

Time: 8-10PM, Tuesday, Oct. 8th, 2024

Your Name: _____

Your PSU Access ID: _____

Your Recitation: _____

INSTRUCTIONS:

1. Please clearly write your full name, your PSU Access User ID (i.e., xyz1234), and the recitation you are in (001–010) in the box above.
2. This exam contains 16 questions.
3. For questions 1–12 (i.e., multiple-choice questions) exactly one choice is correct.
4. Your answer to questions 1–12 MUST be recorded in the table on top of page 2.
5. For questions 13–16, aim to complete your answer within the space provided on the current page and the next page. If additional space is needed, use the last 4 pages (pages 13–16), and clearly indicate which problem your answer corresponds to.

Your answer to questions 1–12:

Question	1	2	3	4	5	6	7	8	9	10	11	12
Your Answer												

1. (3 pts.) What is the time complexity of the following procedure?

```

1: for  $i \leftarrow 1$  to  $n$  do
2:    $j \leftarrow 1$ 
3:   while  $j \leq n$  do
4:      $j \leftarrow j \times 3$ 
5:   end while
6: end for

```

- A. $\Theta(n)$
- B. $\Theta(n^2)$
- C. $\Theta(n^3)$
- D. $\Theta(n \cdot \log n)$

Ans: D

2. (3 pts.) Let S be an array with n distinct integers. Similar to the selection algorithm, we partition S into $n/19$ subarrays, each of which contains 19 numbers. Let x be the median of the medians of the $n/19$ subarrays. How many numbers in S that are guaranteed to be less than or equal to x ? You may assume that n is divisible by 38.

- A. $5n/19$
- B. $11n/38$
- C. $6n/19$
- D. $13n/38$

Ans: A

3. (3 pts.) Consider the two recursive functions: $f(n) = 4f(n/2) + n^3$, $g(n) = 6g(n/2) + n^2 \log n$. Which of the following is correct regarding the growth rates of $f(n)$ and $g(n)$?

- A. $f = \Theta(g)$
- B. $f = O(g)$ but $f \neq \Theta(g)$
- C. $f = \Omega(g)$ but $f \neq \Theta(g)$

Ans: C

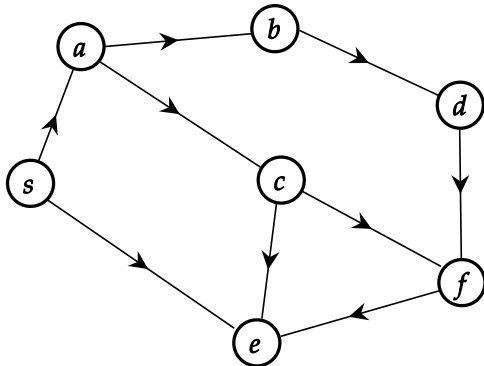
4. (3 pts.) Adding an edge to a DAG creates a cycle.

- A. Always
- B. Sometimes

C. Never

Ans: B

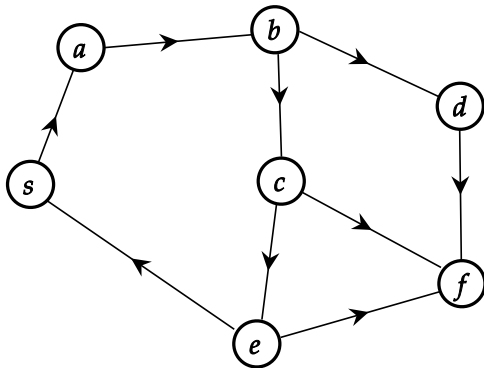
5. (3 pts.) How many linearizations does the following directed acyclic graph (DAG) have?



- A. 2
- B. 3
- C. 4
- D. 5

Ans: B

6. (3 pts.) How many edges in the meta-graph of the following graph?



- A. 2
- B. 3
- C. 4
- D. 5

Ans: B

7. (3 pts.) Consider a binary min-heap represented using an array A . It is known that the priority of one element has changed so that the min-heap-property does not hold. Currently $A = [15, 20, 16, 14, 22, 17]$. Which one of the following procedure can restore its heap-property? Assume A is indexed from 1.

- A. bubble-up($A, 4$);
- B. bubble-up($A, 6$);
- C. sift-down($A, 1$);
- D. sift-down($A, 2$);

Ans: A

8. (3 pts.) Consider running DFS-with-timing on a directed graph $G = (V, E)$. Assume that u is explored before v . Assume also that v can reach u but u cannot reach v in G . Which one of the following is always true?

- A. $pre[u] < pre[v]$ and $post[u] < post[v]$.
- B. $pre[u] < pre[v]$ and $post[u] > post[v]$.
- C. $pre[u] > pre[v]$ and $post[u] < post[v]$.
- D. $pre[u] > pre[v]$ and $post[u] > post[v]$.
- E. None of the above is always true.

Ans: A

9. (3 pts.) Consider the following variation of merge-sort: instead of partitioning the given array A into two subarrays of equal size, we divide A into two subarrays such that the first one consists of only one number, i.e., $A[1]$, and the other one consists of the remaining numbers, i.e., $A[2..n]$. What is the time complexity of this variation of merge-sort (where n is the size of the input array)?

- A. $O(n)$
- B. $O(n \cdot \log n)$
- C. $O(n^2)$
- D. $O(n^2 \cdot \log n)$

Ans: C

10. (3 pts.) Let $G = (V, E)$ be a directed acyclic graph (DAG). Which of the following implies that the vertex v is a sink in G ?

- A. The vertex v with the smallest post number when performing DFS-with-timing on G .
- B. The vertex v that appears last in any linearization of G .
- C. The vertex v with the largest post number when performing DFS-with-timing on G_R , i.e., the reverse graph of G .
- D. All of the above.

Ans: D

11. (3 pts.) Which of the following does NOT satisfy $f = O(g)$?

- A. $f(n) = n^3 \cdot 2^n$, $g(n) = n^2 \cdot 3^n$
- B. $f(n) = \sum_{k=1}^n 1/k$, $g(n) = \log n$
- C. $f(n) = \log(n^{\log n})$, $g(n) = 2^{\log \log n}$
- D. $f(n) = \log(n!)$, $g(n) = n^2$

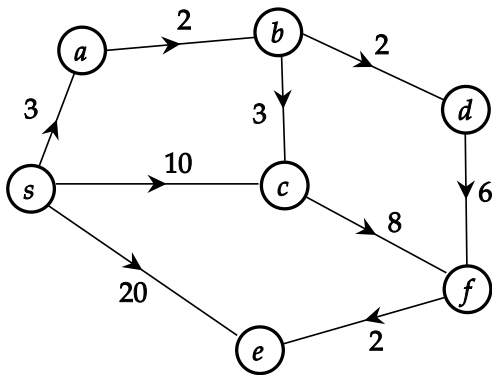
Ans: C

12. (3 pts.) Let $G = (V, E)$ be a directed graph. Assume that G consists of just one connected component, i.e., all vertices in V form a single connected component. Assume also that $|V| \geq 2$. Which of the following is NOT always true?

- A. $|E| \geq |V|$.
- B. $|E| \leq |V|^2/2$.
- C. G cannot be linearized.
- D. Let $u \in V$ be an arbitrary vertex. Then u can reach every other vertex in G .

Ans: B

13. (7 + 3 + 5 + 2 pts.) Consider the directed graph G given below.



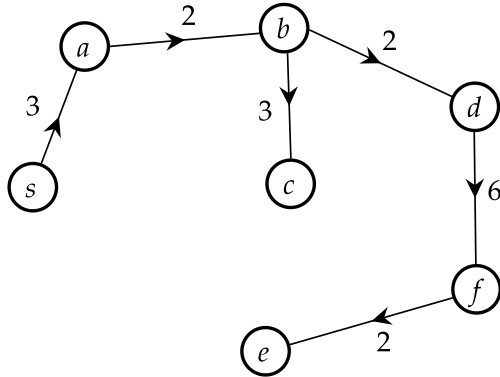
- (a) Run Dijkstra's algorithm on G , starting at the given source vertex s . Whenever there is a choice of vertices with the same dist value, always pick the one that is alphabetically first. Draw a table where each row shows the dist array at each iteration of the algorithm.
- (b) Give the order of vertices following which they are removed from the priority queue.
- (c) Draw one shortest path tree of G with respect to source s .
- (d) Is the shortest path tree of G unique? You just need to answer Yes or No.

Solution:

- (a) Below is a table showing the dist array at each iteration:

Iter.	s	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞	∞
1	0	3	∞	10	∞	20	∞
2	0	3	5	10	∞	20	∞
3	0	3	5	8	7	20	∞
4	0	3	5	8	7	20	13
5	0	3	5	8	7	20	13
6	0	3	5	8	7	15	13

- (b) The order is: s, a, b, d, c, f, e
- (c) The shortest path tree of G with respect to source s is:



- (d) Yes.

14. (5 + 10 pts.) Let $A[1..n]$ be an array with n positive integers, $n \geq 3$. Assume that $A[1] < A[2]$ and $A[n] < A[n-1]$. A location k is said to be a peak if $A[k] \geq A[k-1]$ and $A[k] \geq A[k+1]$. For example, there are five peaks in the following array; the peaks are in boldface.

5, **6**, **6**, 2, 3, **7**, 5, 4, **8**, 3, 3, 4, **10**, 6

- (a) Prove that, in such an array A , peak always exists.
- (b) Now we aim for designing an algorithm to find the location of a peak. If there are multiple peaks, we only need to find the location of one. The framework of the algorithm is given below, where the recursive function $\text{find-peak}(A, i, j)$ is defined to return the peak location in $A[i..j]$. You are asked to fill in the missing two parts in it (2 + 8 pts). The entire algorithm should run in $O(\log n)$ time.

function $\text{find-peak}(A[1..n], i, j)$

- 1: If $i + 2 \geq j$: # FILL IN MISSING PART 1
 - 2: $m \leftarrow \lceil (i + j) / 2 \rceil$
 - 3: $x \leftarrow A[m - 1]$
 - 4: $y \leftarrow A[m]$
 - 5: $z \leftarrow A[m + 1]$
 - 6: # FILL IN MISSING PART 2
-

Solution:

- (a) The maximum number in $A[1..n]$ must be a peak. This is because, the two ends of A , i.e., $A[1]$ and $A[n]$, cannot be the maximum since $A[1] < A[2]$ and $A[n] < A[n-1]$. The maximum therefore must be $A[k]$ for some $1 < k < n$. Hence $A[k] \geq A[k-1]$ and $A[k] \geq A[k+1]$. By definition, k is a peak position.

A different way to prove this is by induction. Assume there is no peak. We now prove by induction that $A[1] < A[2] < \dots < A[n]$ which contradicts with $A[n-1] > A[n]$. Base case: $A[1] < A[2]$. Induction step: Assume $\forall k \geq 1$, if $A[k] < A[k+1]$, then $A[k+1] < A[k+2]$. Because otherwise, position $k+1$ would be a peak. By induction, we have $A[1] < A[2] < \dots < A[n]$ which contradicts with $A[n-1] > A[n]$. Thus we prove that there exists at least a peak.

(b) See the code below.

```
function find-peak( $A[1..n]$ ,  $i$ ,  $j$ )
1: if  $i + 2 \geq j$  then
2:   compare  $A[i], A[i + 1], \dots, A[j]$  and return the  $k$  such that  $A[k]$  is the maximum in  $A[i..j]$ 
3: end if
4:
5:  $m \leftarrow \lceil (i + j)/2 \rceil$ 
6:  $x \leftarrow A[m - 1]$ 
7:  $y \leftarrow A[m]$ 
8:  $z \leftarrow A[m + 1]$ 
9:
10: if  $y \geq x$  and  $y \geq z$  then
11:   return  $m$ 
12: else if  $x > y$  then
13:   return find-peak( $A$ ,  $i$ ,  $m - 1$ )
14: else
15:   return find-peak( $A$ ,  $m + 1$ ,  $j$ )
16: end if
```

15. (6 + 10 pts.) You are given a directed graph $G = (V, E)$. We define a vertex $v \in V$ to be universal-reachable if every other vertex $u \in V \setminus \{v\}$ can reach v . The task is to identify all universal-reachable vertices in G .

- (a) Prove that, if universal-reachable vertices exist, then they form a connected component of G .
- (b) Given $G = (V, E)$, design an algorithm to find all universal-reachable vertices in G . Describe your algorithm (8 pts) and analyze its running time (2 pts). Your algorithm should run in $O(|V| + |E|)$ time. (Hint: think which connected component that universal-reachable vertices belong to.)

Solution:

- (a) Let G^M be the meta-graph of G . If universal-reachable vertices exist, they must form the only sink node (connected component) in G^M .

Proof: Assume conversely that v is a universal-reachable vertex but v belongs to a connected component C in G^M , which is not a sink node. Then, there exists out-edge (C, C') in G^M . Vertices in C' cannot reach any vertex in C , including v , since G^M is a DAG. This is a contradiction to that v is universal-reachable.

If there are two or more sink connected components in G^M , these sink nodes cannot have a path to each other (since their out-degree is 0). Therefore, the vertices in these sink connected components are not universally reachable either.

- (b) From (a), we know that if there is only one sink node in G^M , the answer is the number of vertices in that sink node. If there are two or more sink nodes in G^M , the answer is 0.

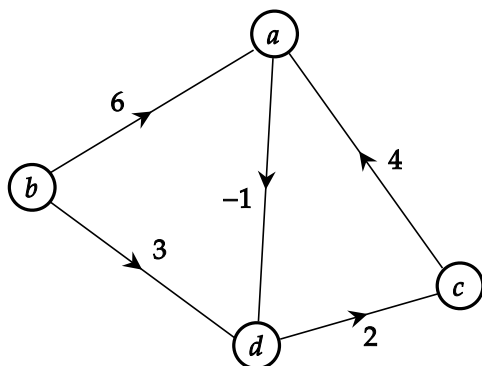
To implement this, we first run the algorithm (in Lecture 11) to find all connected components of G . The result is stored in the *visited* array, where $visited[u] = i$ indicates u belongs to connected component i . Here, $1 \leq i \leq k$, and k be the total number of connected components. We build an array X of size k , where $X[i]$ stores the size (number of vertices) in connected

component i . X can be constructed by scanning *visited* array once. We then determine which connected components are sink in G^M . To do so, we traverse all edges $(u, v) \in E$, and if $visited[u] \neq visited[v]$, then the connected component u belongs to, i.e., $visited[u]$, must be not a sink, and in this case we set $X[visited[u]] = 0$. After that, we check how many positive numbers remain in X . If there is exactly one positive number, which means there is only one sink component, the algorithm returns that number; otherwise the algorithm returns 0.

Time complexity: finding connected components takes $O(|V| + |E|)$ time (this requires two DFS). Traversing all edges to determine sink components takes $O(|E|)$ time; building X and checking positive numbers takes $O(|V|)$ time. Therefore, the overall time complexity is $O(|V| + |E|)$.

16. (4 + 12 pts.) Let $G = (V, E)$ be a directed graph with possibly negative edge length, but without negative cycle. Let $a \in V$. Define $distance_a(u, v)$ as the length of the shortest path from u to v that goes through vertex a .

- (a) See an example given below. Fill out the last row of the matrix, i.e., calculating $distance_a(d, v)$ for each $v \in \{a, b, c, d\}$.
- (b) Given $G = (V, E)$ and $a \in V$, design an algorithm to calculate $distance_a(u, v)$ for all pairs of vertices in G . Describe your algorithm (8 pts), analyze its running time (2 pts), and briefly prove the correctness of your algorithm (2 pts). Your algorithm should run in $O((|V| + |E|) \cdot |V|)$ time. (Hint: consider running Bellman-Ford algorithm twice (on two different graphs).)



input: G and vertex a

	a	b	c	d
a	0	∞	1	-1
b	6	∞	7	5
c	4	∞	5	3
d				

output: $distance_a(u, v)$, for all pairs u and v , represented as a matrix

Solution:

- (a) $distance_a(d, a) = 6$, $distance_a(d, b) = \infty$, $distance_a(d, c) = 7$, $distance_a(d, d) = 5$.
- (b) Note that $distance_a(u, v) = distance(u, a) + distance(a, v)$, where $distance(u, a)$ represents the distance (i.e., the length of the shortest path) from u to a , and $distance(a, v)$ represents the distance from a to v . Therefore, to determine $distance_a(u, v)$ for all pairs of vertices, we only need to determine $distance(u, a)$ for all $u \in V$ and $distance(a, v)$ for all $v \in V$.

We can determine $distance(a, v)$ for all $v \in V$ by running the Bellman-Ford algorithm with a as the source vertex. To compute $distance(u, a)$ for all $u \in V$, consider the reverse graph G_R of G . Clearly, a shortest path from u to a in G will have a corresponding shortest path from a to u in G_R . Thus, we can run Bellman-Ford algorithm in G_R with a as the source vertex,

which will give us $distance(a, t)$ for all $u \in V$. After having $distance(u, a)$ for all $u \in V$ and $distance(a, v)$ for all $v \in V$, we can then compute $distance_a(u, v)$ for all $u, v \in V$ simply by adding $distance(u, a)$ and $distance(a, v)$.

The running time of the above algorithm is $O((|V| + |E|) \cdot |V|)$. Building the reverse graph G_R takes $\Theta(|V| + |E|)$ time. We run the Bellman-Ford algorithm twice to determine $distance(u, t)$ for all $u \in V$ and $distance(t, v)$ for all $v \in V$, which takes $O(|V| \cdot |E|)$ time. The next step of computing $distance(u, t)$ for all $u, v \in V$ takes $O(|V|^2)$ time, as it takes $O(1)$ time to compute $distance_a(u, v)$ for each pair of u and v .