CSE 202 Homework #1

1. **Problem 1: Diameter of a tree**

   Input and output clarification

   The input is a tree T of size n ($n \geq 1$). The tree is rooted at node *root*. The edges are given in the form of adjacency lists, i.e. Children$[u]$ contains all the children of node u.

   The output is an integer indicating the diameter of tree T.

   High-level description

   We will take recursive steps to solve this problem. Denote T_u as the subtree rooted at u, p_u as the longest shortest-path from u to a node in T_u, q_u as the longest shortest-path between two nodes in T_u. p_u and q_u can be computed based on the property of u:

   - If u is a leaf node, i.e. Children$[u] = \emptyset$, we have $p_u = q_u = 0$.
   - If u has only one child, i.e. $|\text{Children}[u]| = 1$, denote this child as v, we have:

   $$\begin{cases} p_u = p_v + 1 \\ q_u = \max\{q_v, p_v + 1\} \end{cases} \tag{1}$$

   - If u has multiple children, i.e. $|\text{Children}[u]| \geq 2$, we have:

   $$\begin{cases} p_u = \max\limits_{v \in \text{Children[u]}} \{p_v\} + 1 \\ q_u = \max\left\{ \max\limits_{v,w \in \text{Children[u]}, v \neq w} \{p_v + p_w\} + 2, \max\limits_{v \in \text{Children}[u]} \{q_v\} \right\} \end{cases} \tag{2}$$

   We start the recursion from the *root*. When all the recursive steps finish, we return q_{root} as the answer.

   Proof of correctness

   We will first get the idea of what a shortest-path looks like in tree T.

   Claim 1. *For any two nodes u, v ($u \neq v$) in tree T. If u is an ancestor of v, the shortest-path between u and v is the path starting from v, keeping moving to its parent node until reaching u.*

Proof. Denote Depth(u) as the node of u in tree T. Any edge in tree T (say the edge connecting w_0 and w_1) ensures the property $|\text{Depth}[w_0] - \text{Depth}[w_1]| = 1$. In order to reach v from u, we need at least $\text{Depth}[v] - \text{Depth}[u]$ edges.

The depth will decrease by 1 if we move from a node to its parent node. Thus if starting from v and keep moving to its parent node until reaching u, we will get the shortest-path between u and v with length $(\text{Depth}[v] - \text{Depth}[u])$. □

Claim 2. *For any two nodes u, v $(i \neq v)$ in tree T. If u is not an ancestor of v and v is not an ancestor of u, the shortest-path between u and v is the concatenation of:*

- *starting from u, keeping moving to its parent node until reaching $LCA(u, v)$*
- *starting from v, keeping moving to its parent node until reaching $LCA(u, v)$*

where $LCA(u, v)$ is the lowest common ancestor (LCA) of u and v.

Proof. For any path between u and v, mark the node appeared in this path with smallest depth as w. u and v **must** be in the subtree of w (i.e. T_w), otherwise we cannot reach w to u or v by not moving to the ancestor nodes of w. By Claim 1, the shortest-path from w to u or v is of length $(\text{Depth}[u] - \text{Depth}[w])$ or $(\text{Depth}[u] - \text{Depth}[w])$, thus the path from u to v through w is at least of length:

$$(\text{Depth}[u] - \text{Depth}[w]) + (\text{Depth}[u] - \text{Depth}[w]) = (\text{Depth}[u] + \text{Depth}[v]) - 2 \times \text{Depth}[w]$$

In order to minimize the length, we must maximize the Depth[w] entry. Thus by assigning w as the lowest common ancestor of u and v, we can obtain the minimum length. □

Now we can prove the correctness of our algorithm by a bottom-up induction.

- Base case. When u is a leaf node (i.e. $\text{Children}[u] = \emptyset$), p_u and q_u is both 0 because T_u contains only u itself.
- Induction step for a node that has only one child. Denote the only child as v.
  - Any shortest-path from u to a node w in T_u (ignore the trivial path from u to u itself) consists an edge from u to v and v to w by Claim 1, thus $p_u = p_v + 1$ holds.
  - Any shortest-path between two nodes in T_u is in one of the two cases: (1) one of the node is u (2) both two nodes are not u. For the first case, it is the same as p_u. For the second case, both two nodes are in T_v where the maximum length is stored in q_v. Thus $q_u = \max\{q_v, p_v + 1\}$ holds.
- Induction step for a node that has multiple children.
  - Any shortest-path from u to a node w in T_u consists an edge from u to a child node v and v to w by Claim 1, thus $p_u = \max_{v \in \text{Children[u]}}\{p_v\} + 1$ holds.

– Any shortest-path between two nodes in T_u is in one of the three cases: (1) one of the node is u (2) both two nodes are in the same T_v where v is a child of u (3) two nodes are in different child subtrees (denoted as T_{v_0} and T_{v_1}). For the first case, it is the same as p_u, but as u has multiple children, we can always continue this path by moving deeper to a child node that is currently not in the path, thus case (1) can be ignored. For the second case, the maximum length is stored in q_v. For the third case, u is the LCA of the two nodes. By Claim 2, the maximum length is $p_{v_0} + 1$ (u to v_0 then to the first node) plus $p_{v_1} + 1$ (u to v_1 then to the second node). By combining case (2) and (3) together, we can obtain q_u shown in Equation 2.

Time complexity

The time complexity is $O(n)$. For every node u, we will call every child of u recursively, and maintain at most two maximum p_v entries and at most one maximum q_v entry to calculate p_u and q_u by Equation 1 and 2. Thus the time complexity is asymptotic to $\sum_{u \in T} |\text{Children}[u]| =$ # of edges in T, which is $O(n)$.

2. **Problem 2: Sorted matrix search**

Input and output clarification

The input is a matrix A of size $m \times n$ containing integers and a integer k. We assume that $m, n \geq 1$, otherwise the matrix is empty (or invalid).
Denote the elements as $M[1,1], \cdots, M[1,n], \cdots M[m,n]$.

The output is a pair of integers. If k is found the matrix, the pair indicates its index, otherwise the pair will be $(0,0)$.

High-level description

We present a searching algorithm by starting at the index $(1,n)$ in the matrix. When we are currently at index (x,y):

- If (x,y) is out of bound, i.e. $1 \leq x \leq m$ or $1 \leq y \leq n$ does not hold, we stop searching and return $(0,0)$.
- If $M[x,y] = k$, we stop searching and return (x,y).
- If $M[x,y] > k$, we decrease y by 1 and continue.
- If $M[x,y] < k$, we increase x by 1 and continue.

Proof of correctness

First we prove that our algorithm will always finish. In each step of the searching process, if we do not return a pair, we will either decrease y by 1 or increase x by 1. As the initial values are $(x,y) = (1,n)$, x and y will be out of bound by m and n steps, respectively. Thus the searching process will return after at most $m+n$ steps.

Then we prove that our algorithm will return the correct answer.

If k does not appear in the matrix, $M[x,y] = k$ cannot be satisfied in any searching step. Thus our algorithm will return $(0,0)$.

If k appears in the matrix, for each searching step, we are at index (x,y), indicating that k must appear at an index (i,j) in the range $x \leq i \leq m$ and $1 \leq j \leq y$:

- If $M[x,y] = k$, we return (x,y) as a correct answer.
- If $M[x,y] > k$, as the y-th column is sorted in ascending order, all the elements that are in the current range and in the y-th column, $M[x,y], M[x+1,y], \cdots, M[m,y]$, are strictly larger than k, thus we can ignore this column by $y \leftarrow y-1$.
- If $M[x,y] < k$, as the x-th row is sorted in ascending order, all the elements that are in the current range and in the x-th row, $M[x,1], M[x,2], \cdots, M[x,y]$, are strictly smaller than k, thus we can ignore this row by $x \leftarrow x+1$.

As we start at $(1, n)$, we will never ignore elements that are equal to k. Thus we will always return a valid index when we find $M[x, y] = k$.

Time complexity

The time complexity is $O(m + n)$. In the worst case, our searching process will take $m + n$ steps, going out of the boundary of the matrix and return false.

3. **Problem 3: 132 pattern**

Input and output clarification

The input is an array a of size n distinct positive integers. We assume that $n \geq 3$, otherwise there will never be 132-patterns.

The output is a boolean value, indicating whether there is a 132 pattern in a.

High-level description

We iterate through array a in inverse order and maintain a variable $best_2$, indicating the largest element that can be regarded as the '2' in 132-pattern. The initial value of $best_2$ is $-\infty$. We also maintain a LIFO stack s.

When we are at index i during the iteration:

- If $a_i < best_2$, we return true as we have found a 132-pattern.
- Otherwise we run into the following loop until s is empty:
  - Denote the top element of the stack as val.
    If $a_i > val$, we update $best_2 \leftarrow \max(best_2, val)$ and pop val out of the stack.
    Otherwise (i.e. $a_i < val$), we exit this loop.
- After the loop, we push a_i into the stack.

If the whole iteration finishes, we will return false.

Proof of correctness

We will first get the idea of how values are popped out from the stack and how $best_2$ is updated.

Claim 3. *For any element a_k in the array a, a_k will be popped out by its previous greater element a_j. Here "previous greater element" indicates that $j < k$ and $a_j > a_k$, in addition that $\nexists j'$ that both $j < j' < k$ and $a_{j'} > a_k$ satisfy.*

Proof. When we are iterating at index k, we will push a_k into the stack.

For the next iteration steps at index $k-1, k-2, \cdots, j+1$, as all the corresponding elements are strictly less than a_k, they will never pop a_k out of the stack.

When we are iterating at index j, the stack contains several elements in $a_{k-1}, a_{k-2}, \cdots, a_{j+1}$ (maybe zero) and then a_k from the top. As $a_j > a_k$ and a_j is also larger than any element in $a_{k-1}, a_{k-2}, \cdots, a_{j+1}$, a_j will pop out **at least** all the elements till a_k. Thus a_k is popped out by its previous greater element a_j. □

Corollary 3.1. *At the beginning of the iteration step at index i, $best_2$ stores the maximum value of a_k $(k > i)$ where a_k has its previous greater element indexed after i.*

We then create a connection between "previous greater element" and 132-pattern.

Claim 4. *If the array a contains a 132-pattern, there will always be a specific 132-pattern (i, j, k) where a_j is the previous greater element of a_k.*

Proof. Let (i, j, k) be any 132-pattern in array a. We have $i < j < k$ and $a_i < a_k < a_j$. Denote $a_{j'}$ as the previous greater element of a_k. Because $a_j > a_k$, the previous greater element of a_k must have the index at or after j, thus $j' \geq j$ always holds. As $i < j \leq j' < k$ and $a_i < a_k < a_{j'}$, (i, j', k) is also a 132-pattern.

By this step, we can transform any given 132-pattern into one which satisfies Claim 4. □

During the iteration step indexed at i, we check whether a_i can be regarded as the '1' in 132-pattern. By Claim 4, it is true if and only if we can find a_j and a_k where $i < j < k$ and $a_i < a_k < a_j$ and a_j is the previous greater element of a_k. By Corollary 3.1, we store the maximum value of an element that has its previous greater element indexed after i, thus $best_2$ is just the maximum possible value of a_k. If $a_i < best_2$, we then find a 132-pattern.

Time complexity

Throughout the whole iteration, every element is pushed into the stack once and is popped out of the stack at most once if its previous greater element is found. As the push and pop operations of a stack has a constant time cost, the total time complexity will be $O(n)$.

4. **Problem 4: Base conversion**

Input and output clarification

The input is an array X of size n, where n is a power of 2. It represents a base b_1 number. Denote the elements as $X_1, X_2, \cdots, X_n$, each digit X_i is in decimal form, i.e., $\forall i, 0 \leq X_i < b_1$.

The output is an array representing X in base b_2.

High-level description

Let $n = 2^T$. Firstly, we define a function TrDigit(x), which transforms a digit to base b_2.

- If $x = 0$, we return a list containing a single 0.
- Otherwise, we go into a loop until x becomes 0. For each step of the loop, we append ($x \bmod b_2$) to the end of a list (initialized as empty) and update x by $x \leftarrow \lfloor \frac{x}{b_2} \rfloor$. When the loop finishes, we return the obtained list.

By this function, we can then pre-compute $b_1, b_1^2, b_1^4, \cdots, b_1^{2^{T-1}}$ in base b_2 by FFT. Denote $B(x, b_2)$ as x in base b_2, we have:

- $B(b_1, b_2) = \text{TrDigit}(b_1)$.
- If we have computed $B(b_1^{2^{t-1}}, b_2)$, we can obtain $B(b_1^{2^t}, b_2)$ by using FFT on $B(b_1^{2^{t-1}}, b_2)$ and $B(b_1^{2^{t-1}}, b_2)$ itself. Denote the result of FFT as C, we must iterate through C to ensure that every element in the result list is less than b_2: if $C[i] \geq b_2$ holds, we will add $\lfloor \frac{C[i]}{b_2} \rfloor$ to $C[i+1]$ and update $C[i]$ by $C[i] \leftarrow C[i] \bmod b_2$. After the iteration, we can assign C to $B(b_1^{2^t}, b_2)$.

We then start the base conversion. Denote $F(i, j)$ as the result of $X_i, X_{i+1}, \cdots, X_j$ representing in base b_2. We will use divide and conquer to solve this problem.

- If $i = j$, it contains a single digit X_i. We can call TrDigit(X_i) to obtain X_i in base b_2.
- If $i \neq j$, we divide $X_i, X_{i+1}, \cdots, X_j$ into two parts of equal length. Let $k = \frac{i+j-1}{2}$, we call $F(i, k)$ and $F(k+1, j)$ recursively to obtain the base b_2 representation of these two parts. When the two calls return, we combine the results by:

$$F(i, j) = F(i, k) + B(b_1^{k-i+1}, b_2) \times F(k+1, j)$$

  where the $\times$ operator is calculated by FFT and the $+$ operator is calculated by element-wise addition of the two operand lists. We should also ensure that every element in $F(i, j)$ is less than b_2 by iterating through the result the same as when pre-computing $B(b_1^{2^t}, b_2)$.

We call $F(1, n)$ and obtain the output result. As $n = 2^T$, we will always split the elements into two parts of equal length, and $k - i + 1 = \frac{j-i+1}{2}$ is always a power of 2, which means $B(b_1^{k-i+1}, b_2)$ is pre-computed.

Proof of correctness

The correctness of function TrDigit(x) is trivial: it is just the procedure of transforming a decimal number x to base b_2. Thus we start by proving the correctness of $B(b_1^{2^t}, b_2)$ entries.

Claim 5. *$\forall t \in [0, T)$, $B(b_1^{2^t}, b_2)$ is correctly pre-computed.*

Proof. For $t = 0$, we call function TrDigit(b_1) directly, thus the correctness is ensured.

For $t > 1$, if $B(b_1^{2^{t_1-1}}, b_2)$ is correctly pre-computed, denote it as $B_0, B_1, \cdots, B_m$, representing the polynomial:

$$\sum_{i=0}^{m} B_i x^i$$

If we use FFT on $B(b_1^{2^{t_1-1}}, b_2)$ and $B(b_1^{2^{t_1-1}}, b_2)$ itself, we have:

$$(\sum_{i=0}^{m} B_i x^i)^2 = \sum_{i=0}^{2m} C_i x^i$$

where $C_0, C_1, \cdots, C_{2m}$ is the result of FFT. In order to obtain the correct $B(b_1^{2^{t_1}}, b_2)$ in which each element is less than b_2, we must adjust $C_0, C_1, \cdots, C_{2m}$. For any C_i that $C_i \geq b_2$ holds, C_i and C_{i+1} represent:

$$C_i x^i + C_{i+1} x^{i+1} \tag{3}$$

in the polynomial. If we do the adjustment $C_{i+1} \leftarrow C_{i+1} + \lfloor \frac{C_i}{b_2} \rfloor$ and $C_i \leftarrow C_i \bmod b_2$, the corresponding polynomial entries become:

$$(C_i - \lfloor \frac{C_i}{b_2} \rfloor \times b_2) x^i + (C_{i+1} + \lfloor \frac{C_i}{b_2} \rfloor) x^{i+1} \tag{4}$$

Although the polynomial becomes different, as we are doing base b_2 integer multiplication, i.e., $x = b_2$, Equation 3 and 4 are exactly of the same value:

$$\begin{aligned}(C_i - \lfloor \frac{C_i}{b_2} \rfloor \times b_2) x^i + (C_{i+1} + \lfloor \frac{C_i}{b_2} \rfloor) x^{i+1} &= (C_i - \lfloor \frac{C_i}{b_2} \rfloor \times b_2) b_2^i + (C_{i+1} + \lfloor \frac{C_i}{b_2} \rfloor) b_2^{i+1} \\ &= C_i b_2^i + C_{i+1} b_2^{i+1} \\ &= C_i x^i + C_{i+1} x^{x+1}\end{aligned}$$

Thus if we adjust every C_i to ensure that $C_i < b_2$, we will obtain the correct pre-computed result of $B(b_1^{2^t}, b_2)$. Note that C_{2m} may also exceed b_2, in this case, $B(b_1^{2^t}, b_2)$ will contain one more element. □

We can now prove the correctness of our recursive function F.

Claim 6. *$F(i, j)$ will return the result of $X_i, X_{i+1}, \cdots, X_j$ representing in base b_2 correctly.*

Proof. For $i = j$, we call function TrDigit(X_i) directly, thus the correctness is ensured.

For $i < j$, let $k = \dfrac{i+j-1}{2}$, the value of $X_i, X_{i+1}, \cdots, X_j$ is:

$$\sum_{u=i}^{j} X_u b_1^{u-i} = \sum_{u=i}^{k} X_u b_1^{u-i} + b_1^{k-i+1} \sum_{u=k+1}^{j} X_u b_1^{u-(k+1)} \tag{5}$$

In order to obtain the base b_2 representation of the above value, we can make use of the recursive steps as well as the pre-computed values:

- The base b_2 representation of $\sum_{u=i}^{k} X_u b_1^{u-i}$ is the return value of the recursive function $F(i, k)$.
- The base b_2 representation of $\sum_{u=k+1}^{j} X_u b_1^{u-(k+1)}$ is the return value of the recursive function $F(k+1, j)$.
- The base b_2 representation of b_1^{k-i+1} is pre-computed by $B(b_1^{k-i+1}, b_2)$, since $k - i + 1$ is always a power of 2.

By using FFT on $B(b_1^{k-i+1}, b_2)$ and $F(k+1, j)$, then applying element-wise addition between the FFT result and $F(i, k)$, we can obtain a polynomial. After a transformation (the same as Claim 5) to ensure every coefficient in the polynomial is less than b_2, we obtain the correct base b_2 representation of $X_i, X_{i+1}, \cdots, X_j$. □

Time complexity

The time complexity of this algorithm is made up by two parts. As b_1 and b_2 are considered constant, we expect the result to be of length $O(n)$.

- Pre-computing has $O(n \log n)$ time complexity. We use FFT $(\log n) - 2$ times to pre-compute $B(b_1^2, b_2), \cdots, B(b_1^{2^{T-1}}, b_2)$, in which the time complexity is bounded by:

$$\sum_{i=1}^{T-1} O(2^i \log(2^i)) = \sum_{i=1}^{T} O(2^i \times i) = O(2^T \times T) = O(n \log n)$$

- The recursive steps has $O(n \log^2 n)$ time complexity. The time complexity can be described recursively as:

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + O(n \log n)$$

  Where $O(n \log n)$ stands for the time complexity of FFT together with the element-wise addition. By master's theorem, we have $T(n) = O(n \log^2 n)$.

Thus the total time complexity is $O(n \log^2 n)$.