1. (•) ***I understand the course policies.***

   Suppose you are given an array A$[1..n]$ of positive integers. Describe and analyze an algorithm to find the smallest positive integer that is not an element of A in $O(n)$ time.

   Solution: A pseudo code of our algorithm is given as follows:

```
SmallestPositiveInteger(A[1..n]):
  for i ← 1 to n+1
      B[i] ← 0
  for i ← 1 to n
      if 1 ≤ A[i] ≤ n
          B[A[i]] ← 1
  for i ← 1 to n+1
      if B[i] = 0
          return i
```

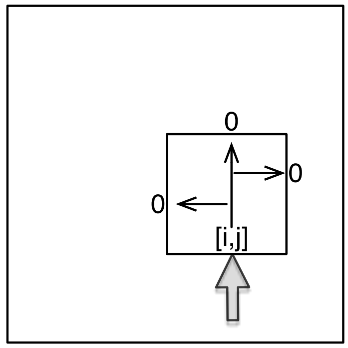
   Running time: $O(n)$

   Space: $O(n)$ ■

2. (•) ***I understand the course policies.***

   Describe and analyze an efficient algorithm to find a solid block in M with maximum area.

Solution: We first observe that given a point at (i, j), we can define a solid block by extending it until encountering a '0', then extend it to left right until the line reaches a '0'. If we can find all these solid blocks defined by (i, j), then we can certainly find the largest solid block among them.



To do this, we first compute the largest solid blocks for a single row and column. Specifically, we define $top(i, j)$, $left(i, j)$ and $right(i, j)$ as the largest (unit width) block from (i, j) to the *top*, *left* and *right* side. For example, for a row `[1 0 1 1 1]`, its corresponding *left* array is `[1 0 1 2 3]`. We can use dynammic programming to compute it as follows:

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

To handle the base case of $left(i,1)$, we just add a sentinel column 0 and let $left(i,0) = 0$. Similarly, $top(i, j)$ and $right(i, j)$ can be computed like $left(i, j)$. We use a 2D-array to store $top(i, j)$ (However, later we only need the *left* and *right* information for the row that we are working on, we can just use a 1D-array to store them). The evaluation order is just from *left* to *right* for $left(i, j)$, *right* to *left* for $right(i, j)$, and *top* down for $top(i, j)$.

Next, we compute a left *extension* and right *extension* for a given point (i, j). They denote the *left*most and *right*most distance the block can extend to after reaching the *top*, respectively. Specifically, *left* extension can be simply computed as:

$$lext(i,j) = min\{lext(i-1,j), left(i,j)\}$$

$rext(i, j)$ is defined similarly. For the base cases, we set $left(0, j) = rext(0, j) = +\infty$ for all j as sentinel. Then, for each pair of i and j, we compute the area of solid block defined by (i, j) as $area(i, j) = (rext(i, j) - lext(i, j) + 1) \times top(i, j)$. We do not need to remember $area(i, j)$ at all beacuse we can just keep track of the largest block we have ever found as we go. This takes $O(mn)$ time. After the scanning if over, we are done. The computation for $top(i, j)$, $left(i, j)$, $right(i, j)$, $lext(i, j)$ and $rext(i, j)$ can be finished in $O(mn)$ time. We can put them together to avoid excessive looping through (i, j) pairs. Hence, the overall runtime is $O(mn)$, so as the space. Note that for the $top[i, j], lext[i, j]$ and $rext[i, j]$ array, since we only need the information for previous row, we can just use a sliding window technique to reduce the final space to $O(m)$. The pseudo code is described as follows. For clarity, we intentionally do not use the sliding window technique in it.

```
LargestSolidBlock(M[1..m, 1..n]):
  for j ← 1 to n                ⟨⟨Base cases⟩⟩
      lext[0, j] ← +∞
      rext[0, j] ← +∞
  largest ← 0
  for i ← 1 to m
      left[0] ← 0
      for j ← 1 to n            ⟨⟨Compute left array⟩⟩
          if M[i, j] = 1
              left[j] ← left[j − 1] + 1
          else
              left[j] ← 0

      right[n] ← 0
      for j ← n down to 1       ⟨⟨Compute right array⟩⟩
          if M[i, j] = 1
              right[j] ← right[j + 1] + 1
          else
              right[j] ← 0

      for j ← 1 to n
          if M[i, j] = 1         ⟨⟨compute top array⟩⟩
              top[i, j] ← top[i − 1, j] + 1
          else
              top[i, j] ← 0
          lext[i, j] ← min{lext[i − 1, j], left[j]}
          rext[i, j] ← min{rext[i − 1, j], right[j]}
          area ← (rext[i, j] − lext[i, j] + 1) × top[i, j]
          largest ← max{largest, area}
  return largest
```

■

3. (•) ***I understand the course policies.***

Describe and analyze an algorithm to compute a maximum weight matching, given the tree T as input.

Solution: We assume that the problem statement is asking for a maximum matching in a weighted graph, i.e., match as many edges as possible, if there is tie, return the matching that has the largest weight. Given a tree T that is rooted in r. We denote it as $T(r)$. Let c be a child of r, we define $T(r) \setminus T(c)$ as the tree that is obtained by removing $T(c)$ from $T(r)$ as shown in Figure 1.

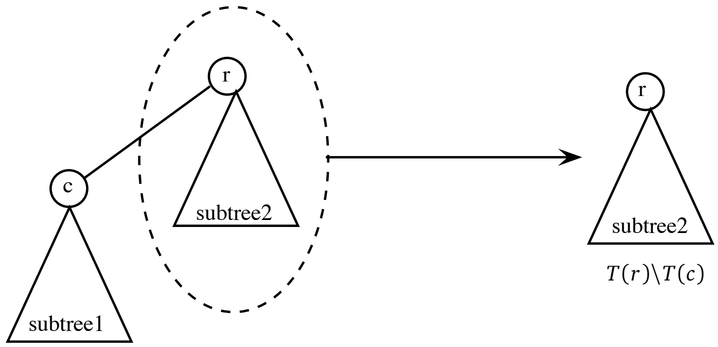
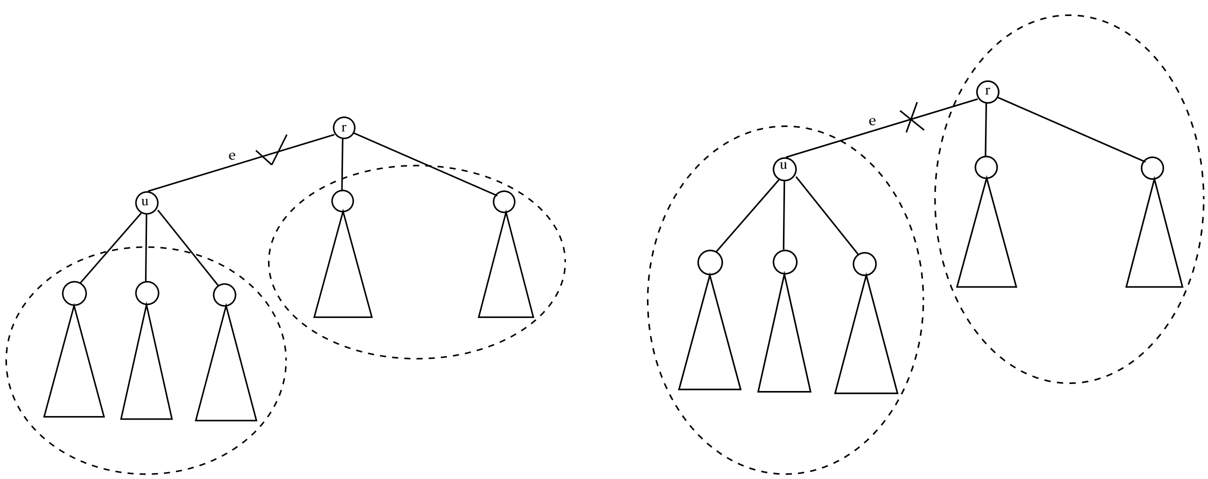


Figure 1. Illustration of $T(r)$ and $T(r)\setminus T(c)$.

Consider an edge $e = (u, r)$ in $T(r)$, there are two possible cases:

(a) **If e is included in the maximum weight matching:** The maximum weight matching of T will then be the summation of the maximum weight matching of all the trees $T(x)$ for all x that is a child of u and the maximum weight matching of all the trees $T(y)$ for all y that is a child of r and $y \neq u$. An illustration is given in Figure 2(a).

(b) **If e is not in the maximum weight matching:** The maximum weight matching of T will then be the summation of the maximum weight matching of $T(u)$ and the maximum weight matching of $T(r) \setminus T(u)$, An illustration is given in Figure 2(b).



(a) Edge e is included in the maximum weight matching. (b) Edge e is not in the maximum weight matching.

Figure 2. Two cases for edge $e = (u, r)$

Hence, we can devise a dynammic programming algorithm to compute the maximum weight matching. We can use the tree itself to memoize the intermediate result. Specifically, for each node v, we can add a new field *v.MWM* to store the weight of the maximum weight matching, and a new field *v.to* to remember to which node v is matched. A pseudo code is given as follows:

```
MaximumWeightMatching(T(v)):
  if v has no child
      return 0
  if v.MWM is computed
      return v.MWM
  u ← any child of v
  e ← (u, v)
  withe ← w(e)
  for each child w of u
      withe ← withe + MaximumWeightMatching(T(w))
  for each child x of v and x ≠ u
      withe ← withe + MaximumWeightMatching(T(x))
  withoute ← MaximumWeightMatching(T(u)) + MaximumWeightMatching(T(v) \ T(u))
  if withe ≥ withoute
      v.MWM = withe
      v.to = u
  else
      v.MWM = withthoute
  return v.MWM;
```

Running time: Because each vertex is visited at most once, the running time of MaximumWeight-Matching($T(v)$) is $O(n)$. To output the maximum weight matching, we just perform a traversal to the tree T and output the matching pairs according to *v.to* for all v. This also takes $O(n)$ time. Hence, the total running time is $O(n)$.

Space: Since we only add a constant amount of storage to each node and there are n nodes, the space complexity is $O(n)$.

∎

4. (•) ***I understand the course policies.***

   Describe and analyze an algorithm that accepts three strings x, y, and z as input, and decides whether z is an interleaving of x and y.

Solution: We first *repeat* x to obtain a string X that has the same length as z, similarly we obtain a string Y by repeating y. We define $\mathit{IsInterleave}(X[1..i], Y[1..j], z[1..i+j])$ as follows:

$$\begin{aligned}&\mathit{IsInterleave}(X[1..i], Y[1..j], z[1..i+j])\\ =\ &\begin{cases}\mathit{IsInterleave}(X[1..i-1], Y[1..j], z[1..i+j-1]) & \text{if } X[i] = z[i+j], \textbf{ or}\\ \mathit{IsInterleave}(X[1..i], Y[1..j-1], z[1..i+j-1]) & \text{if } Y[j] = z[i+j]\\ \mathit{false} & \text{otherwise}\end{cases}\end{aligned}$$

or, in a more compact form:

$$\begin{aligned}&\mathit{IsInterleave}(X[1..i], Y[1..j], z[1..i+j])\\ =\ &[X[i] = z[i+j]]\ \mathit{IsInterleave}(X[1..i-1], Y[1..j], z[1..i+j-1]) \quad \vee\\ &[Y[j] = z[i+j]]\ \mathit{IsInterleave}(X[1..i], Y[1..j-1], z[1..i+j-1])\end{aligned}$$

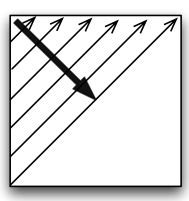
Let i and j denote the length contribution from x and y to z, respectively. In order to find if z is an interleaving of x and y, we only need to check all the possible combination of i and j, and returns *true* if any one of $\mathit{IsInterleave}(X[1..i], Y[1..j], z[1..n])$ is *true*. Clearly we can use a 2D-array to memoize the recurrence. This recurrence contains three base case:

(a) It is trivially true when z is an empty string

(b) It is true if $X[1..i]$ is a *prefix* of z for all $i \in [1..n]$ and $j = 0$

(c) It is true if $Y[1..j]$ is a *prefix* of z for all $j \in [1..n]$ and $i = 0$

Hence, we can just slightly modify the original recurrence to $\mathit{IsInterleave}(X[0..i], Y[0..j], z[0..i+j])$, and add a row and a column with index 0 to the memoization data structure. The evaluation order is illustrated as follows:

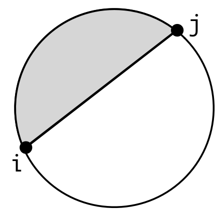


Clearly, this algorithm costs $O(n^2)$ time and uses $O(n^2)$ space. ■

5. (•) ***I understand the course policies.***

   Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

   Solution: We define the maximum reward we can get between a range of snails from i to j as $\mathit{MaxReward}(i, j)$, as the shaded region in the figure:

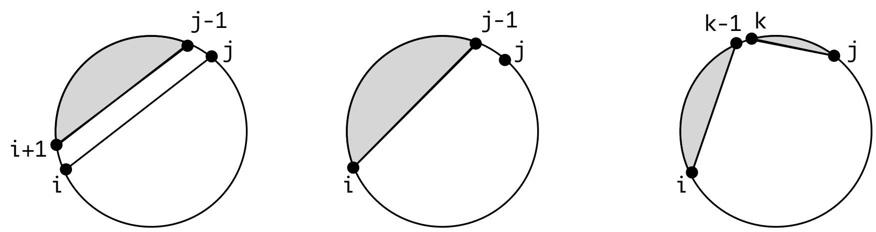
   

   There are three possibilities for $\mathit{MaxReward}(i, j)$:

   (a) If snail i and j make up in the optimal solution, then the optimal solution is equal to $M[i, j] + \mathit{MaxReward}(i+1, j-1)$, where $M[i, j]$ is the reward for pair i and j.

   (b) If snail j is not pair up with some snail in range $[i..j]$, then the optimal solution is just equal to $\mathit{MaxReward}(i, j-1)$.

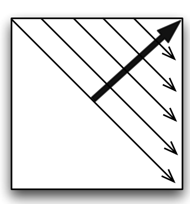
   (c) If snail j pairs up with some snail in range $[i+1..j-1]$, then the optimal solutin will be $max\{MaxReward(i, k-1) + \ MaxReward(k, j)\}$, for all k in $[i+1..j-1]$.

   Hence, the maximum reward can be defined using following recursion:

$$\mathit{MaxReward}(i, j) = \max \left\{ \begin{array}{l} M[i, j] + \mathit{MaxReward}(i+1, j-1) \\ \mathit{MaxReward}(i, j-1) \\ MaxReward(i, k) + \ MaxReward(k, j), \text{ for all } k \in [i+1, j-1]\} \end{array} \right\}$$

   The optimal solution for this problem is $\mathit{MaxReward}(1, n)$. This recurrence has two simple base cases. When i is equal to j, since a single snail can not pair up with itself, thus we have $\mathit{MaxReward}(i, j) = 0$. Also, range $[i..j]$ is invalid when $i > j$, we also set them to 0. We can use a 2D-array to store the values. The evaluation order is illustrated as follows:

   Clearly this algorithm will traverse all pair of i and j where $i < j$, during which a for loop is used to find the maximum value in the third case of the recurrence. Hence, it is a $O(n^3)$ algorithm, which uses $O(n^2)$ space. ■