Algorithms: CSE 202 — Homework 0

For each problem, provide a high-level description of your algorithm. Please make sure to include the necessary details that are crucial for its correctness and efficiency. Prove its correctness and analyze its time complexity.

Problem 1: Maximum sum among nonadjacent subsequences

Find an efficient algorithm for the following problem:

We are given an array of real numbers $V[1..n]$. We wish to find a subset of array positions, $S \subseteq [1...n]$ that maximizes $\sum_{i \in S} V[i]$ subject to no two consecutive array positions being in S. For example, say $V = [10, 14, 12, 6, 13, 4]$, the best solution is to take elements $1, 3, 5$ to get a total of $10 + 12 + 13 = 35$. If instead, we try to take the 14 in position 2, we must exclude the 10 and 12 in positions 1 and 3, leaving us with the second best choice $2, 5$ giving a total of $14 + 13 = 27$.

Solution: Maxium sum of noncontiguous elements

Input: A sequence of $a_1, a_2, \ldots, a_n$ of real numbers for $n \geq 1$.
Output: The maximum sum of noncontiguous elements of the sequence. More preceisly, we would like to output $\sum_{i \in S} a_i$ where S ranges over all subsets of the index set $\{1, 2, \ldots, n\}$ subject to the condition that it does not include the indices j and $j+1$ for any $1 \leq j \leq n$. In other words, S does not include adjacent indices. We define the sum over an empty set to be zero.
High level description: We solve this problem using dynamic programming. For this purpose, we define the following subproblems.
Dynamic Programming Definition:

For $1 \leq i \leq n$, let $\mathbf{M}(i)$ = the maximum noncontiguoous sum for the sequence $a_1, \ldots, a_i$.

Recursive Formulation: For the base case, we define $M(1) = \max\{a_1, 0\}$ and $M(2) = \max\{a_1, a_2, 0\}$. For $i \geq 3$, the maximum noncontiguous sum for the sequence $a_1, \ldots, a_i$ either contains a_i or it does not. If a_i is in the maximum noncontiguous sum, then a_{i-1} will not be part of the sum. In each case, we can express $M(i)$ in terms of M for smaller inputs.

$$\mathbf{M}(i) = \max \begin{cases} \mathbf{M}(i-1), & \text{if } a_i \text{ is not in the maximum noncontiguous sum of the sequence } a_1, \ldots, a_i \\ \mathbf{M}(i-2) + a_i, & \text{if } a_i \text{ is in the maximum noncontiguous sum of the sequence } a_1, \ldots, a_i \end{cases}$$

We compute $M(i)$ for i starting with 1 through n. $M(n)$ is the maximum sum of noncontiguous elements of the input sequence.
Time Complexity: For each $1 \leq i \leq n$, the computation of $M(i)$ during iteration i takes constant number of steps. Hence, the total time complexity of the algorithms is $\Theta(n)$.
Proof of Correctness: The reader is adivsed to supply the straightforward proof by induction.

Problem 2: Maximum difference in an array

Given an array A of integers of length n, find the maximum value of $A(i)-A(j)$ over all choices of indexes such that $j > i$.

Solution: Maximum difference in an array

Algorithm description: The maximum difference in an array can be computed after a single scan of the array from right to left. We use two quantities (variables) to keep track of the progress during the scan.

- *minElement* tracks the current minimum element, that is, the minimum of the elements in the subarray that has been scanned thus far. It is initialized to $A(n)$, the last element of A.
- *maxDifference* tracks the current maximum difference in the subarray scanned so far. It is initialized to $-\infty$. We define the maximum difference of an array of length 1 to be $-\infty$.

For $1 \leq i \leq n-1$, the next element in the array $A(i)$ is processed as follows:

- Calculate the difference $d = A(i) - minElement$. If $d > maxDifference$, set *maxDifference* to d.
- If $A(i) \leq minElement$, $A(i)$ is the current minimum element, that is, *minElement* is set to $A(i)$.

The value of *maxDifference* at the end of the scan is the required value.

Proof of correctness: We prove the following claim by induction to establish the correctness of the algorithm.

Claim 0.1. *For $1 \leq j \leq n$, at the beginning of the j-th iteration we have*

1. *minElement is the minimum of the elements in the subarray $A(n-j+1), \ldots, A(n)$.*
2. *maxDifference is the maximum difference for the subarray $A(n-j+1), \ldots, A(n)$.*

Proof: The proof is by induction on the number of iterations, that is, on the value of j. For the base case consider the state of the execution of the algorithm at the beginning of the first iteration of the loop. At this point since $j = 1$ the array under consideration has exactly one element, namely, the last element $A(n)$. Also at this point we have *minElement* $= A(n)$ and *maxDifference* $= -\infty$. The value of *minElement* is indeed the minimum of the subarray $A(n)$ and the value of *maxDifference* is indeed the maximum difference of the subarray by definition. We have thus proved the claim for the case $j = 1$.

Let $1 \leq k \leq n-1$ be an arbitrary integer. For the inductive step we assume that the claim is true for $1 \leq j \leq k$. Consider the state at beginning of iteration $k+1$. We are considering the subarray $A(n-k), \ldots, A(n)$ at this point.

We first show that *minElement* is the minimum of the subarray $A(n-k), \ldots, A(n)$. By the induction hypothesis, at the beginning of iteration k, *minElement* is the minimum of the subarray $A(n-k+1), \ldots, A(n)$. During the k-th iteration we compared it with $A(n-k)$ and updated it appropriately so we can conclude that *minElement* is the minimum of the subarray $A(n-k), \ldots, A(n)$.

We will now show that at the beginning of iteration $k+1$ the value of *maxDifference* is the maximum difference of the array $A(n-k), \ldots, A(n)$. Indeed the difference of $A(n-k)$ and any element to the right is maximized by the difference between $A(n-k)$ and the minimum of the subarray $A(n-k+1), \ldots, A(n)$. By the induction hypothesis, at the beginning iteration k, the value of *minElement* is equal to the minimum element in the subarray $A(n-k+1), \ldots, A(n)$. During iteration k we compute $d = A(n-k) -$ *minElement* before we update the value of *minElement* to get the maximum difference between $A(n-k)$ and any elements to its right.

We also observe that the maximum difference in the subarray $A(n-k), \ldots, A(n)$ either involves $A(n-k)$ or it does not. By the induction hypothesis, at the beginning of iteration k, *maxDifference* is the maximum difference of the subarray $A(n-k+1), \ldots, A(n)$. During iteration k, after computing d, we compute the maximum of d and *maxDifference* and set the value of *maxDifference* to the larger of the two. Hence, at the beginning of iteration $k+1$, the value of *maxDifference* is indeed the maximum difference of the subarray $A(n-k), \ldots, A(n)$ as claimed.

Problem 3: Maximum difference in a matrix

Given an $n \times n$ matrix $M[i, j]$ of integers, find the maximum value of $M[c, d] - M[a, b]$ over all choices of indexes such that both $c > a$ and $d > b$.

Solution: Maximum difference in a matrix

Let M be the given matrix of integers with n rows and n columns. We use $M[i, j]$ to denote the entry of the matrix in row i and column j. Our goal is to compute $\max_{1 \leq a < c \leq n, 1 \leq b < d \leq n} M[c, d] - M[a, b]$.

For $i \leq k$ and $j \leq l$, the submatrix determined by (i, j) and (k, l) refers to the matrix with entries $M[a, b]$ where $i \leq a \leq k$ and $j \leq b \leq l$.

For each $1 \leq i, j \leq n$ we define

$$T[i, j] = \min_{1 \leq k \leq i, 1 \leq l \leq j} M[k, l].$$

$T[i, j]$ is the minimum value in the submatrix defined by $(1, 1)$ and (i, j).

We define $D[i, j]$ for $1 < i, j \leq n$ as

$$D[i, j] = M[i, j] - T[i-1, j-1]$$

If $i = 1$ or $j = 1$, we define $D[i, j]$ to be $-\infty$.

$D[i, j]$ is the maximum of the differences between $M[i, j]$ and the entries in the submatrix defined by $(1, 1)$ and $(i-1, j-1)$.

For each $1 \leq i, j \leq n$, we show how to compute $T[i, j]$ and $D[i, j]$ in constant time. After we compute the matrix D, we output $\max_{1 \leq i, j \leq n} D[i, j]$ as our answer.

To compute $T[i, j]$ and $D[i, j]$, the algorithm scans the entries of the matrix from row 1 to row n such that each row is scanned from column 1 to column n. We compute $T[i, j]$ using the following formula:

$$T[i, j] = \begin{cases} M[i, j] & \text{if } i = 1 \text{ and } j = 1 \\ \min\{T[i, j-1], M[i, j]\} & \text{if } i = 1 \text{ and } j > 1 \\ \min\{T[i-1, j], M[i, j]\} & \text{if } i > 1 \text{ and } j = 1 \\ \min\{T[i-1, j], T[i, j-1], M[i, j]\} & \text{if } i > 1 \text{ and } j > 1 \end{cases}$$

To compute $T[i, j]$ we rely only on the values of the matrix T that have already been computed. After computing $T[i, j]$, we compute $D[i, j]$ by the following formula.

$$D[i, j] = \begin{cases} -\infty & \text{if } i = 1 \text{ or } j = 1 \\ M[i, j] - T[i-1, j-1] & \text{otherwsie.} \end{cases}$$

It is clear that the algorithm takes linear time in the number of entries in the matrix M. The proof of correctness is left to the reader.

Problem 4: Pond sizes

You have an integer matrix representing a plot of land, where the value at a location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of a pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

Solution: Pond sizes

Denote M as the given integer matrix. Suppose v is some cell of M ($v \in M$), then its corresponding value is M_v and the row and column indices are i_v and j_v, respectively. Let us construct an undirected graph G in the following way:

- G has a vertex v if and only if $v \in M$ and $M_v = 0$;
- For any distinct $v, u \in G$, there is an edge between v and u if and only if $|i_v - i_u| \leq 1$ and $|j_v - j_u| \leq 1$.

Now, it is clear that the connected components of G represent the corresponding ponds, and a *pond size* is the number of nodes in a component. Thus, we can find the pond sizes by running Breadth-First Search Algorithm (BFS) on every connected component of the graph G.

Algorithm: First, we construct the graph G by iterating over M. Notice that for each cell, we check for connectivity with at most eight neighboring cells. Then, we iterate over all the nodes in G and keep track of visited nodes:

- If the current node is not visited, we initialize the pond size to 1 and run BFS on this node while incrementing the pond size and updating the visited nodes. When the BFS is done, we add this pond size to the final answer;
- Otherwise, we continue the loop.

Proof of correctness: For any node of the given connected component, we know that BFS visits all its nodes. Therefore, an unvisited node in every iteration cannot belong to previously encountered connected components. Thus, every BFS runs on a new connected component and calculates its size correctly. For this reason, the aforementioned algorithm is correct.

Time complexity: Suppose n is the number of cells in M. Then, the construction of the graph G takes $O(n)$ time since we iterate over n, make at most eight checks for connectivity and update our graph G in constant time. In the second part, we also iterate over the matrix M and additionally do BFS runs. Since every cell can be visited by BFS at most once, the time complexity for all BFS runs is $O(n)$. Therefore, the total time complexity is $O(n)$.

Problem 5: Perfect matching in a tree

Give a linear-time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each node exactly once.

Solution: Perfect matching in trees

A tree $T = (V, E)$ is represented by a set V of vertices and a set of edges $E \subseteq V \times V$ between them. We assume that the edges are undirected. A vertex is a leaf vertex if its degree is 1.

A perfect matching M of T is a set of edges of T such that now two edges in M have a common end vertex and every vertex in T is incident upon an edge in M. Since every edge has exactly two distinct end vertices, a tree may not have a perfect matching if it has an odd number of vertices.

We now provide a high-level description of an algorithm to check if the graph has a perfect matching. In the algorithm, we delete vertices and their incident edges and as a result the tree may be disconnected. For this reason, we deal with a more general input, a forest of trees. Deleting a vertex from a forest will again give rise to a forest, perhaps an empty forest. An empty forest is a forest with zero vertices.

The strategy is to select edges *greedily* and delete the corresponding end vertices. Specifically, we select a leaf vertex and the edge that connects it to its unique neighbor. Remember that every tree with at least two vertices has at least one leaf vertex. The selected edge will be part of the perfect matching we are trying to construct. If, at any point, there is an isolated vertex (a vertex with degree zero), we conclude that the input forest does not have a perfect matching. On the other hand, if we are left with an empty forest at the end, we conclude that the forest has a perfect matching.

Before we discuss the implementation details of the algorithm, we prove its correctness. We argue that the input forest has a perfect matching if and only if the algorithm terminates with an empty forest.

Claim 0.2. *If the algorithm terminates with an empty forest, then the forest has a perfect matching.*

Proof: Let T be a forest of trees and M be the set of edges selected by the algorithm. Since the end vertices of a selected edge are deleted right after selecting it, no two edges in M will have a common endpoint. Moreover, a vertex is only deleted if an edge incident on it is selected. Since there are no vertices in the forest at termination, we conclude that every vertex is incident on an edge in M. M is thus a perfect matching.

Claim 0.3. *The input forest T has a perfect matching, then the algorithm terminates with an empty forest.*

Proof: We prove this claim by induction on the number of vertices in T. If T has no vertices, the claim is trivially true.

If T has exactly one vertex, it cannot have a perfect matching. Let T be a forest with $n \geq 2$ vertices. Further assume that T has a perfect matching. Since T has a perfect matching, it cannot have any isolated vertices. Hence, every tree in T has at least two vertices. Consider any leaf vertex u of T. In every perfect matching of T, u will be matched with its unique neighbor v. For this reason, $T - \{u, v\}$, the forest obtained by deleting the vertices u and v together with their incident edges, will have a perfect matching. The algorithm selects a leaf vertex u and deletes u together with its unique neighbor v. The algorithm continues its execution with the input $T - \{u, v\}$. By induction we conclude that the algorithm results in an empty forest.

Implementation:

We now describe how to implement this algorithm in linear time. To simplify the presentation, we assume that the input is a rooted Tree. A tree T is a rooted tree if it is a tree where every vertex (except the root) is assigned a unique parent. In a rooted tree, each vertex is endowed with a (possibly null) pointer to the parent and a (possibly empty) list of pointers to the child vertices.

We implement the algorithm by traversing the tree in post order. As we select edges for the matching, we mark the corresponding end vertices. Initially, all vertices are unmarked. Consider the point in time when we visit a vertex for the last time in the post order traversal after visiting all its children. If the vertex is not marked and its parent vertex is not marked, we match the vertex with its parent. We mark the vertex and its parent and continue with the traversal. If the vertex is not marked and its parent is marked, we declare that the tree does not have a perfect matching. If the vertex is marked, we continue with the traversal. Note that the algorithm marks only unmarked vertices, never marks a marked vertex.

We argue that the implementation is consistent with the algorithm outlined above. Observe that the very first vertex marked during the execution must be a leaf vertex. If we pretend that we delete the marked vertex and its marked parent, the subsequent marked vertex is also a leaf node in the forest obtained after deletion. Since the vertices are not actually deleted, post order traversal will continue as before.

Time complexity: Since the post order traversal runs in linear time and the checks and markings take only a constant time per vertex, the entire algorithm runs in linear (in the number of vertices of the tree) time.