CSE 40113: Algorithms
Homework 2

You may complete this homework in groups of 3 or less students. Note that the integrity policy applies: your group should write up your own work, although you're welcome to work on the problems in a larger group. If you have any questions, please re-read both the homework guidelines and the academic integrity policy carefully, and then come discuss any questions or concerns with me.

Required Problems

1. Describe recursive algorithms for the following generalizations of subset sum:

   (a) Given an array of positive integers $X[1..n]$ and an integer T, compute the number of subsets of T whose elements sum to T.

   (b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers with an integer T, where each $W[i]$ represents the *weight* of the element $X[i]$, compute the maximum weight subset of X whose elements sum to T. If no such subset exists, your algorithm should return $-\infty$.

   Do **NOT** analyze or optimize your algorithm's run time after writing the recurrence to describe it; a correct algorithm whose running time is exponential in n is all that I'm requiring for full credit. (You do need to do a proof of correctness and pseudocode, though, as well as writing a recurrence down for the algorithm.)

2. An *addition chain* for an integer n is an increasing sequence of integers that start with 1 and end with n, such that each entry after the second is the sum of two earlier entries.

   More formally, a sequence $x_0 < x_1 < \ldots < x_l$ is an addition chain for n if and only if:

   - $x_0 = 1$
   - $x_l = n$
   - for every index $k \geq 1$, there are two smaller indices $i \leq j < k$ such that $x_k = x_i + x_j$

   We say the *length* of such an addition chain is l (since we don't bother to count the first element). For example: $< 1, 2, 3, 5, 10, 20, 23, 46, 92, 184, 187, 374 >$ is an addition chain for 374 of length 11.

   Describe a recursive backtracking algorithm to compute a minimum length addition chain for a given positive integer n. Again, do **NOT** analyze or optimize your algorithm's run time after writing the recurrence to describe it; a correct algorithm whose running time is exponential in n is all that I'm requiring for full credit.

3. Consider two arrays $X[1..k]$ and $Y[1..n]$, where $k \leq n$. Describe a recursive backtracking algorithm to decide if X a subsequence of Y. For example, the string PPAP is a subsequence of PENPINEAPPLEAPPLEPEN.

   Again, no need to analyze or optimize your algorithm's run time after writing the recurrence to describe it; a correct algorithm whose running time is exponential in n and/or k is all that I'm requiring for full credit.

4. Sample Solved Problem: A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways:
BANANAANANAS, BANANAANANAS, or BANANANANAS.

   Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:
PRODGYRNAM AMMIINCG and DYPRONGARMAMMICING.

   Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

   Solution:

   Recursive formulation: We define a recursive function $Shuf(i, j)$, which is True if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

   - Shuf(i, j) = true if $i = j = 0$
   - Shuf$(0, j-1)$ AND $(B[j] = C[j])$ if $i = 0$ and $j > 0$
   - Shuf$(i-1, 0)$ AND $(A[i] = C[i])$ if $i > 0$ and $j = 0$
   - (Shuf$(i-1, j)$ AND $(A[i] = C[i+j])$) OR (Shuf$(i, j-1)$ AND $(B[j] = C[i+j])$) if $i > 0$ and $j > 0$

   The proof that this formulation is correct can be shown via induction: if you're considering the $(i+j)^{th}$ character of C, it must be from either $A[i]$ or $B[j]$, since the entire prefix $A[1..i]$ and $B[1..j]$ must be included. We are trying both options, and returning true if either works. The base cases handle either A or B being empty, in which case either we've matched everything (and both are 0) or we must exactly match the rest of C to which ever string is left.

   This immediately yields a recursive algorithm, where you'll start with $A[1..m], B[1..n]$, and $C[1..(m+n)]$. At each level, you'll check the indices in $O(1)$ time, and make either 1 or 2 recursive calls - one in two of the bases cases, but two call if $i > 0$ and $j > 0$.

   At a high level, we note that this is exponential, since each call does $O(1)$ work comparing, and then in the worst case makes two recursive calls with inputs that are 1 character smaller (in either A or B, as well as in C). So, this is a Hanoi-like recursive algorithm, and will take exponential time.

   In case you're curious, we can formalize this by letting $k = m+n$, which is the size of the input. We can write the runtime as $T(k) \leq 2T(k-1) + O(1)$, since k reduces by 1 each time in the recursion, yielding $T(k) = O(2^k) = O(2^{m+n})$. (Note that I would not require this last part in your homework.)