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 $d = A(i) - \mathit{minElement}$. If $d > \mathit{maxDifference}$, set *maxDifference* to d.
- If $A(i) \leq \mathit{minElement}$, $A(i)$ 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: Frequent elements

Design an algorithm that, given a list of n elements in an array, finds all the elements that appear more than $n/3$ times in the list. The algorithm should run in linear time. n is a nonnegative integer.

You are expected to use comparisons and achieve linear time. You may not use hashing. Neither can you use excessive space. Linear space is sufficient. Moreover, you are expected to design a deterministic algorithm. You may also not use the standard linear-time deterministic selection algorithm. Your algorithm has to be more efficient (in terms of constant factors) than the standard linear-time deterministic selection algorithm.

Solution: Frequent Elements

1 Problem 5

By Joseph L.

Input: A list of size n
Output: A list of elements that occur more than $\frac{n}{3}$ times. Note that there are between zero and two such elements.

Observations: This is similar to the problem of finding the majority element ($> \frac{n}{2}$ occurrences). The third problem set on the course website includes a solution to this case. Please read and understand "Algorithm 1" of Problem 36, Part 2 before continuing, as understanding that case simplifies the process of learning how to generalize it.

High-level ideas
The key phrase of the $> \frac{n}{2}$ solution is this:

> If an array A has a majority element a, and if we remove any two distinct elements from the array, then the array still contains a majority element, and it is the same majority element as before.

The basic operation, then, was to identify and remove pairs of distinct elements, and then output what remained.

These ideas generalize naturally to the $> \frac{n}{3}$ case:

> If an array A has elements a,b that each occur more than $\frac{n}{3}$ times each, and if we remove any three distinct elements from the array, then the array still contains elements that occur more than $\frac{n}{3}$ times, and these are the same elements as before.

The basic operation is now to identify and remove *triples* of distinct elements, and we will consider our list to be fully reduced once we can no longer remove three distinct elements. In a fully reduced list, there will be at most two distinct elements. We will show that the remaining distinct elements are candidates for the elements that occur with frequency greater than $n/3$.

We represent the fully reduced list with at most two distinct elements as follows: (remaining element 1; count 1; remaining element 2; count 2; index of the next element in the list).

Again, we initialize the fully reduced list (for the empty prefix of elements) with counts 0 (our representation would thus be $(-;0;-;0;1)$). While maintaining a fully reduced list of the elements scanned so far, we process the next element as follows:

1. If the reduced list contains the next element, then we increment the corresponding count by 1.
2. If the reduced list contains openings, add this element to an opening, with count equal to 1.
3. If the reduced list has no openings and does not contain the next element, then we decrement each count by 1.

Let us go through some examples.
Here's one with two solutions: $A = [a, b, a, c, b, a, b, d]$

$$(-;0;-;0;1);(a;1;-;0;2);(a;1;b;1;3);(a;2;b;1;4);(a;1;-;0;5);\\ (a;1;b;1;6);(a;2;b;1;7);(a;2;b;2;8);(a;1;b;1;9)$$

Our tentative solutions are a and b. We then iterate through the list to confirm that each solution does indeed occur $> \frac{n}{3}$ times in the original list; both do, so we output a and b.

Here's one with only one solution: $A = [a, b, a, c, c, a, b, d]$

$$(-;0;-;0;1);(a;1;-;0;2);(a;1;b;1;3);(a;2;b;1;4);(a;1;-;0;5);\\ (a;1;c;1;6);(a;2;c;1;7);(a;1;-;0;8);(a;1;d;1;9)$$

Our tentative solutions are a and d. Upon iterating through the list to confirm each solution, we find that only a occurs $> \frac{n}{3}$ times in the original list; we therefore output only a.

Complexity: As we spend a constant amount of time with each element of the list, we see that our algorithm is linear - $O(n)$.

As per the instructions, we're also interested in the constant factor of our complexity, so we will now compute that. Our solution has the following steps:

1. Iterate through the list once, updating our reduced list at each iteration. Each iteration contains up to 2 comparisons (one for each element in the reduced list) so this step has a total of $2n$ comparisons.
2. Confirm that each tentative solution occurs $> \frac{n}{3}$ times. We need n comparisons for each one. Therefore, this step has a total of $2n$ comparisons.

We make a total of $4n$ comparisons. We would like our solution to be more efficient, in terms of the constant factor, than the deterministic linear-time selection algorithm. That algorithm is said to make up to $24n$ comparisons, so our solution is indeed more efficient.

Correctness: Recall that whenever we remove elements, we remove *three distinct* elements at a time. Suppose that, after terminating on a given run, we've removed elements c times, and have therefore removed $3c$ elements. Whatever c is, we can't have removed more elements than the list originally contained. Thus, we have the following inequality:

$$3c \leq n$$

Equivalently:

$$c \leq \frac{n}{3}$$

Therefore, any instance of an element a could have been removed at most $\frac{n}{3}$ times. If the original list contained $> \frac{n}{3}$ instances of a, then the remaining instances are necessarily in the reduced list.