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 vertices 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 a linear (in the number of vertices of the tree) time algorithm to compute the diameter of a tree.

Algorithm:

Take an arbitrary node p as the root node, do BFS to find the furthest node from p, denoted as q. Then do a second BFS to find the furthest node from q, denoted as w, and $\delta(q, w)$ is the diameter.

Proof of correctness:

We claim that the node found after first BFS must be an endpoint of the longest shortest-path(diameter path). It follows by the property of BFS that the longest path in second BFS is the diameter path.

Claim 1. *Node q is an endpoint of the diameter path.*

Proof. We give the proof by contradiction. Assume q is not the endpoint of diameter path. We denote the endpoints as a and b. There are two cases:

1. The path from p to q intersects with the diameter path on node x. Since $\delta(p, q) \geq \delta(p, b)$, we have $\delta(p, x) + \delta(x, q) \geq \delta(p, x) + \delta(x, b)$, so $\delta(x, q) \geq \delta(x, b)$. Adding $\delta(a, x)$ on each side, we have $\delta(a, x) + \delta(x, q) \geq \delta(a, x) + \delta(x, b)$, so $\delta(a, q) \geq \delta(a, b)$, which contradicts with q not being the endpoint of diameter.

2. The path from p to q does not intersect with the diameter path. In this case a path from node c to d(c on path p to q, d on path a to b) must connect those two paths since it's a tree. So we have $\delta(p, c) + \delta(c, q) \geq \delta(p, c) + \delta(c, d) + \delta(d, b)$. Hence $\delta(a, d) + \delta(d, c) + \delta(c, q) \geq \delta(a, d) + 2\delta(c, d) + \delta(d, b)$, which implies $\delta(a, q) \geq \delta(a, b) + 2\delta(c, d)$ and $\delta(a, q) > \delta(a, b)$. A contradiction.

In both cases we get contradictions. Hence the initial assumption is false and the claim is true. □

Time complexity:

We do two BFS on the entire tree, so the time complexity is $O(|V|)$.

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.

Algorithm:

Denote our matrix as M, and the element to find is s.

Starting at the top-right position, i.e., $M[1, n]$, if $s = M[1, n]$, we find the element and return. Otherwise, if $s > M[1, n]$, move down one cell to $M[2, n]$; if $s < M[1, n]$, move left to $M[1, n-1]$. And repeat the above process.

If we reach the first column or the last row and still have not found s, we terminate the program and return *not found*.

Proof of correctness:

Since at every step we either move down or left and the matrix size is finite, we are guaranteed to reach the first column or the last row if s is not found. Now we claim that if our algorithm returns *not found*, then there is no s in M.

Claim 1. *If the algorithm returns not found, then s is not in M.*

Proof. We give the proof by contradiction. Assume cell $M[i, j] = s(1 \leq i \leq m, 1 \leq j \leq n)$. Since the algorithm traverses from top-right cell to either the first column or the last row, it is guaranteed that it will visit at least one cell on row i or column j. There are two cases:

1. The algorithm first visits a cell on row i, $M[i, k](j < k \leq n)$. According to the ascending-order property, $s = M[i, j] < M[i, k]$, so the algorithm should move left to $M[i, k-1]$. And it will continue to do so until it reaches $M[i, j]$. Thus it finds s and won't return *not found*, a contradiction.

2. The algorithm first visits a cell on row j, $M[v, j](1 \leq v < i)$. According to the ascending-order property, $s = M[i, j] > M[v, j]$, so the algorithm should move down to $M[v+1, j]$. And it will continue to do so until it reaches $M[i, j]$. Thus it finds s and won't return *not found*, a contradiction.

In both cases we get contradictions. Hence the initial assumption is false and the claim is true. □

Time complexity:

In the worst case we have to walk through the longest path (from top-right to bottom-left), so the time complexity is $O(m + n)$.

Problem 3: 132 pattern

Given a sequence of n distinct positive integers $a_1, ..., a_n$, a 132-pattern is a subsequence a_i, a_j, a_k such that $i < j < k$ and $a_i < a_k < a_j$. Design an algorithm that takes as input a list of n numbers and checks whether there is a 132-pattern in the list.

Algorithm:

If $n < 3$, it's trivial to see that there isn't a 132-pattern in the list. If $n \geq 3$, we iterate i from n to 1, and maintain a monotonically-decreasing stack S. We use variable s_3 to keep track of the maximum integer to the right of a_i that could constitute the third element in a 132-pattern. Initially s_3 is set to negative infinity. During every iteration step, first we check if the new a_i is less than s_3, if so, then the new a_i constitutes the first element in a 132-pattern and we return true. Otherwise, we treat the new a_i as the second element. We keep popping S until a_i is less than the top of S or S is empty. We then push a_i into S and take the last popped element as s_3.

If we iterates through the entire list and cannot find a 132-pattern, we return false.

Proof of correctness:

Claim 1. *s_3 is the maximum integer to the right of a_i that could constitute the third element in a 132-pattern.*

Proof. According to the algorithm, s_3 exists (greater than negative infinity) iff there's an $a_j > a_k (i < j < k)$, because then a_k will be popped out and become s_3. Now we prove that s_3 is the maximum a_k among all those (a_j, a_k) pairs. First we state that s_3 will not decrease with new a_i being pushed into S. This is because every new a_i pushed into S is guaranteed to be greater than s_3 (otherwise the algorithm terminates). Thus every element in S after s_3 was popped is greater than s_3, and the new s_3 can only be generated among them. Hence s_3 will not decrease. Now if we denote a_{k^*} in pair (a_{j^*}, a_{k^*}) as the maximum a_k among all (a_j, a_k) pairs, it's obvious that a_{k^*} will be popped out at sometime and become s_3. Since s_3 does not decrease, it retains this result, that is, s_3 is the maximum a_k. □

Claim 2. *The described algorithm returns true iff there is a 132-pattern in the list.*

Proof. If we find an $a_i < s_3$, we know that there exists a pair (i, j, k) where $i < j < k$ and $a_i < a_k < a_j$. Hence it's sufficient.

We can prove that returning true is necessary by proving the contrapositive: if the algorithm returns false, there is no 132-pattern in the list. Because s_3 is the maximum integer to the right of a_i that could constitute the third element, $a_i > s_3$ means a_i cannot be the first element for a 132-pattern. The algorithm returning false means none of a_i can be the first element, hence there isn't a 132-pattern in the list. □

Time complexity:

Time complexity is $O(n)$ because we push and pop each element in the list at most once.

Problem 4: 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.

Algorithm:

We have the input array $[s_0, s_1, ..., s_{n-1}]$ (assume n is a power of 2; otherwise, pad with 0's in the most significant digits), and should return an output array in the form of $[r_0, r_1, ..., r_{m-1}]$ ($m \propto n$), where the following holds:

$$B(n) = \sum_{i=0}^{n-1} s_i * b_1^i = \sum_{i=0}^{m-1} r_i * b_2^i.$$

We can rewrite $B(n)$ in the following form:

$$B(n) = \sum_{i=0}^{\frac{n}{2}-1} s_i * b_1^i + b_1^{\frac{n}{2}} * \sum_{i=0}^{\frac{n}{2}-1} s_{\frac{n}{2}+i} * b_1^i = B(\frac{n}{2}) + b_1^{\frac{n}{2}} * \hat{B}(\frac{n}{2}),$$

since converting $B(\frac{n}{2})$ or $\hat{B}(\frac{n}{2})$ is essentially the same problem as converting $B(n)$ except the problem size is reduced to half, we can apply a recursive algorithm. Base cases of the recursion have a size of 1. Suppose that we have converted $B(\frac{n}{2})$ and $\hat{B}(\frac{n}{2})$, now to convert $B(n)$ we have to compute:

$$\begin{aligned}
\sum_{i=0}^{m-1} r_i * b_2^i &= \sum_{i=0}^{m-1} (r_i - \hat{r_i}) * b_2^i + \sum_{i=0}^{m-1} \hat{r}_i * b_2^i \\
&= \sum_{i=0}^{m-1} (r_i - \hat{r_i}) * b_2^i + b_1^{\frac{n}{2}} * \sum_{i=0}^{m-1} p_i * b_2^i \\
&= \sum_{i=0}^{m-1} (r_i - \hat{r_i}) * b_2^i + \big(\sum_{i=0}^{m-1} q_i * b_2^i\big) * \big(\sum_{i=0}^{m-1} p_i * b_2^i\big).
\end{aligned}$$

To get $\hat{r}_i$, we utilize FFT to multiply two polynomials. Note that $b_1^{\frac{n}{2}} = (b_1^{\frac{n}{4}})^2$, so we can cache point-value representation of $b_1^{\frac{n}{4}}$ in the last recursion and don't need to compute q_i.

Proof of correctness:

Claim 1. *The described algorithm returns the correct array of base b_2 digits.*

Proof. Conversion of the base cases can be easily done by iteratively dividing s_i with b_2. By induction, if we get correct conversions of $B(\frac{n}{2})$ and $\hat{B}(\frac{n}{2})$, we can get correct conversion for $B(n)$ using the above equation. Hence the final result is correct. □

Time complexity:

Let $T(n)$ be the worst time complexity, in every layer of recursion we have to use FFT whose time complexity is $O(nlogn)$, and we need to add up all coefficients which yields $O(n)$, so the total extra time complexity for each layer of recursion is $O(nlogn)$. We have:

$$T(n) = 2T(\frac{n}{2}) + O(nlogn).$$

According to master theorem, we get $T(n) = O(nlog^2n)$.

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.

Algorithm:

We recursively check every sub-tree of the root in a DFS fashion to see whether it has a perfect matching or not. At the same time we record n_i as the number of node i's sub-trees that do not have perfect matchings. We return true if $n_{root} = 1$ and $n_j = 0$ (supposing node j is the root of that exact sub-tree) otherwise false.

Proof of correctness:

We begin by identifying conditions where a perfect matching could appear.

Claim 1. *A graph has a perfect matching **only if** it has N nodes and N is even.*

Proof. Since in a perfect matching each node is touched exactly once and an edge touches two different nodes, if the number of edges in that set is e, then the number of nodes those edges touch is $2e$, which is an even number. □

Claim 2. *A tree has a perfect matching **only if** its root has exactly one sub-tree that does not have a perfect matching.*

Proof. We prove by stating that in all other cases, the tree cannot have a perfect matching.

1. All root's sub-trees have perfect matchings or the root does not have sub-trees. Referring to Claim 1, the total number of nodes in sub-trees must be an even number, so the entire tree has an odd number of nodes. Thus it cannot have a perfect matching.
2. The root has more than two sub-trees that do not have perfect matchings. Since we can only pick one edge which connects the root and a sub-tree in the final matching, at least one sub-tree would be left out that does not have a perfect matching of its own. So there won't be a perfect matching for the entire tree.

□

Our algorithm is correct because according to Claim 2, that's the only scenario where we can pick the edge that connects root and node j, and all other isolating trees have their perfect matchings hence the entire tree has a perfect matching.

Time complexity:

Time complexity is $O(n)$ where n is the number of nodes because we visit each node once in DFS.