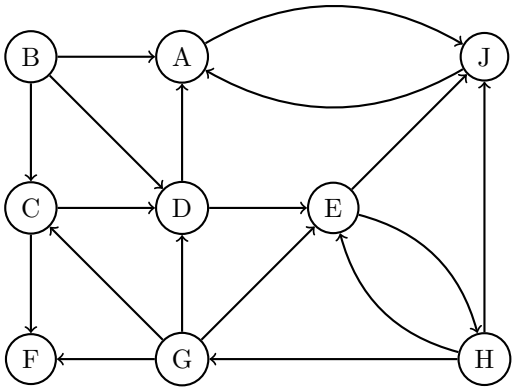
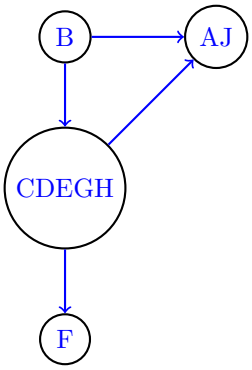
Midterm 1 Solutions

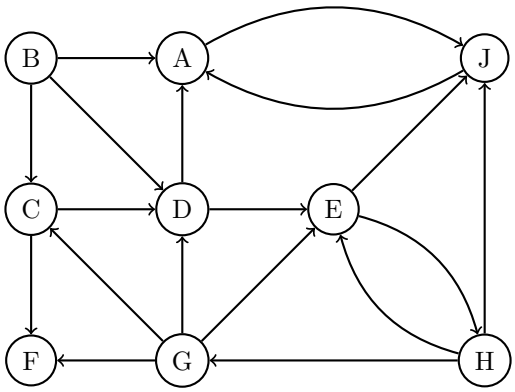
1. **(4 points)** For the directed graph below, find all the strongly connected components and draw the DAG of strongly connected components. Label each strongly connected component with all the nodes it contains.



Draw the DAG in the box below:

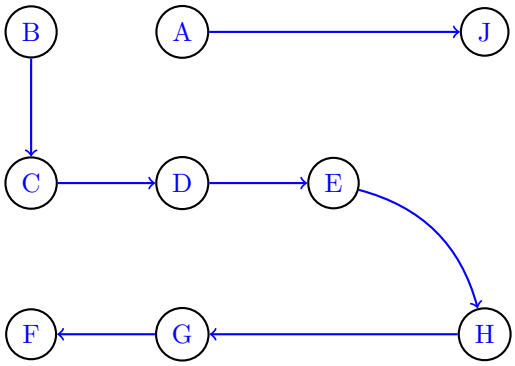


2. **(8 points)** Execute DFS on the same graph (reproduced here for convenience) starting at node A and breaking ties alphabetically. Draw the DFS tree/forest. Mark the pre and post values of the nodes with numbering starting from 1.



Node	pre	post
A		
B		
C		
D		
E		
F		
G		
H		
J		

Draw the DFS Tree/Forest in the box below:

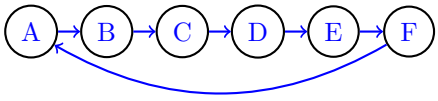


Node	pre	post
A	1	4
B	5	18
C	6	17
D	7	16
E	8	15
F	11	12
G	10	13
H	9	14
J	2	3

3. **(4 points)** In the DFS execution from above, mark the following edges as as **T** for Tree, **F** for Forward, **B** for Back and **C** for Cross. (No justification necessary)

Edge	Type
$G \to C$	B
$A \to J$	T
$B \to A$	C
$B \to D$	F

4. (a) **(4 points)** Draw a strongly connected graph with 6 vertices with the smallest possible number of edges in the box below.



(b) **(2 points)** In general, the minimum number of edges in a strongly connected directed graph with n vertices is n. (no justification necessary)

You can't have less than n edges as it will make the graph disconnected. So the minimum is simply a loop through all the vertices.

5. **(6 points)** Suppose $G = (V, E)$ is an undirected graph with positive integer edge weights $\{w_e | e \in E\}$. We would like to find the shortest path between two vertices s and t with an additional requirement: if there are multiple shortest paths, we would like to find one that has the minimum number of edges.

We would like to define new weights $\{w'_e | e \in E\}$ for the edges so that, a single execution of Dijkstra's algorithm on the graph G with new weights $\{w'_e\}$, starting from s finds the shortest path to t with this additional requirement.

How should we set the new weights w'_e?

$$w'_e = w_e + \frac{1}{|V|}$$

No justification necessary.

The intuition here is that we want to add something to the edges so that we can differentiate a path using 4 edges and a path using 5 edges but with the same distance. In the meantime, we need to make sure the original Dijkstra's doesn't break. Therefore, we add the fraction $\frac{1}{|V|}$ so that the total length of any path will be increased by at most 1. Since all the edges are positive integers, Dijkstra's will run exactly the same as before.

6. **(8 points)** Here is an implementation of Bellman-Ford algorithm:

`Input:` Directed Graph $G = (V, E)$, with edge lengths $\{\ell_e | e \in E\}$.
`Output:` Compute distances $dist(u)$ to each vertex u from a start vertex s.

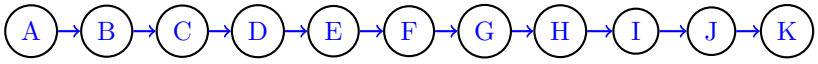
for $i = 1$ to n **do**
  $dist(u) \leftarrow \infty$
  $prev(u) \leftarrow nil$
end for
$dist(s) \leftarrow 0$
$k \leftarrow 0$
repeat
  for $i = 1$ to n **do**
    for each directed edge (i, j) **do**
      update(i, j)
    end for
  end for
  $k \leftarrow k + 1$
until all $dist$ values stop changing OR $(k = n)$

Algorithm 1: Bellman-Ford Algorithm

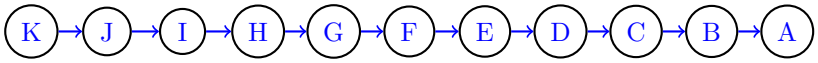
It turns out that the runtime of the above algorithm can be very sensitive to the way in which vertices in a graph are numbered. In other words, the runtime of the algorithm on the same graph can widely vary, if we change the numbering of the vertices.

Give one graph G on 11 vertices and two ways to label the vertices of G, such that in one labelling the algorithm makes 20 calls to update, while in the other labelling the algorithm terminates in 10^2 calls to update.

The following graph will call update exactly 20 times.



The following graph will call update exactly 10^2 times.



7. **(6 points)** We computed the minimum spanning tree T on a graph G with costs $\{c_e\}_{e \in E}$. Unfortunately, after computing the minimum spanning tree, we discover that the costs of all the edges in the graph have changed as follows: the new cost w_e are given by,

$$w_e = \begin{cases} 2 \cdot c_e & \text{if } c_e > 100 \\ 0 & \text{if } c_e \leq 100 \end{cases}$$

Is the tree T that we computed earlier, still a minimum spanning tree of the graph?

Write "yes" or "no": yes

If yes, prove; if no, disprove with a counterexample.

Consider an edge e that is part of the MST T composed from graph G with costs $\{c_e\}_{e \in E}$. We will show that every such edge e must also be part of the MST T' after the weight update.

Case 1
Suppose $c_e \leq 100$, then $w_e = 0$. Since e was part of T, it is an edge with the minimal weight connecting

two sub-MSTs. If we have an edge e' such that $c_{e'} \leq 100$, then in the new MST T', $w_e = w_{e'}$. Thus edge e will still be part of MST T' as adding either e or e' would not add any cost to the total cost of T'. If $c_{e'} > 100$, then $w_e < w_{e'} = 0$ and thus e is part of MST T'.

Case 2

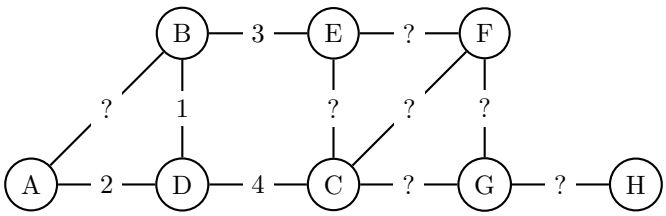
Now suppose $c_e > 100$, then $w_e = 2 \cdot c_e$. Suppose for the sake of contradiction that e is not part of MST T'. There must be some edge e' part of MST T' connecting the two sub-MSTs that e is connecting. If $w_{e'} = 0$, then $c_{e'} \leq 100 < c_e$. Thus e' must be part of MST T rather than e. This is a contradiction because e is part of T and having both e and e' connecting the two sub-MSTs would create a cycle and violate the tree property. Now if $w_{e'}, c_{e'} > 100$ and we assume that e' is part of T' rather than e, $2 * c_{e'} < 2 * c_e$. Thus $c'_e < c_e$. Again e' must be part of MST T, a contradiction.

[The most common mistake was to ignore the case where an edge e of weight $c_e > 100$ may be replaced by an edge e' with prior weight $c_{e'} \leq 100$. For full credit, we needed an explanation of why this case was impossible.]

Alternative solution

A less common, but correct alternative solution was to argue that Kruskal's algorithm produces valid MSTs by considering each edge $e \in G$ in increasing order of weight c_e. The ordering of the edges is preserved in this update. Therefore, if T was a valid solution produced by Kruskal's algorithm prior to the update, then T will remain a valid solution after the update.

8. (**6 points**) In this graph, some of the edge weights are known, while the rest are unknown.



$$cost(A, D) = 2, cost(B, D) = 1, cost(C, D) = 4, cost(B, E) = 3$$

List all edges that **must** belong to a minimum spanning tree, regardless of what the unknown edge weights turn out to be. Justify each of your edges briefly (a sentence or less is enough).

Edges that must belong to every MST	Justification
$G - H$	This is the only edge that can connect H to any other vertex, so it must be included in any MST. Remember that any MST must span all of the vertices.
$B - D$	The cut property. More details: let $S_1 = D$, and $S_2 = V - S_1 = ABCEFGH$. Then three edges from the original graph will connect these two forests, with edge weights: $1, 2, 4$. So any MST must contain the $B - D$ edge of weight 1.
$B - E$	Also by cut property. Consider $S_1 = ABD$, and $S_2 = V - S_1 = CEFGH$. Then the 3 edge is the shortest edge connecting both forests.

Alternative justifications:

Run Prim's MST algorithm, starting from H. $H - G$ is the shortest edge, so it will be added to the MST. Then we know at least one MST will include this edge.

And similarly, we can run Prim's, starting from D. Then the $D - B$ edge is the first edge to be added to the MST, so we know at least one MST will include $D - B$.

But for $B - E$, the Prim's argument gets more complicated. You could argue that running Prim's starting from any of ABD will always add the edge of weight 3, and never the edge of weight 4.

9. Design an efficient algorithm for the following problem

`Input:` n `numbers` $\{a_1, \ldots, a_n\}$
`Goal: Compute the polynomial with` $a_1, \ldots, a_n$ `as its roots. In other words, compute coefficients` $b_0, \ldots, b_n$ `so that` $(x - a_1) \cdot (x - a_2) \cdots (x - a_n) = b_0 + b_1 x + \ldots b_n x^n$.

(Hint: Try divide and conquer & use $O(n \log n)$ time polynomial multiplication algorithm as a blackbox)

(a) **(10 points)** Pseudocode:

procedure PolynomialWithRoots($\{a_1, \ldots, a_n\}$):

if n $= 1$ **then**
  $b_0 \leftarrow -a_1$
  $b_1 \leftarrow 1$
  return b_0, b_1
end if
$q(x) \leftarrow$ PolynomialWithRoots($\{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$)
$r(x) \leftarrow$ PolynomialWithRoots($\{a_{\lfloor n/2 \rfloor}, \ldots, a_n\}$)
$p(x) \leftarrow$ MultiplyPolynomials($q(x)$, $r(x)$)
return coefficients of $p(x)$

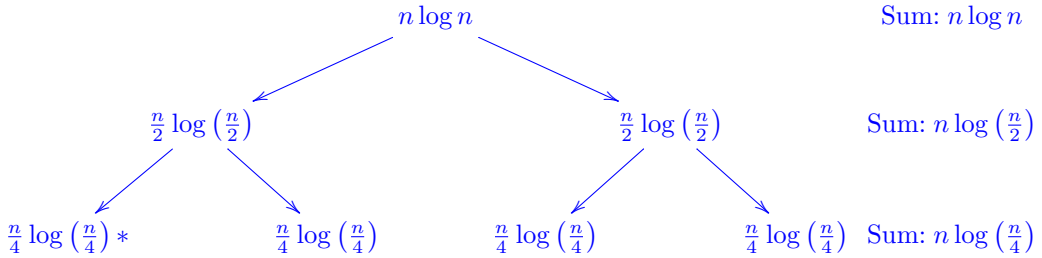
(b) **(3 points)** Write the recurrence for the running time of the algorithm in the box.

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

(c) **(6 points)** Solve the recurrence to compute the running time and put your answer in the box. Show your work below the box.

$T(n) =$ $\boxed{O(n \log^2 n)}$

To show this, we use the recurrence tree:



$\vdots$

The total, then, is

$$
\begin{aligned}
& n \log n + n \log \left(\frac{n}{2}\right) + \log \left(\frac{n}{4}\right) + \cdots \\
&= n \left(\log n + \log \left(\frac{n}{2}\right) + \log \left(\frac{n}{4}\right) + \cdots\right) \\
&= n \left(\log n + (\log n - \log 2) + (\log n - \log 4) + \cdots\right) \\
&= n \left((\log n + \log n + \cdots + \log n) - (\log 2 + \log 4 + \cdots + \log n)\right) \\
&= n \left((\log n)^2 - (1 + 2 + \cdots + \log n)\right) \\
&\approx n \left(\log^2 n - \frac{\log^2 n}{2}\right) \\
&= \frac{n \log^2 n}{2} \\
&= \Theta\left(n \log^2 n\right)
\end{aligned}
$$

We could also use a more general version of the Master Theorem, which states that if $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$, then $T(n) = \Theta\left(n^c \log^{k+1} n\right)$. In this case, $a = b = 2$, $c = 1 = \log_b a = \log_2 2$, and $k = 1$.

10. **(13 points)** You are given the road network $G = (V, E)$ of a country, and the lengths $\{\ell_e | e \in E\}$ of each road in the network.

    Some of the cities have airports, while others don't. Let F be the subset of cities that have an airport in them.

    Devise an algorithm to compute the distance from each city to the nearest airport. (Assume that the graph is directed and that all edge lengths are non-negative).

    Remember every *correct* algorithm will receive a score depending on its runtime. (can you do it with the same run-time as Dijkstra's?).

    (a) **Main Idea:** (try less than 6 sentences if you can, but don't fret if you go over)

    Reverse the graph. Add a dummy source vertex s to the graph. Add a zero weight edge from the source vertex to each airport. Then run Dijkstra on s to find the distance from s to every vertex in the graph. This shortest distance from each city to the nearest airport is the distance from the vertex that represents this city to vertex s.

    It is equivalent to merge all airports into a single vertex or to run a modified Dijkstra's in which all airports are initialized with $dist = 0$.

    Brute force solutions were common, involving running Dijkstra from each vertex or just from each airport vertex. These are correct but inefficient, as the running time will be $O(|V|(|V| + |E|) \log |V|)$ or $O(|F|(|V| + |E|) \log |V|)$, and we have no bound on $|F|$ except that it is at most $|V|$.

    (b) **Runtime of the algorithm** $=$ $O((|V| + |E|) \log |V|)$

    (c) **Proof of Correctness** (try less than 4 sentences if you can, but don't fret if you go over)

    This is a directed graph and the dummy node s that we add is a source node which has only edges going out and no edges going in. Thus adding the node s won't add a path between any two nodes from the original graph. In addition, we set the length of all edges of s to zero, so it won't add any additional length. Through Dijkstra (which we know finds shortest distances from a source node to all others) we will find the shortest distance from each node to node s (because

this is equivalent to the reversed path in the reversed graph), which is the same as the shortest path to any airport.

A brute force solution can be seen to be correct by noting that Dijkstra will find the shortest path between every relevant pair of nodes, via multiple calls, and that we can compare them to find the minimum in each case.

11. **(10 points)** Suppose you are given an array $A[1 \ldots n]$ of sorted integers that has been circularly shifted k positions to the right for some k. For example, $[35, 42, -5, 15, 27, 29]$ is a sorted array that has been circularly shifted $k = 2$ positions, while $[27, 29, 35, 42, -5, 15]$ has been shifted $k = 4$ positions. We can obviously find the largest element in A in $O(n)$ time.

    Assuming all the integers in the array are distinct, describe an $O(\log n)$ algorithm to find the largest element in A.

    Brief but precise description of the algorithm: (try less than 6 sentences if you can, but don't fret if you go over)

    Main idea: We want to leverage the sorted-ness of the array, so we will implement a modification of binary search where you always recurse on the "unsorted" array. Compare A[0] and A[mid] (where mid $= \lceil \frac{n}{2} \rceil$). If $A[0] < A[\text{mid}]$ then the left half of the array is sorted; the max value is either the mid or in the right half of the array. Store the mid as the temporary max and repeat on the right half. If $A[0] > A[\text{mid}]$ then the left half is unsorted and the max is somewhere in that array, so repeat on the left half.
    We can also compare the mid point with the last element. If $A[\text{last}] < A[\text{mid}]$, we look at the right half of the array, including the mid point. If $A[\text{last}] > A[\text{mid}]$, we look at the left half of the array.

    Other splits are also possible. For example, comparing $A[\lceil \frac{n}{4} \rceil]$ with $A[\lceil \frac{3n}{4} \rceil]$. As long as a solution compares 2 points that are far enough apart, it can be used to determine the portion of the array to make the recursive call. If left point is greater than right point, make the recursive call on the element between them. Otherwise the maximum element is in the other part of the array, which can be obtained by joining the two parts in the same relative positioning, i.e. $A[1, \ldots, \lceil \frac{n}{4} \rceil] + A[\lceil \frac{3n}{4} \rceil, \ldots, n]$.

    A common mistake was to compare immediate neighbors. But this can only be used to determine if the left element is the max. It does not have enough information to tell you which side the max is on and thus cannot obtain a $O(\log n)$ solution.