Algorithms: CSE 202 — Homework 0 solutions

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

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

Let $S = x_1, x_2, \ldots, x_n$ be a sequence of nonnegative integers for $n \geq 1$. Let $s = x_i, \ldots, x_j$ be a contiguous subsequence of S for some $1 \leq i \leq j \leq n$. We use the interval $[i, j]$ to represent s. We call i the left index and j the right index of s. We define the area of $s = [i, j]$ as

$$\text{area}(s) = \text{area}[i, j] = \min_{i \leq k \leq j} x_k (j - i + 1)$$

We define $A_S = \max_{1 \leq i \leq j \leq n} \text{area}[i, j]$. A_S is the area of the largest area contiguous subsequence of S. We use the following divide-and-conquer scheme to compute A_S.

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 A_L and A_R.

We will now consider contiguous subsequences which start in L and end in R. We call such contiguous subsequences *middle contiguous subsequences* with respect to L and R. Let A_C denote the area of the largest area middle contiguous subsequence. Below, we will show how to compute A_C in linear time. Finally, we output $\max\{A_L, A_R, A_C\}$ as A_S.

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 middle contiguous subsequence with respect to L and R (type 3).

Assuming the correctness of the algorithm for finding the maximum area middle contiguous subsequence of S, we argue by induction that the overall algorithm produces the correct output since A_L is the maximum area of the contiguous subsequences of type 1, A_R is the maximum area of the contiguous subsequences of Type 2, and A_C 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 A_C: 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 m_i denote the minimum value of the sequence $x_i, \ldots, x_q$. For $j \in V$, let m_j denote the minimum value of the sequence $x_{q+1}, \ldots, x_j$. m_i is nonincreasing as i decreases from q to 1. Similarly, m_j is nonincreasing as j increases from $q+1$ to n. It is easy to see that m_i and m_j can be computed in linear (in the sum of the lengths of L and R) time by scanning the L from right to left and R from left to right.

We will now describe how to compute A_C in linear time. To simplify the presentation, we assume that m_i for $i \in U$ and m_j for $j \in V$ have already been computed, although one can easily interleave the computation of these quantities with that of A_C.

Algorithm for computing A_C: To compute A_C, 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 $m_q \leq m_{q+1}$, we initialize

$$i \leftarrow \arg\min_{1 \leq k \leq q} m_k = m_q \text{ and}$$
$$j \leftarrow \arg\max_{q+1 \leq l \leq n} m_l \geq m_q.$$

If $m_q > m_{q+1}$, we initialize

$$i \leftarrow \arg\min_{1 \leq k \leq q} m_k \geq m_{q+1} \text{ and}$$
$$j \leftarrow \arg\max_{q+1 \leq l \leq n} m_l = m_{q+1}.$$

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 middle contiguous subsequence $x_i, \ldots, x_j$ which is equal to $m_i(j - i + 1)$.

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

$$\begin{array}{ll}
i \leftarrow \arg\min_{1 \leq k < i} m_k = m_{i-1} \text{ and } j \leftarrow j & \text{if } i > 1,\ j < n \text{ and } m_{i-1} \geq m_{j+1} \\
i \leftarrow i \text{ and } j \leftarrow \arg\max_{j < l \leq n} m_l = m_{j+1} & \text{if } i > 1,\ j < n \text{ and } m_{i-1} < m_{j+1} \\
i \leftarrow i \text{ and } j \leftarrow \arg\max_{j < l \leq n} m_l = m_{j+1} & \text{if } i = 1 \text{ and } j < n \\
i \leftarrow \arg\min_{1 \leq k < i} m_k = m_{i-1} \text{ and } j \leftarrow j & \text{if } i > 1 \text{ and } j = n \\
\text{terminate the algorithm} & \text{if } i = 1 \text{ and } j = n
\end{array}$$

We compute area$[i', j']$, update $M \leftarrow \max\{M, \text{area}[i', j']\}$ and repeat.

Remarks about the termination and the time complexity of the algorithm for computing A_C: 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. Let $t \geq 1$. 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 $i > 1$, $m_{i-1} \leq m_j$.
2. If $j < n$, $m_{j+1} \leq m_i$.
3. $M = \max_s \text{area}(s)$ where s ranges over all middle contiguous subsequences in the interval $[i, j]$

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 $m_q \leq m_{q+1}$. The other case can be handled in a similar fashion. By construction, we have $m_i = m_q$ and i is the smallest such index. Also, we have $m_i \leq m_j$. Hence, if $i > 1$, $m_{i-1} < m_i \leq m_j$. Also, if $j < n$, $m_{j+1} < m_i$ since j is the largest index such that $m_i \leq m_j$. At the beginning of the first round, M is equal to the area of the middle contiguous subsequence $x_i, \ldots, x_j$ at this point. Since $m_i = m_q$ and $m_j \geq m_i$, we deduce that m_i is the minimum value of the sequence and no middle contiguous subsequence inside $[i, j]$ can have a larger minimum value. Hence, $M = \max_s \text{area}(s)$ where s ranges over all the middle contiguous subsequences in $[i, j]$.

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, $M = \max_s \text{area}(s)$ where s ranges over all the middle contiguous subsequences in $[i, j]$.

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: $i > 1$**,** $j < n$**,** $m_{i-1} \geq m_{j+1}$**,** $i' \leftarrow \arg\min_{1 \leq k < i} m_k = m_{i-1}$ **and** $j' \leftarrow j$

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

If $j' < n$, $j < n$ and $m_{j'+1} = m_{j+1}$ since $j' = j$. Also, we have $m_{i'} = m_{i-1}$ by construction and $m_{i-1} \geq m_{j+1}$ by case analysis. By chaining these conditions, we conclude $m_{j'+1} \leq m_{i'}$ establishing the second property.

For the third property, we consider a middle contiguous subsequence s in the interval $[i', j']$. If s is also in $[i, j]$, we know that $M' \geq \text{area}(s)$ 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 $x_k, \ldots, x_l$ for some $i' \leq k < i$ and $q+1 \leq l \leq j'$. The minimum value of any such sequence is m_{i-1} since $m_k = m_{i'} = m_{i-1}$ (by construction) and $m_{i-1} \leq m_j = m_{j'}$ (by inductive hypothesis). Since $[i', j']$ is the longest middle contiguous subsequence in $[i', j']$ with the minimum value m_{i-1} and $M' = \max\{\max_{s'} \text{area}(s'), \text{area}[i', j']\}$ where s' ranges over all middle contiguous subsequences in $[i, j]$, we conclude $M' = \max_s \text{area}(s)$ where s ranges over all middle contiguous subsequences in $[i', j']$.

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

If $i' > 1$, $i > 1$ and $m_{i'-1} = m_{i-1}$ since $i' = i$. We also have $m_{j+1} = m_{j'}$. We have $m_{i-1} < m_{j+1}$ by case analysis which implies $m_{i'-1} < m'_j$ establishing the first property.

If $j' < n$, $j < n$ and $m_{j'+1} < m_{j'} = m_{j+1}$ by construction since j' is the largest index such that $m_{j'} = m_{j+1}$. Again, by construction, we have $m_{i'} = m_i$. We have $m_{j+1} \geq m_i$ by inductive hypothesis. which implies $m_{j'+1} \leq m_{i'}$ establishing the second property.

For the third property, we consider a middle contiguous subsequence s in the interval $[i', j']$. If s is also in $[i, j]$, we know that $M' \geq \text{area}(s)$ 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 $x_k, \ldots, x_l$ for some $i' \leq k \leq q$ and $j << l \leq j'$. The minimum value of any such sequence is m_{j+1} since $m_l = m_{j'} = m_{j+1}$ by construction and $m_{j+1} \leq m_i = m_{i'}$ by inductive hypothesis. Since $[i', j']$ is the longest middle contiguous subsequence in $[i', j']$ with the minimum value m_{j+1} and $M' = \max\{\max_{s'} \text{area}(s'), \text{area}[i', j']\}$ where s' ranges over all middle contiguous subsequences in $[i, j]$, we conclude $M' = \max_s \text{area}(s)$ where s ranges over all middle contiguous subsequences in $[i', j']$.

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

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

If $j' < n$, $j < n$ and $m_{j'+1} < m_{j'} = m_{j+1}$ by construction. We also have $m'_i = m_i = m_1$ since $i' = i = 1$. However, by inductive hypothesis we have $m_{j+1} \leq m_i$ which implies $m_{j'+1} \leq m_{i'}$ satisfying the second property of the claim.

For the third property, we consider a middle contiguous subsequence s in the interval $[i', j']$. If s is also in $[i, j]$, we know that $M' \geq \text{area}(s)$ 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 $x_k, \ldots, x_l$ for some $i' \le k \le q$ and $j < l \le j'$. The minimum value of any such sequence is m_{j+1} since $m_l = m_{j'} = m_{j+1}$ (by construction) and $m_{j+1} \le m_i = m_{i'}$ (by inductive hypothesis). Since $[i', j']$ is the longest middle contiguous subsequence in $[i', j']$ with the minimum value m_{j+1} and $M' = \max\{\max_{s'} \text{area}(s'), \text{area}[i', j']\}$ where s' ranges over all middle contiguous subsequences in $[i, j]$, we conclude $M' = \max_s \text{area}(s)$ where s ranges over all middle contiguous subsequences in $[i', j']$.

Case IV: $i > 1$**,** $j = n$**,** $i' \leftarrow \arg\min_{1 \le k < i} m_k = m_{i-1}$ **and** $j' \leftarrow j$

If $i' > 1$, $i > 1$ and $m_{i'-1} < m_{i'} = m_{i-1}$. We also have $m_{j'} = m_j = m_n$ since $j' = j = n$. However, by inductive hypothesis we have $m_{i-1} \le m_j$ which implies $m_{i'-1} \le m_{j'}$ satisfying the first property of the claim.

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

For the third property, we consider a middle contiguous subsequence s in the interval $[i', j']$. If s is also in $[i, j]$, we know that $M' \ge \text{area}(s)$ 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 $x_k, \ldots, x_l$ for some $i' \le k < i$ and $q+1 \le l \le j'$. The minimum value of any such sequence is m_{i-1} since $m_k = m_{i'} = m_{i-1}$ (by construction) and $m_{i-1} \le m_j = m_{j'}$ (by inductive hypothesis). Since $[i', j']$ is the longest middle contiguous subsequence in $[i', j']$ with the minimum value m_{i-1} and $M' = \max\{\max_{s'} \text{area}(s'), \text{area}[i', j']\}$ where s' ranges over all middle contiguous subsequences in $[i, j]$, we conclude $M' = \max_s \text{area}(s)$ where s ranges over all middle contiguous subsequences in $[i', j']$.

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.

Linear-time algorithm

Our algorithm takes as input a sequence of nonnegative real numbers $S = a_1, a_1, ..., a_n$ and outputs the area of the maximum area contiguous subsequence. Let $s = a_i, \ldots, a_j$ be a contiguous subsequence of S for some $1 \le i \le j \le n$. We use the interval $[i, j]$ to represent s. We call i the left index and j the right index of s. Area of $s = [i, j]$ as

$$\text{area}(s) = \text{area}[i, j] = \min_{i \le k \le j} a_k (j - i + 1)$$

We define $A_S = \max_s \text{area}(s)$ where the maximum is taken over all contiguous subsequences of S.

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 $1 \le i \le n$, let $t(i)$ denote the longest contiguous subsequence which contains a_i and whose minimum value is a_i. It is clear that $A_S = \max_i \text{area}(t(i))$.

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

$$l_i = \arg\min_{1 \le j \le i} a_j \ge a_i$$

Similarly, we can express r_i as

$$r_i = \arg\max_{i \le j \le n} a_j \ge a_i$$

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

$$\text{area}(t(i)) = a_i(r_i - l_i + 1)$$

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 a_i and subsequently determines the right index when it first encounters an element less than a_i. We maintain a stack to store items of the form (j, k) where a_j is one of the processed elements $k = l_j$. We call a_j the value of the item, 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 a_j after (j, k) is placed on the stack. After processing the first $i \ge 1$ items, the stack contains items of the form (j, l_j) where $1 \le j \le i$

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 $a_i, a_{i-1}, \ldots, a_1$. We will make this concept precisely in the following.

A linear-time algorithm for computing A_S

Preprocessing phase: We process the elements a_i as i ranges from 1 through n. We also maintain a variable M which is equal to $\max_i \operatorname{area}(t(i))$ where i is the index of an item and it ranges over all items popped from the stack so far. In other words, $M = \max_i \operatorname{area}(t(i))$ 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 a_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 $a_i > a_j$, push $(i, j+1)$ on the stack and exit the loop.
4. If $a_i = a_j$, change the top item (j, k) (with out popping it off the stack) to (i, k) and exit the loop.
5. If $a_i < a_j$, set $l_j = k$, $r_j = i - 1$, compute $\operatorname{area}(t(j))$, and update M. Pop (a_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 a_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 two claims.

Claim 2. 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. $a_{j_1} < \cdots < a_{j_l}$, and
3. if $T < T_i$, $a_{j_p} = \min_{j_{p-1} < h \leq i-1} a_h$ for $1 \leq p \leq l$.
4. if $T = T_i$, $a_{j_p} = \min_{j_{p-1} < h \leq i} a_h$ 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 3. 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, we have $a_j = \min_{k \leq l \leq i-1} a_l$ during round i. Item (j, k) will be replaced by (i, k) by the algorithm only if $a_j = a_i$. We then have $a_i = a_j = \min_{k \leq l \leq i} a_l$ which implies that i and j are equivalent.

Claim 4. 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. a_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 5. The stack is empty at the end of the algorithm.

The correctness of the algorithm follows from the Claims 2, 3, 4, and 5. 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.

Problem 2: Balanced Parentheses

You are given a string consisting of right (}) and left ({) brackets. The string may or may not be balanced. You are allowed to make the following alterations to the string: change a left bracket to a right bracket or a right bracket to a left one. Balance the string with minimum number of changes.

Solution: Balanced Parentheses

Algorithm description: We assume that the input string is of even length, otherwise we cannot balance the parentheses.

The algorithm scans the input string from left to right and uses a stack to store the indices of the unmatched opening parentheses encountered so far. If the next parenthesis is an opening parenthesis, we push its index on the stack. If the next parenthesis is a closing parenthesis, we match it with the top of the stack and pop the top off the stack accordingly. If the stack is empty when we encounter a closing parenthesis, we flip the parenthesis to an opening parenthesis and push its index on the stack. Once we reach the end of the string and there are $k > 0$ unmatched opening parentheses left on the stack, we flip the top $k/2$ of the opening parentheses on the stack to closing parentheses.

Pseudocode:

Algorithm Flip:

```
Input  : String $x_1, \dots, x_n$
Output: Sets $O, C$ of indices where we flip. $O$ are parenthesis we flip from closed to open, $C$ we flip
        from open to closed
unmatched ← empty stack
$C \leftarrow \varnothing$
$O \leftarrow \varnothing$
for $i = 1$ to $n$ do
    if $x_i == \{$ then
        unmatched.push($i$)
    end
    else
        \\$x_i ==\}$
        if unmatched.empty() then
            $O \leftarrow O \cup \{i\}$
            unmatched.push($i$)
        end
        else
            unmatched.pop()
        end
    end
end
size ← unmatched.size for $i = 1$ to size/2 do
    $C \leftarrow C \cup \{$ unmatched.top()$\}$
    unmatched.pop()
end
return $O$, $C$
```

Time analysis: This algorithm runs in time $O(n)$ since we iterate over the string exactly once and only spend constant time per position.

Proof of correctness: To argue correctness we introduce some notation. For a string $z = z_1 \dots z_n$, let $\mathrm{Bal}_z(i)$ be the *balance* of the prefix of length i, that is, the difference between the number of opening parentheses and the number of closing parentheses in the string $z_1 \dots z_i$. We observe that a string z of parentheses is balanced if and only if for all $1 \le i \le n$ we have $\mathrm{Bal}_z(i) \ge 0$ and $\mathrm{Bal}_z(n) = 0$. Furthermore let the *depth* of a string z be $d_z = -\min_{1 \le i \le n} \mathrm{Bal}_z(i)$, i.e., the furthest the balance ever goes negative.

For string z we call $S_z = (O_z, C_z)$ a pair of flips where O_z is a subset of indices of closing parentheses in z and C_z is subset of indices of opening parentheses in z. With a slight abuse of notation, we also use S_z to denote the string obtained by flipping the parentheses in z at the indices given by O_z and C_z. We say that the pair of flips S_z is *valid* for z if S_z is a balanced string of parentheses.

Let $O_x^{\mathbf{Flip}} \subseteq [n]$ be the set of indices of the closing parentheses in x that have been flipped during the execution of the algorithm. Similarly, let $C_x^{\mathbf{Flip}}$ denote the set of indices of the opening parentheses in x that have been flipped. Finally, let $S_x^{\mathbf{Flip}} = (O_x^{\mathbf{Flip}}, C_x^{\mathbf{Flip}})$ be the pair of flips produced by the algorithm.

We argue $S_x^{\mathbf{Flip}}$ is balanced. Furthermore, we will argue that $|O_x^{\mathbf{Flip}}| + |C_x^{\mathbf{Flip}}|$ is the minimum number of flips required to balance x. In the following we provide proofs of these claims.

Claim 6. $S_x^{\mathbf{Flip}}$ is balanced string of parentheses.

Proof. Let $y = S_x^{\mathbf{Flip}}$. We will show that $\mathrm{Bal}_y(i) \ge 0$ for $1 \le i \le n$ and $\mathrm{Bal}_y(n) = 0$. Observe that $\mathrm{Bal}_y(i)$ is exactly the number of indices on the stack after processing the parenthesis x_i. Since the string x has even length, we have an even number of opening parentheses on the stack at the end before we start flipping the open parentheses. Since we are flipping the top half of the parentheses on the stack, we are assured that $\mathrm{Bal}_y(i)$ stays non-negative and becomes zero at the end. □

Let $S_x = (O_x, C_x)$ be a pair of flips for x.

Claim 7. If S_x is valid, we have $|C_x| = \mathrm{Bal}_x(n)/2 + |O_x|$.

Proof. String x has exactly $\frac{n+\mathrm{Bal}_x(n)}{2}$ opening parentheses. After applying the flips to x at the indices specified by O_x and C_x, the string S_x has exactly $\frac{n+\mathrm{Bal}_x(n)}{2} + |O_x| - |C_x|$ opening parentheses. Since S_x is valid, it must have exactly $n/2$ open parentheses. S_x has exactly $n/2$ open parentheses if and only if the equality in the claim is true. ☐

Claim 8. If string x has depth d_x, then for any valid $S_x = (O_x, C_x)$ we have $|O_x| \geq \lceil \frac{d_x}{2} \rceil$.

Proof. Let $i \in O_x$. Flipping the closing parenthesis in position i to an opening parenthesis does not change the balance at any position before i and it increases $\mathrm{Bal}_x(i)$ by exactly 2. Let j be an index at which the maximum negative balance d_x is achieved, that is, $\mathrm{Bal}_x(j) = -d_x$. For the string S_x to be balanced, it is necessary that $\mathrm{Bal}_{S_x}(j) \geq 0$. For this to happen, we must flip at least $\lceil d_x/2 \rceil$ closing parentheses to opening parentheses in the string $x_1 \ldots x_j$. ☐

Claim 9. If the input string x has depth d_x, then $S_x^{\mathbf{Flip}} = (O_x^{\mathbf{Flip}}, C_x^{\mathbf{Flip}})$ satisfies $|O_x^{\mathbf{Flip}}| = \lceil \frac{d_x}{2} \rceil$

Proof. We prove this claim by induction on d_x. If $d_x = 0$, then the algorithm never tries to pop an element off an empty stack, hence $O_x^{\mathbf{Flip}} = \varnothing$.

Let $d_x > 0$ and consider the first position $i \in O_x$. Consider the string x' obtained by flipping the parenthesis at position i in x. The flip at i increases the balance for every position in x' at or after i by exactly 2. Since i is the first position in x with negative balance, the depth of x' is

$$d_{x'} = \begin{cases} d_x - 2 & \text{if } d_x \geq 2 \\ 0 & \text{otherwise.} \end{cases}$$

We now regard the execution of the algorithm as if we started with the string x' and the flip never happened.

By the induction hypothesis, the Algorithm Flip will flip exactly $\lceil \frac{d_{x'}}{2} \rceil$ opening parentheses to achieve balance. Hence the total number of times a closing parenthesis is flipped to an opening parenthesis by Algorithm Flip is $1 + \lceil \frac{d'_x}{2} \rceil = \lceil \frac{d_x}{2} \rceil$. ☐

Claim 10. Algorithm Flip balances its input x with minimal number of flips.

Proof. By Claim 6, $S_x^{\mathbf{Flip}}$ is a valid solution. By Claim 9, the algorithm flips $\lceil d_x/2 \rceil$ closing parentheses to opening parentheses, which is necessary by Claim 8. By Claim 7 if we minimize the number of flips in one direction we also minimize the number of flips in the other direction. We have proved that the algorithm produces a valid solution and that it minimizes the number of flips. ☐

Problem 3: Olay

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. From each well, a spur pipeline is to be connected directly to the main pipeline along a shortest path (either north or south). Given x and y coordinates of the wells, how should the professor pick the optimal location of the main pipeline (the one that minimizes the the total length of the spurs?) Show that the optimal location can be determined in linear time.

Solution: Olay

Algorithm description: Since the main pipeline is running east-west and the different spur pipelines are running north-south, the problem reduces to one in which we have to minimize the sum total of the displacements (along the north-south direction) of each well from the main pipeline. Let $y_1 \le \cdots \le y_n$ be the y-coordinates of the n wells. Assume that the main pipeline is located along the line $y = z$. We have to find a value for z so that the sum of the displacements $S(z) = \sum_{j=1}^{n} |y_j - z|$ is minimized. We claim that setting z to be the median of y_i would provide the optimal solution.

Proof of correctness: We need to minimize $S(z)$ over z. It is clear that the optimal z is between y_1 and y_n, that is, $y_1 \le z \le y_n$. Define $D(i, \delta) = S(y_i + \delta) - S(y_i) = i\delta - (n-i)\delta = (2i-n)\delta$ for $0 \le \delta \le y_{i+1} - y_i$. $\delta \ge 0$ since $y_{i+1} \ge y_i$. If $D(i, \delta) \ne 0$, then its sign is determined by the sign of $(2i - n)$. For each interval $[y_i, y_{i+1}]$, $S(z)$ is nonincreasing in the interval if $2i - n < 0$ and $S(z)$ is nondecreasing in the interval if $2i - n > 0$. In other words, $S(z)$ is nonincreasing for a while and then is nondecreasing as z increases from y_1 to y_n. We consider two cases:

$n \ge 2$ **is even:** $S(z)$ is nonincreasing as z increases from y_1 to $y_{n/2}$, stays constant for $y_{n/2} \le z \le y_{n/2+1}$, and is nondecreasing as z increases from $y_{n/2+1}$ to y_n. This shows $S(z)$ achieves its minimum at every point in the interval $[y_{n/2}, y_{n/2+1}]$. Hence the median, $(y_{n/2} + y_{n/2+1})/2$, is an optimal location.

$n \ge 1$ **is odd:** $S(z)$ is nonincreasing as z increases from y_1 to $y_{\lceil n/2 \rceil}$ and is nondecreasing as z increases from $y_{\lceil n/2 \rceil}$ to y_n. Hence the median, $y_{\lceil n/2 \rceil}$, is the optimal location.

Time Analysis: Since the median finding algorithm takes linear time, time complexity for finding the optimal location is linear as well.

Problem 4: 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 $\max_{1 \le a < c \le n, 1 \le b < d \le n} M[c,d] - M[a,b]$.

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

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

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

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

We define $D[i,j]$ for $1 < i, j \le n$ 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 $1 \le i, j \le n$, we show how to compute $T[i,j]$ and $D[i,j]$ in constant time. After we compute the matrix D, we output $\max_{1 \le i,j \le n} D[i,j]$ 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 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 11. 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 12. 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.