**1** (20 PTS.) Short questions.

**1.A.** (10 PTS.) Give an asymptotically tight solution to the following recurrence, where $T(n) = O(1)$ for $n < 10$, and otherwise:
$T(n) = T(2n/3) + T(n/2) + O(n^2)$.

## Solution:

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

The easiest way to see that is verifying the inductive hypothesis here. Namely, that $T(n) \leq cn^2$, for some constant $c$ sufficient large. The recurrence can be written as $T(n) \leq T(2n/3) + T(n/2) + c'n^2$, for some constant $c'$. As such, we have that

$$T(n) \leq T(2n/3) + T(n/2) + c'n^2 \leq \frac{4}{9}cn^2 + \frac{1}{4}cn^2 + c'n^2 = \left(\frac{4}{9}c + \frac{1}{4}c + c'\right)n^2 \leq cn^2,$$

which holds $\frac{4}{9}c + \frac{1}{4}c + c' \leq c$, which is equivalent to $c' \leq (1 - 25/36)c$, which is equivalent to $c \geq (36/11)c'$.



**1.B.** (10 PTS.) Given a directed graph $\mathsf{G}$, describe a linear time algorithm that decides if there are three distinct vertices $x, y, z$, such that (i) there is a path from $x$ to $y$ in $\mathsf{G}$, (ii) there is a path from $y$ to $z$ in $\mathsf{G}$, and (iii) there is a path from $z$ to $x$ in $\mathsf{G}$.



## Solution:



Compute the SCCs of $\mathsf{G}$. If there is a SCC with three or more vertices in it, then there are three such vertices (any three vertices in this SCC would do).

**2** (20 PTS.) Given a directed graph $\mathsf{G} = (V, E)$ with positive edge lengths. Let $\ell(u, v)$ be the length of edge $(u, v) \in E$, and let $d(u, v)$ be the length of the shortest path from $u$ to $v$ in $\mathsf{G}$. Given two nodes $s$ and $t$, there might be many different paths that realize the shortest path between $s$ and $t$, and let $\Pi$ be the set of all such paths. A vertex is ***useful*** if it lies on any path of $\Pi$. Describe how to compute, as fast as possible, all the useful vertices in $\mathsf{G}$ (given $s$ and $t$). What is the running time of your algorithm.

## Solution:

Compute the shortest path from $s$ to all the vertices of $\mathsf{G}$ using Dijkstra. Denotes this distance by $d(v)$, for each $v \in v$. Next, compute the shortest path distance from $t$ to all the vertices of reverse($\mathsf{G}$) (the reverse edges have the same weight). Let $d'(v)$ be the distance of $t$ to $v$ in the reverse graph. Let $d = d(s, t)$.

A vertex $v$ is useful if $d(v) + d'(v) = d$. Clearly, we can do this check by scanning the vertices in linear time. The overall running time of the algorithm is $O(n \log n + m)$ since we have to do Dijkstra twice.

**3** (20 PTS.) Suppose you are given a sorted array of $n$ distinct numbers that has been rotated right by $k$ steps, for some *unknown* integer $k$ between 1 and $n-1$. That is, you are given an array $A[1...n]$ such that some prefix $A[1...k]$ is sorted in increasing order, the corresponding suffix $A[k+1...n]$ is also sorted in increasing order, and $A[n] < A[1]$.

For example, the below array with $n = 10$ has been rotated by $k = 7$.

| 35 | 65 | 108 | 197 | 303 | 499 | 833 | 3 | 4 | 19 |
|---|---|---|---|---|---|---|---|---|---|

Given a number $x$, describe an algorithm, as fast as possible, that decides if $x$ appears somewhere in the $A$. What is the running time of your algorithm? Argue that your algorithm is correct.

## Solution:

**Solution I: Compute $k$, and then do binary search.**

Let $f(y) = \begin{cases} 1 & y < A[1] \\ 0 & y \geq A[1] \end{cases}$. Clearly, if we apply $f$ to all the elements of $A$, we get an array that is a prefix of 0s followed by a run of 1s. We can find the $k$ such that $f(A[k-1]) = 0$ and $f(A[k]) = 1$ in $O(\log n)$ time using binary search. Now, the two subarray $A[1 \ldots k]$ and $A[k+1 \ldots n]$ are sorted, one can check if $x$ is in either array by doing binary search in both arrays. Since we are doing three binary searches in sorted arrays, this can be done in $O(\log n)$. Observe that one needs to search only in one of two subarray, by inspecting the range of numbers in each subarray, but this is a minor unimportant improvement.



## Solution:

**Solution II: Direct binary search without computing $k$.**



Let $A[1 \ldots n]$ be the given array. A subarray $A[i \ldots j]$ is ***idiotic*** if it is a shifted array of a sorted array as described above. A subarray can be decided if it is idiotic or not by checking if $A[i] > A[j]$.

Consider an idiotic subarray $A[i \ldots j]$. Let $k = \lfloor (i+j)/2 \rfloor$. If $j - i \leq 10$, then we can just check if it contains $x$ by scanning. That takes $O(1)$ time. Otherwise, there are two possibilities:

(A) $A[i \ldots, k]$ is idiotic, and $A[k+1 \ldots j]$ is monotonically increasing.
(B) $A[i \ldots, k]$ is monotonically increasing, and $A[k+1 \ldots j]$ is idiotic.

We can decide if we are in case (A) or case (B) in constant time. We can check if the number $x$ is in the range of the numbers of sorted subarray (by just comparing $x$ to the top and bottom number in this array). If $x$ is in this range, then we just perform binary search on this subarray and return this result. Otherwise, we recurse on the idiotic subarray. Clearly, this takes $O(\log n)$ overall.

**4** (20 PTS.) We are given a sequence of $n$ numbers $A[1], \ldots, A[n]$, and integers $g$ and $\ell$ with $\ell \geq n/g$. We want to choose a subsequence $A[i_1], \ldots, A[i_\ell]$ of length $\ell$, such that $i_1 = 1$, $i_\ell = n$, and $1 \leq i_{j+1} - i_j \leq g$ for all $j = 1, \ldots, \ell - 1$, while minimizing the sum $A[i_1] + \cdots + A[i_\ell]$.

Example: for the input sequence $0, 4, 3, 1, 11, 8, 5, 2$ and $g = 3$ and $\ell = 5$, we could pick $0 + 1 + 8 + 5 + 2 = 15$, but the optimal solution has sum $0 + 3 + 1 + 5 + 2 = 11$.

Describe an algorithm, as fast as possible, to compute the optimal sum, by using dynamic programming. (You do not need to output the optimal subsequence.) 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 as a function of $n$, $g$, and $\ell$.

## Solution:

Let $f(i,t)$ be the cheapest collection of entries of $A[1\ldots i]$ that is a valid solution – that is, $A[1]$ and $A[i]$ are included and the distance between two consecutive values is at most $g$ in the array. We have that

$$f(i,t) = \begin{cases} +\infty & i < 1 \\ A[1] & i = 1 \text{ and } t = 1 \\ +\infty & i = 1 \text{ and } t > 1 \\ +\infty & i > 1 \text{ and } t = 1 \\ A[j] + \min_{j=i-g,\ldots,i-1} f(j,t-1) & \textit{otherwise.} \end{cases}$$

We are interested in computing $f(n,\ell)$, and clearly using memoization this takes $O(ng\ell)$ time. For an explicitly dynamic programming solution, we have:



```
// B[1...n][1...,ℓ] is a global array
eval( i, t):
  if i < 1 then return +∞
  if i = 1 and t = 1 then return A[1]
  if i = 1 and t > 1 return +∞
  if i > 1 and t = 1 return +∞
  return B[i,t]

B[1,1] ← A[1]
for i = 2,...,n do B[i,1] ← +∞

for i = 1,...,n do
  for t = 2,...,ℓ do
    α = ∞
    for j = min(i − g, 1),...,i − 1 do
      α = min(α, eval(j, t − 1))
    B[i,t] ← A[i] + α


return B[n,ℓ]
```

The running time is clearly $O(ng\ell)$.

## Solution:

**A sketchy sketch of a faster solution.** One can solve this in $O(n\ell)$ time, by observing that we essentially need to maintain $\ell$ sliding windows in the array $B$, each of length $g$, and maintain the minimum element in each of the sliding windows. To this end, for an array $C[1 \ldots, g]$ an element $C[i]$ is a *winner*, if it is smaller than all the elements after it (i.e., $C[i+1 \ldots g]$). We maintain a doubly linked list of winners. Note that the list of winners is an increasing list of values. Imagine now inserting a new element $\beta$ at $C[g+1]$ into this data structure. All the winners that are bigger than $\beta$ can be removed from the linked list, and we add $\beta$ to the tail of the list. We then delete $C[1]$ from the list of winners if it in the list. This corresponds to inserting one element into the sliding window and moving the sliding window one position to the right. This takes $O(1)$ time.

We also need to shift the elements of $C$ to the left, but this can be simulated by using a circular array.

Namely, we can easily maintain a sliding window over values where we insert a value, and we maintain the minimum element in the last $g$ values seen. Note, that getting the minimum element can be done in $O(1)$ time, and an insertion takes $O(1)$ time

We can now create such sliding window data structure for each row $B[1 \ldots n][t]$ of $B$. It is now easy to verify that one can modify the dynamic program so that it uses these sliding windows data-structures instead of the inner loop. The running time improves to $O(n\ell)$.



**5** (20 PTS.) You are given a directed graph $\mathsf{G}$ with $n$ vertices and $m$ edges ($m \geq n$), where each edge $e$ has an integer weight $w(e)$ (which could be positive or negative) and each vertex is marked "red" or "blue". You are also given a (small) positive integer $b$.



Describe an algorithm, as fast as possible, to find a walk with the smallest total weight, such that the start vertex is red, the end vertex is red, and the number of blue vertices is divisible by $b$ (with no restrictions on the number of red vertices). Your solution should involve constructing a new graph and applying a known algorithm on this graph. Analyze the running time as a function of $n$, $m$, and $b$.



## Solution:

Let $\mathsf{G} = (\mathsf{V}, \mathsf{E})$. Let

$$\mathsf{V}' = \{(v, i) \mid v \in \mathsf{V}, i = 0, \ldots b-1\} \cup \{s, t\},$$

We define several sets of edges. First the set of edges that do not involve entering a blue vertex – we remain on the same floor:

$$\mathsf{E}_1 = \Big\{(v,i) \to (u,i) \;\Big|\; (v \to u) \in \mathsf{E} \text{ and } u \text{ is not blue, for } i = 0, \ldots, b-1\Big\}.$$

If we enter a blue vertex, we need to go up to the next floor:

$$\mathsf{E}_2 = \Big\{(v,i) \to (u, (i+1) \bmod b) \;\Big|\; (v \to u) \in \mathsf{E} \text{ and } u \text{ is blue, for } i = 0, \ldots, b-1\Big\}.$$

The edges for going to any red vertex in the beginning are

$$\mathsf{E}_3 = \{s \to (v, 0) \mid v \text{ is red}\}.$$

The edges to go from a red vertex to the end vertex:

$$\mathsf{E}_4 = \{(v, 0) \to t \mid v \text{ is red}\}.$$

The resulting graph $\mathsf{H} = (\mathsf{V}', \mathsf{E}_1 \cup \mathsf{E} - 2 \cup \mathsf{E}_3 \cup \mathsf{E}_4)$ has $nb + 2$ vertices, and $2n + mb$ edges.

We set the weights of the edges from $s$ and to $t$ to be zero. All other edges have weight that corresponds to their original weight.

We now need to solve the shortest path problem in $\mathsf{H}$. This can be done in $O(nb \log(nb) + mb)$ time, using Dijkstra. However, the graph might have negative weights, so we need to use Bellman-Ford, and this takes $O((nb)(mb)) = O(nmb^2)$ time.

**Correctness.** A path from $s$ to $(v, i)$ represents a walk that uses $i$ blue vertices (modulo $b$). As such, we are looking for the shortest path that starts at a red vertex (at level 0), and ends at a red vertex at level 0. Such a path corresponds to a shortest path between $s$ and $t$.