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 area contiguous subsequence

Use divide-and-conquer approach to design an efficient algorithm that finds the contiguous subsequence with the maximum area in a given sequence of n nonnegative real values. Analyse your algorithm, and show the results in order notation. Can you do better? Obtain a linear-time algorithm.

The area of a contiguous subsequence is the product of the length of the subsequence and the minimum value in the subsequence.

Solution: Maximum area contiguous subsequence

1 $n \log n$ Algorithm

1.1 Definitions

We are given a nonempty sequence of nonnegative real numbers as input and we are asked to output the area of a contiguous subsequence with the maximum area.

Let $S = x_1, x_2, \ldots, x_n$ be a sequence of nonnegative integers for $n \geq 1$. Let $S_{ij} = x_i, \ldots, x_j$ denote a contiguous subsequence of S where $1 \leq i \leq j \leq n$. We define the area $\mathbf{a}(S_{ij})$ of S_{ij} as

$$\mathbf{a}(S_{ij}) = \min_{i \leq k \leq j} x_k(j - i + 1)$$

We define $\mathbf{A}(S_{ij}) = \max_{i \leq i' \leq j' \leq j} \mathbf{a}(S_{i'j'})$. $\mathbf{A}(S_{ij})$ is the area of a contiguous subsequence of S_{ij} with the maximum area. We use the following divide-and-conquer scheme to compute $\mathbf{A}(S) = \mathbf{A}(S_{1n})$.

1.2 The Algorithm

If $n = 1$, we solve the problem directly and output x_1 as the result. Otherwise, let $q = \lfloor \frac{n}{2} \rfloor$. We partition the sequence S into two subsequences $L = x_1, \ldots, x_q$ and $R = x_{q+1}, \ldots, x_n$. Since $n \geq 2$, we have $q \geq 1$ and $q + 1 \leq n$ which ensure that both L and R have length at least 1. We recursively compute $\mathbf{A}(L)$ and $\mathbf{A}(R)$.

We will now consider contiguous subsequences which start in L and end in R. We call such contiguous subsequences *spanning contiguous subsequences* with respect to L and R. Let $\mathbf{A}_c(L, R)$ denote the area of a spanning contiguous subsequence (with respect to L and R) with maximum area. Below, we will show how to compute $\mathbf{A}_c(L, R)$ in linear time. Finally, we output $\max\{\mathbf{A}(L), \mathbf{A}(R), \mathbf{A}_c(L, R)\}$ as $\mathbf{A}(S_{1n})$.

Proof of correctness: It is clear that a contiguous subsequence of S is

1. entirely in L (type 1) or
2. entirely in R (type 2) or

3. starts in L and ends in R, that is, a spanning contiguous subsequence with respect to L and R (type 3).

Assuming the correctness of the algorithm for finding the maximum area spanning contiguous subsequence (with respect to L and R), we argue by induction that the overall algorithm produces the correct output since $\mathbf{A}(L)$ is the maximum area of the contiguous subsequences of type 1, $\mathbf{A}(R)$ is the maximum area of the contiguous subsequences of Type 2, and $\mathbf{A}_c(L, R)$ is the maximum area of the contiguous subsequences of type 3.

Complexity analysis: Let $T(n)$ denote the number of time steps required by the algorithm in the worst-case on inputs of length n. Clearly, $T(1) = 1$ and $T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn$ for some constant $c > 0$. Solving the recurrence relation, we get $T(n) = O(n \log n)$.

Computing $\mathbf{A}_c(L, R)$: Let $U = \{1, \ldots, q\}$ and $V = \{q + 1, \ldots, n\}$ be the set of indices for the elements of the sequences L and R respectively. For $i \in U$, let l_i denote the minimum value of the sequence $x_i, \ldots, x_q$. For $j \in V$, let r_j denote the minimum value of the sequence $x_{q+1}, \ldots, x_j$. l_i is nonincreasing as i decreases from q to 1. Similarly, r_j is nonincreasing as j increases from $q + 1$ to n. It is easy to see that l_i and r_j can be computed in linear time (in the sum of the lengths of L and R) by scanning the L from right to left and R from left to right.

We use the following notation. For $1 \leq i \leq q$, let L_i denote the contiguous subsequence $x_i, \ldots, x_q$ and for $q+1 \leq j \leq n$ let R_j denote the contiguous subsequence $x_{q+1|}, \ldots, x_j$. We will now describe how to compute $\mathbf{A}_c(L, R)$ in linear time. To simplify the presentation, we assume that l_i for $i \in U$ and r_j for $j \in V$ have already been computed, although one can easily interleave the computation of these quantities with that of $\mathbf{A}_c(L, R)$.

The intution is that the algorithm considers all distinct values of l_i and r_j and for each distinct value y, it finds the longest spanning contiguous subsequence whose minimum is equal to y. Since the minimum value of any spanning contiguous subsequence must equal one such y, correctness of the algorithm is ensured.

Algorithm for computing $\mathbf{A}_c(L, R)$: To compute $\mathbf{A}_c(L, R)$, we scan L from right to left and R from left to right using two pointers i (for the right-left scan) and j (for the left-right scan) to keep track of the current state of the scanning. To initialize i and j, we consider two cases. If $l_q \leq r_{q+1}$, we initialize

$$i \leftarrow \arg\min_{1 \leq k \leq q} l_k = l_q \text{ and}$$
$$j \leftarrow \arg\max_{q+1 \leq k \leq n} r_k \geq l_q.$$

If $l_q > r_{q+1}$, we initialize

$$i \leftarrow \arg\min_{1 \leq k \leq q} l_k \geq r_{q+1} \text{ and}$$
$$j \leftarrow \arg\max_{q+1 \leq k \leq n} r_k = r_{q+1}.$$

During the scan, i will only advance to the left and j will only advance to the right. We also maintain the area of the largest area contiguous subsequence using a variable M, which is initialized with the area of the contiguous subsequence $x_i, \ldots, x_j$ which is equal to $\min\{l_i, r_j\}(j - i + 1)$.

The algorithm proceeds in rounds where in each round we advance the pointers i and j as follows.

$i \leftarrow \arg\min_{1 \le k < i} l_k = l_{i-1}$ and $j \leftarrow j$	if $i > 1$, $j < n$ and $l_{i-1} \ge r_{j+1}$
$i \leftarrow i$ and $j \leftarrow \arg\max_{j < k \le n} r_k = r_{j+1}$	if $i > 1$, $j < n$ and $l_{i-1} < r_{j+1}$
$i \leftarrow i$ and $j \leftarrow \arg\max_{j < k \le n} r_k = r_{j+1}$	if $i = 1$ and $j < n$
$i \leftarrow \arg\min_{1 \le k < i} l_k = l_{i-1}$ and $j \leftarrow j$	if $i > 1$ and $j = n$
terminate the algorithm	if $i = 1$ and $j = n$

We compute $\mathbf{a}(S_{ij})$, update $M \leftarrow \max\{M, \mathbf{a}(S_{ij})\}$ and repeat.

Remarks about the termination and the time complexity of the algorithm for computing $\mathbf{A}_c(L, R)$: It is clear that the algorithm runs in linear time since one of the pointers advances by at least one in each round unless both the pointers are already at their extreme values, in which case the algorithm terminates.

We now turn to the correctness of the algorithm.

Claim 1.1. *Let $t \ge 1$. Let i and j respectively be the values of the left and right pointers at the beginning of round t. The following properties hold at the beginning of round t:*

1. *If $i > 1$, $l_{i-1} < r_j$.*
2. *If $j < n$, $r_{j+1} < l_i$.*
3. *$M = \max_s \mathbf{a}(s)$ where s ranges over all spanning contiguous subsequences with respect to L_i and R_j.*

Note 1.2. The claim is sayng that that $x_i, \ldots, x_j$ is the longest spanning contiguous subsequence with respect to L_i and R_j such that for any $i' < i$ or $j' > j$, $\min(l_{i'}, r_{j'}) < \min(l_i, r_j)$.

Proof: We prove the claim by induction on the number of rounds in the execution of the algorithm. For the base case, consider the values of i, j and M at the beginning of the first round. Assume $l_q \le r_{q+1}$. The other case can be handled in a similar fashion. By construction, we have $l_i = l_q$ and i is the smallest such index. Also, we have $l_i \le r_j$. Hence, if $i > 1$, $l_{i-1} < l_i \le r_j$. Also, if $j < n$, $r_{j+1} < l_i$ since j is the largest index such that $r_j \ge l_i$. At the beginning of the first round, M is equal to the area of the spanning contiguous subsequence $x_i, \ldots, x_j$. Since $l_i = l_q$ and $r_j \ge l_i$, we deduce that l_i is the minimum value of the sequence and no spanning contiguous subsequence with respect to L_i and R_j can have a larger minimum value. Hence, $M = \max_s \mathbf{a}(s)$ where s ranges over all the spanning contiguous subsequences with respect to L_i and R_j.

Assume that the claim holds at the beginning of round t. Let i and j respectively be the values of the left and right pointers at the beginning of round t. By inductive hypothesis, $M = \max_s \mathbf{a}(s)$ where s ranges over all the spanning contiguous subsequences with respect to L_i and R_j.

Let i' and j' be the values of i and j respectively after they have been updated during round t. Let M' be the value of M at the end of round t. We now that prove that the properties in the claim will hold for i', j' and M' by considering each of the cases for updating i' and j'.

Case I: $i > 1$, $j < n$, $l_{i-1} \ge r_{j+1}$, $i' \leftarrow \arg\min_{1 \le k < i} l_k = l_{i-1}$ and $j' \leftarrow j$

If $i' > 1$, $i > 1$ and $l_{i'-1} < l_{i'} = l_{i-1}$ by construction since i' is the smallest index such that $l_{i'} = l_{i-1}$. Again, by construction, $r_j = r_{j'}$. However, we have $l_{i-1} \le r_j$ by inductive hypothesis which implies $l_{i'-1} < r_{j'}$ establishing the first property.

If $j' < n$, $j < n$ and $r_{j'+1} = r_{j+1}$ since $j' = j$. Also, we have $l_{i'} = l_{i-1}$ by construction and $l_{i-1} \ge r_{j+1}$ in this case. By chaining these conditions, we conclude $r_{j'+1} \le l_{i'}$ establishing the second property.

For the third property, we consider a spanning contiguous subsequence s with respect to $L_{i'}$ and $R_{j'}$. If s is also a spanning contiguous subsequene with respect to L_i and R_j, we know that $M' \geq \mathbf{a}(s)$ by inductive hypothesis.

Assume that one of the end points of s lies outside of the interval $[i, j]$. In this case, s must be equal to $x_u, \ldots, x_v$ for some $i' \leq u < i$ and $q + 1 \leq v \leq j'$. The minimum value of any such sequence is l_{i-1} since $l_u = l_{i'} = l_{i-1}$ (by construction) and $l_{i-1} < r_j = r_{j'}$ (by inductive hypothesis). Since $S_{i'j'}$ is the longest spanning contiguous subsequence with respect to $L_{i'}$ and $R_{j'}$ for the minimum value l_{i-1} and $M' = \max\{\max_{s'} \mathbf{a}(s'), \mathbf{a}[i', j']\}$ where s' ranges over all spanning contiguous subsequences with respect to L_i and R_j, we conclude $M' = \max_s \mathbf{a}(s)$ where s ranges over all spanning contiguous subsequences with respect to $L_{i'}$ and $R_{j'}$.

Case II: $i > 1$, $j < n$, $l_{i-1} < r_{j+1}$, $i' \leftarrow i$ **and** $j' \leftarrow \arg\max_{j<l\leq n} m_l = m_{j+1}$

If $i' > 1$, $i > 1$ and $l_{i'-1} = l_{i-1}$ since $i' = i$. We also have $r_{j+1} = r_{j'}$. We have $l_{i-1} < j_{j+1}$ in this case which implies $l_{i'-1} < r_{j'}$ establishing the first property.

If $j' < n$, $j < n$ and $r_{j'+1} < r_{j'} = r_{j+1}$ by construction since j' is the largest index such that $r_{j'} = r_{j+1}$. Again, by construction, we have $l_{i'} = l_i$. We also have $r_{j+1} < l_i$ by inductive hypothesis. which implies $r_{j'+1} < l_{i'}$ establishing the second property.

For the third property, we consider a spanning contiguous subsequence s with respect to $L_{i'}$ and $R_{j'}$. If s is also a spanning contiguous subsequence with respect to L_i and R_j, we know that $M' \geq \mathbf{a}(s)$ by inductive hypothesis.

Assume that one of the end points of s lies outside of the interval $[i, j]$. In this case, s must be equal to $x_u, \ldots, x_v$ for some $i' \leq u \leq q$ and $j <\leq v \leq j'$. The minimum value of any such sequence is r_{j+1} since $l_u \geq m_{j'} = m_{j+1}$ by construction and $r_{j+1} \leq l_i = l_{i'}$ by inductive hypothesis. Since $S_{i'j'}$ is the longest spanning contiguous subsequence with respect to $L_{i'}$ and $R_{j'}$ for the minimum value r_{j+1} and $M' = \max\{\max_{s'} \mathbf{a}(s'), \mathbf{a}[i', j']\}$ where s' ranges over all spanning contiguous subsequences with respect to L_i and R_j, we conclude $M' = \max_s \mathbf{a}(s)$ where s ranges over all spanning contiguous subsequences with respect to $L_{i'}$ and $R_{j'}$.

Case III: $i = 1$, $j < n$, $i' \leftarrow i$ **and** $j' \leftarrow \arg\max_{j<l\leq n} l_l = r_{j+1}$

The first property is trivially true since $i' = i = 1$.

If $j' < n$, $j < n$ and $r_{j'+1} < r_{j'} = r_{j+1}$ by construction. We also have $l'_i = l_i = l_1$ since $i' = i = 1$. However, by inductive hypothesis we have $r_{j+1} \leq l_i$ which implies $r_{j'+1} \leq l_{i'}$ satisfying the second property of the claim.

For the third property, we consider a spanning contiguous subsequence s with respect to $L_{i'}$ and $R_{j'}$. If s is also in in the interval $[i, j]$, we know that $M' \geq \mathbf{a}(s)$ by inductive hypothesis.

Assume that one of the end points of s lies outside of $[i, j]$. In this case, s must be equal to $x_u, \ldots, x_v$ for some $i' \leq u \leq q$ and $j < v \leq j'$. The minimum value of any such sequence is r_{j+1} since $r_v = r_{j'} = r_{j+1}$ (by construction) and $r_{j+1} \leq l_i = l_{i'}$ (by inductive hypothesis). Since $S_{i'j'}$ is the longest spanning contiguous subsequence with respect to $L_{i'}$ and $R_{j'}$ for the minimum value r_{j+1} and $M' = \max\{\max_{s'} \mathbf{a}(s'), \mathbf{a}[i', j']\}$ where s' ranges over all spanning contiguous subsequences with respect to L_i and R_j, we conclude $M' = \max_s \mathbf{a}(s)$ where s ranges over all spanning contiguous subsequences with respect to $L_{i'}$ and $R_{j'}$.

Case IV: $i > 1$, $j = n$, $i' \leftarrow \arg\min_{1\leq k<i} l_k = l_{i-1}$ **and** $j' \leftarrow j$

If $i' > 1$, $i > 1$ and $l_{i'-1} < l_{i'} = l_{i-1}$. We also have $r_{j'} = r_j = r_n$ since $j' = j = n$. However, by inductive hypothesis we have $l_{i-1} < r_j$ which implies $l_{i'-1} \leq r_{j'}$ satisfying the first property of the claim.

The second property is trivially true since $j' = j = n$.

For the third property, we consider a spanning contiguous subsequence s with respect to $L_{i'}$ and $R_{j'}$. If s is also in in the interval $[i, j]$, we know that $M' \geq \mathbf{a}(s)$ by inductive hypothesis.

Assume that one of the end points of s lies outside of $[i, j]$. In this case, s must be equal to $x_u, \ldots, x_v$ for some $i' \leq u < i$ and $q + 1 \leq v \leq j'$. The minimum value of any such sequence is l_{i-1} since $l_u = l_{i'} = l_{i-1}$ (by construction) and $l_{i-1} < r_j = r_{j'}$ (by inductive hypothesis).

Since $S_{i'j'}$ is the longest spanning contiguous subsequence with respect to $L_{i'}$ and $R_{j'}$ for the minimum value l_{i-1} and $M' = \max\{\max_{s'} \mathbf{a}(s'), \mathbf{a}[i', j']\}$ where s' ranges over all spanning contiguous subsequences with respect to L_i and R_j, we conclude $M' = \max_s \mathbf{a}(s)$ where s ranges over all spanning contiguous

subsequences with respect to $L_{i'}$ and $R_{j'}$.

Case V: terminate the algorithm if $i = 1$ and $j = n$

This case will not arise since the program would have terminated before the beginning of the round t.

2 Linear-time algorithm

2.1 Definitions

We are given a nonempty sequence of nonnegative real numbers as input and we are asked to output the area of a contiguous subsequence with the maximum area.

Let $S = x_1, x_2, \ldots, x_n$ be a sequence of nonnegative integers for $n \geq 1$. Let $S_{ij} = x_i, \ldots, x_j$ denote a contiguous subsequence of S where $1 \leq i \leq j \leq n$. We define the area $\mathbf{a}(S_{ij})$ of S_{ij} as

$$\mathbf{a}(S_{ij}) = \min_{i \leq k \leq j} x_k(j - i + 1)$$

We define $\mathbf{A}(S_{ij}) = \max_{i \leq i' \leq j' \leq j} \mathbf{a}(S_{i'j'})$. $\mathbf{A}(S_{ij})$ is the area of a contiguous subsequence of S_{ij} with the maximum area. We use the following divide-and-conquer scheme to compute $\mathbf{A}(S) = \mathbf{A}(S_{1n})$.

2.2 The Algorithm

Although there are $\Theta(n^2)$ contiguous subsequences, we observe that we need only consider n of these subsequences for computing the maximum area. For $1 \leq i \leq n$, let $t(i)$ denote the longest contiguous subsequence which contains x_i and whose minimum value is x_i. It is clear that $\mathbf{A}(S_{1n}) = \max_i \mathbf{a}(t(i))$.

For $t(i)$, let l_i denote its left index and r_i its right index. We can express l_i as

$$l_i = \arg\min_{1 \leq j \leq i} \forall j \leq k \leq i \;\; x_k \geq x_i$$

Similarly, we can express r_i as

$$r_i = \arg\max_{i \leq j \leq n} \forall i \leq k \leq j \;\; x_k \geq x_i$$

For a sequence $t(i)$, its area can be computed once we determine its left and right indices. More specifically,

$$\mathbf{a}(t(i)) = x_i(r_i - l_i + 1)$$

We present an algorithm that processes the elements in the sequence from left to right, determines the left index of $t(i)$ when it first encounters x_i and subsequently determines the right index when it first encounters an element less than x_i. We maintain a stack to store items of the form (j, k) where x_j is one of the processed elements $k = l_j$. We call x_j the value of the item, j its index and k its left index. Furthermore, if an item (j, k) is on the stack, we have not completed processing any element smaller than x_j after (j, k) is placed on the stack. After processing the first $i \geq 1$ items, the stack contains items of the form (j, l_j) where $1 \leq j \leq i$ and $r_j \geq i$. It also turns out that the values of the items on the stack are local minima when we consider the list $x_i, x_{i-1}, \ldots, x_1$. We will make this concept precise in the following.

Linear-time algorithm for computing $\mathbf{A}(S)$

Preprocessing phase: We process the elements x_i as i ranges from 1 through n. We also maintain a variable M which is equal to $\max_i \mathbf{a}(t(i))$ where i is the index of an item and it ranges over all items popped from the stack so far. In other words, $M = \max_i \mathbf{a}(t(i))$ where i corresponds to the index of the items for which the right index has been determined. M is initialized to 0.

Processing phase: For $1 \leq i \leq n$, we process x_i by executing the following loop:

1. If the stack is empty, push $(i, 1)$ on the stack and exit the loop.

2. Otherwise, let (j, k) be the top item in the stack.

3. If $x_i > x_j$, push $(i, j+1)$ on the stack and exit the loop.

4. If $x_i = x_j$, change the top item (j, k) (without popping it off the stack) to (i, k) and exit the loop.

5. If $x_i < x_j$, set $l_j = k$, $r_j = i - 1$, compute $\mathbf{a}(t(j))$, and update M. Pop (j, k) off the stack and repeat.

Post processing phase: When all the elements have been processed, we enter **post processing** phase. At this point, we have $i = n$ and it is the right index of all the items on the stack . We pop the items from the stack one at a time, compute the corresponding area, and update M until the stack is empty.

Final step: Output M.

Time complexity: Assign the cost of each step of the algorithm to the appropriate element in the sequence. It is easy to see that the cost accrued to each element is bounded by a constant and hence the algorithm terminates in linear time.

Proof of correctness: We use *round i* to denote the time period during which the element x_i is processed. Let T_i be the unique point in time at the end of round i at which point the item (i, k) is placed on the stack in step 1, 3 or 4 of the algorithm for some k.

We say that the index i is equivalent to the index j if $t(i) = t(j)$.

We provide the proof of correctness of the algorithm using the following claims.

Claim 2.1. *For each $1 \leq i \leq n$, at any time T during round i, let the contents of the stack from bottom to top be $(j_1, k_1), (j_2, k_2), \ldots, (j_l, k_l)$ for some $l \geq 0$. The following properties hold for the contents of the stack.*

1. $j_1 < \cdots < j_l$,
2. $x_{j_1} < \cdots < x_{j_l}$, *and*
3. *if $T < T_i$, $x_{j_p} = \min_{j_{p-1} < h \leq i-1} x_h$ for $1 \leq p \leq l$.*
4. *if $T = T_i$, $x_{j_p} = \min_{j_{p-1} < h \leq i} x_h$ for each $1 \leq p \leq l$.*
5. $k_1 = l_{j_1} = 1$ *and, for $2 \leq h \leq l$, $k_h = l_{j_h} = j_{h-1} + 1$.*

We regard the stack as empty when $l = 0$. We assume that the properties 1, 2, and 3 will hold for an empty stack. For convenience, we assume $j_0 = 0$.

Claim 2.2. *If an item (j, k) at the top of the stack is replaced with an item (i, k), then i and j are equivalent.*

Proof: Assume that an item (j, k) on the stack is replaced with item (i, k) during round i. By Claim 2.1, we have $x_j = \min_{k \leq l \leq i-1} x_l$ during round i. Item (j, k) will be replaced by (i, k) by the algorithm only if $x_j = x_i$. We then have $x_i = x_j = \min_{k \leq l \leq i} x_l$ which implies that i and j are equivalent.

Claim 2.3. *The following properties hold for each round i for $1 \leq i \leq n$.*

1. *If an item (j, k) is popped off the stack during round i, its right index $r_j = i - 1$. If the item (j, k) is popped off the stack during the post processing phase, its right index $r_j = n$.*
2. *x_i enters the stack during round i. More specifically, the item (i, k) will be on top of the stack for some k at T_i.*

Claim 2.4. *The stack is empty at the end of the algorithm.*

The correctness of the algorithm follows from the Claims 2.1, 2.2, 2.3, and 2.4. More specifically, we know that when an item (j, k) is popped off the stack, its right and left indices have been determined correctly so the area of $t(j)$ is computed correctly. Moreover, the claims guarantee that for every $1 \leq i \leq n$ there is an item (j, k) that is popped off the stack at some point where j is equivalent to i. This implies that all the areas $t(i)$ have been computed correctly. Since M is keeping tracking of the maximum of the areas, the algorithm outputs the correct answer.

Students are strongly encouraged to supply the missing proofs.

Problem 2: 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 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 its 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 (whenever a new vertex is encountered) and marking the visited nodes. When the BFS is done, we add the pond size to the list of pond sizes.
- Otherwise, we continue the loop.

Proof of correctness: Starting with any node of a given connected component, we know that BFS visits all the nodes in the connected component. 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 no 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 (that is, a vertex with degree 1) 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 2.5. *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 2.6. *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.