1 (20 PTS.) Short questions.

1.A. (5 PTS.) Give an asymptotically tight solution to the following recurrence, where $T(n) = 1$ for $n < 14$, and otherwise:
$T(n) = T(3n/7) + T(4n/7) + n^3$.

Solution:

$T(n) = O(n^3)$.

A proof is not required but the easiest way to see this is verifying the inductive hypothesis here. Namely, that $T(n) \leq cn^3$, for some constant c sufficiently large. The recurrence can be written as $T(n) \leq T(3n/7) + T(4n/7) + n^3$. As such, we have that

$$T(n) \leq T(3n/7) + T(4n/7) + n^3 \leq \left(\frac{27}{7^3}\right) cn^3 + \left(\frac{64}{7^3}\right) cn^3 + n^3 = \left(\frac{91}{343}c + 1\right) n^3 \leq cn^2,$$

which holds $c \geq 343/252$

1.B. (10 PTS.) Given a directed graph $\mathsf{G}(V, E)$, describe a linear-time algorithm that outputs all the edges in G that are contained in some cycle.

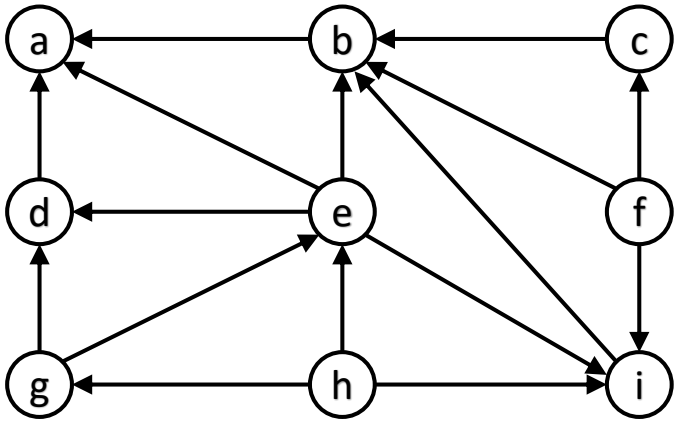
Solution:

Compute the SCCs of G. For each edge $e = (u, v)$, if u and v belong to the same SCC, then e is in some cycle. If u and v belong to different SCCs, then e is not part of any cycle.

A proof is not required but it is easy to see that if u and v belong to the same SCC, then there is a path from v to u. Adding e to that path creates a cycle. If u and v belong to different SCC, then e cannot be in any cycle. If it were, then v can reach u via some path and u can reach v via e. This implies that u and v belong to the same SCC which is a contradiction.

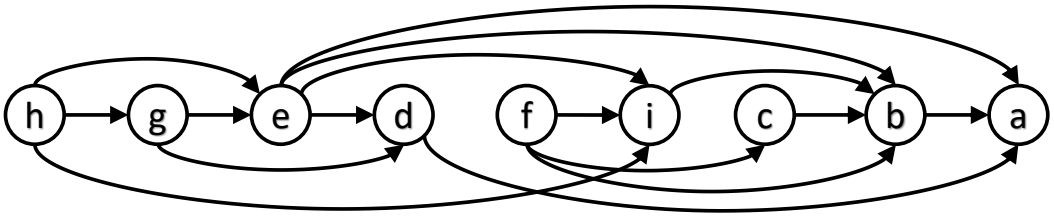
It takes $O(|V| + |E|)$ to compute the SCCs and $O(|E|)$ to check each edge. Hence, the algorithm is linear.

1.C. (5 PTS.) Compute a topological sort of the following directed graph or state that no such order exists if the graph is not a DAG.



Solution:

The graph is a DAG, there are many many possible solutions. Here is one: h, g, e, d, f, i, c, b, a.



2 (20 PTS.) Political Influence

You are given a directed graph $\mathsf{G} = (V, E)$ where every vertex $u \in V$ represents a politician and an edge $e = (u, v)$ indicates that politician u has political influence on v. u can use its political influence to request a favor from v or ask v to leverage its own influence to ask a favor from the politicians it can reach. Each politician possesses a certain political power $p(u) > 0$. The more political power a politician has, the bigger the favor they can provide. Given a vertex s, define $f(s)$ as the biggest political favor that s can ask of any politician it can reach. Define $c(s)$ as the total political capital that s possesses.

Formally, let R be the set of vertices that s can reach (Note that $s \in R$), then

$$\begin{aligned} f(s) &= \max\{p(v) \mid v \in R\} \\ c(s) &= \sum_{v \in R} p(v) \end{aligned}$$

Finally, define the ultimate political capital as total political power in the graph, i.e. $\sum_{v \in V} p(v)$.

2.A. (2 PTS.) Given a vertex s, describe a linear-time algorithm to compute its political capital.

Solution:

Run whatever-first-search or BFS or DFS to find R. Then, compute $c(s)$ by summing the $p(v)$ for $v \in R$.

2.B. (2 PTS.) Suppose G is a DAG. In which case will a single unique politician possess the ultimate political capital?

Solution:

If the DAG has a single source s, then s and only s can reach all nodes in the graph. Hence, s will possess the ultimate political capital.

2.C. (10 PTS.) Suppose G is a DAG. Describe a linear-time algorithm to find for each $u \in V$, the biggest political favor that u can request, i.e. compute $f(u)$ for each $u \in V$. Briefly argue the runtime and correctness of your algorithm.

Solution:

We find a topological ordering of G in linear time. Then we process the vertices in reverse topological order. For each vertex u, we compute the following recurrence:

$$f(u) = \max\{p(u)\} \cup \{f(v) \mid (u, v) \in E\}$$

We will store the values of this function in a one-dimensional array of length $|V|$. The following pseudo-code describes the algorithm.

```
BiggestFavorDAG(G, p)
T ← topological ordering of vertices in G.
f ← an array of length |V|.
for i ← |V|, ··· , 1 do
   u = T(i)
   f(u) = p(u)
   for each v such that (u, v) ∈ E do
      if f(u) < f(v)
         f(u) = f(v)
return f
```

Observe that since we process vertices in reverse topological order, the value of $f(v)$ for all the children of the vertex u is computed before we process u. To see the correctness of this recurrence, note that if the vertex with the most power reachable from u is not u, then it is reachable from u through one of its children. Since every vertex is processed once and every edge is considered only once, the algorithms runs in linear time $O(|V| + |E|)$.

Rubric: 10 points:

- -8 pts if f is computed for a single vertex s.
- -6 pts if use one DFS/BFS per vertex and repeat $|V|$ times.
- -6 pts for any other non-linear time algorithm.
- -4 pts if argument for correctness and/or runtime is missing.
- -4 pts for wrong evaluation order or wrong implementation of the recurrence.
- -2 pts for minor error and unclear explanation.

2.D. (6 PTS.) Suppose G is a general directed graph. Describe a linear-time algorithm to find for each $u \in V$, the biggest political favor that u can request, i.e. compute $f(u)$ for each $u \in V$. Briefly argue the runtime and correctness of your algorithm. (Hint: Think of how you would solve it if G was strongly connected and then leverage the algorithm from the previous part as a black box.)

Solution:

Find the strongly connected component for G and build the meta graph G_{SCC}. This can be done in linear time. Define $p' : V(\mathsf{G}_{SCC}) \to \mathbb{R}$ which the maximum political power among vertices in each strongly connected component to the vertex corresponding to that strongly connected component in the meta-graph G_{SCC}. Formally, for every strongly connected component C of G, we define $p'(C) = \max\{p(v) \mid v \in V(C)\}$.

Since the meta-graph G_{SCC} is a DAG, we can run the algorithm from the previous part on this graph. Then, for each vertex in a strongly connected component, we assign the values of f computed for its strongly connected component. The following pseudo-code implements the algorithm.

```
BiggestFavor(G, p)
G_SCC ← strongly connected component meta-graph of G.
p' ← an array of length |V(G_SCC)|.
f' ← an array of length |V(G_SCC)|.
f ← an array of length |V(G)|.
for C ∈ V(G_SCC) do
   p'(C) = max {p(v) | v ∈ V(C)}
f' = BiggestFavorDAG(G_SCC, p')
for C ∈ V(G_SCC) do
   for u ∈ V(C) do
      f(u) = f'(C)
return f
```

Computing the meta-graph takes $O(|V| + |E|)$. Computing p' takes $O(|V|)$. Running the previous algorithm on a meta-graph takes $O(|V(\mathsf{G}_{SCC})| + |E(\mathsf{G}_{SCC})|) = O(|V| + |E|)$. Finally, assigning the final f takes $O(|V|)$. Hence, the entire algorithm runs in $O(|V| + |E|)$. To see the correctness of the algorithm, observe that every vertex in a strongly connected component has the same reachability and hence all vertices belonging to the same component will have the biggest favor f. Also observe that if v can reach u then v can reach any vertex in the strongly connected component containing u.

Rubric: 6 points:

- -6 pts if f is computed for a single vertex s.
- -4 pts if use one DFS/BFS per vertex and repeat $|V|$ times.
- -4 pts for any other non-linear time algorithm.
- -3 pts for not correctly generating the DAG meta-graph.
- -3 pts for wrong translation between p and f and p' and f'.
- -2 pts if argument for correctness and/or runtime is missing.
- -2 pts for minor error and unclear explanation.

3 (20 PTS.) 5G Cellular Deployments

A key difference between 4G cellular networks and 5G cellular networks is small cell deployments. This means instead of only having large cellular base stations that cover large areas like west Champaign, 5G also has small cellular base stations that can cover small areas like few houses or a building. Small cells in 5G are supposed to be much more efficient than 4G. However, a major challenge facing 5G is real estate cost.

GlobalCell company wants to deploy b 5G small cell base stations along Green St. There are n possible locations $x_1, x_2, \cdots, x_n$. Suppose that x_i represents the distance in meters from the start of the street and the locations are indexed in increasing distance, i.e. $0 \leq x_1 < x_2 < \cdots < x_n$. Deploying a base station at location x_i costs c_i. The goal is to minimize the cost of deploying the b base stations. However, we cannot deploy the base stations too close to each other since they would interfere and result in poor performance. Hence, any two base stations must be at least D meters apart. We also cannot deploy less than b base stations. Otherwise, the cellular network performance would be poor.

Describe and analyze an algorithm as fast as possible that GlobalCell can use to minimize the cost needed to deploy b base stations while satisfying the distance constraints. If no feasible solution exists, your algorithm should output ∞. (You do not need to output the locations where the base stations need to be deployed, just the minimum cost.)

Give a clear English description of the function you are trying to evaluate, and how to call your function to get the final answer, then provide a recursive formula for evaluating the function (including base cases). If a correct evaluation order is specified clearly, pseudocode is not required. Analyze the running time and space as a function of n and b.

Solution:

Let $C(i, j)$ be the minimum cost needed to deploy j base stations while satisfying the distance constraints using locations $x_i, \cdots, x_n$ for $1 \leq i \leq n+1$ and $0 \leq j \leq b$

$$C(i,j) = \begin{cases} 0 & j = 0 \\ \infty & i > n \\ \infty & i + j - 1 > n \text{ and } j \neq 0 \\ \min\{c_i + C(next(i), j-1), C(i+1, j)\} & \text{Otherwise} \end{cases}$$

where $next(i) = \min\{k \mid x_k \geq x_i + D\}$ is the next available location that is at least D distance away from the current location x_i. If every location after x_i is less than D meters away, $next(i) = n + 1$. (Note that only one of the $i + j - 1 > n$ and $i > n$ base cases is necessary. We list both here for completeness.)

We solve the problem by evaluating $C(1, b)$ which represents the cost to deploy b base stations using locations $x_1, \cdots, x_n$.

To memoize the problem, we can use a two-dimensional array $C[1, \cdots, n+1; 0, \cdots, b]$. We evaluate the array in column-major order: column by column left to right, each column from down to top. In other words: for $j = 0, \cdots, b$; for $i = n+1, \cdots, 1$;. This would require $O(nb)$ space. However, note that a subproblem in the column indexed by j only depends on

subproblems in the same column or the previous column indexed by $j-1$. Therefore, only two instead of b columns need to eb memoized and the space required is only $O(n)$.

There are $O(nb)$ distinct subproblems. However, each requires evaluating $next(i)$. Naively, finding $next(i)$ for each subproblem can take $O(n)$ in the worst case each time. This would lead to a runtime of $O(n^2b)$. We can find $next(i)$ faster using binary search. This would lead to a runtime of $O(nb \log n)$. However, it is possible to find $next(i)$ in $O(1)$ by preprocessing and precomputing an array of $next(i)$ locations. The preprocessing step takes $O(n)$ which can be achieved via the following simple for loop.

```
m = 2
x_{n+1} = ∞
for i = 1, ··· , n do
    while x_i + D > x_m do
        m = m + 1
    next(i) = m
```

Hence, precomputing an array to store $next(i)$ would reduce the runtime to $O(n+nb) = O(nb)$. It will also incur an additional $O(n)$ space. Hence, the total space required is still $O(n)$.

<u>*Rubric:*</u> Standard DP rubric. 20 points total. Little penalty for less efficient algorithm. In particular:

- -1 pts for $O(nb \log n)$ instead of $O(nb)$.
- -2 pts for $O(n^2b)$ instead of $O(nb)$.
- -2 pts for $O(nb)$ space instead of $O(n)$.
- -4 pts for slower but correct solution.
- -10 pts for no recurrence or pseudo code but correct description and idea presented.
- -10 pts for ignoring the constraint of exactly b base stations.
- -20 pts for ignoring the D constraint.
- -2 pts for minor errors or missing base case.

4 (20 PTS.) Medical Supplies.

You're a purchaser with the Illinois Emergency Management Agency in Springfield, tasked with acquiring a supply of medical equipment to help with the Covid-19 outbreak. Through your personal connections, you've found an opportunity to bid in a auction with a private broker in Chicago, but you have to hurry since there are competing bidders and the offer ends soon. The broker wants to help, but is old-fashioned and will not accept any form of electronic agreement... instead the only way you can place your bid is to meet up with the broker at a McDonald's parking lot. If you and the broker both start driving at the same time, can you reach a McDonald's before the auction ends?

4.A. (10 PTS.) You are given a roadmap in the form of a weighted directed graph G. The n nodes represent locations, and the m edge weights represent the time it takes to travel from one location to another ($m > n$, and all weights are non-negative). You can assume you have access to a flag for each node, $q(u)$, which is 1 if node u is a McDonald's and 0 otherwise. (You cannot assume anything about the number of McDonald's except that are no more than n of them). Your starting location is node S, and the broker's starting location is node C. The auction ends at time $t > 0$ and both you and the broker begin driving at time 0.

Describe an algorithm, as fast as possible, to determine whether or not there is a McDonald's that you and the broker can both reach before the auction ends. Analyze the running time of your algorithm as a function of n and m.

Solution:

Use Dijkstra's algorithm first to compute the $d_S(u)$ the length of the shortest weighted path from S to u for each node u, and next to compute $d_C(u)$ the length of the shortest weighted path from C to u. If there is any u among the nodes such that $q(u) = 1$ and $max(d_C(u), d_S(u)) < t$, then we can meet the broker in time.

This takes $O(m+n \log n)$ for two passes of Dijkstra's algorithm. The running time analysis of Dijkstra's is applicable because the weights are non-negative.

Rubric: 10 points:

- -7 pts for anything that runs $O(nm)$ or $O(n^2 \log n + nm)$
- -7 pts for finding the shortest path to each Mac and then from each Mac finding the shortest path to the broker.
- -4 pts for using $d_C(u) + d_S(u) < t$ or min instead of max.
- -3 pts for not checking the $q(u) = 1$.
- -2 pts for minor errors.
- -0 pts for doing $\min_u(\max(d_C(u), d_S(u))) < t$ for $q(u) = 1$.

4.B. (10 PTS.) Before meeting up with the broker, you first have to stop at a bank along the way. Banks are indicated on your graph by a flag $b(u)$, wich is 1 if node u is a bank and 0 otherwise. The time it takes to stop at a bank is a constant t_b, but you may consider it 0. With this additional constraint, answer the same question as in A.

Solution:

We can adapt the answer to part A by graph transformation. We create a new graph G', which consists of two copies of G, call them G_0 and G_1. G_0 corresponds to the path before visiting the bank, and G_1 is the path after visiting the bank. There are 0 weight edges $u_0 \to u_1$ from each $u_0 \in G_0$ to the corresponding node $u_1 \in G_1$ whenever $b(u) = 1$, such that the only way to reach G_1 is to drive by a bank. Thus the shortest path to u that passes by a bank is the shortest path from $S_0 \in G'$ to $u_1 \in G'$. The size of the graph is now $2m + n$ edges and $2n$ nodes, so the running time does not change.

Rubric: 10 points:

- -8 pts for anything that run in $O(n^2m)$ or $O(n^3)$.
- -4 pts for anything that runs $O(nm)$ or $O(n^2 \log n + nm)$
- -4 pts for finding the shortest path to each bank and then from each bank finding the shortest path to each Mac.
- -4 pts if find all pair shortest paths and then compute the bank and Mac that satisfy the constraint in $O(n^2)$.
- -2 pts for using $d_C(u) + d_S(u) < t$ or min instead of max.
- -2 pts for adding an edge for all nodes and not just $b(u) = 1$.
- -2 pts for minor errors.

5 (20 PTS.) Functions

5.A. (8 PTS.) Suppose $f : \mathbb{N} \to \mathbb{Z}$ is a function with a single *strict-zero-crossing* point $n \in \mathbb{N}$, such that:

- $f(x) \geq 0$ is non-negative on the range $0 \leq x < n$,
- $f(n) < 0$ is strictly negative at $x \geq n$,

Furthermore, suppose that $f(x)$ can be computed in running time $O(1)$ for any x. Describe and analyze algorithm that can compute n, the strict-zero-crossing of f. Your algorithm must terminate in time $O(\log n)$.

As an example, given a function f partially defined by the following table, your algorithm should output $n = 4$.

$f(0)$	$f(1)$	$f(2)$	$f(3)$	$f(4)$	$f(5)$	$f(6)$	...
3	0	1	9	-1	0	-2	...

Note that although you can assume n exists, you do not know it n, hence you can not use it as a parameter to your algorithm.

Solution:

We don't have an upper bound for n, but we can find one by exponential search... that is we evaluate $f(2^0), f(2^1), ...$ until reaching a k such that $f(2^k) < 0$. Now we can do standard binary search to find the smallest value $0 \leq n \leq 2^k$ such that $f(n) < 0$. Since 2^k is at most $2n$, this requires $O(\log n)$ evaluations of f, and each can be bounded by $O(1)$, this is bounded by $O(\log n)$ in total.

Rubric: 8 points:

- -8 pts for anything larger than $O(\log n)$.
- -8 pts for assuming n is known.
- -8 pts for wrong algorihtm.
- -6 pts for assuming an upper bound on n is known.
- -4 pts for correct algorithm but wrong analysis.
- -2 pts for minor errors.

5.B. (6 PTS.) Consider the discrete integral of f, that is $g(x) = \sum_{i=0}^{x} f(i)$. Suppose also that $g(x)$ also can be computed directly in time $O(1)$ for any x.

Consider the sequence $G_b = (g_0, g_1, ..., g_b)$ of the first $b+1$ evaluations of g, that is, $g_i = g(i)$, for some fixed $b \in \mathbb{N}$. You can assume $b > n$, hence this range includes the zero-crossing of f.

Describe and analyze an algorithm that takes an integer query $c \in \mathbb{Z}$ and computes the number of times that c occurs in the sequence G_b, in running time $O(\log b)$.

As an example, continuing the example from Part A with $b = 6$ and query $c = 12$, your algorithm should output 2.

g_0	g_1	g_2	g_3	g_4	g_5	g_6	...
3	3	4	13	12	12	10	...

Solution:

Notice that the zero-crossing n of f is also the global maximum of g - before n, g is increasing, after n, g is decreasing. Hence, if $c > g(n)$, output 0. Otherwise, consider the subsequence $[s_0, s_{n-1}]$ over which g is (not strictly) increasing, and $[s_n, s_b]$ over which g is decreasing. Since each is sorted, we can use binary search to find the index of the first and last instance of c in each if it exists.

The algorithm takes $O(1) + O(\log n) + O(\log(b-n)) = O(\log b)$.

Rubric: 6 points:

- -6 pts for anything larger than $O(\log b)$.
- -6 pts for doing binary search on $[s_0, s_b]$.
- -3 pts for correct algorithm but wrong analysis.
- -2 pts for ignoring the fact that c might appear multiple times in $[s_0, s_{n-1}]$.
- -2 pts for minor errors.

5.C. (6 PTS.) Consider the following claim about the skewness of f: just the top 25% largest values of $f(x)$ over the range $0 \leq x < n$ account for more than 90% of the total growth from 0 to $g(n-1)$. Suppose, you can still compute $g(x)$ and $f(x)$ in O(1). Describe and anlyze an algorithm to check whether this claim is true or not. Your algorithm should run in time $O(n)$.

Continuing the example from Part A, B, the largest 25% values are just $\{f(3) = 9\}$, but $g(n-1) = 13$, hence the top 25% largest values account for $\frac{9}{13} = 0.692....$ of the total growth, which is not quite 90% so the claim is false.

Solution:

We can note that the total sum $\sum_{x=0}^{n-1} f(x)$ is just $g(n-1)$, which can be computed in constant time, but the hard part is finding the sum of the top 25% of values. We know that f is non-negative everywhere on this range, but it may not be monotonic. In $O(n)$ time we can compute $f(x)$ on the entire range $[0, n]$. Using $Select(0.75n, f)$ would find us the break point B for the largest 25% of such values. In a second linear pass, we can add up all of the values $f(x)$ such that $f(x) \geq B$. Finally we can compare this sum to $0.9g(n-1)$ to evaluate the claim.

The algorithms takes $O(n)$ to compute $f(x)$, another $O(n)$ to run select, an a final $O(n)$ to compute the sum. Hence, it runs in $O(n)$.

Rubric: 6 points:

- -6 pts for anything larger than $O(n)$.
- -3 pts for correct algorithm but wrong analysis.
- -2 pts for misusing select.
- -2 pts for minor errors.