CS 3510: Design and Analysis of Algorithms	Georgia Tech
Final Exam	
Instructor: Shahrokh Shahi	*Summer 2022*

GT Username:	**Full Name:**

Instructions:

1. Write your **name** and **GT username** on each page very clearly. Then, complete the exam.
2. This exam is closed-book, and collaboration is NOT permitted.
3. You are allowed to use **one sheet of notes**, i.e., both sides of a letter-sized paper, during the exam.
4. No calculator is required.
5. You have **120 minutes** to complete this exam.
6. It is recommended to read all the questions before starting. Please read the questions carefully. Misunderstanding the question is not a valid excuse for losing points.
7. If you find it necessary, make reasonable assumptions but make sure to state them clearly.
8. You can use the back of each sheet as scratch paper.
9. Write your solution in the space provided. In case you need more space, you can use back of the same sheet, and make a notation on the front of the sheet.
10. The exam has **80 + 4** points in total.

Good luck!

Number	Problem	Points	Grade
1	Short Answer Questions	30 pts	
2	Divide-and-Conquer	10 pts	
3	Dynamic Programming	10 pts	
4	Shortest Path Problem	10 pts	
5	Shortest Path Problem	10 pts	
6	Flow Network	10 pts	

GT Username:	Full Name:

1 Short answer questions [30 pts]

1.1 Asymptotic Notations and Master Theorem [6 pts]

(a) (2 pts) Rank the following functions by increasing order of asymptotic growth, that is find an arrangement $f_1, f_2, f_3, \ldots$ such that $f_1 \in O(f_2), f_2 \in O(f_3), \ldots$. Explanation/justification is NOT required.

$$n^{10000001}, \quad \sqrt{n}, \quad n.2^n, \quad 1024, \quad n^n, \quad \log n, \quad 2^n, \quad n^3 + \log n^2, \quad n, \quad \left(\frac{3}{2}\right)^n$$

$1024, \quad \log n, \quad \sqrt{n}, \quad n, \quad n^3 + \log n^2, \quad n^{10000001}, \quad \left(\frac{3}{2}\right)^n, \quad 2^n, \quad n.2^n, \quad n^n$

Grading: each error -1/2, zero credit for more than 4 mistakes.

(b) (4 pts) For the following divide-and-conquer programs, give the recurrence relation describing their running time and apply the Master Theorem to calculate the running time. Grading: recurrence 1 pt, running time 1 pt

```
def func(n):
    if n==0: stop

    func(n/3)

    func(n/3)

    do_something in O(1)
```

Recurrence relation: $T(n) = 2T(n/3) + O(1)$
Using master theorem: $a = 2, b = 3, d = 0$, and $a > b^d$. Therefore, $T(n) = n^{\log_b a} = n^{\log_3 2}$

```
def func(n):
    if n==0: stop

    func(n/4)
    do_something in O(1)

    func(n/4)
    do_something in O(n)

    func(n/4)
    do_something in O(n^2)
```

Recurrence relation: $T(n) = 3T(n/4) + O(n^2)$
Using master theorem: $a = 3, b = 4, d = 2$, and $a < b^d$. Therefore, $T(n) = n^d = n^2$

GT Username:	**Full Name:**

1.2 Algorithm Paradigms [5 pts]

Complete the following table by writing the used design paradigm (e.g., divide-and-conquer, dynamic programming, etc.) and the application of each of the following algorithms discussed in class.

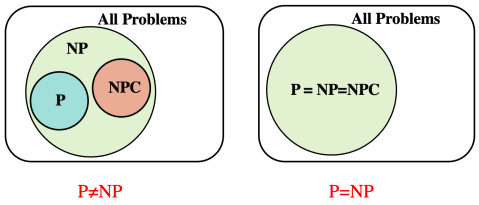
Algorithm	Design paradigm	Application
Kruskal	Greedy	MST
Dijkstra	Greedy	Shortest path
Prim	Greedy	MST
Bellman-Ford	Dynamic Programming	Shortest path
Floyd-Warshall	Dynamic Programming	Shortest path (all pairs)

Grading: for Floyd-Warshall, only "Shortest path" is accepted.

1.3 NP Completeness [10 pts]

(a) (4 pts) Briefly describe the two possibilities for the relationships among complexity classes P, NP, and NP-complete. Also, show these two possibilities using Venn diagram (in terms of set inclusion).

(1) P = NP = NPC, (2) P $\subseteq$ NP but NP $\not\subseteq$ P, so P $\neq$ NP



Grading: note there is also another version of this figures in Lecture 16 / Slide #48

(b) (6 pts) Suppose there is a polynomial-time reduction from problem A to problem B ($A \leq_p B$). Specify whether the following statements are **True** or **False**.

#	**Statement**	**True / False**
1	Problem B is NP-hard.	F
2	Polynomial-time algorithm for solving B can be used to solve A in polynomial time.	T
3	If B has no polynomial-time algorithm then neither does A.	F
4	If A is NP-hard and B has a polynomial-time algorithm then P=NP.	T
5	If B is NP-hard then A is NP-hard.	F
6	If B reduces to C then A reduces to C.	T

GT Username:	Full Name:

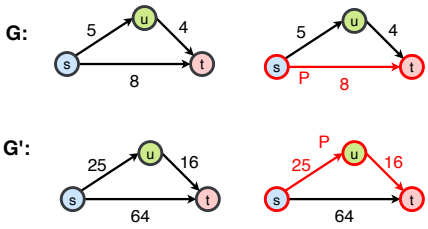
1.4 General True/False Questions [10 pts]

For each of the following statements, decide whether it is **True** or **False**. If it is true, provide a short explanation and if it is false, give a counterexample. Grading: each true/false 1 pt, reasoning/counterexample 1 pt

1. An input array that gives the best running time in the `Insertion-sort` algorithm can give the worst running time in the `Quick-sort` algorithm.
   True; a sorted array gives the best running time ($O(n)$) in the `Insertion-sort` but the worst running time $O(n^2)$ in the `Quick-sort` algorithm.

2. Let $G = (V, E)$ be a weighted graph and let T be a minimum spanning tree of G. The path in T between any pair of vertices u and v must be a shortest path in G.
   False. Counterexample: $V = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, a)\}$, where $w(a, b) = 3$, $w(b, c) = 3$, and $w(c, a) = 4$. Clearly, $T = \{(a, b), (b, c)\}$. But the shortest path between c and a is the edge (c, a) of weight 4 and not the path from the MST which has a total weight of $3 + 3 = 6$.

3. Given a directed and weighted graph $G = (V, E)$, let P be a minimum weight (shortest distance) $s - t$ path between two given s and t nodes. Assume we replace each edge weight, w_e by its square w_e^2 for all $e \in E$, thereby creating a new graph G' with the same vertices but different edge weights. Then, P must still be a minimum weight (shortest) path between s and t for the new graph G'.
   False. As the counter example consider the following graphs G and G':



4. The maximum spanning tree (spanning tree of maximum weight) can be computed by negating the cost of all the edges in the graph and then computing minimum spanning tree.
   True; this works, and none of the algorithms presented for finding an MST depended on edge weights being non-negative.

5. The heaviest edge in a graph cannot belong to the minimum spanning tree.
   False. This edge may be connecting two otherwise-disconnected subgraphs. Any counterexample that can show that is accepted. (Note this is different from the case that the heaviest graph is on a cycle in the given graph (Exam 2 – problem 4).)

GT Username:	Full Name:

2 Divide-and-Conquer [10 pts]

Georgiana has two sorted arrays $A = [v_1, \ldots, v_n]$ and $B = [v_1, ..., x, ..., v_n]$, where $0 < v_1 < \ldots < v_n$ and the second array, B, is obtained by inserting an integer number x into the first array, A, where $x \neq v_i$ for all $1 \leq i \leq n$. Therefore, A has n elements and B has $n+1$ elements. Note that x can be inserted at any position i, $1 \leq i \leq n$, into the first array. She is tasked with finding the index of the inserted element x in the second array B. Let's help Georgiana design an efficient algorithm to solve this problem.

Example 1: index of the inserted number = 4

$$A = [2, 3, 6, 9, 10, 18, 20, 23]$$
$$B = [2, 3, 6, 100, 9, 10, 18, 20, 23]$$

Example 2: index of the inserted number = 1

$$A = [2, 3, 6, 9, 10, 18, 20, 23]$$
$$B = [47, 2, 3, 6, 9, 10, 18, 20, 23]$$

Example 3: index of the inserted number = 9

$$A = [2, 3, 6, 9, 10, 18, 20, 23]$$
$$B = [2, 3, 6, 9, 10, 18, 20, 23, -4]$$

Design an efficient algorithm to find the index of the inserted number x in B. The running time of your algorithm should not be larger than $O(\log n)$. You can assume that the elements of the arrays $(v_1, \ldots, v_n)$ are distinct, and x also is not equal to any of them.

(a) (5 pts) Explain your algorithm in words.

Key observation: For any i $(1 \leq i \leq n)$, if $A[i] == B[i]$, then we can be sure that the index of the inserted number, say k, is greater than i $(k > i)$, i.e., k is in the right side of i. Otherwise, $k \leq i$. We can use this observation to construct a binary search algorithm and at each step compare $A[mid]$ and $B[mid]$, where $mid = (lo + hi)/2$. Therefore,

```
# Python implementation
def find_difference(A,B):
    n = len(A)
    l, r = 0, n
    while l < r:
        m = (l+r)//2
        if B[m]==A[m]:
            l = m+1
        else:
            r=m
    return l
```

GT Username:	**Full Name:**

(b) (5 pts) Provide the pseudocode describing your algorithm.

Algorithm 1: `FindIndex`

Input: $A = \{a_1, a_2, ..., a_n\}$ and $B = \{a_1, a_2, ..x..., a_n\}$
Result: the index of the inserted zero

$lo \leftarrow 0$
$hi \leftarrow n$

while *(*$lo < hi$*)* **do**
  $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
  if *(B[i]==A[i])* **then**
    $lo \leftarrow m + 1$
  else
    $hi \leftarrow m$
  end
end

return l

Note: Other implementations, e.g., recursive approach is also accepted.

GT Username:	Full Name:

3 Dynamic Programming [10 pts]

The `Coin-changing` algorithm discussed in class lectures provides the minimum number of coins required to make change for S cents with an infinite supply of coins whose values are represented by an array, i.e., `Coins` = $\{v_1, v_2, \ldots, v_n\}$. Modify this dynamic programming algorithm to return the total number of combinations of coins that can make up the total amount of S cents. If that amount of money cannot be obtained by any combination of the given coins, then the algorithm should return 0.

Example 1: Let the total amount $S = 5$ and `Coins` = $\{1, 2, 5\}$, `return 4`
The combinations that we can make up the total amount of 5:
(1) $1 + 1 + 1 + 1 + 1$,
(2) $1 + 1 + 1 + 2$,
(3) $1 + 2 + 2$,
(4) 5

Example 2: Let the total amount $S = 5$ and `Coins` = $\{2\}$, `return 0`
Because we cannot make up 5 cents using any number of the coin of 2.

Design the `Coin-changing-combinations` algorithm to return the total number of combinations of coins that can make up the total amount of S cents.

(a) (5 pts) Discuss the optimal substructure of the `Coin-changing-combinations` problem, and give the recurrence relation including the base case(s).

As mentioned in class, this problem can be solved with a 2D dynamic programming approach but it can also be reduced to a 1D case. Here we describe the latter, but a 2D solution is also accepted for this problem.

1D approach:
Let $OPT[T]$ denotes the number of combinations of coins to make up T cents, where $T \leq S$, and our goal is finding $OPT[S]$. Clearly, the problem has optimal substructure, where the solution to the problem can be constructed using the solutions of the subproblems. The base case is when the given value is zero (we have no money so we need no coin for change). This will be just one combination, i.e., using no coin. Therefore, $OPT[0] = 1$. Also, another base case is when we have no coin, i.e., `Coins` = $\{\}$. In this case, for any $0 < T \leq S$, we will have zero combination that can make up T cents; thus, $OPT[T] = 0$, for $0 < T \leq S$. The optimal substructure can be observed when we pick up, for instance the i-th coin with the value of v_i, and use it to make up the amount T cent. In this case, the number of combinations is the sum of the following two choices:

1. Using coin i to make up the T cent. In this case, the number of combinations is equal to the number of combinations to make up $T - v_i$ cent, i.e., $OPT[T - v_i]$.
2. Not using coin i and make up the T cent with the previous $i - 1$ coins, where in this case, the number of combinations remain $OPT[T]$ obtained in previous step.

Therefore, the recurrence relation can be written as $OPT[T] = OPT[T] + OPT[T - v_i]$, and for each coin of value v_i, we can iterate over all amounts $v = \{v_i, v_i + 1, v_i + 2, \ldots, S\}$ and compute the number of combinations. (Base cases: $OPT[0] = 1$, and $OPT[T] = 0$, for $0 < T \leq S$.)

GT Username:	Full Name:

2D approach:

Note one could define $OPT[i, T]$ as the number of combinations of coins to make up T cents with the first i coins, and treat this problem as a 2D dynamic programming problem. In this case, the number of combinations is the sum of the following two choices:

1. Using coin i to make up the T cent. In this case, the number of combinations is equal to the number of combinations to make up $T - v_i$ cent, i.e., $OPT[i, T - v_i]$.
2. Not using coin i and make up the T cent with the previous $i - 1$ coins, where in this case, the number of combinations is $OPT[i - 1, T]$.

Therefore, $OPT[i, T] = OPT[i - 1, T] + OPT[i, T - v_i]$, and the base case can be written as $OPT[i, 0] = 1$ and $OPT[0, T] = 0$, where $0 < T \leq S$.

(b) (3 pts) Give the pseudocode of a bottom-up or top-down implementation of the dynamic programming algorithm using the recurrence relation from part (a).

Bottom-up implementation:

Algorithm 2:
`Coin-changing-combinations(1D)`

Input: S and `Coins` = $\{v_1, v_2, \ldots, v_n\}$
Result: the total number of combinations that we can build S using the coin values.

$opt \leftarrow [1, 0, 0, \ldots, 0]$

for $(v_i \in \{v_1, \ldots, v_n\})$ **do**
  for $(t \in \{v_i, \ldots, S\})$ **do**
    $opt[t] \leftarrow opt[t] + opt[t - v_i]$
  end
end

return $opt[S]$

Algorithm 3:
`Coin-changing-combinations(2D)`

Input: S and `Coins` = $\{v_1, v_2, \ldots, v_n\}$
Result: the total number of combinations that we can build S using the coin values.

$opt \leftarrow zeros(n + 1, S + 1)$
for $(i \in \{0, \ldots, n\})$ **do**
  $opt[i][0] \leftarrow 1$
end

for $(i \in \{1, \ldots, n\})$ **do**
  $v_i \leftarrow$ `Coins[i]` **for** $(t \in \{1, \ldots, S + 1\})$ **do**
    $opt[i][t] \leftarrow opt[i - 1, t]$
    if $t - vi \geq 0$ **then**
      $opt[i][t] \leftarrow opt[i][t] + opt[t - v_i]$
    end
  end
end

return $opt[n][S]$

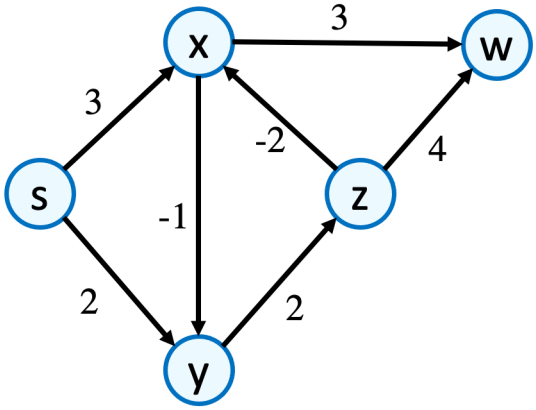
(c) (2 pts) Discuss the time and space complexities of your algorithm.

Time complexity $O(nS)$, Space complexity $O(S)$ for 1D implementation and $O(nS)$ with 2D implementation.

GT Username:	Full Name:

4 Shortest Path [10 pts]

Consider the following graph $G = (V, E)$, where $E = \{(s, x), (s, y), (x, y), (x, w), (y, z), (z, x), (z, w)\}$ and $V = \{s, x, y, z, w\}$, and answer the following questions.



We want to run the `Bellman-Ford` algorithm explained in class, from node s to find the shortest distance from this node to every other nodes in this graph. The following tables show the **shortest distance estimation array,** $d_s[v]$ and **the predecessors array,** $\pi[v]$, at the end of the first iteration ($i = 1$). Perform the next iteration of `Bellman-Ford` algorithm and give the values of these two arrays at the end of the second iteration ($i = 2$). Note for the relaxing procedure, visit edges in the EXACT order as they appeared in the edge set E presented above.

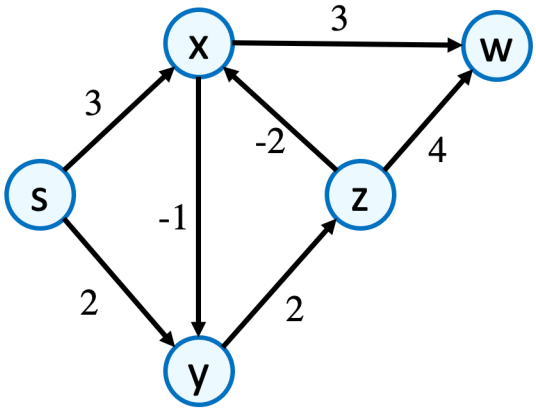
You may use the next page to show your work for partial credit.

v	$\pi[v]$ $(i = 1)$	$\pi[v]$ $(i = 2)$
s	ϕ	ϕ
x	z	z
y	s	x
z	y	y
w	x	x

i	$d[s]$	$d[x]$	$d[y]$	$d[z]$	$d[w]$
0	0	∞	∞	∞	∞
1	0	2	2	4	6
2	0	1	1	3	5

As a side note, this graph has one negative cycle which is reachable from the source node ($x \to y \to z \to x$). Therefore, after relaxing all edges for $|V| - 1$ times, the Bellman-Ford algorithm returns `False`. However, this does not affect the solution of this problem.

GT Username:	**Full Name:**



$V = \{s, x, y, z, w\}$,

$E = \{(s, x), (s, y), (x, y), (x, w), (y, z), (z, x), (z, w)\}$

GT Username:	Full Name:

5 Shortest Path [10 pts]

Let $G = (V, E)$ be a directed graph with positive edge . Let $t \in V$. Give an algorithm that runs in $O(|V|^2)$ time for finding shortest paths between all pairs of nodes, such that these paths pass through t. You can assume the graph G is represented using either adjacency matrix or adjacency list.

(a) (8 pts) Describe your algorithm in words, and justify the correctness of your approach. No pseudocode is required.

Let $\delta_t(u, v)$ denote the length of the shortest path from u to v passing through t. We must have that $\delta_t(u, v) = \delta(u, t) + \delta(t, v)$, where $\delta(u, t)$ denotes the shortest distance from u to t and $\delta(t, v)$ represents the shortest distance from t to v. Therefore, to determine $\delta_t(u, v)$ for all pairs of vertices, we only need to determine $\delta(u, t)$ for all $u \in V$ and $\delta(t, v)$ for all $v \in V$. This can be done by running `Dijkstra`'s algorithm twice on the given graph G and its reverse G^T.

More specifically, we can determine $\delta(t, v)$ for all $v \in V$ by running `Dijkstra`'s algorithm on graph G, where t is set as the source node. Then, we can reverse the graph (creating G^T), and run `Dijkstra`'s algorithm on the reverse graph G^T, with t as the source node. Clearly, a shortest path from t to u in the reverse graph G^T corresponds to a shortest path from node u to t in the original graph G. Therefore, the second execution of `Dijkstra`'s algorithm on the reverse graph G^T will give us $\delta(u, t)$ for all $u \in V$. Once we have both $\delta(u, t)$ for all $u \in V$ and $\delta(t, v)$ for all $v \in V$, we can compute the all pairs shortest distance $\delta_t(u, v)$ for all $u, v \in V$ by adding $\delta(u, t)$ and $\delta(t, v)$.

To find the shortest path from u to v passing through t, we can simply concatenate the shortest path from u to t and the shortest path from t to v, where both of which can be obtained within the `Dijkstra`'s algorithm execution using backtracing the predecessor array $\pi()$.

Algorithm 4: `AllPairsShortestPath`
(Note pseudocode is NOT required. It is just added for clarity.)

Input: $G = (V, E), L, t$, the given graph, edge length, the given vertex t
Result: D, a $|V| \times |V|$ matrix, where $D(u, v)$ indicates the length of shortest path from u to v passing through t

Run `Dijkstra`(G, L, t) to obtain the array SP of size $|V|$ including the length of shortest paths from t to all vertices $v \in V$

Run `Dijkstra`(G^T, L, t) to obtain the array SP^T of size $|V|$ including the length of shortest paths from all vertices $u \in V$ to the given vertex t

for $(u = 1$ **to** $|V|)$ **do**
  for $(v = 1$ **to** $|V|)$ **do**
    $D(u, v) \leftarrow SP^T(u) + SP(v)$
  end
end
return D

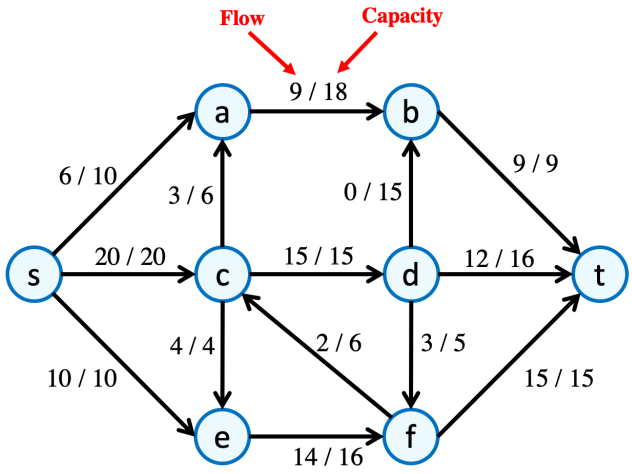
(b) (2 pts) Discuss the running time of your algorithm.

The running time of the explained algorithm is $O(|V|^2)$. We need to run the `Dijkstra`'s algorithm twice, where each run takes $O(|V|^2)$ with adjacency matrix representation (the priority queue can be an unordered array in this case), or with an adjacency list representation it takes $O(|E| \log |V|)$. Also, reversing the graph takes $O(|E| + |V|)$ with an adjacency list representation or $O(|V|^2)$ if we use the adjacency matrix representation. The adding operations for $\delta(u, t) + \delta(t, v)$ to create the all pairs shortest distance matrix take $O(|V|^2)$. Therefore, the overall running time is dominated by $O(|V|^2)$.

GT Username:	Full Name:

6 Flow Network: Ford-Fulkerson Algorithm (10 pts)

Consider the following st-flow network and the given feasible flow f.



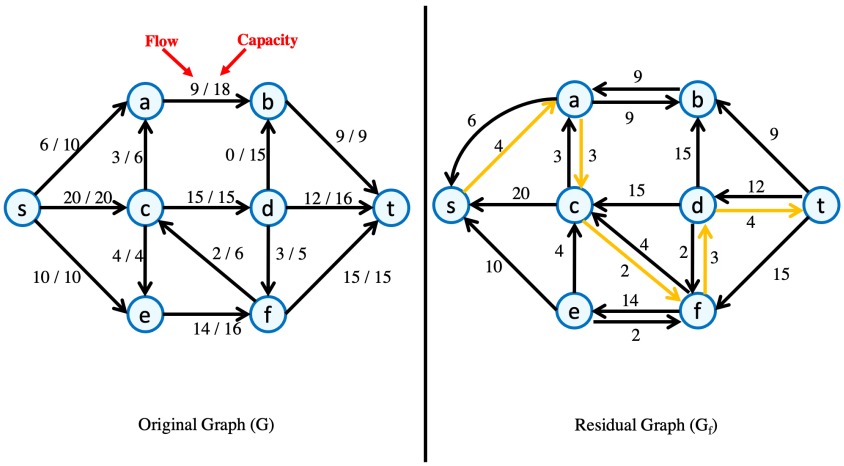
(a) (1 pts) What is the value of the current flow f?

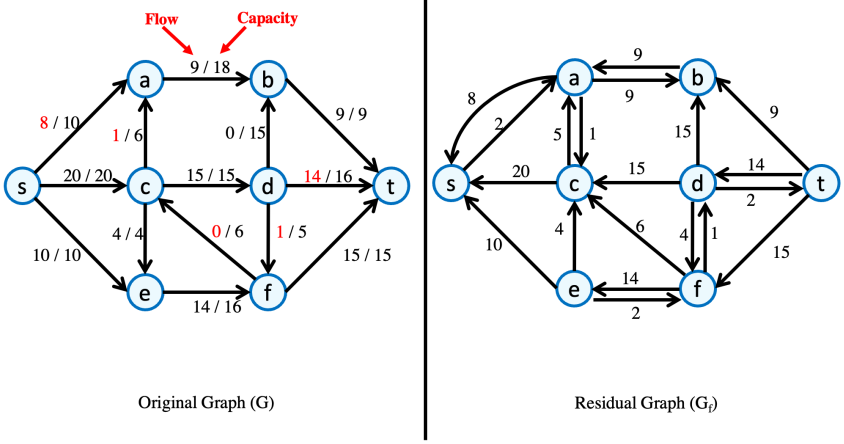
$6 + 20 + 10 = 36$

(b) (2 pts) What are the two constraints of a flow? Verify that f is a feasible flow in this network.

(1) (capacity) $0 \leq f(e) \leq c(e), \quad \forall e \in E$
(2) (flow conservation) $\sum_{e \to v} f(e) = \sum_{e \leftarrow v} f(e), \quad \forall v \in V - \{s, t\}$
These two conditions hold for all edges and intermediate nodes in the given graph.

(c) (3 pts) Perform one iteration of the Ford-Fulkerson algorithm, starting from the flow f. Give the sequence of vertices on an augmented path. Draw the residual graph, and show the path you chose.





(d) (2 pts) What is the value of the maximum flow? (Justify your answer using the final residual graph)

$val(f) = 9 + 14 + 15 = 38$

There is no augmenting path left, so the `Ford-Fulkerson` algorithm is terminated and according to the augmenting path theorem, the flow f is the max flow.

(e) (2 pts) List vertices on the s side of a minimum cut. (Hint: use your work from part c) What is the capacity of the minimum cut?

We can choose the nodes that are reachable from s in the final residual graph. Therefore, set $A = \{s, a, b, c\}$

Capacity $= c(s, e) + c(c, e) + c(c, d) + c(b, t) = 10 + 4 + 15 + 9 = 38$.