CSE 202 Homework #0

1. **Problem 1: Maximum sum among nonadjacent subsequences**

   Input and output clarification

   The input is an array V of size n containing real numbers.

   The output is the maximum sum as well as the corresponding subset indices described in the problem statement.

   We assume that the subset could be empty.

   High-level description

   We can use dynamic programming to solve this problem.

   Denote $f[i]$ as the maximum sum when we choose nonadjacent indices from $1, \cdots, i$. To find $f[i]$, we can make a choice on whether to choose index i or not:

   - If we choose index i, we **must not** choose index $i-1$. Thus, the rest indices we can choose are among $1, \cdots, i-2$.
   - If we do not choose index i, it is free to choose indices among $1, \cdots, i-1$.

   Thus we could write down the transition equation:

   $$f[i] = \max(f[i-2] + V[i], f[i-1])$$

   When $i = 1$, both $f[i-2]$ and $f[i-1]$ are undefined entries. We can find it specifically by:

   $$f[1] = \max(V[1], 0)$$

   When $i = 2$, $f[i-2]$ is undefined, but we could assign $f[0] = 0$, which means the maximum sum among nonadjacent subsequences within an empty set of indices is 0.

   The maximum sum is stored in $f[n]$.

   To obtain the corresponding subset indices, we can iterate the f entries in reverse order, starting from $i = n$:

   - If $i = 0$, we stop the iteration.
   - If $i = 1$, we add 1 to the subset if and only if $f[1] = V[1]$. We then stop the iteration.

- If $i \geq 2$, we add i to the subset if and only if $f[i] = f[i-2] + V[i]$. If this equation satisfies, we decrease i by 2, otherwise we decrease i by 1.

Proof of correctness

We could prove by induction that $f[i]$ stores the maximum sum among nonadjacent subsequences in $1, \cdots, i$. By this induction, we can also prove that the subset returned is correct.

- Base case. For $i = 1$, we can either choose or not choose $V[1]$, so:

  $$f[1] = \max(V[1], 0)$$

  holds. If $f[1] = V[1]$, we choose index 1.

  For $i = 2$, we have three options:

  - choose $V[1]$.
  - choose $V[2]$.
  - choose nothing.

  If we assign $f[0] = 0$ as apply $i = 2$ to the previous transition equation, we would get:

  $$f[2] = \max(V[2], f[1])$$

  which is just the maximum among $V[1]$, $V[2]$ and nothing. If $f[2] = V[2]$, we choose index 2, otherwise we give the option to $i = 1$.

- Induction Step. For $i > 2$, if we have calculated both $f[i-2]$ and $f[i-1]$, we can find $f[i]$ by:

  - either choose $V[i]$. Then we must not choose $V[i-1]$, but we can choose from $1, \cdots, i-2$ with no extra constraints. The sum will be $V[i] + f[i-2]$.
  - or do not choose $V[i]$. Then we can choose from $1, \cdots, i-1$ with no extra constraints. The sum will be $f[i-1]$.

  Since $f[i]$ is the maximum between these two entries, it fits the transition equation. If $f[i] = V[i] + f[i-2]$, we choose index i and give the option to $i \leftarrow i-2$, otherwise we give the option to $i \leftarrow i-1$.

Time complexity

The time complexity of this algorithm is $O(n)$. In the first iteration, we calculate n entries to by dynamic programming, and the transition equation has a constant time cost. In the second iteration, we iterative through every index at most once in reverse order. Thus the total time complexity is $O(n)$.

2. **Problem 2: Maximum difference in an array**

Input and output clarification

The input is an array A of size n containing integers. Denote the elements as $A[1], \cdots, A[n]$.

The output is the maximum value described in the problem statement, denoted as T.

We assume that $n \geq 2$, otherwise the answer is undefined.

High-level description

We can find T by using two iterations over array A.

Denote $opt[i]$ as $\max_{1 \leq k \leq i}(A[k])$. The first iteration is used for calculating all $opt[i]$ $(1 \leq i < n)$ entries by the following transition equation:

$$opt[i] = \begin{cases} A[i], & i = 1 \\ \max(opt[i-1], A[i]), & 2 \leq i < n \end{cases}$$

The second iteration is used for finding T. We initialize T as $-\infty$. During the iteration, when we are at index i $(2 \leq i \leq n)$, we calculate $opt[i-1] - A[i]$ and set T as the maximum of this entry and T itself.

After the two iterations, we will obtain the final answer in T.

Proof of correctness

By definition, we can find T by:

$$T = \max_{1 \leq i < j \leq n}(A[i] - A[j]) = \max_{2 \leq j \leq n}\left(\max_{1 \leq i < j}(A[i] - A[j])\right) = \max_{2 \leq j \leq n}\left(\max_{1 \leq i < j}(A[i]) - A[j]\right) \quad (1)$$

In the first iteration, we can prove $opt[i] = \max_{1 \leq k \leq i}(A[k])$ by induction:

- Base case. For $i = 1$, we have $opt[1] = A[1]$ as definition.
- Induction step. For $i \geq 2$, the maximum entry among $A[1], \cdots, A[i]$ is either the maximum entry among $A[1], \cdots, A[i-1]$ (which can be represented by $opt[i-1]$) or $A[i]$ itself, which fits the transition equation.

Thus Equation 1 can be rewritten as:

$$T = \max_{2 \leq j \leq n}\left(opt[j-1] - A[j]\right)$$

which describes exactly what the second iteration does in math.

Time complexity

The time complexity of this algorithm is $O(n)$. In the first iteration, we calculate $n-1$ opt entries and the transition equation has a constant time cost. In the second iteration, we iterate index i through $2, \cdots, n$ and each iteration step has a constant time cost. Thus the total time complexity is $O(n)$.

3. **Problem 3: Maximum difference in a matrix**

Input and output clarification

The input is an array M of size $n \times n$ containing integers.
Denote the elements as $M[1,1], \cdots, M[1,n], \cdots M[n,n]$.

The output is the maximum value described in the problem statement, denoted as T.

We assume that $n \geq 2$, otherwise the answer is undefined.

High-level description

We can find T by using two iterations over array M.

Denote $opt[i,j]$ as $\min_{1 \leq a \leq i, 1 \leq b \leq j}(M[i,j])$. The first iteration is used for calculating all $opt[i,j]$ $(1 \leq i,j < n)$ entries by the following transition equation:

$$opt[i,j] = \min(opt[i-1,j], opt[i,j-1], M[i,j])$$

When $i = 1$ or $j = 1$, $opt[i-1,j]$ and $opt[i,j-1]$ may be undefined. We can assign all of them (i.e. $i < 1$ or $j < 1$) as ∞.

The second iteration is used for finding T. We initialize T as $-\infty$. During the iteration, when we are at grid (i,j) $(2 \leq i,j \leq n)$, we calculate $M[i,j] - opt[i-1,j-1]$ and set T as the maximum of this entry and T itself.

After the two iterations, we will obtain the final answer in T.

Proof of correctness

By definition, we can find T by:

$$\begin{aligned} T &= \max_{1 \leq a < c \leq n, 1 \leq b < d \leq n} (M[c,d] - M[a,b]) \\ &= \max_{2 \leq c,d \leq n} \big(\max_{1 \leq a < c, 1 \leq b < d} (M[c,d] - M[a,b]) \big) \\ &= \max_{2 \leq c,d \leq n} \big(M[c,d] - \min_{1 \leq a < c, 1 \leq b < d} M[a,b] \big) \end{aligned} \tag{2}$$

In the first iteration, we can prove $opt[i,j] = \min_{1 \leq a \leq i, 1 \leq b \leq j}(M[i,j])$ by induction:

- Base case. For $(i,j) = (1,1)$, we have $opt[1,1] = \min(\infty, \infty, M[1,1]) = M[1,1]$ as definition.
- Induction step on the first row. For $i = 1$ and $j \geq 2$, the minimum entry among the first j elements in the first row is either the minimum entry among the first $j-1$ elements (which can be represented by $opt[1][j-1]$) or $M[i,j]$ itself. As the transition equation for $(1,j)$ is $opt[1,j] = \min(\infty, opt[1,j-1], M[i,j]) = \min(opt[1,j-1], M[i,j])$, it is expected to be correct.

- Induction step on the first column. For $i \geq 2$ and $j = 1$, the minimum entry among the first i elements in the first column is either the minimum entry among the first $i-1$ elements (which can be represented by $opt[i-1][1]$) or $M[i,j]$ itself. As the transition equation for $(i,1)$ is $opt[i,1] = \min(opt[i-1,1], \infty, M[i,j]) = \min(opt[i-1,1], M[i,j])$, it is expected to be correct.
- Induction step otherwise. For $i, j \geq 2$, the transition equation shows the minimum among $opt[i-1,j]$, $opt[i,j-1]$ and $M[i,j]$. We can find that the union of the range of $(1,1), \cdots, (i-1,j)$ and $(1,1), \cdots, (i,j-1)$ covers every grid in $(1,1), \cdots, (i,j)$ but (i,j). Thus with the extra entry $M[i,j]$, the transition equation is expected to be correct.

Thus Equation 2 can be rewritten as:

$$T = \max_{2 \leq c,d \leq n} \big(M[c,d] - opt[c-1,d-1]\big)$$

which describes exactly what the second iteration does in math.

Time complexity

The time complexity of this algorithm is $O(n^2)$. In the first iteration, we calculate $(n-1)^2$ opt entries and the transition equation has a constant time cost. In the second iteration, we iterate index both i and j through $2, \cdots, n$ and each iteration step has a constant time cost. Thus the total time complexity is $O(n^2)$.

4. **Problem 4: Pond sizes**

Input and output clarification

The input is a matrix A of size $m \times n$ containing positive integers or 0.
Denote the elements as $A[1,1], \cdots, A[1,n], \cdots, A[m,n]$.

The output is a list which contains the sizes of all ponds in the matrix.

High-level description

We iterate through every grid (i,j) in the matrix. If we find a water grid $A[i,j] = 0$, we can start to find the pool containing this water grid.

To find a pool, we can use breadth-first search with a FIFO queue:

- We first push (i,j) into the queue.
- In each search step, we pop a location (i_{cur}, j_{cur}) from the queue and check its adjacent grids. Here "adjacent" means any of the following 8 grids:
  - $(i_{cur}-1, j_{cur}-1), (i_{cur}-1, j_{cur}), (i_{cur}-1, j_{cur}+1)$
  - $(i_{cur}, j_{cur}-1), (i_{cur}, j_{cur}+1)$
  - $(i_{cur}+1, j_{cur}-1), (i_{cur}+1, j_{cur}), (i_{cur}+1, j_{cur}+1)$

  If an adjacent grid (denoted as (i_{adj}, j_{adj})) is inside the matrix and $A[i_{adj}, j_{adj}] = 0$, we push (i_{adj}, j_{adj}) into the queue for further search.
- To prevent redundant search, whenever a grid index is pushed into the queue, we must modify its corresponding value in the matrix from 0 to any positive integer. Thus, we can use a counter to store the number of indices that is pushed into the queue. When the search finishes, the counter indicates the pool size, and we can store its value into the output list.

Proof of correctness

For every pool in the matrix, it is "activated" by one of its water grids during the iteration.

- Every water grid is explored at least once because we search all the 8 directions.
- Every water grid is explored at most once because we modify the corresponding grid value in the matrix during the search.

Thus, every water grid is counted exactly once, and we will get the correct size of the pool after the breadth-first search.

Time complexity

Every land grid is visited once in the iteration. Every water grid is visited at most once in the iteration and at most 9 times in the breadth-first search (it is popped from the queue exactly once, and when any of its 8 neighbors is popped from the queue, the adjacency check will visit this grid). Thus the time complexity is $O(m \times n)$.

5. **Problem 5: Perfect matching in a tree**

Input and output clarification

The input is a tree of size n, the nodes are labeled $1, \cdots, n$. We assume that n is even, otherwise there will be no perfect matching since every edge touches exactly 2 nodes.
We also assume the structure of the tree given is an array of adjacency lists, i.e., $adj[i]$ contains all the nodes that are directly connected to node i.

The output is a boolean value which indicates whether there is a perfect matching or not.

High-level description

We can reformulate the problem as follows:

- Find $n/2$ pairs of nodes (u_i, v_i) $(1 \leq i \leq n/2)$. u_i and v_i are neighbors (i.e. there is an edge between u_i and v_i). Each node is appeared in exactly one pair.

The edges between u_1 and v_1, u_2 and v_2, $\cdots$, $u_{n/2}$ and $v_{n/2}$ will then form a perfect matching.

To find the matching, we use a breadth-first search with a FIFO queue:

- We denote $edgeCount[u]$ as the number of neighbors of u. The initial value of $edgeCount[u]$ is just the size of $adj[u]$.
  If a node is later included in a pair, we will set its $edgeCount$ value as 0.
- We first put every node u into the queue where $edgeCount[u] = 1$ holds.
- In each search step, we pop a node u from the queue:
  - If $edgeCount[u]$ is 0, we ignore this node and move to the next step.
  - If $edgeCount[u]$ is 1, we iterate through the original neighbors of u by $adj[u]$ and find the only current neighbor v by checking whether $edgeCount[v] > 0$. We add 1 to a pair counter, and then iterate through the original neighbors of v by $adj[v]$. Denote a neighbor of v as w, we decrease $edgeCount[w]$ by 1. If $edgeCount[w] = 1$ afterwards, we push w into the queue.
    After the iteration, $edgeCount[u]$ is 0 since $u \in adj[v]$. Finally we set $edgeCount[v] = 0$ to indicate that v is included in a pair.
  - $edgeCount[u]$ cannot have values other than 0 or 1.

When the breadth-first search ends, we check the value of the pair counter. There is a perfect matching if and only if the value is exactly $n/2$.

Proof of correctness

We will prove three statements:

- The breadth-first search process will always end.
- If there is no perfect matching, our algorithm will not incorrectly find one.

- If there is a perfect matching, our algorithm will definitely find one.

To prove the first statement, we can see that u is pushed into the queue when *edgeCount*[u] is 1. Since the value of *edgeCount*[u] is decreasing, every u will be pushed to the queue at most once. Thus when all the nodes are popped out, the process ends.

To prove the second statement, we can show that our algorithm will always find valid (u, v) pairs and never use a node in multiple pairs. When u is popped out from the queue and *edgeCount*[u] is 1, we find its only neighbor v by checking whether *edgeCount*[v] > 0. Since $v \in$ *adj*[u], there is an edge connecting u and v. After (u, v) is paired, we set both *edgeCount*[u] and *edgeCount*[v] as 0 as if they are moved from the tree, so they will not be involved in the next search steps. Thus every node will be used at most once.

To prove the third statement, we can show that there is always at least one u that *edgeCount*[u] $=$ 1 holds before the counter value reaches $n/2$.

- Initially, every leaf in the given tree has *edgeCount* value as 1. There is at least one leaf in the tree.
- After every step of finding a (u, v) pair, we will remove node u and v together with all the incident edges. Thus the tree which contains u and v will be split into one or more trees, indicating that the rest nodes and edges always form a forest. Every leaf node that is in a tree of size larger than 1 has *edgeCount* value 1.

  If there is a perfect matching, we will **never** fall into the case where we find less then $n/2$ pairs but all the rest trees have size 1.

  - If currently there is only one node u that *edgeCount*[u] $= 1$ holds, we can choose nothing but to form a pair between u and its only neighbor v.
  - If there are more than one nodes, our algorithm will choose the first node in the front of the queue. Actually the **order** is not important. We can choose any node u that has a *edgeCount* value 1, because the (u, v) pair is already **fixed**: we can either form a pair immediately or postpone the pair to later steps. Thus, every order of pairing will result in a correct perfect matching.

Time complexity

We have proved that every node is pushed into the queue at most once. When a node is popped out from the queue, we will iterate through its original neighbors by *adj*. Since the sum of the length of *adj* lists equals 2 times the number of edges, and a tree with n nodes has $n - 1$ edges, the total time complexity is $O(n)$.