CS570 Spring2022: Analysis of Algorithms   Exam II

	Points		Points
Problem 1	20	Problem 4	16
Problem 2	6	Problem 5	16
Problem 3	20	Problem 6	22
	Total	**100**	

Instructions:
1. This is a 2-hr exam. Open book and notes. No electronic devices or internet access.
2. If a description to an algorithm or a proof is required, please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.
5. Do not detach any sheets from the booklet. Detached sheets will not be scanned.
6. If using a pencil to write the answers, make sure you apply enough pressure, so your answers are readable in the scanned copy of your exam.
7. Do not write your answers in cursive scripts.
8. This exam is printed double sided. Check and use the back of each page.

1) 20 pts
Mark the following statements as **TRUE** or **FALSE** by circling the correct answer. No need to provide any justification.

[TRUE/[✓ FALSE]]
In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will always be optimal.

- Explanation: Consider the following knapsack problem: $W = 40$,
  $w_0 = 10,\ w_1 = 20,\ w_2 = 30$
  $v_0 = 40,\ v_1 = 60,\ v_2 = 50$
  Then choosing items 0 and 2 uses up all the capacity of the knapsack, but only achieves value 90; the optimal choice is items 0 and 1, which uses capacity 30 and achieves value 100.

[TRUE/[✓ FALSE]]
Suppose that a new max-flow algorithm operating on a flow network G with integer capacities finds a non-integer max-flow f, i.e., the flow assigned to each edge is not necessarily an integer. Then there may exist some s-t cut (A,B) in G where $f^{out}(A) - f^{in}(A)$ is non-integer.

- Explanation: value of max flow in a flow network will always be an integer and the sum of flow going through each cut is always equal to the value of flow

[[✓ TRUE]/FALSE]
For every min-cut (A,B) and max-flow f in a flow network G, if $e=(u,v)$ is a directed edge with $u \in A$ and $v \in B$, then $f(e) = c_e$.

- Explanation: By the Max-Flow Min-Cut Theorem, if (A,B) is a min-cut and f is a max-flow, then $v(f) = c(A,B)$. However, if $f(e) < c(e)$ for some edge $e = (u,v)$ with u in A and v in B, then $v(f) \leq \sum_{e \text{ out of } A} f(e) < \sum_{e \text{ out of } A} c_e = c(A,B)$, a contradiction.

[TRUE/[✓ FALSE]]
When the Ford-Fulkerson algorithm terminates, there must not be any directed edge going out of the source node S in the residual graph.

- Explanation: This only happens if a min cut is right next to S but this may not be the case.

[[✓ TRUE]/FALSE]

If removing an internal node *u* from a flow network *G* disconnects its source from its sink, then the number of iterations required for Ford-Fulkerson to find max-flow in G will be bounded by the sum of the capacities of the edges going into *u*.
Explanation: All flow from S to T has to go through u

[TRUE/FALSE]

In a flow network, if all s-t cuts have integer capacities then all edge capacities must be integers.

- Explanation: See Manhattan walk example in discussion problems

[TRUE/FALSE]
If all edge capacities in a flow network are the same constant integer, then a maximum flow can be found in linear time using the Ford-Fulkerson algorithm.

- Explanation: Ford Fulkerson will still take O(mn)

[TRUE/FALSE]
The Sequence Alignment problem discussed at length in lecture (between two strings of size *m* and *n*) can be solved given only *O(k)* memory space where *k=min(m,n)*

- Explanation: We can always split the larger string in the middle and hold columns corresponding to the size of the smaller string

[TRUE/FALSE]
The time complexity of the space-efficient version of the sequence alignment algorithm is *O(m + n)* (between two strings of size *m* and *n*)

- Explanation: complexity remains as *O(mn)*

[TRUE/FALSE]
A flow network with unique edge capacities may have several min cuts.

- Explanation: Consider the flow network $G = (V,E)$, with $V = \{s, a, b, t\}$, and $E = \{(s,a,1), (b,s,2), (a,t,3), (t,b,4)\}$. Then there are two min-cuts, $(\{s\}, \{a,b,t\})$ and $(\{s,b\}, \{a,t\})$, each with value 1.

2) 6 pts

a) Consider the following 0-1 knapsack problem. Given a knapsack capacity W=40, what is the maximum value that can be obtained for the following set of items? You do not need to show your work. (3 pts)

$w_0 = 10$	$w_1 = 20$	$w_2 = 30$
$v_0 = 40$	$v_1 = 60$	$v_2 = 50$

(a) 60
(b) 90
(c) 100
(d) 110
(e) 150

- Explanation: possible combinations of items with weight<=40 [no item, (0) with value 40, (1) with value 60, (2) with value 50, (0,1) with value 100, (0,2) with value 90, (1,2) exceeds weight].

b) If each edge capacity of a flow network G is multiplied by the same positive integer >1, then which of the following will change? (Assume that G has a non-zero max flow) (3 pts)

A. The location of min cut, or locations of min cuts if there is more than one
B. The capacity of any min-cut
C. The maximum value of flow from source to sink
D. A and B
E. A and C
F. B and C
G. A and B and C

3) 20 pts

A company has *n* software applications. Each application *i* has *F(i)* unique features (or functionalities), i.e. features are not shared across different applications. Each feature requires one DB connection from a given subset of databases *D(j)*. For example: An application *i* with *F(i)=3* features could have these database requirements: {{1, 3}, {2, 4, 5}, {6}}. This means that the first feature requires access to either database 1 or 3, the second feature requires access to databases 2, or 4, or 5, and the third feature requires access to database 6. Note that a database may be required for more than one feature.

Assuming that each database *k* can only accommodate *C(k)* connections at a given time, design a network flow solution to determine if there is an assignment of features to databases in which each application will at most have one feature without a DB connection. In other words, each application *i* should have at least *F(i)-1* of its features up and running.

a) Describe the complete construction of your network. (12 pts)

Solution 1 **Constructing a flow network:**

[+1] Create source node S and sink node T
[+1] Create *n* nodes representing the n applications
[+2] Connect S to each node representing application *i* with capacity *F(i)-1*
[+1] Create *F(i)* nodes for each application *i* representing its feature set
[+2] Connect the node representing application *i* to each of its feature nodes with capacity of 1
[+1] Create a node for each database (superset of DB's used by all features)
[+2] Connect each feature node to the subset of the DB's it can use with capacity 1
[+2] Connect an edge from each database *k* to *T* with capacity *C(k)*

Solution 2 **Constructing a circulation:**

[+1] Create source node S and sink node T. Create a node for each database (superset of DB's used by all features)
[+1] Create *n* nodes representing the n applications. Create *F(i)* nodes for each application *i* representing its feature set
[+2] Connect S to each node representing application *i* with capacity *F(i)-1*
[+2] Set the demand of source node S to $-\sum(F(i) - 1)$ and the demand of sink node T to $\sum(F(i) - 1)$ (otherwise, connect the sink to source node with unlimited capacity.)
[+2] Connect the node representing application *i* to each of its feature nodes with capacity of 1
[+2] Connect each feature node to the subset of the DB's it can use with capacity 1
[+2] Connect an edge from each database *k* to *T* with capacity *C(k)*

b) Which problem will you solve in this network and what algorithm will you use to solve it? (4 pts)

[+2] Find Max flow (If the solution for (a) is Max Flow) / Find the feasible circulation (If the solution for (a) is Circulation)
[+2] Scaled version of Ford Fulkerson or Edmonds Karp or Orlin or KTR

c) Describe how the solution to your network flow problem can be used to determine whether each application can have at least all but one of its features up and running. No proof is necessary. (4 pts)

Solution 1 **Constructing a flow network:**
[+1] If value of max flow = ($\sum$ *F(i) for all i)-n* **([+3])**, we have a solution, i.e., we can have all but one features in all applications up and running (If the solution for (a) is Max Flow)

Solution 2 **Constructing a circulation:**
[+4] iff a feasible circulation can be found (If the solution for (a) is Circulation)

4) 16 pts

You are given an unsorted array $A[1..n]$ of positive integers. You want to maximize the number of points you get by performing the following operations.
Pick any $A[i]$ and delete it to earn $A[i]$ points. Afterwards, you must delete elements $A[i-1]$ and $A[i+1]$ (if they exist) for which we won't get any points. This process will be called one move. Develop an efficient dynamic programming solution to return the maximum number of points you can earn with exactly k moves, where $k<n$.

Example: $A=[1,8,2]$, $k=1$ Solution: 8
Example: $A=[1,8,2]$, $k=2$ Solution: 3 (if we pick 8 we won't be able to make 2 moves because all elements will be deleted with that first move)

Note: if the array is of size 1 or 2 we cannot make more than one move. If the array is of size 3 or 4 we cannot make more than 2 moves, etc.

I. Define (in plain English) the subproblems to be solved. (3 pts)

OPT(i,j) = max points than can be earned for the subarray A[1..i] with exactly j moves

Similar answers: OPT(i,j) = max points than can be earned for the subarray A[i..n] with exactly j moves

II. Write a recurrence relation for the subproblems (5 pts)

OPT(i,j) = Max(OPT(i-1,j), A[i]+OPT(i-2,j-1))

Similar answers: OPT(i,j) = Max(OPT(i+1,j), A[i]+OPT(i+2,j-1)

III. Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum number of points that can be earned. (6 pts total)
Make sure you specify:

i. the base cases and their values (2 pts)

- Initialize $OPT(i,0) = 0$ for all integers $i \geq 0$, i.e., the maximum score achievable with exactly zero moves is always zero.
- Initialize $OPT(0,j) = -\infty$ for all integers $j \geq 1$, i.e., it is infeasible to make $j \geq 1$ moves given an array A with length 0.
- Initialize $OPT(1,1) = A[1]$.
- Initialize $OPT(1,j) = -\infty$ for all integers $j \geq 2$, i.e., it is infeasible to make $j \geq 2$ moves given an array A with length 1.

ii. where the final answer can be found (e.g. *OPT(n)*, or *OPT(0,n)*, etc.) (1 pt)

Answer is found at OPT(n,k)

iii. Rest of pseudocode (3 pt)

Pseudocode

```
Initialization given above
for i=2 to n
        for j=1 to k

        OPT(i,j) = Max(OPT(i-1,j), A[i]+OPT(i-2,j-1))
        endfor
endfor

Return OPT(n,k)
```

IV. What is the run time complexity of your solution? (2 pts)

O(kn) or *O(n^2)* both OK

Answer 2:

1. Define (in plain English) the subproblems to be solved. (3 pts)

    OPT(i,j,h) = max points that can be earned for the subarray A[i..j] with exactly h moves

2. Write a recurrence relation for the subproblems (5 pts)

    OPT(i,j,k) = max{ OPT(i,u-2,v) + OPT(u+2,j,h-1-v) + A[u]}
    where i<=u<=j, 0<=v<=h-1.

3. Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum number of points that can be earned. (6 pts total)
Make sure you specify:
    a. the base cases and their values (2 pts)

    Initialize OPT(i,j,w) = 0, for any $-1 \le i \le n+2$, $-1 \le j \le n+2$, $0 \le w \le k$.
    Initialize OPT(i,j,w) = -Inf for any $i > j$ and $0 < w < k$.
    (Explanation: no operations can be done for these intervals.)

    b. where the final answer can be found (e.g. *OPT(n)*, or *OPT(0,n)*, etc.) (1 pt)

    OPT(1,n,k)

    c. Rest of pseudocode (3 pt)

```
For h = 1 to k do
   For v = 0 to h - 1 do
      For i = 1 to n do
         For j = i to n do
            For u = i to j do
               OPT(i,j,h) = max{OPT(i,j,h),
                                  OPT(i,u-2,v) + OPT(u+2,j,k-1-v) + A[u]}
```

4. What is the run time complexity of your solution? (2 pts)

$O(n^3k^2)$

Answer 3:

1. Define (in plain English) the subproblems to be solved. (3 pts)

OPT(i,h) = max points that can be earned for the subarray A[1..i] with exactly h moves where the last operation is on element i.

2. Write a recurrence relation for the subproblems (5 pts)

OPT(i,h) = max{ OPT(j,h-1) + A[i] } for $1 \le j \le i-2$

3. Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum number of points that can be earned. (6 pts total)
   Make sure you specify:
   a. the base cases and their values (2 pts)

      Initialize OPT(i,h) = 0, for any i and h
      Initialize OPT(1,1) = A[1], OPT(2,1) = max(A[2],A[1])

   b. where the final answer can be found (e.g. *OPT(n)*, or *OPT(0,n)*, etc.) (1 pt)

      max{ OPT(i,k) } for 1<=i<=n

   c. Rest of pseudocode (3 pt)

```
Initialize as mentioned.
For h = 1 to k do
    For i =3 to n do
        For j = 1 to i - 2 do
            OPT(i,h) = max{OPT(i,h), OPT(j,h-1) + A[i]}
return max{OPT(i,k): 1<=i<=n}.
```

4. What is the run time complexity of your solution? (2 pts)

$O(n^2k)$

Wrong Case Explanation: why the following dp algorithms are not correct. Note that, it means **zero score** for the answers similar to the following cases.

1. OPT(i) = max points that can be earned for the subarray A[i].
   a. Reason: not considering the constraint that “exact k moves”.
2. OPT(i,j) = max points that can be earned for the subarray A[i..j].
   a. Reason: not considering the constraint that “exact k moves”.

Rubrics (In general):

1. 3 pts total
   - (-3pts) If the definition of subproblem is not correct, that is not satisfying the optimal-substructure property.

    - (-1pts) If the definition of subproblem has some extra dimensions, or it is not clearly defined.
    - (-0pts) Correct Answer.
2. 5 pts total
    - (-5pts) If the previous question is not answered, or, the recurrence relation is not correct, or, no answer.
    - (-2pts) If the coefficient/loop condition is not correct, but the formula (and the definition of subproblems) is almost correct.
    - (-0pts) Correct Answer.
3. 6 pts total

   (1). 2 pts
    - (-2pts) If the previous questions are not answered correctly (that is, 0 pts for both subquestion 1 and 2), or no answer.
    - (-1pts) If one pair of the base cases and their values is incorrect.
    - (-0pts) Correct Answer for all situations. Note that, since the problem didn't mention the correct output when k>(n+1)/2, the -Inf initializations

   (2). 1 pts
    - (-1pts) If the previous questions are not answered correctly (that is, 0 pts for both subquestion 1 and 2), or no answer, or answer is not correct (e.g.,max OPT[i][k] for k in 1..n is considered as a wrong answer).
    - (-0pts) Correct Answer.

   (3). 3 pts
    - (-3pts) If the previous questions are not answered correctly (that is, 0 pts for both subquestion 1 and 2), or no answer, or answer is not correct.
    - (-1pts) Some implementation details of the algorithm are not correct, with the main algorithm and the recurrence formula being correct. Note that, the wrong coefficients such as OPT(i-3,k) is regarded as this case.
    - (-0pts) Correct Answer.

4. 2 pts total
    - (-2pts) No answer for the previous question, or incorrect answer (such as O(n)).
    - (-0pts) Correct Answer.

5) 16 pts

A string is called *palindromic* if it is equal to its reverse, e.g., the string *racecar* is palindromic. Design a dynamic programming algorithm for finding the length of the longest palindromic subsequence of a given input string *S* of length *n*.

Example:

S='bbbab', output=4 (the longest palindromic subsequence is bbbb)

S ='abbcab', output=4 (the longest palindromic subsequence is abba)

I. Define (in plain English) subproblems to be solved. (3 pts)

OPT(i,j) = the length of the longest palindromic subsequence of S[i,...,j]

II. Write a recurrence relation for the subproblems (5 pts)

- If S[i]==S[j] then OPT(i,j)=OPT(i+1,j-1)+2 (If items at the ends are the same, the length of the the longest palindromic subsequence is added by 2 and then we need to check the smaller subsequence starting from i+1 and ending in j-1)

- If S[i]!=S[j] then OPT(i,j)=max(OPT(i, j-1), OPT(i+1,j)) (If items at the ends are not the same, then we will check both subsets one starting from i+1 and ending in j and the other one starting from i and ending in j-1)

So we end up with this recurrence formula:

```
OPT(i,j) = {          OPT(i,j)=OPT(i+1,j-1)+2       if S[i]==S[j]
                      Max(OPT(i, j-1), OPT(i+1,j))  otherwise     }
```

III. Using the recurrence formula in part b, write pseudocode using iteration to compute the length of the longest palindromic subsequence. (6 pts total)

Make sure you specify:

i. the base cases and their values (2 pts)

Initialize the diagonal terms: OPT(i,i) = 1 for i=1,n

Also need to initialize terms right below the diagonal to zero:

OPT(i,i-1) = 0 for i=2,n

ii. where the final answer can be found (e.g. *OPT(n)*, or *OPT(0,n)*, etc.) (1 pt)

Final answer: OPT(1,n)

iii. Rest of pseudocode (3 pt)

Pseudocode (Similar to matrix chain multiplication in lecture 8)

```
Initialization given above
for j=2 to n
        for i=j-1 to 1 by -1

                OPT(i,j) = {OPT(i,j)=OPT(i+1,j-1)+2 if S[i]==S[j]
                            Max(OPT(i, j-1), OPT(i+1,j))  otherwise}
        endfor
endfor

Return OPT(1,n)
```

IV. What is the complexity of your solution? (2 pts)

$O(n^2)$

Rubrics (In general):

1. 3 pts total
   - (-3pts) If the definition of subproblem is not correct, that is not satisfying the optimal-substructure property.
   - (-1pts) If the definition of subproblem has some extra dimensions.
   - (-0pts) Correct Answer.
2. 5 pts total
   - (-5pts) If the previous question is not answered, or, the recurrence relation is not correct, or, no answer.
   - (-2pts) If the coefficient/loop condition is not correct, but the formula (and the definition of subproblems) is almost correct.
   - (-0pts) Correct Answer.
3. 6 pts total
   1) 2 pts
      - ❖ (-2pts) If the previous questions are not answered correctly (that is, 0 pts for both subquestion 1 and 2), or no answer.
      - ❖ (-1pts) If one pair of the base cases and their values is incorrect.
      - ❖ (-0pts) Correct Answer for all situations. Note that, if they transform this problem into the longest common subsequence, the initial should be OPT[i,0]=OPT[0, j]=0 or other reasonable forms.
   2) 3 pts

- ❖ (-3pts) If the previous questions are not answered correctly (that is, 0 pts for both subquestion 1 and 2), or no answer, or answer is not correct.
- ❖ (-1pts) Some implementation details of the algorithm are not correct, with the main algorithm and the recurrence formula being correct. Note that, the wrong coefficient is regarded as this case.
- ❖ (-0pts) Correct Answer.

3) 1 pts
   - ❖ (-1pts) If the previous questions are not answered correctly (that is, 0 pts for both subquestion 1 and 2), or no answer, or answer is not correct.
   - ❖ (-0pts) Correct Answer.

4. 2 pts total
   - (-2pts) No answer for the previous question, or incorrect answer (such as O(n)).
   - (-0pts) Correct Answer.

Alternative 1: longest common sequence (LCS)
OPT(i, j) = the longest common sequence (LCS) of S[1..i] and its reverse string S'[1…j]; or in any other reasonable forms.

init: OPT[0, j]=OPT[i, 0]=0

```
        OPT(i,j) = {          OPT(i,j)=OPT(i-1,j-1)+1        if S[i]==S'[j]
                              Max(OPT(i, j-1), OPT(i-1,j))   otherwise     }
```

in this case, final answer should be OPT[n, n]
code:

```
int n = s.length(); string t = s; reverse(t.begin(),t.end());
int OPT[n+1][n+1];
for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
                if (i == 0 || j == 0)
                        OPT[i][j] = 0;
                else if (s[i-1] == t[j-1])
                        OPT[i][j] = OPT[i-1][j-1] + 1;
                else
                        OPT[i][j] = max(OPT[i][j-1], OPT[i-1][j]);
return table[n][n];
```

Alternative 2: utilize LCS distance (a type of edit distance LCSD)
LCS distance allows deletion and insertion but not substitution.
LCSD (S1, S2) = the minimum operations/cost of transforming S1 to S2
thus, LCSD = deletion+insertion = (n - LCS) + (m - LCS) (in this problem, n=m)

LCS = n - LCSD/2

OPT(i, j) = the min operations/cost/LCSD of transforming S[1…i] to its reverse string S’[1…j] that allows deletion and insertion; or other forms.

init: OPT[0, j]=j, OPT[i, 0]=i

```
        OPT(i,j) = {            OPT(i,j)=OPT(i-1,j-1)         if S[i]==S'[j]
                                Min(OPT(i, j-1), OPT(i-1,j))+1  otherwise    }
```

in this case, final answer should be (n - OPT[n, n]/2)

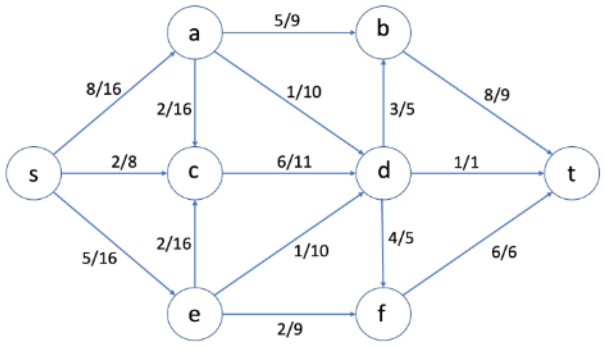
code:

```
int n = s.length(); string t = s; reverse(t.begin(),t.end());
int OPT[n+1][n+1];
for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
                if (i == 0)
                        OPT[i][j] = j;
                else if (j == 0)
                        OPT[i][j] = i;
                else if (s[i-1] == t[j-1])
                        OPT[i][j] = OPT[i-1][j-1];
                else
                        OPT[i][j] = min(OPT[i][j-1], OPT[i-1][j]) + 1;
return n - OPT[n][n]/2;
```

5) 22 pts

Consider the flow-network below, for which an *s*-*t* flow has been computed. The numbers *x/y* on each edge shows that the capacity of the edge is equal to *y*, and the flow sent on the edge is equal to *x*.

[redacted]



[redacted]

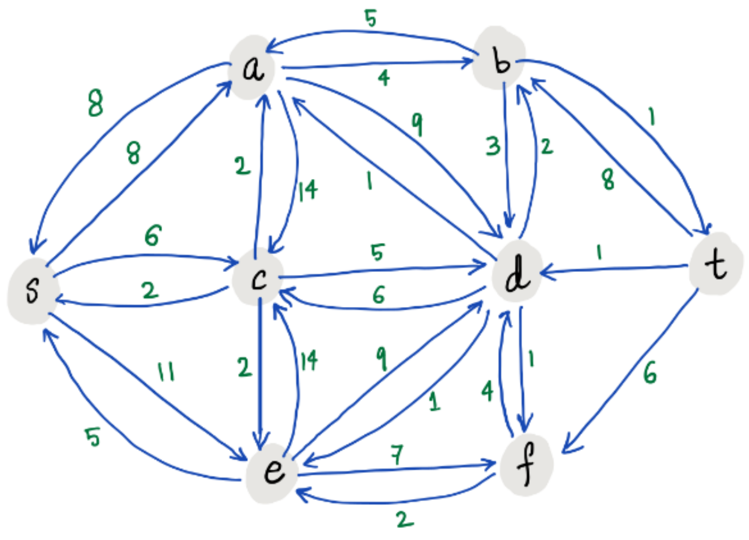
a. What is the current value of the flow? (2 pts)

Answer = 8+2+5 = 8+1+6 = 15

+2 points for correct answer (no partial marking)

b. Draw the corresponding residual graph. (6 pts)

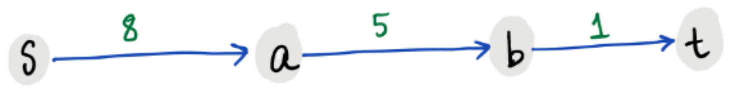
Answer:

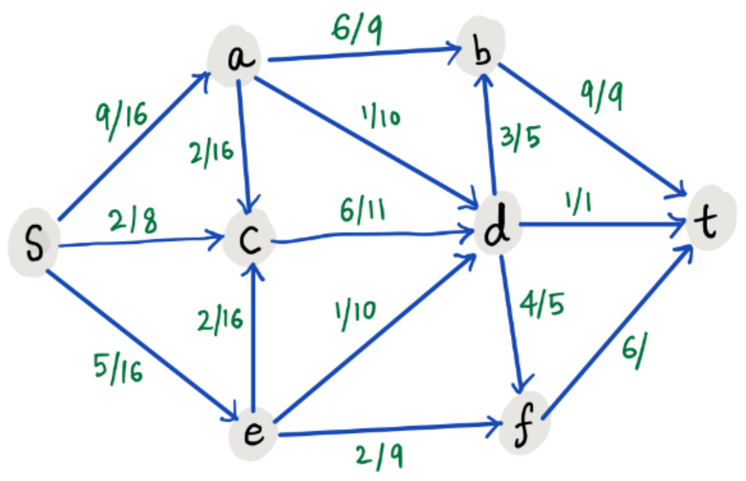


+2 points for correctly including all the nodes
    -1 for missing some nodes
+2 points for including the correct edges (backward and forward)
    -1 for missing some edges
+2 points for correctly writing the residual capacities
    -1 for wrongly writing some residual capacities

c. Starting from the given flow, perform as many iterations of the Ford-Fulkerson algorithm as required to find a max flow. Draw the augmenting path at each step and draw the final max flow (6 pts)

Answer:





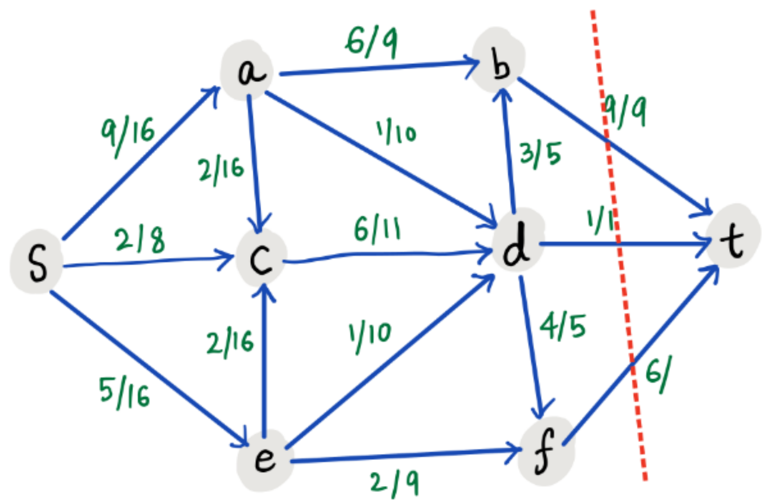
The augmenting path is s-a-b-t (there can be others)
We augment 1 unit of flow through this path

After this there will be no s-t path with positive capacity on residual graph
Therefore, value of max flow = 15+1 = 16

+3 points for identifying a correct augmenting path and augmenting 1 unit
    -1 if they perform multiple augmentations or don't start from current flow
    -2 for wrong augmenting value
    -2 point for partial correct path
    -3 if they don't draw/write the augmenting path anywhere
+3 points for correct graph
    -1 for wrong flow values in some of the edges
    -2 points if they draw the residual graph instead of the flow network
    -3 if they draw a completely unrelated/wrong graph, or don't draw a graph at all

d. Use a drawing to show a min cut in the above flow network (2 pts)

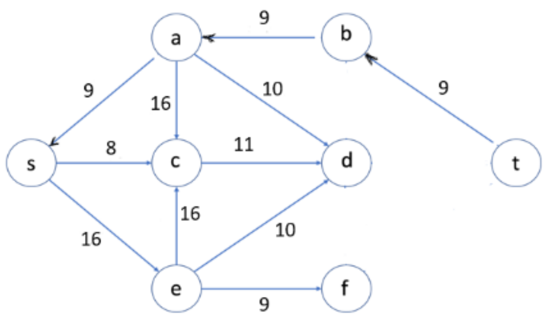
Answer:



2 points for the correct cut

e. Assuming that there is no flow currently going through the network (i.e., zero flow going through each edge), use the scaled version of Ford Fulkerson to complete one augmentation step and draw how the network $G_f(\Delta)$ appears right after the first augmentation step. (6 pts)

No augmentation is possible for $\Delta = 16$
First augmentation step will be with $\Delta = 8$
Augmentation path: SABT, bottleneck value=9
$G_f(\Delta)$ after first augmentation step:



2 points for correct SABT path at $\Delta = 8$ and flow=9

- -2 point for wrong augmenting value or wrong path
- -1 point for wrong delta or minor errors

4 points for the correct Gf($\Delta = 8$)

- -2 point for including an extra edge, missing an edge, minor errors, etc.
- -4 points for wrong graph

Additional Space

Additional Space

Additional Space