Readings

- [Lecture Notes Chapter 7: Divide & Conquer and Recurrence Relations]
- [Lecture Notes Chapter 13: Stacks & Queues]

Problems

Problem 1. Local Maxima

You are given an integer array \( A[1..n] \) with the following properties:

- Integers in adjacent positions are different

A position \( i \) is referred to as a local maximum if \( A[i-1] < A[i] \) and \( A[i] > A[i+1] \). You may assume \( n > 2 \).

Example: You have an array \([0, 1, 5, 3, 6, 3, 2]\). There are multiple local maxes at 5 and 6.

Propose an efficient algorithm that will find a local maximum and return its index.

Solution.

A naive solution to this problem is to simply iterate through the array and check each element with its neighbors until we find a possible solution, which runs in linear time. Instead, we aim for a more efficient way of solving this problem.

The first thing to notice is that given the properties of the array, there must be a local maximum. A simple proof of this is to consider a maximum element in the array. Since neighbors are distinct and since the endpoints cannot be maximum, we are guaranteed that this element will be a local maximum.

If we want to approach this problem with the Divide & Conquer methodology, we need to figure out how we can cut the problem into subproblem(s). We do this by cutting the problem in half every time.

We let our base case for this algorithm be when the array is of size 3, in which case we return the middle element index. Otherwise, consider the middle element of the array, say at index \( m \). If that element meets the requirements of a local maximum, then we return \( m \). If not, notice that this element must have a larger neighbor, WLOG say the right neighbor is larger.

Observe that the subarray \( A[m..n] \) also satisfies the given property of the input array—the endpoints are smaller than their neighbors and every pair of adjacent elements are distinct. Therefore, by our claim above, the subarray must contain a local maximum. Since we are strictly reducing this problem at every recursive call, we must eventually hit our base case (which we know to be correct) or terminate early by finding a local maximum.

The running time of our algorithm is thus \( T(n) = T\left(\frac{n}{2}\right) + O(1) \). Solving this recurrence yields \( O(\log n) \).
**Problem 2. Largest Subarray Sum**

Given an integer array (containing both positive and negative values), return the sum of the largest contiguous subarray which has the largest sum.

**Solution.**

One naive way to solve this problem is to use two for-loops. The outer loop runs through the elements in the array, while the inner loop finds the maximum contiguous subarray sum starting at the current outer loop element. If this sum is bigger than the best running maximum, we update and continue through the process. This runs in $O(n^2)$.

A better solution is to apply our knowledge of the Divide & Conquer approach, and see if we can find a more efficient solution to this problem!

One thing that we can intuitively notice is that the optimal sub-sequence either lies in the left half of the array, the right half of the array, or runs along the center of the array and cuts through the middle element. Logically, these are the only three options we have. Thus, we can compute all three of these values and the maximum of them will be our solution!

To do this, we must recursively divide the array into two halves and find the maximum subarray sum in both halves. This can be done easily with two recursive calls. Lastly, we need to efficiently compute the maximum cross sum. This can be done in $O(n)$ time by starting at the middle element and calculating the maximum sum to the left of the middle element, doing the same with the right and combining the two!

To calculate the run time of our algorithm, we notice that it breaks down the work into two sub problems, each with half the size as input and then checks the cross sum in linear time. Thus, we get the following recurrence: $T(n) = 2T(n/2) + O(n)$.

Does this look familiar? It should! It’s the same recurrence you saw for the running time analysis of Mergesort! This evaluates to $O(n \log n)$.

**Problem 3**

**Given:** A binary tree $T$.

**Objective:** Print the level order traversal of the tree $T$.

**Example:**

**Solution**

**Algorithm:** We use a queue to hold nodes that are to be visited. We first start with the queue containing the root node of the tree. While the queue is not empty, we **enqueue** an element from the queue, mark it as visited, and then **enqueue** its children into the queue.

- For the tree above, we first start with node 1 in the queue. We remove 1, mark it as visited, and add 2, 3 to the queue.
- We then remove 2 and add 7, 6 to the queue. We remove 3 and add 5, 4 to the queue.
- Since all nodes in the queue at this point are leaves, we remove each node one by one until the queue is empty.

**Time and space complexity:** If the tree $T$ contains $n$ nodes, this solution takes $O(n)$ time since we are enqueuing and dequeuing each of the $n$ nodes once and $O(n)$ space for the queue.
Problem 4
You are given two stacks $S_1$ and $S_2$, each of size $n$.

Implement a queue using $S_1$ and $S_2$. Your queue’s **enqueue** and **dequeue** methods should be implemented using only your stacks’ **push**, **pop**, and/or **peek** methods. What are the running times of your new queue’s **enqueue** and **dequeue** methods?

Solution
**enqueue**(x):
1. **push** x into $S_1$.

**dequeue**:
1. If $S_2$ is empty, **pop** all elements from $S_1$ and **push** them into $S_2$.
2. If $S_2$ is still empty, return Nil.
3. Else **pop** an element from $S_2$ and return it.

**Running Time:** The running time of **enqueue**(x) is clearly $O(1)$. The running time for **dequeue** is a bit trickier. If we consider that each element will be in each Stack exactly once, then we realize that each element will be pushed exactly twice and popped exactly twice. Thus, the amortized running time of **dequeue** is $O(1)$.
Additional Practice Problems

Problem 1. Element Index Matching

You are given a sorted array of \( n \) distinct integers \( A[1...n] \). Design an \( O(\lg n) \) time algorithm that either outputs an index \( i \) such that \( A[i] = i \) or correctly states that no such index \( i \) exists.

Solution.

A naive solution would be to iterate through the array, checking if \( A[i] = i \) at every stage. This runs in \( O(n) \), but we can probably do better. Since we have a sorted array and an \( O(\lg n) \) runtime constraint, a modified binary search seems to be a good choice.

Notice that at any index \( i \), if \( A[i] \neq i \), we can narrow down the possibilities of where a potential candidate lies. More specifically, if \( A[i] < i \), since integers are distinct and sorted, it’s impossible to find an index \( m < i \) such that \( A[m] = m \). We can see this as follows:

\[
A[i - k] \leq A[i] - k < i - k
\]

The same reasoning applies to the right side when \( A[i] > i \). Therefore, letting \( i \) be the middle index, we can be sure that the specified half will never contains an index-matching element, and instead recurse on the other.

For each stage of our algorithm, we are performing a constant time comparison, and then dividing our problem in half. Thus, our recurrence is \( T(n) = T(\frac{n}{2}) + O(1) \), which we know is \( O(\lg n) \) from binary search.

Problem 2

Given: A binary tree \( T \).
Objective: Print the spiral order traversal of the tree \( T \).
Example:

![Figure 2: For this tree, your function should print 1, 2, 3, 4, 5, 6, 7.](image)

Hint: Try using 2 stacks.
Solution
We will use two stacks, \( S_1 \) and \( S_2 \). We will use \( S_1 \) to hold elements in the same level that are being printed from left to right, and we will use \( S_2 \) to hold elements in the same level that are being printed from right to left. We observe that these stacks are disjoint (i.e., they contain no overlapping elements), and if a given node \( n \) in \( T \) is in \( S_1 \), then its two children should be in \( S_2 \) (and vice versa).

Algorithm: First, push the root of the tree \( T \) onto stack \( S_2 \). The following procedure will loop until both \( S_1 \) and \( S_2 \) are empty.
- While \( S_2 \) is not empty, pop the top element \( n \) from \( S_2 \). Print \( n \). If \( n \) has a right child, push it onto the other stack \( S_1 \). Then, if \( n \) has a left child, push it onto \( S_1 \). Continue this step until \( S_2 \) is empty.
- While \( S_1 \) is not empty, pop the top element \( n \) from \( S_1 \). Print \( n \). If \( n \) has a left child, push it onto the other stack \( S_2 \). Then, if \( n \) has a right child, push it onto \( S_2 \). Continue this step until \( S_1 \) is empty.

Time and space complexity: If the tree \( T \) contains \( n \) nodes, this solution takes \( O(n) \) time and \( O(n) \) extra space.

Problem 3
Given a full stack \( S_1 \) of size \( n \) and an empty stack \( S_2 \) of size \( n \), sort the \( n \) elements in ascending order in \( S_2 \). You may only use the given 2 stacks \( S_1 \) and \( S_2 \) (each of size \( n \)) and \( O(1) \) additional space. What is the running time of your sorting procedure?

Example:

```
  4 3 1 5 2  
→ 1 2 3 4 5
```

Hint: Start with a simpler example:

```
  3 2 1  
→ 1 2 3
```

Solution
To solve this problem, we will use the two given stacks, \( S_1 \) and \( S_2 \), and two extra variables \( \text{max} \) and \( \text{size} \).

Algorithm: Initialize \( \text{max} \) to \(-\infty \) and \( \text{size} \) to 0.

1. pop all elements from \( S_1 \) and push them onto \( S_2 \). While pop’ing, keep track of the maximum element we have seen so far in \( \text{max} \). Once we have push’ed all elements into \( S_2 \), the absolute maximum element will be stored in \( \text{max} \).
2. pop all elements from \( S_2 \) and push all except the maximum element \( \text{max} \) back into \( S_1 \).
3. push the maximum element (stored in \( \text{max} \)) into \( S_2 \). Now \( S_1 \) contains \( n - 1 \) unsorted elements, and \( S_2 \) contains 1 sorted element.
4. Increment \( \text{size} \) by 1. We will use \( \text{size} \) to keep track of the number of sorted elements in \( S_2 \) so that we don’t pop them.
5. Repeat steps 1-4 until \( \text{size} = n \). In Step 2, take care to only pop elements from \( S_2 \) until \( S_2 \) contains exactly \( \text{size} \) elements. (The bottom \( \text{size} \) elements in \( S_2 \) have already been sorted.)
When the procedure terminates, $S_1$ will be empty, and $S_2$ contains the elements in non-decreasing order.

**Running Time:** The running time of our sorting procedure is $O(n^2)$, since for each element that we sort, we must push and pop at most $n$ elements.