Name: ______________________________ Student ID: ________________________________

Question	Points	Score
1	10	
2	10	
3	10	
4	10	
5	10	
Total:	50	

INSTRUCTIONS: Be clear and concise. Write your answers in the space provided. Use the backs of pages, and/or the scratch page at the end, for your scratchwork. All graphs are assumed to be simple. Good luck!

You may freely use or cite the following subroutines from class[1]:

- *dfs*(*G*)
  This returns three arrays of size |V|: *pre*, *post*, and *cc*. If the graph has *k* connected components, then the *cc* array assigns each node a number in the range 1 to *k*.

- *bfs*(*G*, *s*)
  This returns two arrays of size |V|: *dist* and *prev*.

SOLUTIONS

[1]Recall from class/text the time complexities (1) *dfs*: O(|V| + |E|); (2) *bfs*: O(|V| + |E|).

(A VERSION) QUESTION 1.

(a) (*5 points*) Consider the following pseudocode:

```
function example(n):
     x = 0
     if n = 1:
          return
     endif

     for i = 1 to n:
          x = x + 1
     endfor

     return example(n/2)
```

State the recurrence relation for the running time $T(n)$ of the function `example(n)`. Solve the recurrence using the Master Theorem.

There is one loop that runs n times. The innermost "unit time" operation is thus performed for a total of n times. The problem is then divided into one subproblem of size $n/2$. The recurrence is thus

$$T(n) = T(n/2) + n$$

$$a = 1, b = 2, d = 1.$$

$$\log_b a = \log_2 1 < 1 = d$$

Using Master theorem we get

$$T(n) = O(n)$$

(b) (*5 points*) An algorithm solves problems of size n by recursively solving two subproblems of size $n-2$ and then combining the subproblems in constant time. Write the recurrence relation for the running time $T(n)$ of this algorithm. Give the running time in big-O notation. (Show your work.)

$$\begin{aligned} T(n) &= 2T(n-2) + 1 \\ &= 2(2T(n-4) + 1) + 1 \\ &= 4T(n-4) + 2 + 1 \\ &= 4(2T(n-6) + 1) + 2 + 1 \\ &= 8T(n-6) + 4 + 2 + 1 \end{aligned}$$

In general,

$$T(n) = 2^k T(n-2k) + 2^{k-1} + \ldots + 2 + 1$$

$$= 2^k T(n-2k) + 2^k - 1$$

Substituting $k = (n-1)/2$ gives

$$T(n) = 2^{(n-1)/2} T(0) + 2^{(n-1)/2} - 1$$

$T(0)$ is a constant amount of work. It follows that:

$$T(n) = O(2^{(n-1)/2}) = O(2^{n/2})$$

(B VERSION) QUESTION 1.

(a) (*5 points*) Consider the following pseudocode:

```
function example(n):
     x = 0
     if n = 1:
          return
     endif

     for i = 1 to n:
          for j = 1 to n:
               x = x + 1
     endfor

     return example(n/3)
```

State the recurrence relation for the running time $T(n)$ of the function `example(n)`. Solve the recurrence using the Master Theorem.

There are two nested loops that run for n times each. The innermost "unit time" operation is thus performed for a total of $n * n = n^2$ times. The problem is then divided into one subproblem of size n/3. The recurrence is thus

$$T(n) = T(n/3) + n^2$$

$$a = 1, b = 3, d = 2.$$

$$\log_b a = \log_3 1 < 1 = d$$

Using Master theorem we get

$$T(n) = O(n^2)$$

(b) (*5 points*) An algorithm solves problems of size n by recursively solving two subproblems of size $n - 3$ and then combining the subproblems in constant time. Write the recurrence relation for the running time $T(n)$ of this algorithm. Give the running time in big-O notation. (Show your work.)

$$T(n) = 2T(n-3) + 1$$

$$= 2(2T(n-6) + 1) + 1$$

$$= 4T(n-6) + 2 + 1$$

$$= 4(2T(n-9) + 1) + 2 + 1$$

$$= 8T(n-9) + 4 + 2 + 1$$

In general,

$$T(n) = 2^k T(n-3k) + 2^{k-1} + \ldots + 2 + 1$$

$$= 2^k T(n-3k) + 2^k - 1$$

Substituting $k = (n-1)/3$ gives

$$T(n) = 2^{(n-1)/3} T(1) + 2^{(n-1)/3} - 1$$

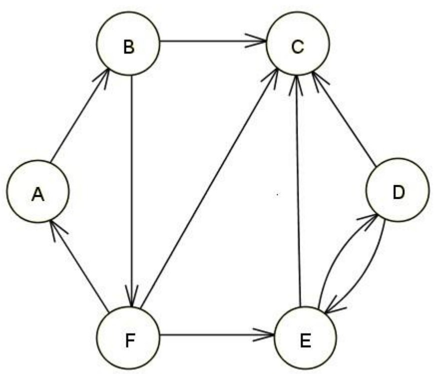
$T(1)$ is a constant amount of work. It follows that:

$$T(n) = O(2^{(n-1)/3}) = O(2^{n/3})$$

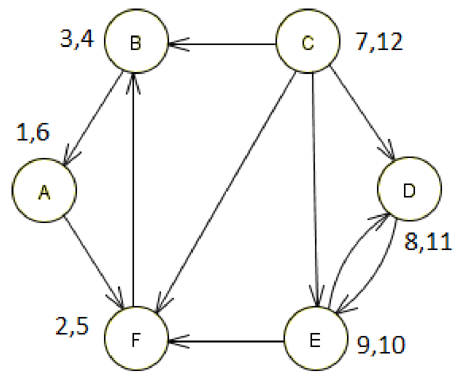
(A VERSION) QUESTION 2.

Refer to the graph $\boldsymbol{G}$ shown below and answer the following questions.

(a) *(5 points)* In the right side of the figure, draw the reverse graph $\boldsymbol{G}^R$ and execute DFS in $\boldsymbol{G}^R$ starting from vertex $\boldsymbol{A}$, breaking all ties in lexicographic order. Write the *pre* and *post* labels for each vertex in $\boldsymbol{G}^R$.



Graph $\boldsymbol{G}$

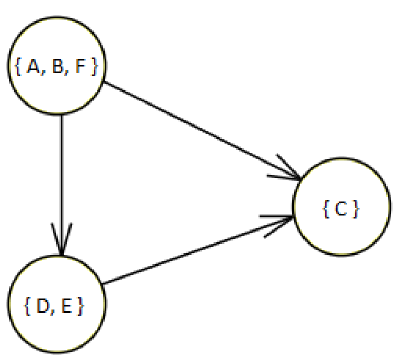


Graph $\boldsymbol{G}^R$

(b) *(3 points)* In the following table, write down the SCC(s) of $\boldsymbol{G}$ according to whether they are source SCC(s), sink SCC(s), or neither source nor sink SCC(s).

SCC(s) that are **source SCC(s)** in $\boldsymbol{G}$	SCC(s) that are **sink SCC(s)** in $\boldsymbol{G}$	SCC(s) that are **neither sink nor source SCC(s)** in $\boldsymbol{G}$
{A, B, F}	{C}	{D, E}

In case it helps, here is the meta-graph that you may have drawn:



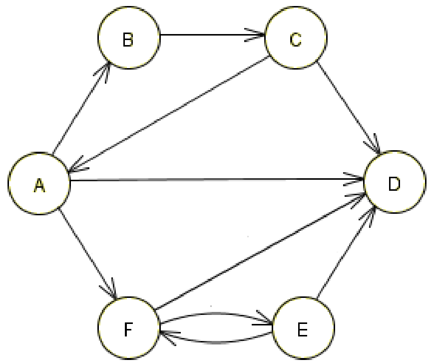
(c) *(2 points)* What is the **minimum** number of edges that, when added to $\boldsymbol{G}$, will make it strongly connected?

ONE edge from {C} to {A, B, F} is sufficient.

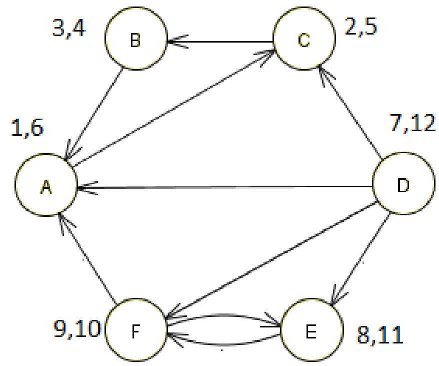
(B VERSION) QUESTION 2.

Refer to the graph $\boldsymbol{G}$ shown below and answer the following questions.

(a) *(5 points)* In the right side of the figure, draw the reverse graph $\boldsymbol{G}^R$ and execute DFS in $\boldsymbol{G}^R$ starting from vertex $\boldsymbol{A}$, breaking all ties in lexicographic order. Write the *pre* and *post* labels for each vertex in $\boldsymbol{G}^R$.



Graph $\boldsymbol{G}$

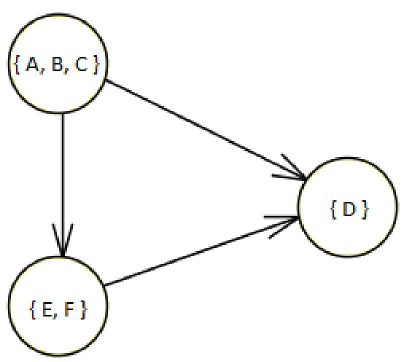


Graph $\boldsymbol{G}^R$

(b) (3 points) In the following table, write down the SCC(s) of $\boldsymbol{G}$ according to whether they are source SCC(s), sink SCC(s), or neither source nor sink SCC(s).

SCC(s) that are **source SCC(s)** in $\boldsymbol{G}$	SCC(s) that are **sink SCC(s)** in $\boldsymbol{G}$	SCC(s) that are **neither sink nor source SCC(s)** in $\boldsymbol{G}$
{A, B, C}	{D}	{E, F}

In case it helps, here is the meta-graph that you may have drawn:

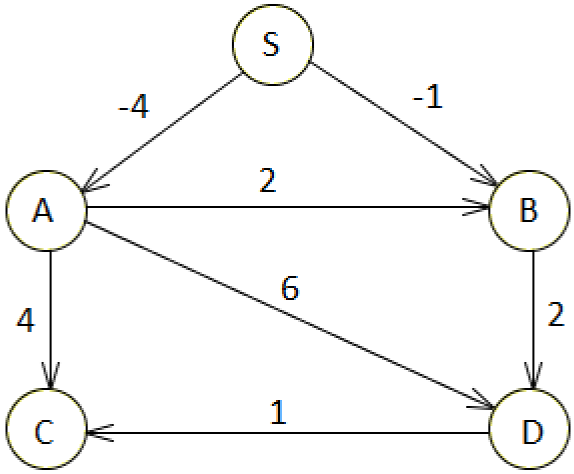


(c) *(2 points)* What is the **minimum** number of edges that, when added to $\boldsymbol{G}$, will make it strongly connected?

ONE edge is sufficient ({D} to {A, B, C})

(A VERSION) QUESTION 3.

Consider the following directed graph $\boldsymbol{G}$, with source vertex S and negative-weight edges only present from the source vertex.



(a) *(5 points)* Suppose Dijkstra's algorithm is executed on the graph, with S as the source vertex. Fill in the table with the intermediate distance values of all the vertices at each iteration of the algorithm. [Note: You know from Homework #2, Problem 7 that Dijkstra's algorithm correctly finds all source-sink shortest path lengths when any negative-weight edges in the graph are only from $\boldsymbol{S}$, as in this case.]

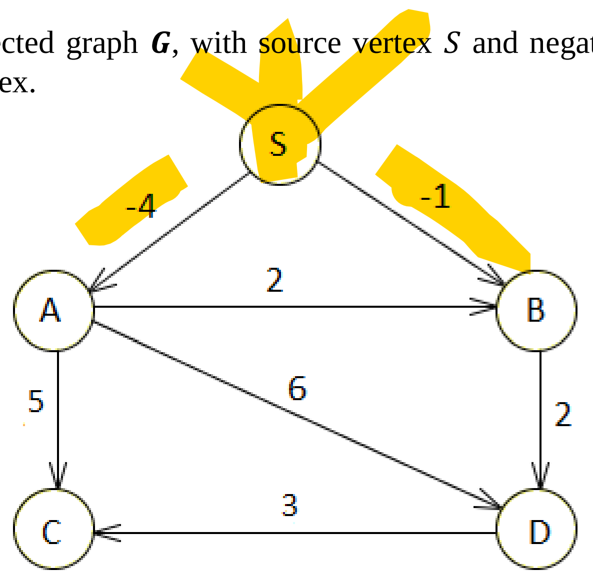
Label / Iteration	l(S)	l(A)	l(B)	l(C)	l(D)
0	0	∞	∞	∞	∞
1	0	-4	-1	∞	∞
2	0	-4	-2	0	2
3	0	-4	-2	0	0
4	0	-4	-2	0	0

(b) *(5 points)* Suppose the Bellman-Ford algorithm is executed on the same graph, with S as the source vertex. Fill in the table with the intermediate distance values of all the vertices at each iteration of the algorithm.

Label / Iteration k	l_S^k	l_A^k	l_B^k	l_C^k	l_D^k
$k = 0$	0	∞	∞	∞	∞
$k = 1$	0	-4	-1	∞	∞
$k = 2$	0	-4	-2	0	1
$k = 3$	0	-4	-2	0	0
$k = 4$	0	-4	-2	0	0

(B VERSION) QUESTION 3.

Consider the following directed graph $\boldsymbol{G}$, with source vertex S and negative-weight edges only present from the source vertex.



(a) *(5 points)* Suppose Dijkstra's algorithm is executed on the graph, with S as the source vertex. Fill in the table with the intermediate distance values of all the vertices at each iteration of the algorithm. [Note: You know from Homework #2, Problem 7 that Dijkstra's algorithm correctly finds all source-sink shortest path lengths when any negative-weight edges in the graph are only from $\boldsymbol{S}$, as in this case.]

Label / Iteration	l(S)	l(A)	l(B)	l(C)	l(D)
0	0	∞	∞	∞	∞
1	0	-4	-1	∞	∞
2	0	-4	-2	1	2
3	0	-4	-2	1	0
4	0	-4	-2	1	0

(b) *(5 points)* Suppose the Bellman-Ford algorithm is executed on the same graph, with S as the source vertex. Fill in the table with the intermediate distance values of all the vertices at each iteration of the algorithm.

Label / Iteration k	l_S^k	l_A^k	l_B^k	l_C^k	l_D^k
$k = 0$	0	∞	∞	∞	∞
$k = 1$	0	-4	-1	∞	∞
$k = 2$	0	-4	-2	1	1
$k = 3$	0	-4	-2	1	0
$k = 4$	0	-4	-2	1	0

QUESTION 4.

You are given an array $A[1...n]$ of n elements. A *majority* element of A is any element that occurs strictly more than $n/2$ times (so, if $n = 6$ or $n = 7$, any majority element will occur in at least four positions). For example, in the following array, 5 is a majority element:

$$[1, 5, 5, 5, 2, 6, 5, 1, 5]$$

Assume that elements **cannot be sorted**, but **can only be compared for equality**.

(Thus, for example, you aren't allowed to say "Sort the array in $O(n\ log\ n)$ time, then go through the sorted array once in $O(n)$ time to see if an element occurs more than $n/2$ times".)

(a) (*4 points*) Describe in words a divide-and-conquer algorithm to find a majority element in A (or determine that no majority element exists) in $O(n \log n)$ time. Note: You **MUST** use a DQ algorithm for this problem.

Observe that if we divide array A into two parts A1 and A2, then if a majority element M exists in A, then M will be a majority element in at least one of A1 and A2. Further, array A can have at most one majority element, as at most one element can occur more than $n/2$ times.

Using this observation, we can write a DQ algorithm as follows.

Divide array A into two parts A1 and A2 of size $n/2$ each. Recursively find a majority element (if one exists) in A1 and A2. For the merge step, there are four cases.

Case 1: Neither A1 nor A2 returns a majority element.
By the above observation, no majority element exists in A → return no majority element exists (this takes $O(1)$ time).

Case 2: A1 returns a majority element M1 and A2 returns no majority element.
Find the number of occurrences of M1 in A (this takes $O(n)$ time). If M1 occurs more than $n/2$ times, then it is a majority element in A. Else, return no majority element exists.

Case 3: A2 returns a majority element M2 and A1 returns no majority element.
(Symmetric to Case 2.)

Case 4: Both A1 and A2 return majority elements M1 and M2 respectively.
Find the number of occurrences of M1 and M2 in A (this takes $O(n)$ time). If either of M1 and M2 occurs more than $n/2$ times then return it as a majority element. Else, return no majority element exists.

(b) (*4 points*) Give pseudocode for your algorithm in (a).

```
procedure majority(A[1...n])

Input: Integer array A[1...n].

Output: Majority element M of A.


if n == 0:
     return NIL;
if n == 1:
     return A[1];

M1 = majority(A[1...⌊n/2⌋]) // Part A1
M2 = majority(A[(⌊n/2⌋+1)...n]) // Part A2

if M1 == M2:
     return M1;

if M1 != NIL:
      count_M1 = count occurrences of M1 in A[1...n] in linear
time.
else: // Case 3: Only A2 returns majority.
     count_M1 = 0

If M2 != NIL:
      count_M2 = count occurrences of M2 in A[1...n] in linear
time.
else: // Case 2: Only A1 returns majority.
     count_M2 = 0

// Case 4: both A1 and A2 return majority, check which
// element is majority in A[1...n]
if count_M1 > n/2:
     return M1;
if count_M2 > n/2:
     return M2;
return NIL; // Case 1: Neither side returns majority.
```

(c) (*2 points*) Write the recurrence $T(n)$ that characterizes the running time of your algorithm.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

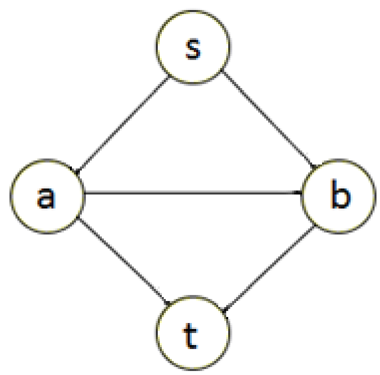
QUESTION 5.

A shortest path between two vertices $s \in V$ and $t \in V$ in an undirected, unweighted graph is a path from s to t that contains the least number of edges. Often there are multiple shortest paths between two vertices of a graph. Give a linear-time algorithm for the following task (i.e., your algorithm must run in $O(|V| + |E|)$ time).

Input: Undirected, unweighted graph $\boldsymbol{G} = (V, E)$; vertices $s, t \in V$.

Output: The number of distinct shortest paths from s to t.

For the following graph, your algorithm should return a value of 2, as there are two distinct shortest s-t paths (of length 2 edges: $s \to a \to t$ and $s \to b \to t$). The path $s \to a \to b \to t$ has length 3 edges, and is not a shortest s-t path.



(a) *(4 points)* Describe your algorithm in words.

Executing BFS on G with s as source vertex finds the shortest path between s and t. We can modify BFS by adding a counter for each vertex to keeps track of the number of shortest paths from s. The algorithm is then:

1. During the execution of BFS from s, when a vertex v is encountered for the first time, we know that this is the shortest $s - v$ path. Record this shortest path distance to v. If v was discovered through the edge (u, v) for some vertex u, set the counter value of v to that of u, indicating that there are as many shortest paths to v (through u) as there are to u.

2. Every subsequent time that vertex v is encountered, through some edge (u', v) for some vertex u', calculate the traversed path distance to v. If it is equal to the distance recorded in Step 1, increment the counter value of v by the counter value of u', indicating that we have discovered as many additional shortest paths to v (through u') as there are to u'.

3. After the execution of BFS completes, return the counter value of t.

(b) *(4 points)* Give pseudocode for your algorithm in (a).

```
procedure bfs(G, s, t)

Input: Undirected graph G = (V,E); vertices s,t ∈ V.
Output: Integer value count.

for all u ∈ V :
  dist(u) = ∞
  count(u) = 0
dist(s) = 0
count(s) = 1

Q = [s] (queue containing just s)
while Q is not empty:
  u = eject(Q)
  for all edges (u,v) ∈ E:
    if dist(v) == ∞:
      inject(Q, v)
      dist(v) = dist(u) + 1
      count(v) = count(u)
    else if dist(v) == dist(u) + 1:
      count(v) = count(u) + count(v)
return count(t)
```

(c) *(2 points)* Provide a time-complexity analysis in big-O notation based upon your pseudocode.

The first for loop iterates over all vertices in V, setting each vertex's distance and count values to 0. This takes $O(|V|)$ time.

Within the $while$ loop, a vertex is removed from the queue and the for loop iterates over its adjacency list. There are $O(|V|)$ queue operations and the total number of edges that the for loop iterates over is $O(|E|)$. Within the for loop, the modifications add a constant number of operations, which preserves the loop's time complexity of $O(|E|)$.

Therefore, the overall time complexity of the algorithm is $O(|V| + |E|)$, i.e., linear time.

SCRATCH PAGE. DO NOT REMOVE.