1 Practice Problems

Problem 1.
Each of the following statements is false. Provide an explanation or counterexample for each.

(a) Suppose we have a disjoint forest data structure where we use the path compression heuristic. Let $x$ be a node at depth $d$ in a tree, then calling $\text{FIND}(x)$ a total of $n$ times takes $\Theta(nd)$ time.

(b) Using Huffman encoding, the character with the largest frequency is always compressed down to 1 bit.

(c) Consider a variation on the set cover problem where each set is of size at most 3. Then the greedy set cover algorithm discussed in class is optimal. (You can choose how the greedy algorithm breaks ties)

Solution
(a) Consider a chain of $d$ elements. The first $\text{FIND}(x)$ will take $O(d)$ work while every subsequent call will take $O(1)$ work. So in total, it takes $O(d + n)$ work.

(b) Consider $A, B, C, D$ each with frequency 0.25. We will connect $A, B$ into a supernode first. Then connect $C, D$ in a supernode next and then connect $A, B, C, D$ into one big supernode. Now each takes 2 bits to encode

(c) Consider elements $\{1, 2, 3, 4, 5, 6\}$ and set $\{1, 2, 3\}, \{1, 4\}, \{2, 5\}, \{3, 6\}$. Then greedily gives us 4 sets to cover it while I can just throw away $\{1, 2, 3\}$ which only needs 3 to cover.

Problem 2.
We know that a sequence of $n$ union/find operations using union by rank and path compression takes time $O(n \log^* n)$. What if all the union operations are done first? Show that a sequence of $n$ unions followed by $m$ finds on a graph with $n$ vertices takes time $O(n + m)$, assuming the unions always are of heads.

Solution
Doing $n$ unions on head costs $O(n)$ times. Now doing $m$ finds will at most change $n - 1$ edges and after that every operations takes $O(1)$. So $m$ finds take $O(m + n)$ time.

Problem 3.
You are given a perfect binary tree with $2^d - 1$ nodes. Suppose that each node $x$ has a weight $w_x$ and that all the weights are distinct for the nodes of the tree. A node is called a local minimum if it is smaller than its parent and both its children (if they exist). The root is a local minimum if it is smaller than its two children, and a leaf is a local minimum if it is smaller than its parent. Given a pointer to the root node of this binary tree, give a $O(d)$ algorithm for finding any local minimum.
Solution
Let \( r \) be the root of the tree. If \( r < r.left \) and \( r < r.right \), then \( r \) is a local minimum and we are done. Otherwise, we recurse on the subtree rooted by the smaller of \( r.left \) and \( r.right \). We know that the weight of this child we recurse on will be smaller than the weight of \( r \), so we are in the same situation as before where we check whether this child is a local minimum and the recurse on its children appropriately. This algorithm is correct because it either finds a local minimum or it reaches a leaf, in which case the leaf is smaller than its parent and that would be a local minimum. The run-time is \( O(d) \) because we go down the tree at most \( d \) levels.

Problem 4.
(From MIT 6.046 Exam, Spring 2015) Finding the median of a sorted array is easy: return the middle element. But what if you are given two sorted arrays \( A \) and \( B \), of size \( m \) and \( n \) respectively, and you want to find the median of all the numbers in \( A \) and \( B \)? You may assume that \( A \) and \( B \) are disjoint.

(a) Give a naive algorithm running in \( O(m + n) \) time.

(b) If \( m = n \), give an algorithm that runs in \( O(\log m) \) time.

(c) For any \( m, n \), give an algorithm that runs in \( O(\log \min(m, n)) \) time.

Solution
(a) Merge the two sorted arrays (which takes \( O(m + n) \) time) to preserve the order of the array and find the median directly.

(b) Pick the median \( m_1 \) for \( A \) and median \( m_2 \) for \( B \). Note that the overall median has to be between \( m_1 \) and \( m_2 \), inclusive. This is because the median being smaller than both \( m_1 \) and \( m_2 \) would mean at least half the elements in both \( A \) and \( B \) are bigger than the median, which makes no sense. We can say symmetrically the median isn’t bigger than both \( m_1 \) and \( m_2 \). Therefore, if \( m_1 = m_2 \) we just return \( m_1 \). Else, we look at the bigger half of \( A \) and the smaller half of \( B \) if \( m_1 < m_2 \) and the other way if \( m_1 > m_2 \) and recursively compute the median of the union of these two halves. This works because we have removed an equal number of terms that we know are at least the median and at most the median. We repeat this \( O(\log m) \) times.

(c) Assume WLOG that \( m > n \). Then, we can remove the first \( \frac{m-n}{2} \) and last \( \frac{m-n}{2} \) elements since even if the elements of \( B \) are extremely large or extremely small, these elements must be in the smaller half and larger half, respectively. Then, we have two arrays of size \( n \), so use part b.

Problem 5.
Given a character array (The number of distinct character is a constant.) representing \( n \) tasks CPU need to do. Tasks can be done in any order and each task can be finished in 1 unit time. For each unit time, CPU could finish one task or just be idle. However, there is a non-negative cooling interval \( k \) that means between two same tasks, there must be at least \( k \) units of time that CPU are doing different tasks or just be idle. Give a greedy approach to find the least amount of time the CPU will take to finish all the given tasks.

Solution
Greedy Property. Let \( t \) be the optimal time and \( S \) be an optimal sequence, so \( |S| = t \). We claim
that for any prefix \(S[1..i]\), we have a solution \(S'\) with the same prefix as of \(S[1..i]\) and \(S'[i+1]\) is the task with most remaining copies (Namely, it appears the most in \(S[i+1..t]\)).

We use the “switch argument” to prove the claim. Let the indices of the most frequent task in the remaining sequence be \(i_1, \ldots, i_a\) and the indices of \(S[i+1]\) be \(j_1, \ldots, j_b\). By definition, \(a \geq b\). Let \(c\) be the smallest index such that \(i_c < j_c\). If such \(c\) does not exist, then let \(c = b + 1\). Then we construct \(S'\) from \(S\) by switching the tasks at \(i_1\) and \(i_1, i_2\) and \(j_2, \ldots\) until \(i_{c-1}\) and \(j_{c-1}\). In can be verified that \(S'\) is still a valid sequence and \(S'[i+1]\) is the task with most remaining copies.

**Algorithm.** Based on the greedy property, first, we can have an \(O(t)\) solution. We keep track of the remaining frequency and the cooling time of each task. Then keep choosing the most frequent task and the cooling time is over.

It is possible to do in the constant time except reading the input. It needs some observations more than the greedy property. The idea is to calculate the idle slots. We know if we only consider the most frequent task, the remaining sequence be \(S\) and the cooling time is over.

**Problem 6.**
Suppose we are given a string of zeros and ones of length \(n\), \(S = a_0...a_{n-1}\). We are asked to find if there exists three well spaced ones in \(S\), say, \(a_i = 1\), \(a_j = 1\), and \(a_k = 1\) so that \(k - j = j - i \geq 1\). For example, the string 10100101 does not have such a pattern while the string 01010011 does.

(a) Give a simple \(O(n^2)\) algorithm to find at least one such triple of ones in \(S\).

The goal in the next two parts is to use the FFT to find an algorithm that runs in \(O(n \log n)\) time to find a well spaced triple of ones.

(b) Suppose we convolve \(S\) by itself, \(C = C_0...C_{2n-2}\) is the sequence gotten by squaring the polynomial \(a_0 + a_1x + ... + a_{n-1}x^{n-1}\). Give an interpretation of \(C_k\) both in the case when \(k\) is odd and even.

(c) Use your interpretation to count the number of well spaced triples in \(O(n \log n)\) time. More specifically, assuming that such a convolution is given for free, your algorithm has to run in \(O(n)\) time.

**Solution**
(a) Let \(x\) run from 0 to \(n - 1\) and \(y\) run from 1 to \(n/2\). Check for each \((x, y)\) whether the indices \(x, x+y, x+2y\) are all between 0 and \(n - 1\) and that they are all 1’s. If this ever happens, we will find a triple and return YES, otherwise we return NO.

(b) Note that \(C_k = \sum_{x+y=k,0 \leq x,y \leq n-1} a_xa_y\). Then, if \(k\) is odd, this sum is even, because for every \(a_xa_y\), there is a corresponding \(a_ya_x\). For \(k\) even, there is a term \(a_{k/2}a_{k/2}\), which is 1 if \(a_{k/2} = 1\) and 0 otherwise. Therefore, \(C_k\) is odd if and only if \(a_{k/2} = 1\), assuming \(k\) is even.

(c) If we look at any coefficient \(C_{2j}\), it equals \(a_j^2\) plus the sum over all \(i \neq k, i + k = 2j\) of \(a_ia_k\). This is the same as \(a_j^2\) plus the sum over all \(i < j < k\) where \(i, j, k\) are a well spaced triple,
of $2a_i a_k$, since $a_i a_k + a_k a_i = 2a_i a_k$. Therefore, if there are $r$ well spaced triples of ones with middle element $a_j$, the value of $C_{2j}$ will be $2r + 1$, as there will be a term $a_j^2$ and $r$ terms of the form $2a_i a_k$ where $i, j, k$ are in an arithmetic progression (i.e. $k - j = j - i ≥ 1$). If there are no such well spaced triples going through $a_j$ as the middle element, then either $a_j = 0$, so $C_{2j}$ is even, or $a_j = 1$ and $C_{2j} = 1$.

Therefore, to count the total number of well spaced triples of ones, we just have to look at all elements $C_{2j}$ and check which ones are odd and at least 3. If $C_{2j} = 2r + 1$ for some $r ≥ 1$, then there are $r$ triples going through $j$ as a middle element, so our answer is the sum over all $C_{2j}$ that are odd of $\frac{C_{2j} - 1}{2}$. This can easily be computed in $O(n)$ time after computing the FFT in $O(n \log n)$ time.

**Problem 7.**

(From MIT 6.006, Spring 2011) Given a log of wood of length $k$, Woody the woodcutter will cut it once, in any place you choose, for the price of $k$ dollars. Suppose you have a log of length $L$, marked to be cut in $n$ different locations labeled $1, 2, \ldots, n$. For simplicity, let indices $0$ and $n + 1$ denote the left and right endpoints of the original log of length $L$. Let the distance of mark $i$ from the left end of the log be $d_i$, and assume that $0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L$.

Give an algorithm to determine the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody.

**Solution**

**Definition:** Let $X[i, j]$ be the minimum cost to break the segment from cut point $i$ to $j$ into $j - i$ pieces. We are looking for $X[0, n + 1]$.

**Recurrence:** We search over where the optimal first cut will take place:

$$X[i, j] = \begin{cases} 
0 & j = i + 1 \\
\min_{i + 1 \leq k \leq j - 1} X[i, k] + X[k, j] + (d_j - d_i) & \text{otherwise}
\end{cases}$$

This recurrence is correct because we are exhaustively checking all the possible places where the first cut can be made, and then recursively finding the best way to cut the resulting two pieces.

$X[i, j]$ finds the cost of the best sequence of cuts, but if we wanted the actual cost, we could have $X[i, j]$ store the location of the first cut point to make on this segment from $i$ to $j$. In other words, $X[i, j]$ is the $k$ that we chose in the minimum above. In order to recover the entire sequence of cut points, we would find $k = X[i, j]$ and recursively compute the best sequence of cut points for the segment between $i$ and $k$ as well as the segment between $k$ and $j$, concatenating those together and appending to $[k]$.

**Analysis:** The runtime is $O(n^3)$ because there are $n$ possible inputs to $X$ and each one takes $O(n)$ time to compute (to find that optimal $k$). The space complexity is $O(n^2)$ because we store 1 number per $X[i, j]$.

**Problem 8.**

Consider the 0-1 knapsack problem: We have $n$ items, each of which has sizes $s_i > 0$ and value $v_i > 0$. We have a knapsack of size $M$ and want to maximize the sum of the values of the items
inside the knapsack without the sum of the sizes of the items exceeding $M$. Let $V = \sum_{i=1}^{n} v_i$ be the sum of all the values of the items.

(a) Describe a solution that computes the maximum achievable value in $O(nM)$ time.

(b) Describe a solution that computes the maximum achievable value in $O(nV)$ time.

Solution

(a) **Definition**: Let $X[k, m]$ be the highest achievable value when we are trying to pack the first $k$ items into a knapsack with capacity $m$. We are looking for $X[n, M]$.

**Recurrence**: When we are calculating $X[k, m]$, we can either choose to use or not use the $k$th item. If we choose to use it, then we must subtract $s_k$ from the capacity of the knapsack, but add on $u_k$ amount of value. If we do not choose to use it, then the capacity of the knapsack does not change. In either case, we recurse on the first $k - 1$ elements with a possibly smaller knapsack.

$$X[k, m] = \begin{cases} 0 & k = 0, m \geq 0 \\ -\infty & m < 0 \\ \max\{X[k - 1, m], X[k - 1, m - s_k] + u_k\} & \text{else} \end{cases}$$

**Analysis**: The runtime is $O(nM)$ because there are $n \cdot M$ possible inputs to $X$ and each one takes $O(1)$ time to compute. The space complexity seems to be $O(nM)$ at first, but we can use bottom-up dynamic programming to improve upon the space. We notice that in order to calculate $X[k, m]$, we only need the row $X[k - 1, \cdot]$. Therefore, the space can be improved to $O(M)$ by only keeping one row $X[k - 1, \cdot]$ at a time.

(b) **Definition**: Let $X[k, v]$ be the minimum size of a subset of the first $k$ items whose values sum up to exactly $v$. To calculate the final answer, we look at $X[n, v]$ for all $v$ between $V$ and 0 in order and find the largest value of $v$ such that $X[n, v] \leq M$.

**Recurrence**: We again consider whether or not we will be taking the $k$th item when computing $X[k, v]$. If we decide to take it, then we need to compute the minimum size where we use a subset of the first $k - 1$ items and try to sum to achieve the value $v - v_k$. If we don’t decide to take it, then we change $k$ to $k - 1$ and recurse.

$$X[k, v] = \begin{cases} 0 & k = 0, v \geq 0 \\ \infty & v < 0 \\ \min\{X[k - 1, v], X[k - 1, v - v_k] + s_k\} & \text{else} \end{cases}$$

**Analysis**: The runtime is $O(nV)$ because there are $n \cdot V$ possible inputs and each takes $O(1)$ time to compute. The space complexity is $O(V)$ because we only need to store $X[k - 1, \cdot]$ in order to compute $X[k, \cdot]$. 

5