CS570
Analysis of Algorithms
Spring 2010
Exam II

Name: ______________________
Student ID: __________________

DEN Student YES / NO

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	20	
Problem 4	20	
Problem 5	20	
Total	100	

2 hr exam
Close book and notes

If a description to an algorithm is required please limit your description to within 200 words, anything beyond 200 words will not be considered.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from source to sink.
TRUE

In a flow network whose edges all have capacity 1, the maximum flow value equals the maximum degree of a vertex in the flow network.
FALSE.

Memoization is the basis for a top-down alternative to the usual bottom-up approach to dynamic programming solutions.
TRUE

The time complexity of a dynamic programming solution is always lower than that of an exhaustive search for the same problem.
FALSE

If we multiply all edge capacities in a graph by 5, then the new maximum flow value is the original one multiplied by 5.
TRUE.

For any graph G with edge capacities and vertices s and t, there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from s to t. (Assume that there is at least one path in the graph from $\boldsymbol{s}$ to $\boldsymbol{t}$.)
FALSE.
There might be more than one min-cut, by increasing the edge capacity of one min-cut doesn't have to impact the other one.

Let G be a weighted directed graph with exactly one source s and exactly one sink t. Let (A, B) be a maximum cut in G, that is, A and B are disjoint sets whose union is V, $s \in A, t \in B$, and the sum of the weights of all edges from A to B is the maximum for any two such sets. Now let H be the weighted directed graph obtained by adding 1 to the weight of each edge in G. Then (A, B) must still be a maximum cut in H.
FALSE
There could exist other edge which has many more cross-edges, which was not max-cut previously, to become the new max-cut.

A recursive implementation of a dynamic programming solution is often less efficient in practice than its equivalent iterative implementation.
We give points for both TRUE and FALSE answers due to the ambiguity of "often".

Ford-Fulkerson algorithm will always terminate as long as the flow network G has edges with strictly positive capacities.
FALSE
If some edges are irrational numbers, Ford-Fulkerson may never terminate.

Any problem that can be solved using dynamic programming has a polynomial time worst case time complexity with respect to its input size.
FALSE

2) 20 pts

Judge the following statement is true or false. If true, prove it. If false, give a counter-example.

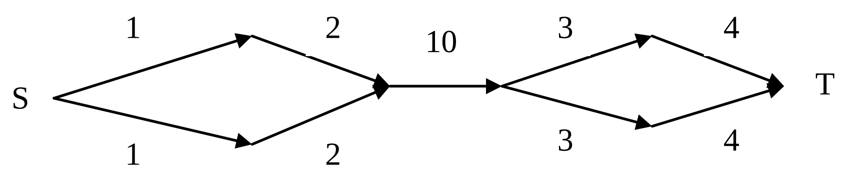
Given a directed graph, if deleting edge *e* reduces the original maximum flow more than deleting any other edge does, then edge *e* must be part of a minimum s-t cut in the original graph.

Solution:

False.

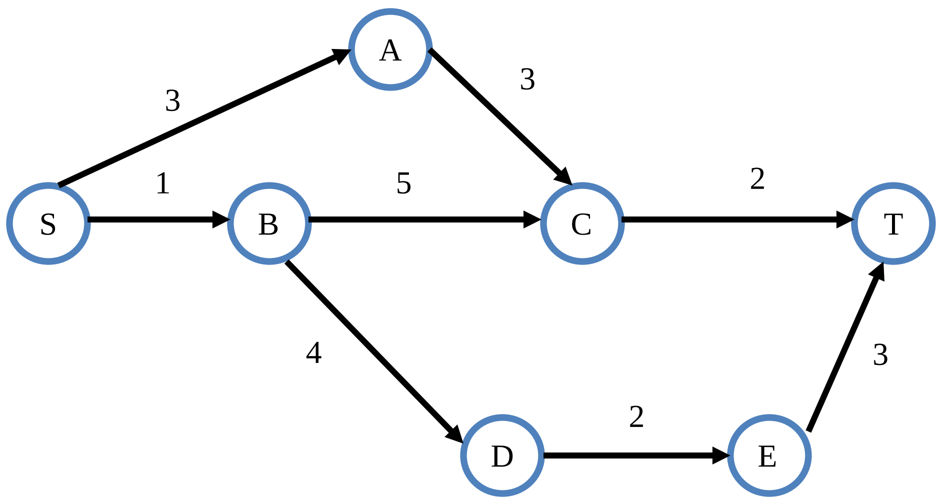
Counter-example:

Apparently deleting the edge with capacity 10 will reduce the flow by the most, while it is not part of minimum s-t cut.



Comment: some of you misunderstood the problem in different ways. Some counter-examples have multiple min s-t cuts and the edge "e" is actually in one of them. And some others failed to satisfy the precondition "more than… any other", and have some equal alternatives, which are also incorrect.

3) 20 pts



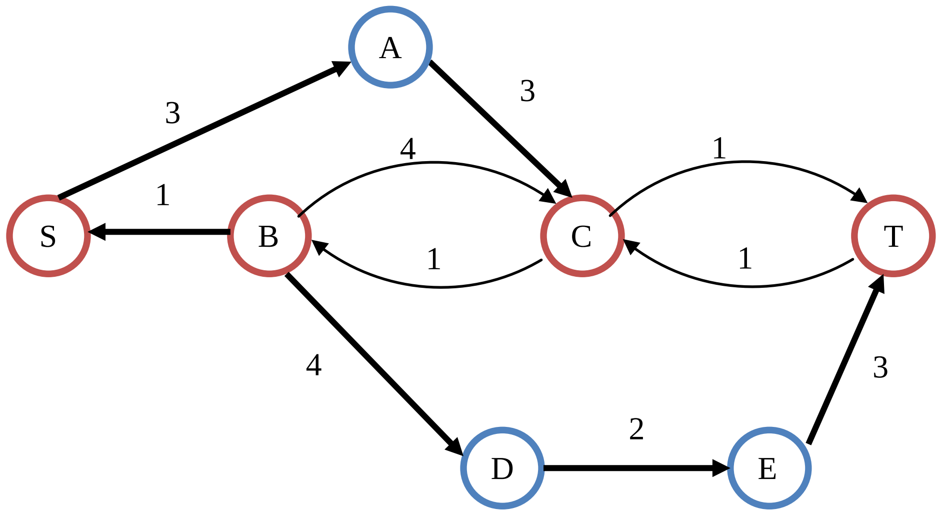
a- Show all steps involved in finding maximum flow from S to T in above flow network using the Ford-Fulkerson algorithm.

b- What is the value of the maximum flow?

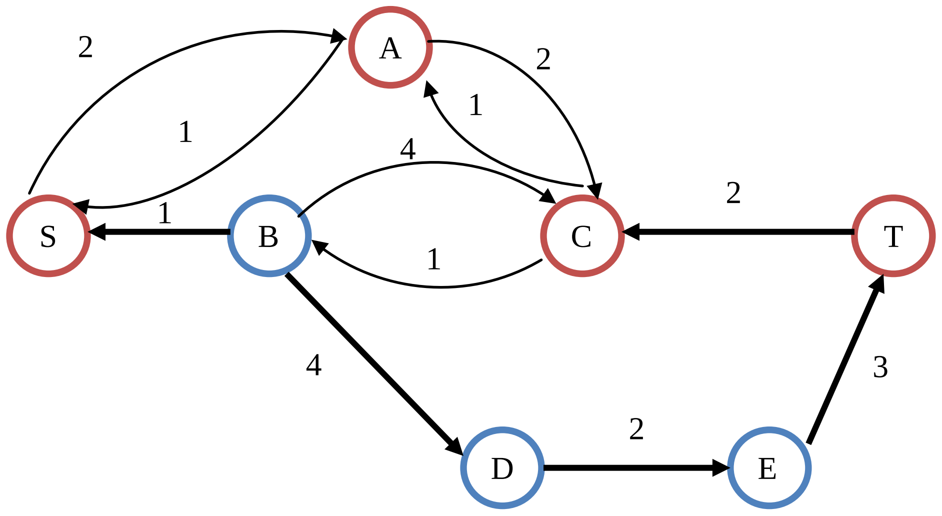
c- Identify the minimum cut.

a.

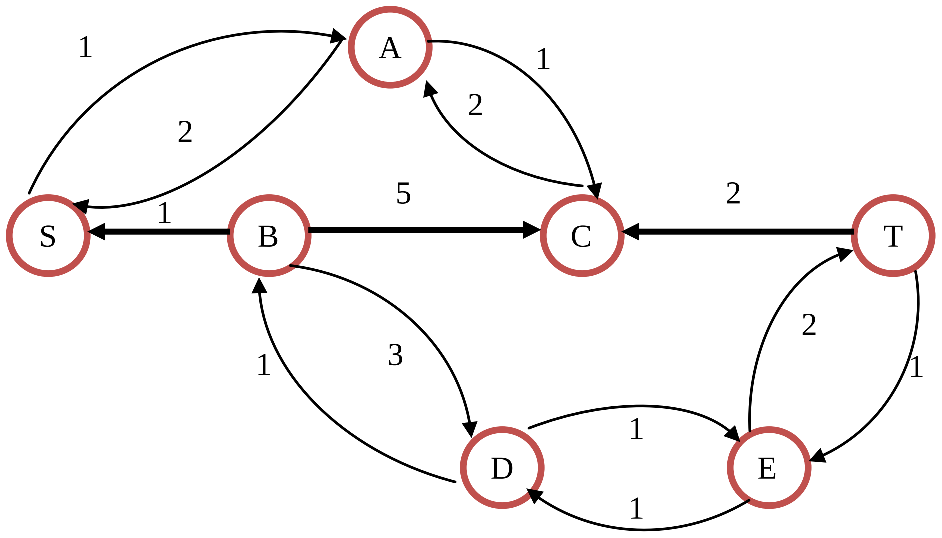
1. Augment path: [S, B, C, T], bottleneck is 1, residual graph:



2. Augment path: [S, A, C, T], bottleneck is 1, residual graph:



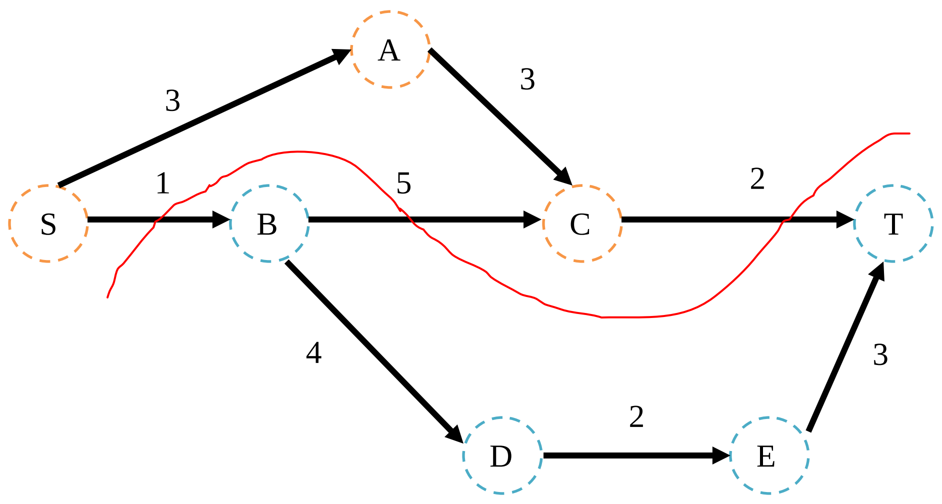
3. Augment path: [S, A, C, B, D, E, T], bottleneck is 1, residual graph:



4. No forward path from S to T, terminate.

b. The maximum flow is 3.

c. The Min-Cut is: [S, A, C] and [B, D, E, T].



4) 20 pts
The words `computer' and `commuter' are very similar, and a change of just one letter, p→m will change the first word into the second. The word `sport' can be changed into `sort' by the deletion of the `p', or equivalently, `sort' can be changed into `sport' by the insertion of `p'.

The edit distance of two strings, s1 and s2, is defined as the minimum number of point mutations required to change s1 into s2, where a point mutation is one of:

1. change a letter,
2. insert a letter or
3. delete a letter

a) Design an algorithm using dynamic programming techniques to calculate the edit distance between two strings of size at most *n*. The algorithm has to take less than or equal to $O(n^2)$ for both time and space complexity.
b) Print the edits.

The following recurrence relations define the edit distance, $d\ (s1, s2)$, of two strings s1 and s2:

1. $d\ ('', '')\ =\ 0$ (for empty strings)
2. $d\ (s, '')\ =\ d\ ('', s)\ =\ |s|$ (length of s)
3. $d\ (s1 + ch1, s2 + ch2)\ =\ \boldsymbol{min}(d(s1, s2) + \begin{cases} 0, if\ ch1 = ch2 \\ 1, else \end{cases}, d(s1 + ch1, s2) + 1, d(s1, s2 + ch2) + 1\ )$

The first two rules above are obviously true, so it is only necessary consider the last one. Here, neither string is the empty string, so each has a last character, $ch1$ and $ch2$ respectively. Somehow, $ch1$ and $ch2$ have to be explained in an edit of $s1 + ch1$ into $s2 + ch2$.

- If $ch1$ equals $ch2$, they can be matched for no penalty, which is 0, and the overall edit distance is $d(s1, s2)$.
- If $ch1$ differs from $ch2$, then $ch1$ could be changed into $ch2$, costing 1, giving an overall cost $d(s1, s2) + 1$.
- Another possibility is to delete $ch1$ and edit $s1$ into $s2 + ch2$, $d(s1, s2 + ch2) + 1$.
- The last possibility is to edit $s1 + ch1$ into $s2$ and then insert $ch2$, $d(s1 + ch1, s2) + 1$.

There are no other alternatives. We take the least expensive, ***min***, of these alternatives.

Examination of the relations reveals that $d(s1, s2)$ depends only on $d(s1', s2')$ where $s1'$ is shorter than $s1$, or $s2'$ is shorter than $s2$, or both. This allows the *dynamic programming* technique to be used.

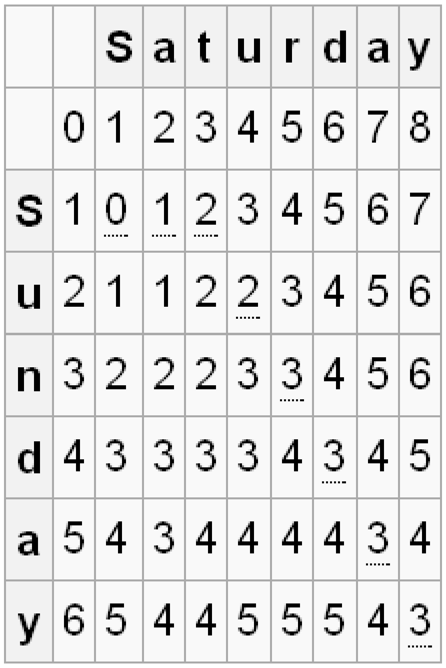
A two-dimensional matrix, m[0..|s1|,0..|s2|] is used to hold the edit distance values:

```
m[i,j] = d(s1[1..i], s2[1..j])

m[0,0] = 0
m[i,0] = i,  i=1..|s1|
m[0,j] = j,  j=1..|s2|

m[i,j] = min(m[i-1,j-1] + if s1[i]=s2[j] then 0 else 1,
             m[i-1, j] + 1,
             m[i, j-1] + 1 ),  i=1..|s1|, j=1..|s2|
```
$m[,]$ can be computed *row by row*. Row $m[i,]$ depends only on row $m[i-1,]$. The time complexity of this algorithm is $O(|s1| * |s2|)$. If $s1$ and $s2$ have a similar length n, this complexity is $O(n^2)$.

b)
Once the algorithm terminates, we have generated the edit distance matrix m, we can do a trace-back for all the edits, e.g. Figure 1:



		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 1 Edit distance matrix (from Wikipedia)

Here we are using a head recursion so that it prints the path from beginning to the end.

```
Print_Edits($n1$, $n2$)
if $n = 0$ then
    Output nothing
else
    Find $(i,j) \in \{(n1-1, n2), (n1, n2-1), (n1-1, n2-1)\}$ that has the minimum value from
    $m[i,j]$.
    Print_Edits($i, j$)
    if $m[i,j] = m(n1, n2)$ then
```

```
        Do nothing
    else
        if $(i, j) = (n1 - 1, n2 - 1)$ then
            Print "$change\ s1's\ n1^{th}\ letter\ into\ s2's\ n2^{th}\ letter$"
        else if $(i, j) = (n1, n2 - 1)$ then
            Print "$insert\ s2's\ n2^{th}\ letter\ after\ s1's\ n1^{th}\ position$"
        else
            Print "$delete\ s1's\ n1^{th}\ letter$"
```

The trace-back process has the complexity of $O(n)$.

5) 20 pts
Given a 2-dimensional array of size $m \times n$ with only “0” or “1” in each position, starting from position [1, 1] (top left corner), every time you can only move either one cell to the right or one cell down (of course you must remain in the array). Give an *O(mn)* algorithm to calculate the number of different paths without reaching “0” in any step, starting from [1, 1] and ending in [*m*, *n*].

Solution:

We can solve this problem using dynamic programming. We denote a[i][j] as the array, num[i][j] as the number of different paths without reaching "0" in any step starting from [1, 1] and ending in [i, j]. Then the desired solution is num[m][n].
We have

$$num[i][j] = \begin{cases} 0 & \text{i=0 or j=0 or a[i][j]=0} \\ a[1][1] & \text{i=1 and j=1} \\ num[i-1][j] + num[i][j-1] & other \end{cases}$$

This is simply because we can reach cell [i, j] by coming from either [i-1, j] or [i, j-1]. When a[i][j]=1, num[i][j] is just the sum of the two num's from the two different directions.
By calculating num row by row, we can get num[m][n] at last. This is an O(mn) solution since we use O(1) time to calculate each num[i][j] and the size of array num is O(mn).

Comment: this is different from the "number of disjoint paths" problem which can be solved by being reduced to max flow problem. And some used max function instead of plus. Some other used recursion, but without memorization which would course the complexity become exponential. And some other tried to find paths and count, which is also an approach with exponential complexity.