Algorithms: CSE 202 — Homework 1 Solutions

Problem 1: Diameter of a tree (CLRS)

The *diameter* of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ is the shortest path length between the nodes u and v. That is, the diameter of the tree is the length of the longest shortest-path between any two nodes in the tree. Give an efficient algorithm to compute the diameter of a tree and analyze its running time.

We are expecting a linear (in the number of nodes of the tree) time algorithm for this problem.

Solution: Diameter of a tree (CLRS)

Assume that the tree is rooted. If not, we create a rooted tree in linear time.

Note that for any two nodes, there is a unique path in the tree. As a consequence, the longest of the shortest paths between any two nodes in a tree must either run through the root of the tree or be entirely contained in a subtree rooted at one of the children. This leads to the followingt recursive formulation. At each level of the recursion, we deal with a subtree T_v rooted at node v and we keep track of the longest path P_v contained in T_v and the longest path p_v from the root of T_v to a node in T_v. Let L_v be the length of the path P_v and l_v be the length of the path p_v. Assume that we computed L_u (and P_v) and l_u (and p_u) for each T_u where u is a child of v. We then compute L_v and l_v as follows:

$$l_v \leftarrow \max_{\text{child } u} l_u + 1$$
$$L_v \leftarrow \max(\max_{\text{child } u} L_u, \max_{\text{children } u \neq u'} l_u + l_{u'} + 2)$$

Informally speaking, to compute L_v, we take the maximum of two quantities: maximum of L_u over the children u of v and the length of the longest path (through v) that can be formed by combining the paths p_u and $p_{u'}$ for children u and u' together with the edges from u and u' to their parent v. If v has only one child u, then the second quantity is just $l_u + 1$. If v has no children, then $l_v = L_v = 0$.

Note that we need to maintain additional information to determine the paths P_v and p_v.

The reader is encouraged to supply the missing detail.

Complexity: The algorithm takes linear time since each of the steps in the computation can be attributed to an appropriate node (the root of the subtree under question) and each node acrrues at most c steps for some constant c.

Proof of Correctness: We argue that L_v and l_v are computed correctly for each subtree T_v by induction on the height of the tree. If T_v has height 0 (in which case it has only one node v), we have $L_v = l_v = 0$ since all paths have length 0 in this tree. Our algorithm also computes the same values for l_v and L_v.

Assume that the height of v is greater than 0. Let $u_1, \ldots, u_k$ be its children for $k \geq 1$. Assume that L_{u_i} and l_{u_i} have been computed correctly for $1 \leq i \leq k$. Either the longest path in T_v does not contain v or it contains v. If it does not contain v, then it must be entirely in one of the subtrees T_{u_i} and its length is $\max_{1 \leq i \leq k} L_{u_i}$. Consider the case that the longest path in T_v contains v. Such a path must look like $v_1, \ldots, u_i, v, u_j, \ldots, v_2$ where the nodes $v_1, \ldots, u_i$ are in the subtree T_{u_i} and $u_j, \ldots, v_2$ are in another subtree

T_{u_j} for some $1 \leq i \neq j \leq k$. In the special case where v has only one child, $v_2 = v$. The length of such a path is indeed $l_{u_i} + l_{u_j} + 2$, which is exactly what our algorithm computes. Similarly, we can argue the correctness for the computaiton of l_v. The reader is strongly encouraged to supply the missing details in the proof.

Problem 2: Sorted matrix search

Given an $m \times n$ matrix in which each row and column is sorted in ascending order, design an algorithm to find an element.

If the element occurs multiple locations in the matrix, your algorithm is allowed to return the element from any location.

Solution: Sorted Matrix Search

Algorithm description:
Let $A_{i,j}$ be a 2-dimensional matrix where $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$ such that every row and every column of A is sorted in increasing order. We search for an element x in A by following a walk in the matrix as follows:

- Let c be the element in the top right corner of the matrix.
- If $x = c$, we declare that x is found.
- If $x < c$, then we move one column to the left while remaining at the same row and search in the submatrix obtained by deleting the last column.
- If $x > c$, then we move one row down while remaining at the same column and search in the submatrix obtained by removing the first row of the matrix.
- This process is repeated until either the element is found or the matrix is an empty matrix without any rows and columns.

Proof of Correctness: If x is not in A it is clear that the algorithm is correct. If x is in A, then the correctness of the algorithm follows from this claim.

Claim 0.1. *Let c be the element in the top right corner of a matrix whose rows and columns are sorted in increasing order and x be an element in the matrix. If $x < c$, then x is in the submatrix obtained by deleting the last column. If $x > c$, then x is in the submatrix obtained by removing the first row of the matrix.*

We argue this claim by considering the two cases seperately. If $x < c$, then x is strictly less than every element in the last column of the matrix since the column is sorted in increasing order and c is its first element. Hence, it must be in the submatrix obtained by deleting the last column of the matrix.

Now suppose instead that $x > c$. Then x is greater than every element in the first row of the matrix since the row is sorted in increasing order and c is its last element. Hence, it must be in the submatrix obtained by deleting the first row of the matrix.

Complexity: In this algorithm, at every step we eliminate either a row or a column or terminate if the target is found. So, we look at a maximum of $m + n$ elements in the matrix. So the time complexity is $O(m + n)$. Since, we do not use any extra space, the space complexity is $O(1)$.

Problem 3: 132 pattern

Given a sequence of n distinct positive integers $a_1, \ldots, a_n$, a 132-pattern is a subsequence a_i, a_j, a_k such that $i < j < k$ and $a_i < a_k < a_j$. For example: the sequence $31, 24, 15, 22, 33, 4, 18, 5, 3, 26$ has several 132-patterns including 15, 33, 18 among others. Design an algorithm that takes as input a list of n numbers and checks whether there is a 132-pattern in the list.

Solution: 132 Pattern

1 Solution I

This solution is based on the submission of Yuwei Wang.
Note: Students are advised to supply all the missing details and proofs in the following.

High level description

Let $a_1, \ldots, a_n$ be a sequence of distinct integers. We will let $a_0 = \infty$ for the rest of the discussion as it simplifies our definitions without affecting correctness.

For $1 \le i \le n$, let $\pi(i) = \max\{j \mid 0 \le j < i \text{ and } a_j > a_i\}$. In other words, $\pi(i)$ is the index of the nearest greater element to the left. For all $1 \le i \le n$, $\pi(i)$ is defined since $\{j \mid 0 \le j < i \text{ and } a_j > a_i\}$ is not empty. Furthermore for $1 \le i \le n$, $0 \le \pi(i) < i$.

For $1 \le i \le n$, let $\sigma(i) = j$ where j is such that $a_j = \min_{0 \le k < i} a_k$. In other words, $\sigma(i)$ is the index of the smallest element to the left of position i. We have $0 \le \sigma(i) < i$ for all $1 \le i \le n$. Note that $\sigma(i) \ge 1$ for all $i \ge 2$ since there is at least one element to the left of position i.

For $i \ge 1$, let $T(i) = (a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$. For each i, the triple $T(i)$ contains a_i as the third component, its previous greater element $a_{\pi)i)}$ as the second component, and the smallest element before the previous greater elements as the first component.

Fact 1.1. *For $i \ge 3$ such that $\pi(i) \ge 2$, we have $1 \le \sigma(\pi(i)) < \pi(i) < i$.*

Fact 1.2. *For $i \ge 3$ such that $\pi(i) \ge 2$, if $a_{\sigma(\pi(i))} < a_i$, then $T(i)$ is a 132 pattern.*

For $1 \le k < j < i$, let the triple $S = (k, j, i)$ be such that (a_k, a_j, a_i) is a 132 pattern, that is, $a_k < a_i < a_j$. We say S is canonical if

- $i \ge 3$ is the smallest index among all such triples,
- $j \ge 2$ is the largest index among all triples with the smallest i, and
- $k = \sigma(j)$.

Fact 1.3. *If the sequence has a 132 pattern, then there is a canonical triple (k, j, i) of indexes such that (a_k, a_j, a_i) is a 132 pattern.*

Here is a high-level sketch of the algorithm which outputs a triple for each i132-pattern for each .

1. Compute the $\pi(i)$ for $1 \le i \le n$.
2. Compute $\sigma(i)$ for $1 \le i \le n$.
3. For each i starting with $i = 3$. check if $\pi(i) \ge 1$ and $a_{\sigma(\pi(i))} < a_i$ If the check is positive, the algorithm returns **True** (and also the triple $(a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$.

For every position $1 \le i \le n$, our algorithm checks if the triple $(a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$ forms a 132 pattern. In particular, we test if $a_{\sigma(\pi(i))} < a_i$. We output true if the test succeeds for some $3 \le i \le n$, false otherwise.

For the computation of $\pi(i)$, we refer to the solution of the "Next Greater Element" problem which is similar to the problem of finding the previous greter element.

Correctness proof

We assume the correctness of the first two steps of the algorithm. Under this assumption, we show that the algorithm returns true if and only if there is a 132 pattern. Readers are strongly encouraged to provide a detailed algorithm for each of the first two steps and prove its correctness.

We first show that if the algorithm returns true, then there is a 132 pattern. We prove this by induction on the iteration number of the third step of the algorithm. Assume that the algorithm returns true during iteration i for some $3 \leq i \leq n$. Consider the triple $(\sigma(\pi(i)), \pi(i), i)$. Let $j = \pi(i)$ and let $k = \sigma(j)$. We know $j < i$. Since the algorithm returned true during iteration i, we have $j \geq 1$ and $a_{\sigma(j)} < a_i$, which imply $j \geq 2$, and $1 \leq k < j$. Combining the inequalities we get $1 \leq k < j < i \leq n$. Since $a_j > a_i$ and $a_k < a_i$ (and thereby $a_k < a_j$), we conclude (k, j, i) forms a 132 pattern.

By Fact 1.3, we know that there is a canonical triple which forms a 132 a pattern if there is a 132 pattern. We show that if $T = (k, j, i)$ is a canonical triple that forms a 132 pattern (that is, $1 \leq k < j < i$ and $a_k < a_i < a_j$), the algorithm returns true during iteration i. Since $a_j > a_i$ and j is the largest such index we conclude $j = \pi(i)$. We also have $k = \sigma(j) \geq 1$ since T is canonical. Therefore algorithm returns true during iteration i and would not return true in any earlier iteration since i is the smallest index for which a 132 pattern exists.

Runtime analysis

$\sigma(i)$ can be computed by a single scan of the array which takes linear time. Similarly, it turns out that $\pi(i)$ can also be computed in a single scan in linear time. The last step of the algorithm also takes linear time. We conclude that the algorithm runs in time $O(n)$.

Solution: 132 pattern

2 Solution II

As in the previous solution, for $i \geq 0$, $\sigma(i)$ denotes the index of the smallest element to the left of i. We define $\sigma(0)$ to be ∞.
Algorithm description: Let $a_0, \ldots, a_n$ be a sequence of distinct positive integers. To find the existence of 132 pattern in the sequence, we employ the following two step algorithm.

- For $1 \leq i \leq n$, we compute $\sigma(i)$ in linear time by scanning the array from left to right.
- For each location $1 \leq i \leq n - 1$ we check if a_i is the middle element of a 132 pattern by scanning the list from right to left while maintaining a balanced binary search tree of all the elements seen so far. For each i we check if a_i is greater than the smallest number to its left, that is, if $a_i > a_{\sigma(i)}$. If so, we check if there is an element in the binary search tree whose value is between a_i and $a_{\sigma(i)}$. We do this as follows. We conduct the search for both a_i and $a_{\sigma(i)}$ in parallel. If at some point the searches lead to different branches of the tree, we conclude that there is an element between the two and terminate the process. Otherwise, we insert a_i in the binary search tree and continue the scan. If we reach a_0 in the process, we terminate and return False.

Here is a description of the algorithm wiht a bit more detail.

1. Compute $\sigma(i)$ for each $i \geq 1$ by scanning the list from left to right.
2. Scan the list from right to left and maintain a binary tree of all the elements scanned so far. Let T_i be the binary search tree just before we scan the i-th element where T_n is an empty tree. Check if $a_{\sigma(i)} < a_i$. If so, check if T_i has an element whose value is between $a_{\sigma(i)}$ and a_i. If the checks are satisfied, output **True**.

Proof of correctness:

We assume that $\sigma(i)$ is computed correctly for $i \geq 1$.

For $i < j < k$, we say that (a_i, a_j, a_k) is a *standard* 132 pattern if

- $a_i < a_k < a_j$,
- j is the largest index among all triples with 132 pattern,
- k is such that $\pi(k) = j$, and
- $i = \sigma(j)$.

where π and σ are as defined in the previous solution.

Fact 2.1. *If the array has a 132 pattern, then it has a standard 132 pattern.*

Fact 2.2. *Let T be a binary search tree and $\rho_1, \ldots, \rho_l$ be the sorted sequence of its keys. If two queries $q < q'$ (not in the search tree) take the same path in T, then one of the following holds:*

- $\exists 1 \leq i \leq l-1$ *such that* $\rho_i < q < q' < \rho_{i+1}$
- $q < q' < \rho_1$
- $\rho_l < q < q'$

Claim 2.3. *The algorithm returns true if there is at least one 132 pattern in the array.*

Proof: Since there is at least one 132 pattern, there exists a standard 132 pattern and let (a_i, a_j, a_k) be a standard 132 pattern where $i < j < k$, $j = \pi(k)$ and $i = \sigma(j)$. By definition, there is not 132 pattern where the index of the middle element is greater than j. We know that $a_{\sigma(j)} = a_i < a_j$. If the algorithm returns true before the j-iteration, the claim holds trivially. Consider the iteration (during the second step of the algorithm) when the element a_j is processed. At this point, the algorithm will search the binary search tree with the queries a_i and a_j since the condition $a_{\sigma(j)} = a_i < a_j$ is satisfied. Since the binary search tree contains at least one element (that is, a_k) which is in between a_i and a_j, the two queries must necessarily disagree at some node in the binary search tree which in turn implies that the algorithm returns true.

Claim 2.4. *If the algorithm returns true, there is at least one 132 pattern in the array.*

Proof: If the algorithm returned true, there is some iteration during which it returned true. Let a_j be the element processed during the iteration and $i = \sigma(j)$. Since the algorithm returned true during the iteration, the following statements hold:

- $\sigma(j) = a_i < a_j$, and
- during the iteration the algorithm searched the binary search tree with queries a_i and a_j and succeeded in finding a_k where $k \geq j$ and $a_i < a_k < a_j$.

This implies that there are elements a_i, a_j, and a_k in the array such that $i < j < k$ and $a_i < a_k < a_j$, which is a 132 pattern.

Complexity Analysis:

- The first step involves a single scan from left to right with constant time per element. The complexity of this step is $O(n)$.
- The second step involves a single scan from right to left with at most $O(n)$ find and insertion operations on the binary search tree. The complexity of this step is $O(n \log n)$.

Hence, the overall time complexity of the algorithm is $O(n \log n)$.

Problem 4: Next greater element

Given an array, print the Next Greater Element (NGE) for every element. The Next Greater Element of an item x is the first greater element to its right in the array.

Solution: Next greater element

Algorithm description: Let $a_1, a_2, \ldots, a_n$ be a list of n numbers. Let us assume without loss of generality that a_i is a positive integer for $1 \leq i \leq n$. If a_i does not have any greater element to its right, we define its next greater element to be -1. We compute the next greater element (NGE) for every element a_i by scanning the list from left to right and storing the elements for which we have not yet found the next greater element on the stack.

Here are the details of the algorithm. For each element a_i starting with $i = 1$ and an empty stack, we process a_i as follows until it is pushed onto the stack.

We push a_i onto the stack if the stack is empty or $a_i \leq t$ where t is the top of the stack. Otherwise, if $a_i > t$, we pop t from the stack and mark a_i as its next greater element and repeat the process of pushing a_i onto the stack.

After a_n is processed, we pop all the elements of the stack and mark -1 as their next greater element.

Proof of Correctness: We prove the correctness of the algorithm by induction on the number of iterations of the algorithm. Each iteration exactly involves processing one element of the sequence. More precisely,

Claim 2.5. *For $0 \leq i \leq n$, at the end of processing a_i, the following holds:*

1. *If the stack is not empty, indexes of the elements on the stack belong to the set $\{1, 2, \ldots, i\}$, are increasing and the values are decreasing as we go from the bottom of the stack to the top of the stack.*
2. *The elements on the stack do not have a next greater element among $a_1, \ldots, a_i$.*
3. *If an element e is popped out of the stack during iteration i, then a_i is the next greater element for e.*

We regard the beginning of iteration j as the end of iteration $j - 1$ for $1 \leq j \leq n$. Although we only push indexes onto the stack, with a slight abuse of the notation, we use the phrase top value on the stack to refer to the value at the index given by the top element of the stack.

Proof.
Base case ($i = 0$): At the end of iteration 0 the stack is empty, so properties $1, 2,$ and 3 are satisfied vacuously.
Inductive Step: Assume that the properties hold at the end of iteration j for all $0 \leq j \leq i < n$.

We will prove that the properties hold at the end of iteration $i + 1$. Consider the state of the program at the beginning of the iteration $i + 1$. There are several possibilities: the stack is empty; if not, a_{i+1} is less than the top value on the stack; or a_{i+1} is greater than the top value on the stack. In all scenarios, we prove that all three properties will hold.

- The stack is empty: In this case, we simply push a_{i+1} onto the stack during iteration $i + 1$. Property 1 holds trivially since there is only one element on the stack. Property 2 holds since the element a_{i+1} cannot have its next greater element in the list $a_1, \ldots, a_{i+1}$. Property 3 holds vacuously since no element has been popped during iteration $i + 1$ since the stack is empty at the beginning of the iteration.

- a_{i+1} is less than the top value on the stack: We push a_{i+1} onto the stack in this case. Since $i + 1$ is the largest index of all the indices considered and a_{i+1} is smaller than the top value on the stack, it follows from induction hypothesis that Property 1 is satisfied. Again by induction hypothesis and since a_{i+1} is smaller than the top value we conclude that none of the elements on the stack have a next greater element in $a_1, \ldots, a_{i+1}$. Since no elements have been popped from the stack, Property 3 holds vacuously.

- a_{i+1} is greater than the top value on the stack: Let $x_1, \dots, x_k$ be the indices of the elements on the stack with x_1 at the bottom and x_k at the top of the stack. Clearly $i+1 > x_k$ since $i+1$ is largest index we have accessed so far. We pop elements from the stack till the stack is empty or a_{i+1} is less than the top value on the stack. Hence, property 1 still holds at the end of iteration $i+1$.

  Let $a_{x_j}, \dots, a_{x_k}$ be the elements that were popped during the iteration. This implies that $a_{i+1} > a_{x_j} > \dots > a_{x_k}$. Additionally, we claim that for every element a_{x_m} where $m \in [j,k]$, there is no other element with index y such that $i+1 > y > m$ and $a_y > a_m$, since from property 2 at the end of iteration i we know that the elements on the stack do not contain their next greater element in the array $a_1, \dots, a_i$. Therefore all the popped elements have found their next greater element. Since a_{i+1} is smaller than the remaining elements on the stack, we conclude that values on the stack (including a_{i+1}) do not have their next greater element in $a_1, \dots, a_{i+1}$.

$\square$

Terminating Step: At the end of the loop, if the stack contains any elements (with indices $x_1, x_2, \dots, x_k$) then $x_1 < x_2 < \dots < x_k$ and $a_{x_1} > a_{x_2} > \dots > a_{x_k}$. Since the properties in the claim hold at the end of iteration n, the next greater element does not exist for the elements on the stack in $a_1, \dots, a_n$ so we set their next greater values to -1.

Complexity Analysis: Every element in the array is pushed exactly once onto the stack and popped exactly once from the stack. Overall, there are only constant number of operations per element. Therefore, the algorithm runs in $\Theta(n)$ time.

Problem 5: Base conversion

Give an algorithm that inputs an array of n base b_1 digits representing a positive integer in base b_1 in little endian format (that is, the least significant digit is at the lowest array index) and outputs an array of base b_2 digits representing the same integer in base b_2 (again in little endian format). Get as close as possible to linear time. Assume b_1, b_2 are fixed constants.

If we do the conversion digit-by-digit, the number of operations is really $O(n^2)$, since, for example, once we convert each digit we need to add up all the resulting $O(n)$ bit numbers. We can do better using a divide-and-conquer algorithm using a FFT integer multiplication algorithm as a sub-routine. Assume n is a power of 2; otherwise, pad with 0's in the most significant digits.

Solution: Base conversion

Convert(input: $X[0], \dots, X[n-1]$, an array of n digits in base $b_1 > 1$; output: an array of digits in base $b_2 > 1$)

1. If $n = 1$, output $X[0]$ in base b_2
2. $Z[0], \dots, Z[n/2] \leftarrow (b_1)^{n/2}$ (1 followed by $n/2$ 0's; the array $Z[0], \dots, Z[n/2]$ is $b_1^{n/2}$ in base b_1).
3. $A \leftarrow$ **Convert**$(X[0], \dots, X[n/2-1])$
4. $B \leftarrow$ **Convert**$(X[n/2], \dots, X[n-1])$
5. $C \leftarrow$ **Convert**$(Z[0], \dots, Z[n/2])$
6. Return **Add**(**Mult**$(A, C), B)$ (where **Mult** is the $O(n \log n)$ FFT multiplication algorithm from class and **Add** is the $O(n)$ grade-school addition algorithm).

If X has n digits in base b_1, $X < b_1^n$ and X has at most $\lceil \log_{b_2} X \rceil < n \log_{b_2} b_1 + 1 \leq cn$ digits in base b_2 for some constant $c > 0$. Therefore, the **Mult** and **Add** algorithms in the last line are called on $O(n)$ digit arrays, and their total time is $O(n \log n)$. Thus, the recurrence from the above is $T(n) = 3T(n/2) + O(n \log n)$. We solve the recurrence relation to get $T(n) = O(n^{\log_2 3})$. This is an improvement, but we can do much better.

Note that the above algorithm repeatedly converts $b_1, b_1^2, \ldots, b_1^{2^i}, \ldots, b_1^{n/2}$ to base b_2. Instead of computing them recursively, we will compute them once and save them for reuse. We can then do the recursion with only two calls.

We will compute the powers of b_1 in base b_2 and store them in a two dimensional array $P[i, j], 1 \leq i \leq \log n$, $1 \leq j \leq cn$. $P[i, j]$ is the j-th digit of $b_1^{2^i}$ in base b_2 for $0 \leq i \leq \log n$. We compute $P[i, j]$ recursively. Let $P[1, j]$ is the j-th digit of b_1 in base b_2. $P[i + 1, j] = \mathbf{Mult}(P[i, *], P[i, *])$ where $P[i, *]$ denotes the sequence of digits $P[i, 0], \ldots, P[i, cn]$. This recursivion can be implemented in $O(n)$ memory. It takes at most $O(n \log n)$ time to perform each multiplication, so we get an $O(n \log^2 n)$ upper bound on the total time for to compute P. (Actually, since the length of $P[i, *]$ is increasing exponentially in i, the total time is just $O(n \log n)$, but the rest of the algorithm is $O(n \log^2 n)$, so we won't analyze this precisely.)

We now present the main conversion procedure.

Convert(input: $X[0], \ldots, X[n - 1]$, an array of n digits in base b_1; output: an array of digits in base b_2.

1. If $n = 1$, output $X[0]$ in base b_2.
2. $A \leftarrow \mathbf{Convert}(X[0], \ldots, X[n/2 - 1])$
3. $B \leftarrow \mathbf{Convert}(X[n/2], \ldots, X[n - 1])$
4. $C \leftarrow P[\log n - 1, *]$
5. Return $\mathbf{Add}(\mathbf{Mult}(A, C), B)$ (where **Mult** is the $O(n \log n)$ FFT multiplication algorithm from class and **Add** is the $O(n)$ grade-school addition algorithm).

The time complexity $T(n)$ of the main recursive procedure is $T(n) = 2T(n/2) + O(n \log n) = O(n \log^2 n)$.