1. Suppose you are given a two-dimensional array $M[1..n, 1..n]$ of numbers, which could be positive, negative, or zero, and which are *not* necessarily integers. The ***maximum subarray problem*** asks to find the largest sum of elements in any contiguous subarray of the form $M[i..i', j..j']$.

   (a) For any indices i and j, let *Sum*(i, j) denote the sum of all elements in the subarray $M[1..i, 1..j]$. Describe an algorithm to compute *Sum*(i, j) for all indices i and j in $O(n^2)$ time.

Solution: The function *Sum* satisfies the following recurrence:

$$Sum(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ M[i,j] + Sum(i-1,j) & \text{otherwise} \\ \qquad + Sum(i,j-1) - Sum(i-1,j-1) & \end{cases}$$

We can memoize thins function into a two-dimensional array $Sum[0..n, 0..n]$, which we can evaluate in standard row-major order in $O(n^2)$ time.

```
ComputeSums($M[1..n, 1..n]$)
  for $i \leftarrow 0$ to $n$
      for $j \leftarrow 0$ to $n$
          if $i = 0$ or $j = 0$
              $Sum[i,j] \leftarrow 0$
          else
              $Sum[i,j] \leftarrow M[i,j] + Sum(i-1,j)$
                                    $+ Sum(i,j-1) - Sum(i-1,j-1)$
```

■

Rubric: 5 points, standard DP rubric (scaled). Max 3 points for an $O(n^3)$-time algorithm; scale partial credit.

(b) Describe a simple(!!) algorithm to solve the maximum subarray problem in $O(n^4)$ time, using the output of your algorithm for part (a).

Solution: After running the algorithm from part (a), we can compute the sum of the elements in any subarray $M[i\,..\,i', j\,..\,j']$ in constant time by inclusion-exclusion:

$$Sum[i-1, j-1] - Sum[i', j-1] - Sum[i-1, j'] + Sum[i', j']$$

Thus, we can find the maximum subarray in $O(n^4)$ time by pure brute force.

```
MaxBlock($M[1\,..\,n, 1\,..\,n]$)
  ComputeSums($M$)
  $best \leftarrow -\infty$
  for $i \leftarrow 1$ to $n$
      for $i' \leftarrow i$ to $n$
          for $j \leftarrow 1$ to $n$
              for $j' \leftarrow j$ to $n$
                  $sum \leftarrow Sum[i-1, j-1] - Sum[i', j-1]$
                                    $- Sum[i-1, j'] + Sum[i', j']$
                  $best \leftarrow \max\{best, sum\}$
  return $best$
```

■

Rubric: 5 points.

(c) Let L be a horizontal line through M that splits the rows (roughly) in half. Describe an algorithm to find the maximum-sum subarray ***that crosses L*** in $O(n^3)$ time, using the output of your algorithm for part (a). *[Hint: Consider the top half and the bottom half of M separately.]*

Solution (fix left and right sides): For simplicity, assume n is even, and that the line L splits the rows exactly in half. For all indices i and i', we define two functions:

- $BestUp(j, j')$ is the maximum sum of a subarray of the form $M[i\,..\,n/2, j\,..\,j']$
- $BestDn(j, j')$ is the maximum sum of a subarray of the form $M[n/2+1\,..\,i', j\,..\,j']$

Given the indices j and j', we can compute each of these functions in $O(n)$ time by brute force, using the same inclusion-exclusion formula from part (b). The maximum sum of a subarray of the form $M[i\,..\,i', j\,..\,j']$ that crosses L is then

$$BestUp(j, j') + BestDn(j, j').$$

So we can solve the stated problem by brute force in $O(n^3)$ time. We can remove the subexpressions in gray because they cancel out in the final sum $bestup + bestdn$.

```
MaxCrossingSubarray($M[1..n, 1..n]$):
  ComputeSums($M$)
  $best \leftarrow -\infty$                      ⟨⟨global best block sum⟩⟩
  for $i \leftarrow 1$ to $n$
      for $i' \leftarrow i$ to $n$
          $bestup \leftarrow -\infty$            ⟨⟨best block sum above L for fixed i and i'⟩⟩
          for $i \leftarrow 1$ to $n/2$
              $sumup \leftarrow Sum[i-1, j-1] - Sum[i-i, j']$
                                  $- Sum[n/2, j-1] + Sum[n/2, j']$
              $bestup \leftarrow \max\{bestup, sumup\}$
          $bestdn \leftarrow -\infty$            ⟨⟨best block sum below L for fixed i and i'⟩⟩
          for $i' \leftarrow n/2+1$ to $n$
              $sumdn \leftarrow Sum[i', j'] - Sum[i', j-i]$
                              $- Sum[n/2, j'] + Sum[n/2, j-1]$
              $bestdn \leftarrow \max\{bestdn, sumdn\}$
          $best \leftarrow \max\{best, bestup + bestdn\}$
  return $best$
```

■

Rubric: 5 points.

Solution (fix top and bottom sides): For simplicity, assume n is even, and that the line L splits the rows exactly in half. For all indices i and i', let $MaxSum(i,i')$ denote the maximum sum of a subarray of the form $M[i\,..\,i', j\,..\,j']$. Suppose we define a new array $B[1\,..\,n]$ where

$$B[k] = \sum_{h=i}^{i'} M[h,k]$$

for every index k. Then the sum of elements in $M[i\,..\,i', j\,..\,j']$ is also the sum of elements in $B[j\,..\,j']$. Thus, computing $MaxSum(i,i')$ is equivalent to solving the *one-dimensional* maximum subarray problem for array B. So we need two ingredients here:

- For any indices i and j, let $Above(i,j) = \sum_{h=1}^{i} M[h,j]$. We can easily compute $Above(i,j)$ for all i and j in $O(n^2)$ time by dynamic programming. Then we immediately have

$$\sum_{h=i}^{i'} M[h,k] = Above(i',k) - Above(i-1,k).$$

  Thus, for any fixed indices i and i', we can populate the one-dimensional array B in $O(n)$ time. Alternatively, since we can compute each $B[k]$ in constant time, we don't have to *build* the array B at all; we can just compute the entries on the fly when we need them.
- The one-dimensional maximum subarray problem can be solved in $O(n)$ time using the following algorithm of Kadane, as described on Wikipedia:

```
KadaneMaxSubarray($B[1..n]$):
    $best \leftarrow 0$
    $sum \leftarrow 0$
    for $i \leftarrow 1$ to $n$
        $sum \leftarrow \max\{0, sum + A[i]\}$
        $best \leftarrow \max\{best, sum\}$
    return $best$
```

Putting these ingredients together, we obtain the following algorithm, which clearly runs in $O(n^3)$ time.

```
MaxCrossingSubarray($M[1..n, 1..n]$):
    for $j \leftarrow 1$ to $n$
        $Above[0,j] \leftarrow 0$
        for $i \leftarrow 1$ to $n$
            $Above[i,j] \leftarrow M[i,j] + Above[i-1,j]$
    $best \leftarrow 0$
    for $i \leftarrow 1$ to $n/2$
        for $i' \leftarrow n/2+1$ to $n$
            $sum \leftarrow 0$
            for $j \leftarrow 1$ to $n$
                $sum \leftarrow \max\{0, sum + Above[i',j] - Above[i-1,j]\}$
                $best \leftarrow \max\{best, sum\}$
    return $best$
```

■

Rubric: 5 points. No, you don't have to prove Kadane's algorithm is correct.

(d) Describe a divide-and-conquer algorithm to find the maximum-sum subarray in M in $O(n^3)$ time, using your algorithm for part (c) as a subroutine. *[Hint: Why is the running time $O(n^3)$ and not $O(n^3 \log n)$?]*

Solution: Intuitively, the maximum subarray either crosses L, lies entirely above L, or lies entirely below L. We use the algorithm in part (c) to find the best block that crosses L, and we recursively find the best blocks above L and below L.

Suppose we use the first solution for part (c). After the first level of recursion, the array is no longer square, so we have to consider rectangular arrays. If the input array has m rows and n columns, the subroutine MaxCrossingSubarray runs in $O(m^2 n)$ time. Implementing the divide-and-conquer strategy directly would give us the run-time recurrence

$$T(m,n) = O(m^2 n) + 2T(m/2, n),$$

which implies $T(m,n) = O(m^2 n \log m)$ (via recursion trees) and therefore $T(n,n) = O(n^3 \log n)$.

To avoid the additional logarithmic factor, we modify the algorithm to alternately cut along horizontal and vertical lines in successive levels of recursion. (The algorithm for cutting along vertical lines is almost identical to part (c).) After two levels of recursion, the resulting subproblems are again square, so we don't need to worry about arbitrary rectangular arrays. The running time of the resulting algorithm satisfies the recurrence

$$T(n) = O(n^3) + 4T(n/2),$$

which implies $T(n) = O(n^3)$ (via recursion trees), as required. ■

Solution: Intuitively, the maximum subarray either crosses L, lies entirely above L, or lies entirely below L. We use the algorithm in part (c) to find the best block that crosses L, and we recursively find the best blocks above L and below L.

Suppose we use the second solution for part (c). After the first level of recursion, the array is no longer square, so we have to consider rectangular arrays. If the input array has m rows and n columns, the subroutine MaxCrossingSubarray runs in $O(m^2 n)$ time. Implementing the divide-and-conquer strategy directly would give us the run-time recurrence

$$T(m,n) = O(mn^2) + 2T(m/2, n),$$

which implies $T(m,n) = O(mn^2)$ and therefore $T(n,n) = O(n^3)$, as required. ■

Rubric: 5 points = 3 for algorithm + 2 for time analysis. Max 3 points for $O(n^3 \log n)$-time algorithm = 2 for algorithm + 1 for time analysis.

2. Describe and analyze an algorithm to determine who wins the "Hello, Sweetie!" game, assuming both players play perfectly.

Solution (dynamic programming): We start by topologically sorting the input dag G, because that's *always* the first thing one does with a dag. Topological sort labels the vertices with integers from 1 to V, so that every edge points from a lower label to a higher label. Because s is the only source, its label is 1, and because t is the only sink, its label is V.

We represent the two players with booleans: True means the Doctor, and False means River. For any vertices d and r and any boolean *who*, let $\mathit{WhoWins}(d, r, \mathit{who})$ denote the winning player when the Doctor's token starts on d, River's token starts on r, player *who* moves first, and both players play perfectly. We need to compute $\mathit{WhoWins}(s, t, \text{True})$.

If the game is not over, then the Doctor wins moving first if and only if *at least one* move by the Doctor leads to a position where the Doctor wins moving second, and the Doctor wins moving second if and only if *every* move by River leads to a position where the Doctor wins moving first.[1] Thus, the function *WhoWins* can be computed by the following recursive algorithm:

```
WhoWins(d, r, who):
  if $d = r$
      return True
  else if $d = t$ or $r = s$
      return False
  else if $who$ = True
      return $\bigvee_{d \to v} \mathit{WhoWins}(v, r, \text{False})$
  else if $who$ = False
      return $\bigwedge_{v \to r} \mathit{WhoWins}(d, v, \text{True})$
```

Thanks to our vertex labeling, we can memoize this function into a $V \times V \times 2$ array, indexed by the variables d, r, and *who* in that order. We can fill the array with two nested for loops, decreasing d in one loop and increasing r in the other, considering both players inside the inner loop. The nesting order of the two for-loops doesn't matter. Explicit pseudocode appears on the next page.

For any node v in G, let $\mathit{indeg}(v)$ denote the number of edges entering v (the *in-degree* of v), and let and $\mathit{out}(v)$ denote the number of edges leaving v (the *out-degree* of v). For almost every pair of vertices d and r, our algorithm considers all $\mathit{outdeg}(d)$ possible moves for the Doctor and then all $\mathit{in}(r)$ possible moves for River. Thus, the total running time of our algorithm is at most

$$\sum_{d=1}^{V}\sum_{r=1}^{V} O\left(\mathit{outdeg}(d) + \mathit{indeg}(r)\right).$$

Ignoring the big-Oh constant, we can evaluate this sum in two pieces:

$$\sum_{d,r} \mathit{outdeg}(d) = \sum_{r}\left(\sum_{d} \mathit{outdeg}(d)\right) = \sum_{r} E = VE$$

$$\sum_{d,r} \mathit{indeg}(r) = \sum_{d}\left(\sum_{r} \mathit{indeg}(r)\right) = \sum_{d} E = VE$$

[1]This is the recursive *definition* of "play perfectly", for *any* finite two-player game that cannot end in a draw.

Less formally, our algorithm considers all VE pairs (Doctor's position, River's move) and all VE pairs (Doctor's move, River's position), spending $O(1)$ time on each pair. We conclude that our algorithm runs in $\boldsymbol{O(VE)}$ ***time***.

```
WhoWins(V, E):
  label vertices of $G$ in topological order
  for $d \leftarrow V$ down to 1
      for $r \leftarrow 1$ to $V$
          if $d = r$
              $WhoWins[d, r, \text{True}] \leftarrow \text{True}$
              $WhoWins[d, r, \text{False}] \leftarrow \text{True}$
          else if $d = t$ or $r = s$
              $WhoWins[d, r, \text{True}] \leftarrow \text{False}$
              $WhoWins[d, r, \text{False}] \leftarrow \text{False}$
          else
              $WhoWins[d, r, \text{True}] \leftarrow \text{False}$
              for all edges $d \rightarrow v$
                  $WhoWins[d, r, \text{True}] \leftarrow WhoWins[d, r, \text{True}] \vee WhoWins[v, r, \text{False}]$
              $WhoWins[d, r, \text{False}] \leftarrow \text{True}$
              for all edges $v \rightarrow r$
                  $WhoWins[d, r, \text{False}] \leftarrow WhoWins[d, r, \text{False}] \wedge WhoWins[d, v, \text{True}]$
  return $WhoWins[s, t, \text{True}]$
```

■

Rubric: 10 points, standard dynamic programming rubric. −1 for looser time bounds like $O(V^3)$ or $O(V^2 D)$ where D is maximum degree.

This is not the only correct description of this algorithm. For example, instead of topologically sorting the dag at the beginning, we can discover the correct traversal orders on the fly via depth-first search:

```
WhoWins(V, E):
  for all vertices $d$ in postorder                  ⟨⟨via $DFS(G, s)$⟩⟩
      for all vertices $r$ in reverse postorder      ⟨⟨via $DFS(rev(G), t)$⟩⟩
          ⟨⟨and so on⟩⟩
```

We can also arguably simplify the recurrence logic by asking whether the *first* player wins and always NANDing the results of recursive calls, instead of alternating between ANDs and ORs.

This solution is more detailed than necessary for full credit. In particular, explicit iterative pseudocode is **not** required for full credit; see the dynamic programming rubric in HW1.

No penalty for implicitly assuming that the input graph has more than 10^{100} vertices; we can solve the small cases by brute force in $O(1)$ time. No penalty for assuming without proof that $V = O(E)$; because G has only one source, G must be connected, and therefore $E \geq V - 1$.

Solution (configuration graph search): First we construct the configuration graph $H = (V', E')$, which contains a vertex for every possible game configuration and a directed edge for every legal move. Specifically:

- $V' = V \times V \times \{0, 1\}$. Each vertex (d, r, who) represents the configuration where the Doctor's token is at node d, River's token is at node r, and it is the Doctor's turn if and only if $who = 1$. The number of vertices in H is $2V^2$.
- $E' = \big\{(d,r,1)\to(d',r,0) \bigm| d\to d' \in E\big\} \cup \big\{(d,r,0)\to(d,r',1) \bigm| r'\to r \in E\big\}$. Each edge represents a legal move by the appropriate player. The number of edges in H is $\sum_r \sum_d outdeg(d) + \sum_d \sum_r indeg(r) = 2VE$.

If H contains a directed cycle, it must have the form

$$(d_0, r_0, 1)\to(d_1, r_0, 0)\to(d_1, r_1, 1)\to\cdots\to(d_k, r_k, 1)\to(d_0, r_k, 0)\to(d_0, r_0, 1).$$

But then the original input dag would contain the cycles

$$d_0\to d_1\to\cdots\to d_k\to d_0 \qquad \text{and} \qquad r_0\to r_k\to\cdots\to r_1\to r_0,$$

which is impossible. We conclude that H is also a dag.

For each vertex (d, r, who) let $WhoWins(d, r, who) = \text{True}$ if the Doctor wins from the configuration (d, r, who) if both players play perfectly; otherwise, let $WhoWins(d, r, who) = \text{False}$. We need to compute $WhoWins(s, t, 1)$. We can evaluate this function recursively as follows:

```
$WhoWins(d, r, who)$:
  if $d = r$
      return True
  else if $d = t$ or $r = s$
      return False
  else if $who = 1$
      return $\bigvee_{(d,r,1)\to(d',r,0)} WhoWins(d', r, 0)$
  else if $who = 0$
      return $\bigwedge_{(d,r,0)\to(d,r',1)} WhoWins(d, r', 1)$
```

We can memoize this function into the vertices of H themselves, and we can evaluate this function at every vertex of H by considering the vertices in reverse topological order (or equivalently, in DFS postorder). The resulting algorithm runs in $O(V' + E') = \boldsymbol{O(VE)}$ ***time***. ■

Rubric: 10 points = 5 for correctly defining H (= 2 for vertices + 2 for edges + 1 for proving H is a dag) + 5 for solving the problem on H (dynamic programming rubric)