Practice Problem 1. Coat Check

Suppose that \( n \) women check their coats at a concert. However, at the end of the night, the attendant has lost the claim checks and doesn’t know which coat belongs to whom. All of the women came dressed in black coats that were nearly identical, but of different sizes. The attendant can have a woman try a coat, and find out whether the coat fits (meaning it belongs to that woman), or the coat is too big, or the coat is too small. However, the attendant cannot compare the sizes of two coats directly, or compare the sizes of two women directly. Describe how the attendant can determine which coat belongs each woman in expected \( O(n \log n) \) time. Give a brief analysis of the running time of your algorithm.

Solution: We solve this using a modified version of quicksort:

1. Choose a random woman (whom we will call the “pivot woman”) and have her try on all the coats. Separate the coats into three groups: the coats that are too small for her, the coats that are too big for her, and her own coat, which we will call the “pivot coat.”

2. Have all the remaining women try on the pivot coat. Separate them into two groups: those who are too small for the pivot coat and those who are too big for the pivot coat.

3. Recursively solve the two remaining subproblems, matching the women and coats who are smaller than the matched pair and those who are larger than the matched pair.

This requires exactly twice as many operations as a standard randomized quicksort, so this algorithm still runs in expected \( O(n \log n) \) time.
Practice Problem 2. The Power of Quicksort

To get concrete evidence of the power of randomized quicksort, you want to create discrete probability distributions of how many comparisons are needed to perform a randomized quicksort on \( n \) elements. Specifically, if we let \( X_n \) be a random variable denoting how many comparisons are needed, you want to determine \( P(X_n = x) \) for all integers \( x \geq 0 \).

For example, the distribution for \( X_3 \) is

\[
X_3 = \begin{cases} 
2 & \text{with probability } \frac{1}{3} \\
3 & \text{with probability } \frac{2}{3}
\end{cases}
\]

because there is a \( \frac{1}{3} \) probability we will only need 2 comparisons (if the chosen pivot is the middle element) and a \( \frac{2}{3} \) probability that we will need 3 comparisons (if the chosen pivot is the smallest or largest element).

The goal for this problem is to come up with an efficient algorithm that computes the distribution for \( X_n \).

(a) Given two random variables \( X \) and \( Y \) whose domains are the non-negative integers such that \( 0 \leq X, Y \leq m \) for a known parameter \( m \) and their probability distributions, give an efficient algorithm that computes the probability distribution for \( X + Y \). State the running time of this algorithm in terms of \( m \).

Solution: Note that

\[
P(X + Y = a) = \sum_{i=-\infty}^{\infty} P(X = i) \cdot P(Y = a - i).
\]

Therefore we can solve this by convolving two lists of length \( m + 1 \) where the \( i \)th element in each list is the probability that the associated random variable equals \( i \). This can be done in \( O(m \log m) \) time using an FFT.

(b) Give a polynomial-time dynamic programming algorithm that computes the distribution for \( X_n \).

Hint: Start by expressing \( X_n \) in terms of \( X_0, X_1, \ldots, X_{n-1} \).

Solution: Going from the hint, the recurrence is:

\[
X_n = (n - 1) + \begin{cases} 
X_0 + X_{n-1} & \text{with probability } 1/n \\
X_1 + X_{n-2} & \text{with probability } 1/n \\
X_2 + X_{n-3} & \text{with probability } 1/n \\
\vdots & \text{with probability } 1/n \\
X_{n-1} + X_0 & \text{with probability } 1/n
\end{cases}
\]

We can compute the distributions for each of the \( n \) cases using dynamic programming using the base case that \( X_0 = 0 \) with probability 1. Since randomized quicksort runs in \( \Theta(n^2) \) in the worst case, computing each distribution takes \( O(n^2 \log n^2) = O(n^2 \log n) \) time. Each subproblem therefore requires \( O(n^3 \log n) \) time.

Since there are \( \Theta(n) \) subproblems in total, the final running time is \( O(n^4 \log n) \).
Practice Problem 3. The Sensational 6046

The Sensational 6046 is a group of superheroes in charge of fighting off an alien invasion of Earth. Their most powerful ability involves linking their bodies together to form various weapons (including swords, loganberries, and Turing machines). In the heat of battle, they must be able to form these weapons as quickly as possible. Thanks to your expertise in algorithms, you feel that you can help them.

There are $n$ heroes numbered 1 through $n$. Each hero starts at some location in 3D space and, to form a particular weapon, they need to move to new points, which are also numbered from 1 through $n$. However, in order to maintain the structural integrity of the weapon, a hero must arrive at point $j$ at or before time $t \cdot j$ for some fixed constant $t$. You are given an $n \times n$ matrix $T$ where $T_{ij}$ denotes the amount of time it takes for hero $i$ to move to point $j$.

The problem can be put in mathematical terms as follows:

Let $\sigma_1, \ldots, \sigma_n$ be a permutation of 1, $\ldots$, $n$ and $t$ a positive constant. This permutation is said to be $t$-valid if $T_{i\sigma_i} \leq t \cdot \sigma_i$ for $i = 1, \ldots, n$.

(a) Give an efficient algorithm that, given $T$ and $t$, finds a $t$-valid permutation if one exists, or reports that one doesn’t exist.

**Solution:** This can be formulated as a matching problem where we attempt to match heroes to destination points, which can in turn be formulated as a max-flow problem. Let $G = (V, E)$ be our flow network where $V = \{s, t, h_1, \ldots, h_n, d_1, \ldots, d_n\}$, where $h_i$ corresponds to hero $i$ and $d_j$ corresponds to destination point $j$. Create edges from $s$ to each $h_i$ and from each $d_j$ to $t$, all with capacity 1. For each $i, j$, create an edge with capacity 1 from $h_i$ to $d_j$ if and only if $T_{ij} \leq t \cdot j$. Run Ford-Fulkerson on this flow network. If the flow has size $n$, the $h_i \rightarrow d_j$ edges that are at capacity correspond to a $t$-valid permutation $\sigma$ with $\sigma_i = j$. Otherwise no $t$-valid permutation exists.

This algorithm runs in $O(n^3)$ time since the maximum flow can have size at most $n$.

The commander is mainly interested in knowing how quickly a weapon can be formed, i.e., the smallest value $t_0$ such that there exists a $t_0$-valid permutation.

(b) We will call $t_c$ a critical value if there exist values $i, j$ such that $T_{ij} = t_c \cdot j$. Show that $t_0$ must be a critical value.

**Solution:** We prove this by contradiction. First assume that $t_0$ is smaller than any critical value. Then by definition we have $t_0 \cdot j < T_{ij}$ for all $i, j$, which implies that there can never be a $t_0$-valid permutation. Contradiction.

Now assume that $t_0$ is not a critical value. Let $t_c^*$ be the largest critical value that is smaller than $t_0$. Observe that if $T_{ij} \leq t_c^* \cdot j$, then $T_{ij} \leq t_0 \cdot j$ as well because otherwise there would have to be a $t'$ in between such that $T_{ij} = t' \cdot j$, which contradicts the definition of $t_c^*$. The converse also holds by virtue of the fact that $t_c^* < t_0$. Therefore if $\sigma$ is $t_0$-valid, then $\sigma$ is $t_c^*$-valid as well. But since $t_c^* < t_0$, we’ve contradicted the definition of $t_0$. Therefore $t_0$ must be a critical value.

(c) Give an efficient algorithm that finds $t_0$.

**Solution:** There are at most $n^2$ distinct critical values, one for each pair of values for $i$ and $j$. Therefore we may sort the list of critical values (which takes $n^2 \log n$ time) and perform binary search to find the smallest critical value $t_c$ such that there exists a $t_c$-valid permutation. Each query takes $O(n^3)$ time, and since there are $O(n^2)$ elements, we only need to perform $O(\log n^2) = O(\log n)$ queries. The total running time of this algorithm is $O(n^3 \log n)$. 


With a bit more cleverness, you can bring this back down to $O(n^3)$. Instead of starting with a brand new flow every time, modify the existing flow network by removing edges if a $t_0$-valid permutation was found or by adding edges if one was not found. This prevents us from performing the same computations many times.