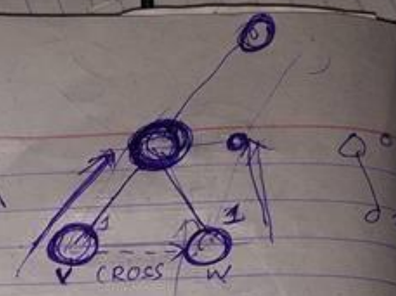


A.parent(null, null)

B.parent(V, e)

* Cycle





startBFS( ) ~~cnt ← 0~~

~~v~~. s.level = ~~cnt~~ 0 S_1 =

s.parent = null S_2

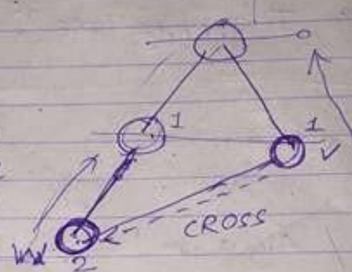
~~preVertexVisit(G, v)~~

~~cnt ← cnt+1~~

preDiscoveryEdgeVisit(G, v, e, w)

w.level ← v.level+1

w.parent ← v ~~parent~~



crossEdgeVisit (G, v, e, w)

if v.level > w.level:

while v.level ¬= w.level

S1.push(v)

v ← v.parent

else if v.level < w.level

while v.level ¬= w.level

S2push(w)

w ← w.parent

while ~~(w)~~ w ¬= v :

S2.push(w)

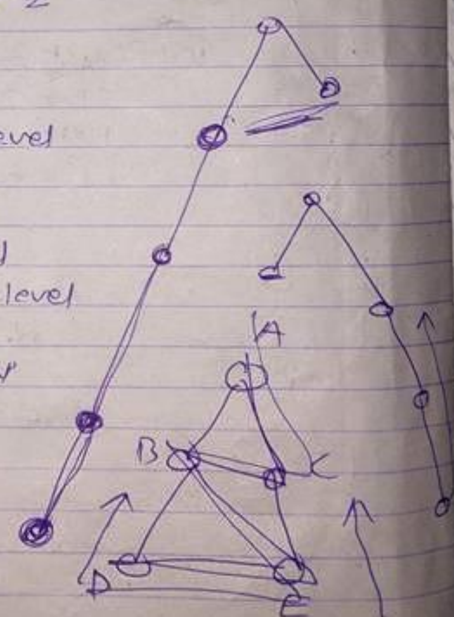
S1 push (v)

w ← w.parent

v ← v.parent

S1.push(w) // (v)

~~while S2 is empty~~



① checkMST(G, K)

$\quad T \leftarrow$ MST(G)

$\quad$ sum $\leftarrow$ sum of edges of T $\leftarrow$ $\left\{ \begin{array}{l} \text{sum} \leftarrow 0 \\ \text{for all } e \in G.\text{edges}() \\ \quad \text{sum} \leftarrow \text{sum} + e \end{array} \right.$

$\quad$ if sum > K

$\quad\quad$ return no

$\quad$ else

$\quad\quad$ return yes

② checkShortestPath(G, K)

$\quad T \leftarrow$ ShortestPath(G)

$\quad$ sum $\leftarrow$ sum of edges of T

$\quad$ if sum > K

$\quad\quad$ return no

$\quad$ else

$\quad\quad$ return yes

③ checkSubsetSum(S, max, min)

sequence

$\quad$ sum $\leftarrow$ sum of all elements of S

$\quad$ if min $\le$ sum $\le$ max:

$\quad\quad$ return yes

$\quad$ return no

④ LCS(S_1, S_2, k)

$\quad C \leftarrow$ LCS(S_1, S_2)

$\quad$ if $C \ge K$ return yes $\quad$ (at least k)

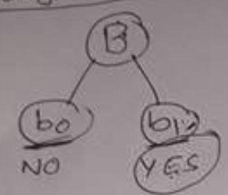
$\quad$ return no

P–NP Reduction

① Easy (Reduction)

- MST, ShortestPath, SubsetSum, LCS,

- Language Problem



$B(b_1)$ ↓

LCS (S1, S2, K)

a ← LCS(S1, S2

LCS

```
Algorithm LCS2B(S1, S2, K)
    a ← LCS(S1, S2)
    if a ≤ K then
        return b1
    else
        return b0
```

(10)

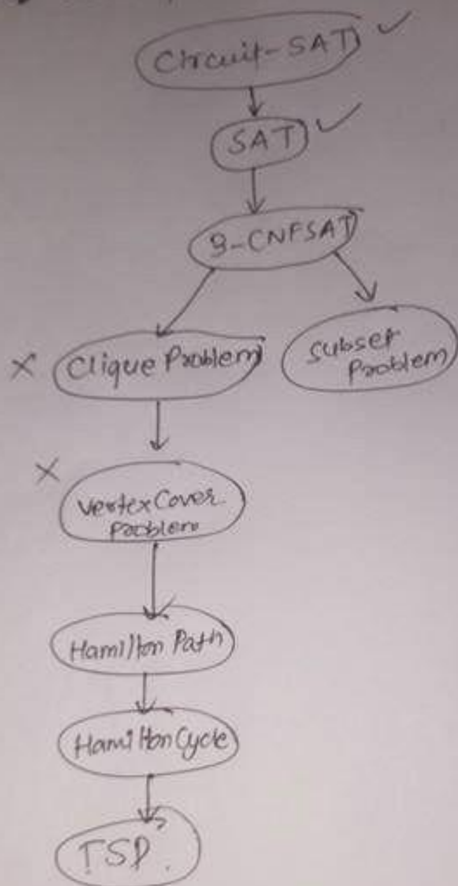
LCS

Brute Force = $2^n * m$

DP = $O(n*m)$

P → Sorting, searching, MST, LCS, Shortest Path

NPC → 01 knapsack,



NPH → Halting Problem

Algorithms CS...

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 $B \in P$

False

e.g. A → MST(P)

B → SSum(NPC) ∴ ($B \notin P$)

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

False ∴ $A \notin P$

e.g. A → SSum (NPC → NP)

B → 01Knapsack (NPC → NP)

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

True

e.g. B ∈ NPC, NPH

B ⊂ NPH

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

False

A → MST(P) $A \in P \notin NPC$

B → SSum(NPC)

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

False A → B $A, B \in P \notin NPC$

A → MST(P) B → A

B → ShortestPath(P)

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

False B ↛ A

A → some problem in P

B → some problem in NPC

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

e.g. A → 0-1 Knapsack (NPC) $B \notin NPC$

B → Halting Problem (NPH) $B \in NPH \notin NPC$

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

A → Halting Problem A → B

B → Some other problem in NPH which is not in NPC $B \notin NPC$

	Edge List	Adjacency List	Adjacency Matrix
Space	$n+m$	$n+m$	n^2
Incident Edges			

Spanning tree of a connected graph is a spanning subgraph that is tree. Spanning tree is not unique unless the graph is a tree.
No cycle.

$A \to_f B$ $B \to NPC$, $A \to$

P → P, NP, NPC, NPH
NP → P, NP, NPC, NPH
NPC → NPC, NPH, NP
NPH → NPH, NPC

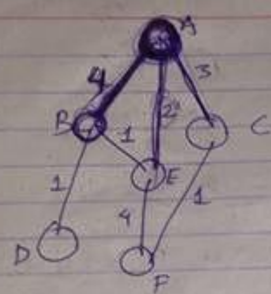
P → LCS, MST,
NPC → SAT, 3-SAT, Subset Problem, TSP, Hamitonian cycle/path.

NPH → Halting problem.

BFS/DFS $O(n+m)$

Shortest $m/\log n$

(1)



S [(A,0),
(B,1),
(C,1),
(D,1),
(E,1),
(F,1)]

(A,-1)

Algorithm initResult(G)
  S ← new Empty sequence $O(1)$

Algorithm preVertexVisit(G, V)
  min ← -1 $O(1)$

Algorithm preEdgeVisit(G, v, e)
  if min = -1 or weight(e) < min: $O(1)$
    min = weight(e) $O(1)$

Algorithm postVertexVisit(G, V):
  S.insertLast((V, min)) $O(1)$

Algorithm result(G)
  return S; $O(1)$

∴ Running: $O(n+m)$

(b,w), (b,v) size()

② 0-1 Knapsack

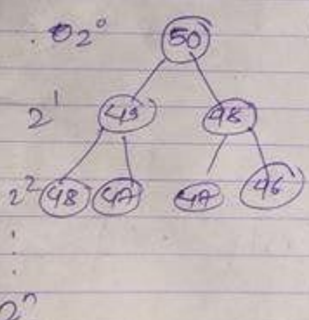
$B[k,w] = 0$ if $k=0$ $w=0$

n $= B[k-1,w]$ if $w_k < w$

$= \max(B[k-1, w-1], B[k-1][w] + b_k)$

BruteForce

Algorithm 01knapsack(S, k, w)




```
if k=0 or w=0
    return 0
else S
    bk, wk ← S.removeFirst()
    if wk < w:
        return 01knapsack(S, k-1, w)
    else
        return max(01knapsack(S, k-1, w-1),
                   01knapsack(S, k-1, w) + bk)
```

$\boxed{2^n}$

Efficient

M (n*k)

50