Name(last, first): Preet Modi

ID (rightmost 4 digits): 0307

UCLA Computer Science Department

CS 180 Algorithms & Complexity

Final Exam Total Time: 3 hours December 10, 201~~8~~9

*** Write all algorithms in bullet form (as done in the past) ***

You need to prove EVERY answer that you provide.

There are a total of 8 pages including this page.

Name(last, first): Modi, Preet

1. (20 points: each part has 10 points)

a. Consider a S-T network N. **Prove** that if **f** is a maxflow in the network N then there is a cut C with its capacity equal to **f**.

Consider flow f and consider the min cut c.

Two cases:

① $f < |C|$

② $f > |C|$

→ ① If flow is less than capacity of min cut, it means there is still some amount of flow possible from the set of vertices A that contain source s and the set of vertices B that contain sink t where A and B are the two partitions of vertices created by partitioning along the edges of the min cut. This means we can push more flow and that f is not a max flow which is a contradiction.

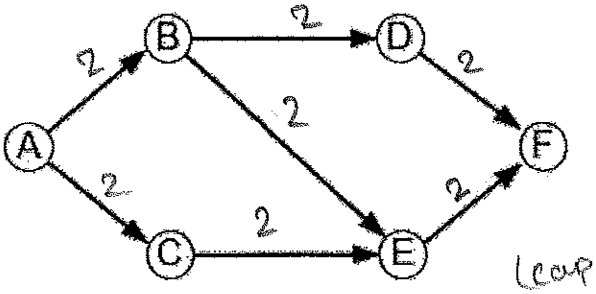
→ ② If flow is greater than capacity of min cut, there must be f = maxflow units of flow going from s to t. However if we partition the vertices into A and B where $s \in A$ and $t \in B$ along the edges of the min cut, we get a capacity $|C| < f$. There cannot be flow greater than capacity for any set of edges. Thus f cannot be more than c.

→ Therefore f must be equal to c

→ Thus, if f is a maxflow in the network N, there is a cut (namely the min cut) with its capacity equal to f.

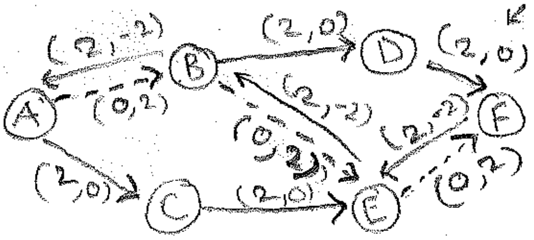
Name(last, first): Modi, Preet

b. Consider the following network with source **A** and sink **F**. If the Ford-Fulkerson max flow algorithm initially finds the path **A,B,E,F** in the network below and sends **2 units** of flow on it, **show the residual network** (also known as the augmented network) and all subsequent steps of Ford-Fulkerson algorithm on this network (all capacities are equal to 2).



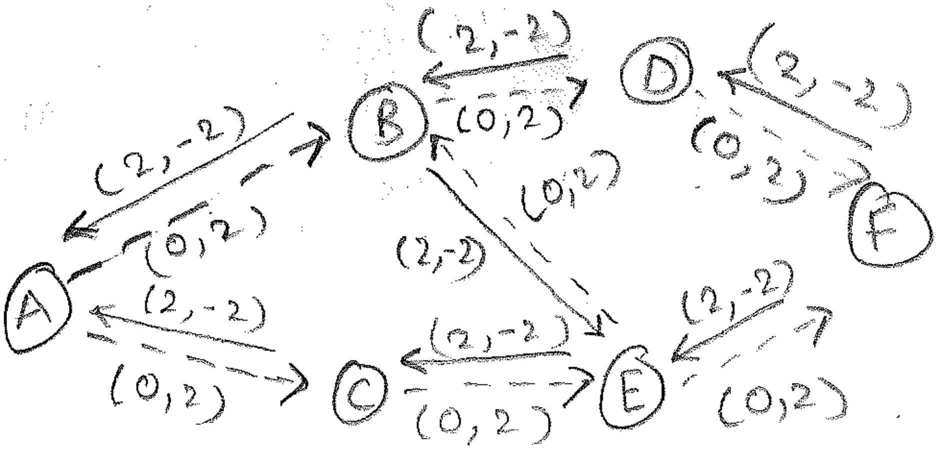
(cap, flow)

1)



flow = 2

2)



flow = 4

Name(last, first): Modi, Preet

2. (15 points) a. Given a n x n matrix where all numbers are distinct, design an **efficient algorithm** that finds the maximum length path (starting from any cell) such that all cells along the path are in increasing order with a difference of 1.

b. Analyze the time complexity of your algorithm

We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or (i-1, j) or (i, j-1) with the condition that the adjacent cells have a difference of 1.

1 2 9 5 3 8 6 7

```
Input:  mat[][] = {{1, 2, 9}
                   {5, 3, 8}
                   {4, 6, 7}}
Output: 4
The longest path is 6-7-8-9.
```

```
① store all mat[][] in a new 1-d array arr[]
   Define OPT() and set Opt(i) = 0 for all i <= 0, Opt(i) = 1 for all i > 0
② for( i = 1 to n^2 )
       a = 0, b = 0
       if ( |arr[i-1] - arr[i]| == 1 )      <- abs. value |...|
           a = Opt(i-1)
       if ( |arr[i-n] - arr[i]| == 1 )
           b = opt(i-n)
       if ( |arr[i-1] - arr[i-n]| == 2 )
           Opt(i) = Opt(i) + a + b
       else
           Opt(i) = Opt(i) + max(a, b)
   endfor
   - maintain a global max for ~~[illegible]~~ opt() called m and keep updating it when doing step ②
③ Follow Opt(m) backwards, considering Opt(m-1) and Opt(m-n) to generate path P.
   - Check if Opt(m-1) ~~[illegible]~~ Opt(m-n) = Opt(m-1),
       if yes add both to P
   - Else add the max to P breaking ties arbitrarily
④ Output P
```

Proof: - In the original array: mat[][], arr[i-1] represents the element to the left and arr[i-n] represents the element above.

- We consider both of them
- arr [i+1] represents the element to the right and arr [i+n] represents the element below in mat [ ] [ ]
- Our algorithm doesn't consider these when at i but does consider them when at i+1 and i+n respectively
- We do this until the end exhaustively checking all possibilities and choose the max.
- Our algorithm also ensures that arr [i-1] and arr [i-n] are in the right order with respect to arr [i].

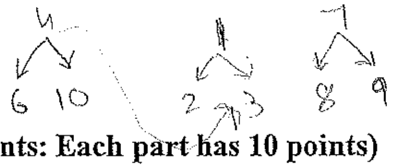
b) <u>Time Complexity</u>

① $O(n^2)$ to construct arr

② $O(n^2)$ to loop through arr[ ]

③ $O(n^2)$ at worst when all n^2 elements are a part of P.

Overall: $O(n^2)$



Name(last, first): Modi, Preet

1 2 3
7 8 9
4 6 10

3. (20 points: Each part has 10 points)

a. Consider **d** sorted arrays of integers each containing n1, n2, .. nd numbers. The numbers ni's can be very different. The total number of all elements is **n** (sum of all ni's). Design an **O(n log d)** algorithm that merges all arrays into one sorted list. You may wish to use a data structure that we have discussed in class. → heap?

for all (n)

b. Prove a lower bound on sorting **n** numbers in the decision tree model (using comparison exchange).

a) Create a min heap h.
① Check the ~~first~~ element of each of the d arrays and ~~find the min~~ store them in the min heap h.
For the array that the min came from say di, merge it with h rebalancing as needed

b) Consider a tree of numbers $(1, \ldots, N)$.
- There are at most 2^h leaves where h is the height of the tree
- With n numbers we have n! possible leaves.
  $2^h \geq n!$ - so we pick n! as we want the lower bound.
- To sort such a tree of numbers we must have at least $\log_2(n!)$ considerations
- $\log_2(n!) \approx n \log n$
- Thus sorting is $\Omega(n \log n)$

Name(last, first): Modi, Preet

4. (15 points)
Consider an array a1,...,an of n integers, that is hidden from us. We have access to this array through an procedure knapsack(.,.). For a set S ⊆ {1,...,n} and an integer k, knapsack(S,k) will output "yes" if there is a subset T ⊆ S such that the numbers indexed in T add up to k, and it will output "no" otherwise.

(-1, 2, 4, 7), 5

Design an algorithm that calls knapsack only O(n) times and outputs a set S ⊆{1,...,n} such that the numbers indexed in S add up to k, if such a set exists. You can use ONLY the knapsack function (e.g., you cannot sort the numbers or do any other operations on them).

For example, suppose a1 = 2, a2 = 4, a3 = 3, a4 = 1, and k = 7. Then, knapsack({1,2,3,4},7) returns "yes" and knapsack({1,3,4},7) returns "no". In this case your algorithm can output either of the sets {1,2,4} or {2,3}. Note that for example {1,2,4} are indices of the numbers, that is, a1, a2, and a4.

Let a[ ] be the array of numbers, S be the output set.

```
for (i = 1 to n)
    flag = knapsack( a[ ] without a[i])
    if( flag = "Yes")
        remove a[i] from all future considerations of a[ ]
    else
        add i to S
→ endfor check if knapsack(S,k) returns yes. If yes then return
    S, else return empty set.
```

Proof: There are 2 possibilities after removing an element

① Yes: This means the element was not essential and it is safe to remove as there is a subset without it that adds up to k.

② No: This means there is no subset that adds up to k without this element so it must be present in the final set.

- We finally check if the set S obtained can add up to k to deal with the case where none of the subsets add up to k, so we would have gotten No for each iteration.

Run time: O(n) - we pass through one for loop of O(n).

Name(last, first): Modi, Preet

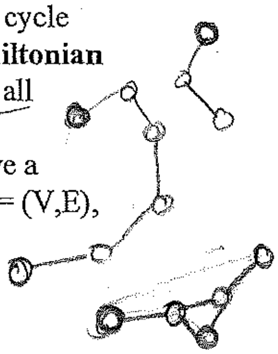
5. (15points) → undirected

A **Hamiltonian cycle** in a graph with **n** vertices is a cycle of length n, i.e., it is a cycle that visits all vertices exactly once and returns back to the starting point. A **Hamiltonian path** in a graph with n vertices is a path of length n−1, i.e., it is a path that visits all vertices of the graph exactly once.

Hamil-cycle problem is defined as follows: Given a graph G = (V,E), does it have a Hamiltonian cycle? Hamil-path problem is defined as follows: Given a graph G = (V,E), does it have a Hamiltonian path?

Prove that Hamil-path is polynomial-time transformable to Hamil-cyle.

That is Hamil-path ≤P Hamil-cycle.



- without loss of generality, take any arbitrary graph.
- Check the in degree and out degree of each vertex.
- Pick all paired vertices such that in degree ≠ out degree
- Say the pair we have is (u, v)
- Run DFS/BFS to find if u has a path to w, construct an edge between
- Say the graph produced by these operations is G', checking if G' has a hamil cycle is the same as checking if G has a hamil path.

Proof: - If G had a hamil path, there would be only one pair of vertices with in ≠ out, namely the first and the last vertex on the hamil path. They would also be reachable from one another. We construct an edge between them to make a hamil cycle.

- If G didn't have a hamil path, there would be an odd number vertex pairs with in ≠ out. Moreover, the in degree of at least one vertex would be 1 and out degree would be zero. This vertex would not have a new edge in our transformation, thus we would have a way to enter this vertex but not leave it. This would yield no hamil cycles.

Thus Hamil-Path ≤P Hamil-cycle.

Name(last, first): Modi, Preet

6. (15 points)

for any vertex I pick, I cannot pick ~~e~~ its children, or its parents

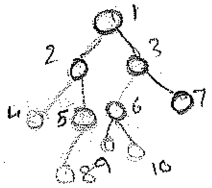
You are given a tree T where every node **i** has weight **wi** ≥ 0.

a. Design a polynomial time algorithm to find the weight of the largest weight independent set in T: among all independent sets one with maximum sum of the weights (an **independent set** is a subset of vertices where there are no edges between any of them).

For example, suppose in the following picture w1 = 3, w2 = 1, w3 = 4, w4 = 3, w5 = 6.
The maximum independent set has nodes 3,4,5 with weight 4 + 3 + 6 = 13.

①,⑥,⑦,④,⑤

```
1
| \
2  3
|  \
4   5
(1 is connected to 2 and 3.  And 2 is connected to 4 and 5)
```



b. Analyze the time complexity of your algorithm.

```
a) ① for each vertex v
        exclude its children and parent from consideration.
   ② for each remaining vertex w
        - compare its weight with the sum of the weights of its parent and its children.
        - remove the lesser set from consideration
        - for every vertex removed from consideration
          bring back its parent and children into consideration and accordingly update other vertices using the same idea.
     endfor
   ③ Calculate weight of all vertices remaining and keep a global max weight and update it when necessary.
   endfor
```

Proof :- In a tree, a vertex can only be connected to its parent and its children.

- So we can either include a vertex or include its parent and children
- Our algorithm considers both these choices ~~and~~ and ~~makes~~ picks the best possible outcome

PTO →

b) Run time

① $O(n^{\cancel{2}})$ - we loop for each vertex,

② $O(n^2 \log n)$ - we loop for $O(n)$ vertices, each update can take $O(n)$ and we have at most $\log n$ updates

③ $O(n)$ - we must $O(n)$ vertices

Overall: $O(n^3 \log n + n^2) = O(n^3 \log n)$