1 More Sorting!

We are given an unsorted array $A$ with $n$ numbers between 1 and $M$ where $M$ is a large but constant positive integer. We want to find if there exist two elements of the array that are within $T$ of one another.

1. Design a simple algorithm that solves this in $O(n^2)$.
2. Design a simple algorithm that solves this in $O(n \log n)$.
3. How could you solve this in $O(n)$? (Hint: modify bucket sort.)

1. Compare all pairs of numbers to see if any are within $T$ of each other.
2. Sort the array, then compare only adjacent elements to see if they are within $T$ of each other.
3. Because we know that the elements are from 1 to $M$, if they are integers, we can simply bucket sort the elements and check subsequent elements to see if they are $T$ apart. If we cannot create $M$ buckets due to memory constraints, or if the array elements are real numbers, we could split the items up into buckets of size $T$. If any bucket has 2 or more items, then those elements are within $T$ of each other. Otherwise, each bucket holds at most 1 element (indeed, the elements are sorted) and we only need to check pairs of elements in adjacent buckets, and there are at most $n - 1$ such pairs.

2 Uniqueness of BST Structure

You are given a binary tree structure with $n$ nodes and a set of $n$ distinct keys (numbers). Prove or disprove: There is exactly one way to assign keys to the given tree structure such that the resulting tree is a valid binary search tree.

Example: You are given the binary tree drawn below and the set of keys 1, 2, 3, 4, 5, 6. The question asks whether there is exactly one way to assign the keys to nodes such that the tree will be a binary search tree. (If you prove the statement, it should be for any input and not just this example.)
True. We prove by induction on \( n \).

**Inductive Hypothesis:** Given a tree with \( n \) nodes, there is only one way to assign a set of \( n \) distinct values to the nodes such the resulting tree is a valid BST.

**Base case:** If \( n = 0 \), then the tree contains no nodes, and there is only one way to assign no value to the nonexistent node. (A silly base case, but it works and is necessary).

**Inductive step:** Let \( k > 1 \) be an integer. Assume the inductive hypothesis holds for up to \( n \) nodes, where \( n < k \). Consider a tree with \( k \) nodes. Denote by \((a_1, \ldots, a_k)\) the set of keys in sorted order \((a_i < a_{i+1})\).

Let \( r \) denote the root. Denote by \( l \) the number of nodes in the left subtree of the root. Since the nodes in the left subtree must store a key that is less than the root and the nodes in the right subtree must have keys that are greater than the root, we conclude that the root must store the key \( a_l + 1 \).

In addition, the left subtree of the root must contain the keys \( \{a_1, \ldots, a_l\} \) and the right subtree must contain the keys \( \{a_{l+2}, \ldots, a_k\} \). Each of these subtrees is a binary tree with strictly less than \( k \) nodes, and by our induction hypothesis, we know that there is only one way to assign the values to the nodes in each subtree. We conclude that there is only one way to assign the \( k \) values to the nodes of the tree in a way that will result in a valid binary search tree.

**Conclusion:** We conclude that the inductive hypothesis holds for all \( n \).

### 3 Randomly Built BSTs

In this problem, we prove that the average depth of a node in a randomly built binary search tree with \( n \) nodes is \( O(\log n) \). A randomly built binary search tree with \( n \) nodes is one that arises from inserting the \( n \) keys in random order into an initially empty tree, where each of the \( n! \) permutations of the input keys is equally likely.

Let \( d(x, T) \) be the depth of node \( x \) in a binary tree \( T \) (the depth of the root is 0). Then, the average depth of a node in a binary tree \( T \) with \( n \) nodes is

\[
\frac{1}{n} \sum_{x \in T} d(x, T).
\]

a. Let the total path length \( P(T) \) of a binary tree \( T \) be defined as the sum of the depths of all nodes in \( T \), so the average depth of a node in \( T \) with \( n \) nodes is equal to \( \frac{1}{n} P(T) \). Show that \( P(T) = P(T_L) + P(T_R) + n - 1 \), where \( T_L \) and \( T_R \) are the left and right subtrees of \( T \), respectively.

Let \( r(T) \) denote the root of tree \( T \). Note the depth of node \( x \) in \( T \) is equal to the length of the path from \( r(T) \) to \( x \). Hence, \( P(T) = \sum_{x \in T} d(x, T) \).

For each node \( x \) in \( T_L \), the path from \( r(T) \) to \( x \) consists of the edge \((r(T), r(T_L))\) and the path from \( r(T_L) \) to \( x \). The same reasoning applies for nodes \( x \) in \( T_R \). Equivalently, we have

\[
d(x, T) = \begin{cases} 
0, & \text{if } x = r(T) \\
1 + d(x, T_L), & \text{if } x \in T_L \\
1 + d(x, T_R), & \text{if } x \in T_R 
\end{cases}
\]
Let \( P \) be a randomly built binary search tree with \( n \) nodes. Show that \( P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n-1). \)

Let \( T \) be a randomly built binary search tree with \( n \) nodes. Without loss of generality, we assume the \( n \) keys are \( \{1, \ldots, n\} \).

By definition, \( P(n) = \mathbb{E}_T[P(T)] \). Then, \( P(n) = \mathbb{E}_T[P(T_L) + P(T_R) + n-1] = n-1 + \mathbb{E}_T[P(T_L)] + \mathbb{E}_T[P(T_R)], \) where \( T_L \) and \( T_R \) are the left and right subtrees of \( T \), respectively. Note

\[
\mathbb{E}_T[P(T_L)] = \sum_{i=1}^{n} \mathbb{E}_T[P(T_L)]r(T) = i \cdot Pr(r(T) = i).
\]

Since each element is equally likely to be the root of \( T \), \( Pr(r(T) = i) = \frac{1}{n} \) for all \( i \). Conditioned on the event that element \( i \) is the root, \( T_L \) is a randomly built binary search tree on the first \( i-1 \) elements. To see this, assume we picked element \( i \) to be the root. From the point of view of the left subtree, elements \( 1, \ldots, i-1 \) are inserted into the subtree in a random order, since these elements are inserted into \( T \) in a random order and subsequently go into \( T_L \) in the same relative order. Hence, \( \mathbb{E}_T[P(T_L)r(T) = i] = P(i-1) \). Putting these together, we get

\[
\mathbb{E}_T[P(T_L)] = \sum_{i=1}^{n} \frac{1}{n} P(i-1).
\]

Similarly, we get \( \mathbb{E}_T[P(T_R)] = \sum_{i=1}^{n} \frac{1}{n} P(n-i) \). Then,

\[
P(n) = n-1 + \mathbb{E}_T[P(T_L)] + \mathbb{E}_T[P(T_R)]
= n-1 + \frac{1}{n} \sum_{i=1}^{n} [P(i-1) + P(n-i)]
= n-1 + \frac{1}{n} \sum_{i=0}^{n-1} [P(i) + P(n-i-1)],
\]

where we changed the indexing of the sumamation in the last equality.

c. Show that \( P(n) = O(n \log n) \). You may cite a result previously proven in the context of other topics covered in class.
This is the same recurrence that appears in the analysis of Quicksort (see lecture 4 notes section 4.1)

d. Design a sorting algorithm based on randomly building a binary search tree. Show that its (expected) running time is $O(n \log n)$. Assume that a random permutation of $n$ keys can be generated in time $O(n)$.

The algorithm is 1) construct a randomly built binary search tree $T$ by inserting given elements in a random order; and 2) do the inorder traversal on $T$ to get a sorted list. Note step 2 can be done in $O(n)$ time. We argue that step 1 takes $O(n \log n)$ time in expectation. We observe that given the final state of tree $T$, we can compute the amount of work spent to construct $T$. To insert a node $x$ at depth $d$, we traversed exactly the path from the root to the parent of $x$, at depth $d - 1$, to insert it. Hence, we can upper bound the total work done to construct $T$ by $O(P(T))$. From part (c), we know that $P(T) = O(n \log n)$ in expectation. It follows that Step 1 takes $O(n \log n)$ time in expectation. Overall, the algorithm runs in $O(n \log n)$ in expectation.