#

HKUST – Department of Computer Science and Engineering

COMP3711: Design and Analysis of Algorithms – Fall 2016

Final Examination – Solution Key

Date: Friday December 16, 2016 Time: 16:30-19:30 Venue: LG3 Multi-purpose Room

Name:	____________________	Student ID:	________________
Email:	____________________	Lecture	L1 / L2

Instructions

- This is a closed book exam. It consists of 24 pages and 9 questions .
- Please write your name, student ID and ITSC email and circle your lecture section (L1 is Tu,Th 12-13:20 and L2 is Tu,Th 16:30-15:50) at the top of this page.
- For each subsequent page that you write on, please write your student ID at the top of the page in the space provided.
- Please sign the honor code statement on page 2.
- Answer all the questions within the space provided on the examination paper. You may use the back of the pages for your rough work. The last 2 pages are scrap paper and may also be used for rough work. Each question is on a separate page most have at least one extra page for writing answers This is for clarity and is not meant to imply that each question requires all of the blank pages. Many can be answered using much less space.

Questions	1	2	3	4	5	6	7	8	9	Total
Points	4	6	10	12	12	14	13	14	15	100
Score										

Student ID: ___________

As part of HKUST's introduction of an honor code, the HKUST Senate has recommended that all students be asked to sign a brief declaration printed on examination answer books that their answers are their own work, and that they are aware of the regulations relating to academic integrity. Following this, please read and sign the declaration below.

```
I declare that the answers submitted for
this examination are my own work.

I understand that sanctions will be
imposed, if I am found to have violated the
University regulations governing academic
integrity.

Student's Name:      __________________________

Student's Signature:   _____________________
```

Student ID: ___________

1. **Sorting** [4 pts]

   The leftmost array gives 8 4-digit numbers. Run Radix sort on these numbers. Show the result after sorting the array on each digit. The rightmost array should be your final result.

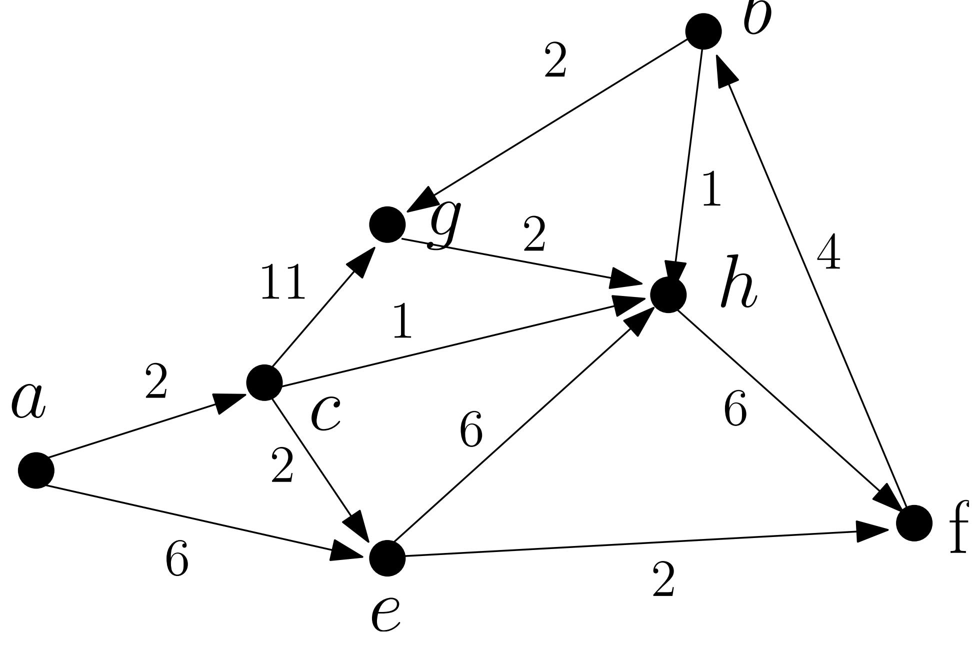
9 1 6 2				
8 2 3 4				
1 2 3 4				
4 3 2 1				
3 5 2 1				
3 6 2 1				
7 1 8 2				
2 9 8 5				

Solution:

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

Student ID: ________

2. **Dijkstra's Algorithm** [6 pts]
Run Dijskstra's algorithm on the graph below, starting from vertex a.



List the vertices by row in the order that Dijkstra's algorithm takes them out of the heap. Recall that $v.d$ is the current value that Dijkstra's algorithm stores with that vertex. Fill in the table below. We have filled in the first two rows for you.

	$a.d$	$b.d$	$c.d$	$e.d$	$f.d$	$g.d$	$h.d$
$-$	0	∞	∞	∞	∞	∞	∞
a	0	∞	2	6	∞	∞	∞

	$a.d$	$b.d$	$c.d$	$e.d$	$f.d$	$g.d$	$h.d$
$-$	0	∞	∞	∞	∞	∞	∞
a	0	∞	2	6	∞	∞	∞
c	0	∞	2	4	∞	13	3
h	0	∞	2	4	9	13	3
e	0	∞	2	4	6	13	3
f	0	10	2	4	6	13	3
b	0	10	2	4	6	12	3
g	0	10	2	4	6	12	3

3. **Stable Matchings** [10 pts]

Consider the following preference lists for the stable marriage problem for men A, B, C, D and women a, b, c, d.

Man	1st	2nd	3rd	4th
A	b	a	c	d
B	a	d	c	b
C	a	d	b	c
D	d	b	a	c

Woman	1st	2nd	3rd	4th
a	B	D	C	A
b	A	C	D	B
c	D	B	C	A
d	A	D	C	B

(a) Explain why A-b, B-c, C-a, D-d is NOT a stable matching.

Solution: Recall that a matching is not stable if an unstable pair exists. An unstable pair is an M-w pair in which M prefers over his current matched woman and w prefers M to over current matches man.

In this example B-a is an unstable pair because

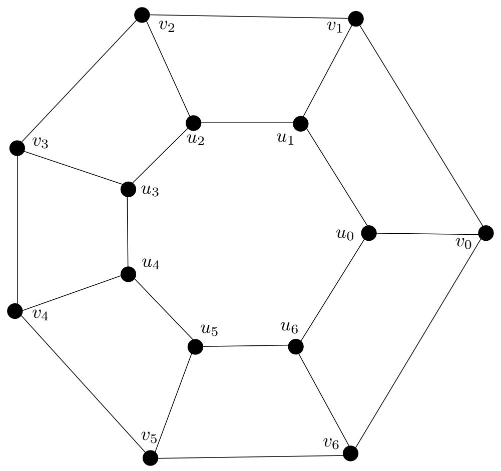
– "B" prefers "a" over his current match "c"
– "a" prefers "B" over her current match "C"

(b) Give the stable matching produced by the Gale-Shapely Propose-And-Reject algorithm on the preference lists (with men proposing).

A-b, B-a, C-c, D-d.

Student ID: _________

4. **BFS and DFS** [12 pts] (Problem continues onto next page)
Consider the undirected *Annulus Graph* A_7 with 14 vertices $V = \{u_0, \ldots, u_6, v_0, \ldots, v_6\}$ and the associated ordered adjacency lists as shown below:

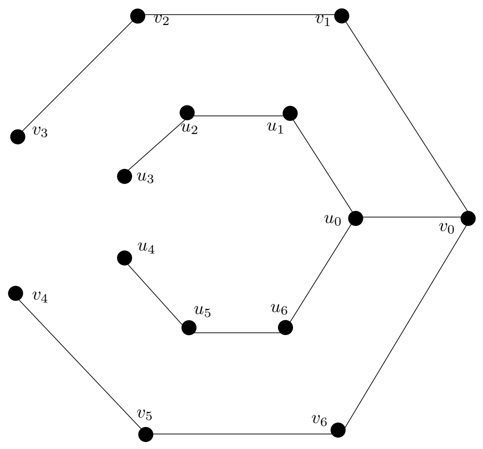


For all i, $0 \le i < 7$,

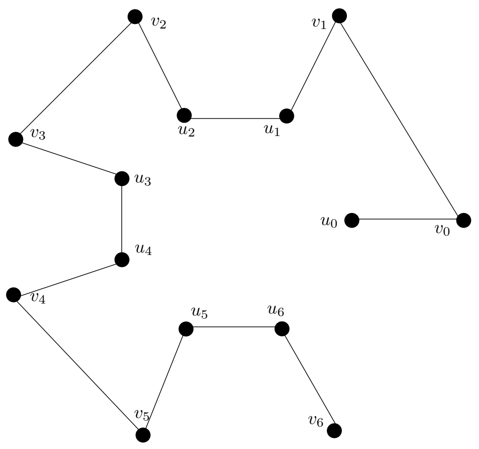
$$u_i \; : \; v_i, u_{(i+1) \bmod 7}, u_{(i-1) \bmod 7}$$
$$v_i \; : \; u_i, v_{(i+1) \bmod 7}, v_{(i-1) \bmod 7}$$

For instance, the adjacency list of u_1 is v_1, u_2, u_0 and the adjacency list of v_1 is u_1, v_2, v_0.

(a) Run BFS and DFS on A_7. Start both algorithms at u_0. Draw the edges of the resulting BFS and DFS trees on the appropriate graph below.

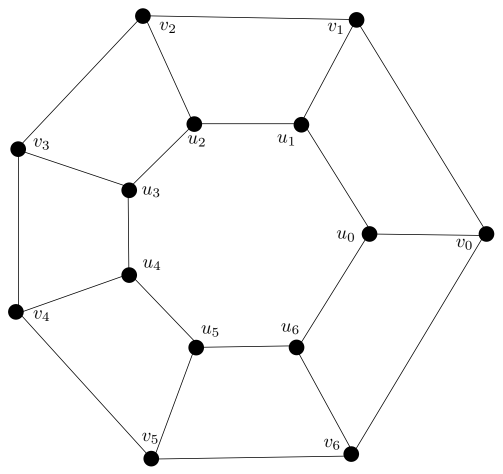


Draw BFS Tree edges connecting vertices above



Draw DFS Tree edges connecting vertices above

(b) The graph A_7 remains the same but we change the order of neighbors in the adjacency lists so that they are now as below:

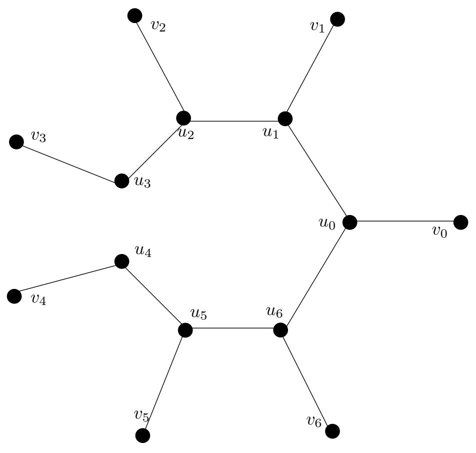


For all i, $0 \le i < 7$,

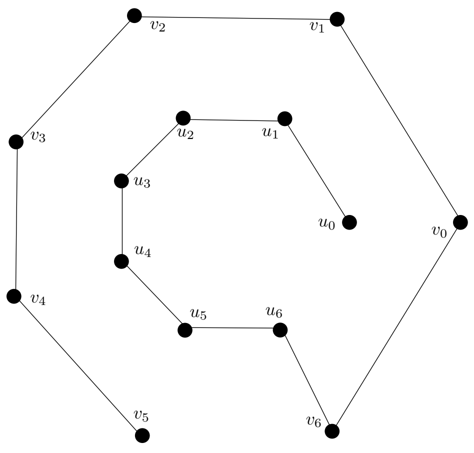
$$u_i \ : \ u_{(i+1) \bmod 7}, u_{(i-1) \bmod 7}, v_i$$
$$v_i \ : \ v_{(i+1) \bmod 7}, v_{(i-1) \bmod 7}, u_i$$

For instance, the adjacency list of u_1 is u_2, u_0, v_1 and the adjacency list of v_1 is v_2, v_0, u_1.

Now run BFS and DFS on A_7 using these new adjacency lists. Start both algorithms at u_0. Draw the edges of the resulting BFS and DFS trees on the appropriate graph below.



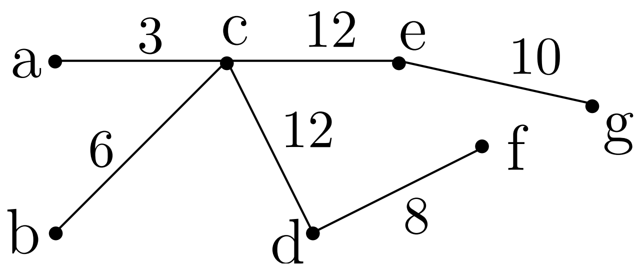
Draw BFS Tree edges connecting vertices above



Draw DFS Tree edges connecting vertices above

5. **Graphs** [12 pts]
A spanning tree of a graph G is a connected acyclic subgraph containing all vertices of G (not to be confused with *MST*, which is the spanning tree with the minimum sum of edge weights).

Let T be a spanning tree of an undirected weighted graph G. The *bottleneck weight* of T is the largest weight of any edge in T. As an example, in



the edges (c, e) and (c, d) have the bottleneck weight 12.

Design an $O(E)$ algorithm that, given connected, undirected, weighted graph G and value B, either returns a spanning tree with bottleneck weight at most B or reports back that no such spanning tree exists.

You must explain why your algorithm is correct and justify its running time.

You may use any algorithm given in class as a subroutine (without having to describe the internal details of that algorithm). If you do use such an algorithm, you must explicitly state the algorithm you are using and its running time.

The problem is asking whether there is a spanning tree composed of edges of weight B or less. This can be solved as follows:

(a) *Preprocess by running through all the adjacency lists once, throwing away all edges of weight $> B$.*

(b) *Run BFS or DFS starting from any vertex.*

(c) *Check if BFS/DFS sees all of the edges in one connected component. If it does, return "YES" and report the spanning tree output by the DFS or BFS. If it doesn't, report "N".*

- *The running time of both steps (a) and (b) are $O(V + E)$.*
- *Since G is connected, $V = O(E)$, so $O(V + E) = O(E)$.*
- *Step (c) can either be done within the BFS/DFS call or in $O(V) = O(E)$ time afterwards.*
- *Also note that (a) wasn't actually necessary. It was also possible to modfy BFS/DFS so that, in the loop that checks all neighbors of a vertex, it would ignore all edges with weight $> B$.*

6. **Minimum Spanning Trees** [14 pts]
Let G be a weighted undirected connected graph with distinct edge weights. Recall that we proved that such a graph has a unique Minimum Spanning tree. Let T be this unique MST.

   (a) Prove the *Cut Lemma* that we saw in class.

   Cut Lemma: Let S be any subset of nodes in G, and let e be the min-cost edge with exactly one endpoint in S. Then the MST of G must contain e.

   Your proof should be from first principals.

   (b) Either prove or disprove the following statement:

   The edge with the second smallest weight in graph G must always be in T

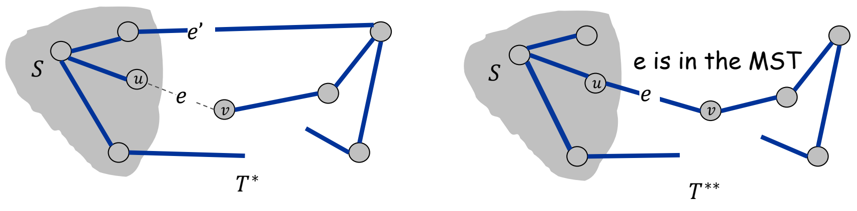
   (c) Either prove or disprove the following statement:

   The edge with the third smallest weight in graph G must always be in T

For parts (b) and (c) you should assume that G has more than 3 vertices. A proof of correctness may use the Cut-Lemma as a subroutine. Any other lemma must be proven from scratch. A proof that one of the statements is not correct requires a counter-example, e.g., find a MST that does not contain the edge with the second or third smallest weight.

Solution:

(a) *Here is the proof of the cut Lemma from the class notes.*



Let T^* be the MST.

- *Let $e = (u, v)$ and suppose $e \notin T^*$*
- *There must be path in T^* that connects u to v. Since u is in one side of the cut and v is on the other, this path must cross the cut separating S from $S - V$ using some other edge $e' \in T^*$. (If more than one cut crossing edge exists in T^*, you can set e' to be any such edge).*
- *Since e is the min-cost edge with exactly one endpoint in S, $w(e) < w(e')$.*
- *Remove e' from T^*. Add e to T^*. Because adding e creates a cycle with the path that connected u and v in T^*, removing any edge on that path creates a new spanning tree. In particular, removing e' creates a new spanning tree that we will call T^{**}. But*

$$cost(T^{**}) = cost(T^*) - w(e') + w(e) < cost(T^*)$$

contradicting the fact that T^* is a minimum *spanning tree.*

One **incorrect** *way of answering this question was to note that T^* must contain some edge e' crossing the cut, removing this edge e' and inserting e. The reason that this is incorrect is that it's quite possible that if there are multiple edges crossing the cut then this process would not create a tree (because e' is NOT on the cycle that inserting e would create). You needed to specifically reference the path in T^* that connects the endpoints of e in order to find the proper e'.*

(b) *True.*

Let $e_1 = (u, v)$ be the edge with the smallest weight in the graph and $e_2 = (u', v')$ be the edge with the second smallest weight.

There are two cases that need to be examined

(i) e_1 **and** e_2 **do not share any vertices.** *In this case define $S = \{u', u, v\}$. e_2 is then the min-cost edge with exactly one endpoint in S (because it is cheaper than every edge except for e_1 and e_1 has both edges in S). Thus, by the Cut Lemma, e_2 must be in the MST of G.*

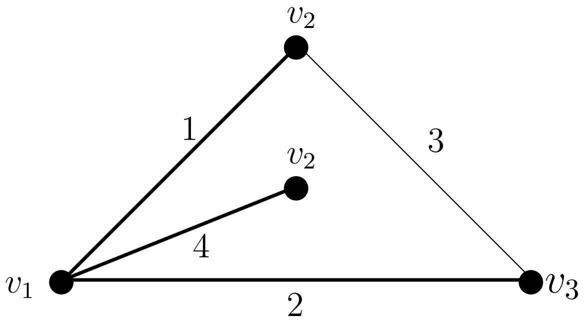
(ii) e_1 **and** e_2 **do share a vertex.** *Since they can't share both vertices (otherwise they would be the same edge) they only share one vertex. Without loss of generality suppose this is $u = u'$. Now define $S = \{u', v\}$. In this case again e_2 is the min-cost edge with exactly one endpoint in S (because it is cheaper than every edge except for e_1 and e_1 has both edges in S). Thus, by the Cut Lemma, e_2 must be in the MST of G.*

Since (i) and (ii) include all cases that can occur we have just shown that, in all cases, e_2 is in the MST of G.

An alternative proof is to consider what happens if e_2 is not in T^* already and is inserted into T^*. Since T^* is connected, this creates a cycle. Since a cycle must have at least three edges, one of those three edges must be an edge e'' which is NOT e_1 or e_2. Removing e'' leaves a new Tree T^{***} with lower cost than T^*, causing a contradiction.

(c) *False.*

Consider the counterexample below. The three heavy edges are the MST, which does not include the third cheapest edge (v_2, v_3).



7. **Greedy Algorithms** [13 pts]
Given two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ design an algorithm that determines if X is a *subsequence* of Y. You should assume $m \leq n$.

For example, if $X = BCBA$ and $Y = ABCBDAB$ then the answer is "yes". But, if $X = BDBA$, then the answer is "no". You should describe how your algorithm works, either with documented pseudocode or in words. For full credit, your algorithm should run in $O(n)$ time.

The algorithm walks through the items x_i checking $i = 1 \ldots m$ in order and walks through the items y_j, $j = 1, \ldots n$ in order. It always tries to match the current x_i to the first $y_j = x_i$ to the right of the current y_j and then moves j to the new index. If it manages to match all of the x_i it suceeds. If it runs out of items in Y before it finishes matching x_m, it fails.

Another way of writing this is that it sets

$$j_1 = \min\{j' : y_{j'} = x_1\}$$

and, for all i, $1 < i \leq m$ it sets

$$j_i = \min\{j' > j_{i-1} : y_{j'} = x_i\}$$

If all of the minimums exist, then it suceeds. If one doesn't, it fails.

The code can be written as

```
(a) $i = 1$; $j = 1$;
(b) While( $(i \leq m)$ AND $(j \leq n)$)
(c)     If $(x_i == y_j)$
(d)         $i = i + 1$;    $j = j + 1$;
(e)     ELSE $j = j + 1$
(f) If $i == m + 1$ then
(g)     print "success"
(h) ELSE print "Failure
```

8. **Max-Flow** [14 pts]

A community of n people is trying to set up a neighborhood patrol. They will send out m patrols each week and each patrol needs three participants. Every community member has agreed to participate in at least 2 patrols a week, when he or she is available and $2n = 3m$. The availability information is provided as a set of variables $A_{i,j}$, for $1 \le i \le n$, $1 \le j \le m$,

$$A_{i,j} = \begin{cases} \text{TRUE} & \text{if person } i \text{ is available to participate in patrol } j \\ \text{FALSE} & \text{Otherwise} \end{cases}$$

The *Roster Problem* is to decide whether it is possible to create a feasible schedule. A feasible schedule is one in which:

- Exactly 3 people are assigned to each patrol
- Each person is assigned to exactly two patrols, and
- Person i can only be assigned to patrol j if $A_{i,j} = \text{TRUE}$

(a) Describe how to solve the roster problem by creating an associated max-flow problem and solving that problem using the Ford-Fulkerson algorithm. You must describe the max-flow problem you create (listing the edges and capacities) and show how to transform the solution to the max-flow problem into a solution to the roster problem. It is not necessary to prove correctness of your algorithm

(b) What is the running time of your algorithm as a function of n?

(c) In part (a) you used the Ford-Fulkerson algorithm as a subroutine. As explained in class, if edge capacities are all integers, the Ford-Fulkerson algorithm returns a flow in which every edge value is also an integer. Suppose that your computer system provides another Max-Flow solver, not Ford-Fulkerson, that will always return a Max-Flow, but does not provide the same guarantee, i.e., even if the capacities are all integers it might return a Max-Flow with some non-integer values.

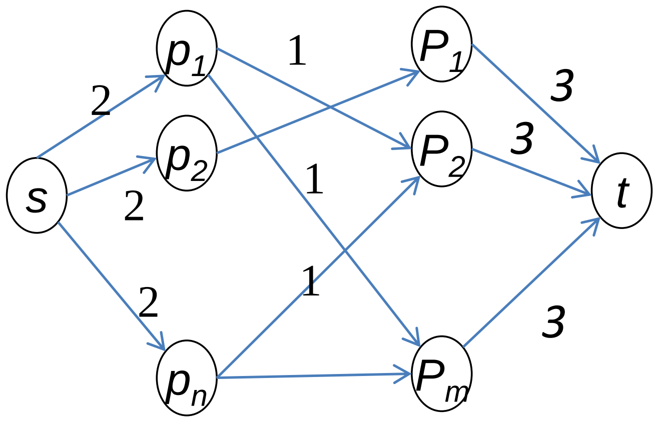
If you replace the Ford-Fulkerson algorithm in part (a) with your system's Max-Flow solver will your algorithm still always be able to find a correct solution to the roster problem? Provide a brief explanation as to why it will or will not be able to find a correct solution. If the answer is that it will not, give a brief example as part of your explanation, showing where your algorithm breaks down.

Solution

(a) *This is very similar to the extensions of Max-Bipartite matchings we saw in class. Create a flow network with*

- *n vertices p_i, $i = 1, \ldots, n$ corresponding to the people.*
- *m vertices P_j, $j = 1, \ldots, n$ corresponding to the patrols*
- *A source vertex s and a sink vetex t*
- *n edges (s, p_i), $i = 1, \ldots, n$, each with capacity 2*
- *m edges (P_j, t), $j = 1, \ldots, n$, each with capacity 3*
- *An edge (p_i, P_j) for every pair $A_{i,j}$ = True, with each such edge having capacity 1*

as illustrated below



Run the Ford-Fulkerson algorithm on this network. If the value of the maxflow $= 2n = 3m$, then a feasible schedule exists. Otherwise, no feasible schedule exists.

(b) *The Ford-Fulkerson algorithm runs in time $O(|f^*|E)$ where $|f^*|$ is the value of the max-flow. In our example $|f^*| \leq 2n$. Also, since $m = \Theta(n^2)$ (because $2n = 3m$) we have $E = O(mn + m + n) = O(n^2)$. So, the running time of the algorithm in terms of n is $O(|f^*|n) = O(nn^2) = O(n^3)$.*

(c) *The answer is YES. It will be able to solve the problem. The value of the max-flow will be the same regardless of whether or not the solution has integer flow values or not. So, even with the different solver, al y ou need to do is check whether the max-flow $= 2n$ or not.*

What it will NOT be able to do is find the assignments of the people to patrols. This is because the algorithm for finding the assignments assigns person p_i to patrol P_j if and only if the flow from p_i to P_j in the max flow equals 1. If the algorithm you are using does not guarantee that the flows are integers you can't use the flows to figure out which people get assigned to which patrol. For example, if $A_{i,j} =$ True for every pair i, j then we can set the flow $f(i, j) = \frac{2}{m} = \frac{3}{n}$ for every pair i, j. There would be no way to figure out which person would go to which patrol from this.

Although the correct answer was YES, we gave full credits to anyome who said NO and gave an explanation as to why it could not find the assignments.

9. **Dynamic programming** [15 pts]
Suppose you are given three strings of characters: $X = x_1x_2 \cdots x_n$, $Y = y_1y_2 \cdots y_m$, $Z = z_1z_2 \cdots z_p$, where $p = n + m$. Z is said to be a *shuffle* of X and Y iff Z can be formed by interleaving the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string. The goal of this problem is to design a dynamic-programming algorithm to determine whether Z is a shuffle of X and Y.

   (a) Show that *cchocohilaptes* is a shuffle of *chocolate* and *chips*, but that *chocochilatspe* is not.

   (b) For any string $A = a_1a_2 \cdots a_r$, let $A_i = a_1a_2 \cdots a_i$ be the substring of A consisting of the first i characters of A. For example, if A is *chocolate*, then A_3 is *cho* and A_6 is *chocol*.

   Define $f(i, j)$ to be 1 if Z_{i+j} is a shuffle of X_i and Y_j, and 0 otherwise. Derive a recurrence relation for $f(i, j)$. Remember to include the basis cases. Briefly explain your derivation.

   (c) Give pseudocode for an algorithm for determining whether Z is a shuffle of X and Y. Analyze the running time of your algorithm. For full marks in this part and part (b) your algorithm should run in $O(nm)$ time.

Solution

   *(a) Consider c*__choco__*hi*__la__*p*__te__*s*
   The letters from "chocolate" are in bold, while the letters from "chips" are not.
   For the second part set X = chocolate, Y = chips and Z = chocochilatspe, so $n = 9$, $m = 5$ and $p = 14$. There are at last two correct approaches to showing that Z is not a shuffle of X and Y.
   The first is to note that the letters p, s both only appear only in Y and not in X. Therefore, if Z is a shuffle of X, Y the letters p, s must both be coming from Y. But, they appear in the order ps in Y and the reverse order sp in X, which is impossible in a shuffle.
   The second approach is to prove that Z is not a shuffle of X and Y by working backwards.

- *$z_{14} = e = x_9$ but $z_{14} \neq y_5 = s$.*
  $\Rightarrow z_{14}$ matches to x_9.
- *This means that either $z_{13} = x_8 = t$ or $z_{13} = y_5 = s$.*
- *But $z_{13} = p$ so neither of those is possible and therefore Z is not a shuffle of X and Y.*

(b) *The boundary conditions are the values of $f(i,0)$ and $f(0,j)$.*

When $j = 0$, $f(i,0) = 1$ is just the statement that $X_i = Z_i$. This can be checked recursively by setting $f(1,0) = 1$ if and only if $x_1 = z_1$ and, for $i > 1$ $f(i,0) = 1$ if and only if $f(i-1,0) = 1$ and $x_i = z_i$. A similar set of basis cases holds for $f(0,j)$.

For $i,j > 1$, if X_i and Y_j can be shuffled to result in Z_{i+j} then one of the two following cases must occur

- *The last item in Z_{i+j} is the last item in X_i.*
  This can happen if and only if $\mathbf{z_{i+j} = x_i}$ and Z_{i+j-1} is a shuffle of X_{i-1} and Z_j, i.e., $\mathbf{f(i-1,j) = 1}$.
- *The last item in Z_{i+j} is the last item in X_j.*
  This can happen if and only if $\mathbf{z_{i+j} = x_j}$ and Z_{i+j-1} is a shuffle of X_i and Z_{j-1}, i.e., $\mathbf{f(i,j-1) = 1}$.

The initial conditions are then

$$f(i,0) = \begin{cases} 1 & \text{if } i = 1 \text{ AND } x_1 = z_1 \\ 1 & \text{if } i > 1 \text{ AND } f(i-1,0) = 1 \text{ AND } x_i = z_i \\ 0 & \text{otherwise} \end{cases}$$

$$f(0,j) = \begin{cases} 1 & \text{if } j = 1 \text{ AND } y_1 = z_1 \\ 1 & \text{if } j > 1 \text{ AND } f(0,j-1) = 1 \text{ AND } y_j = z_i \\ 0 & \text{otherwise} \end{cases}$$

and the general recurrence is, for $i,j > 1$

$$f(i,j) = \begin{cases} 1 & \text{if } (f(i-1,j) = 1 \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{if } f(i,j-1) = 1 \text{ AND } z_{i+j} = y_j) \\ 0 & \text{otherwise} \end{cases}$$

One common error was to write that $f_{i,j} = 1$ if

$$(f(i-1,j) = 1 \text{ OR } f(i,j-1) = 1) \text{ AND } (z_{i+j} = x_i \text{ OR } z_{i+j} = y_j).$$

To see that this is not correct, consider $X = abc$, $Y = def$ and $Z = abdeff$. For $i = 3,$, $j = 3$ this satisfies the condition above which would imply $f(3,3) = 1$ but $f(3,3) = 0$

(c)

```
i.    Set all $f(i,j) = 0$;
ii.   If $x_1 = z_1$ set $f(1,0) = 1$;
iii.  If $y_1 = z_1$ set $f(0,1) = 1$
iv.   For $i = 2$ to $n$
v.        If ($(f(i-1,0) = 1)$ AND $(x_i = z_i)$)
vi.           set $f(i,0) = 1$;
vii.  For $j = 2$ to $m$
viii.     If ($(f(0,j-1) = 1)$ AND $(y_j = z_j)$)
ix.           set $f(0,j) = 1$;
x.    For $i = 1$ to $n$
xi.       For $j = 1$ to $m$
xii.          If ($f(i-1,j) = 1$ AND $z_{i+j} = x_i$)
                OR (if $f(i,j-) = 1$ AND $z_{i+j} = y_j$)
xiii.            set $f(i,j) = 1$
```