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 X whose elements sum to T.

   Solution: This can actually be solved by a simple modification to the algorithm on page 78 of the textbook. In the base case if $T = 0$, return 1. In the base case where $i = 0$ or $T < 0$, return 0 (since you have not found a subset). Finally, calculate with and without as shown, but then return with + without in the last line.

   Runtime: Unchanged from the book, so still $T(n) \leq 2T(n-1) + O(1) = \Theta(2^n)$.

   Correctness: Induction on n, the size of the array:

   Base case: If $T = 0$, you have one subset that works (the empty set), so return 1. If $T < 0$ or ($n = 0$ and $T \neq 0$), you have failed, so you find 0 subsets in this recursive structure.

   Inductive Hypothesis: Assume the recursion fairy can solve correctly on sets of size $n-1$.

   Inductive step: Either $X[n]$ is in some subsets, or it is not. Try both: count the number of subsets that include $X[n]$, which means you'd have a subset of $X[1..n-1]$ that sums to $T - X[n]$, which the by the induction hypothesis you can compute correctly in the recursive call to compute with. Then count the number that don't include $X[n]$, which means you have a subset of $X[1..n-1]$ that sums to T, which again we compute correctly by the inductive hypothesis. Since these are disjoint sets, we can use the rule of sum and add the result together, and this must be the number of subsets of $X[1..n]$ that 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. Your algorithm should return just the weight of this maximum subset, and if no such subset exists, your algorithm should return $-\infty$.

   Solution: Again, we modify the helper function from the book slightly, so that it also include the weight array W. We will define the recursive function as follows, where we return the value of the Max Weight Subset summing to T in $X[i..n]$, given inputs X and i, as well as W and T:

```
MaxWeightSubsetSum($X, W, i, T$):
  if $T = 0$
      return 0
  else if $T < 0$ or $i = 0$
      return $-\infty$
  else
      with ← W[i] + MaxWeightSubsetSum($X, W, i + 1, T - X[i]$)
      without ← MaxWeightSubsetSum($X, W, i + 1, T$)
      return max(with,without)
```

Note that this does assume that $-\infty$ plus any other number is still $-\infty$; if necessary we can add an if statement to track this condition separately.

Correctness: Induction on the size of the array, n: Base case: If $T = 0$, the maximum subset has weight 0, with 0 items included. If $T < 0$ or ($n = 0$ and $T \neq 0$), you have failed, so we correctly return $-\infty$.

Inductive Hypothesis: Assume the algorithm works for inputs $\leq n - 1$.

Inductive Step: Consider an input of size n. The maximum weight subset will either include $A[n]$ or will not; our algorithm tries both possibilities and recurses on a smaller array, so by the inductive hypothesis we know that those calls return the correct answer for the smaller array; we then return the larger, which must be the maximum weight subset.

Runtime: Same as the original algorithm and part (a), as we are still doing a constant amount of work in the function (just now additions instead of boolean operations) and recursing at most twice.

■

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.

Solution: This recursion can be formalize either way, but I felt the most natural was "backwards": We know the last element must equal n, and the first must equal 1. The second element x_1 must equal 2, since that is the only way to generate a sum from two smaller index elements. For an arbitrary index i, consider a partially generated chain $x_0, x_1, \ldots, x_i$. We generate all possible "next" elements by summing every possible pair of earlier elements, recursing on a larger addition chain, and keep the largest, which we then return. The recursion will terminate when we reach a value $x_i = n$, when we return the length i of the chain.

```
AddChain(n):
  if $n = 1$
      return 1
  initialize an array $X$
  $X[0] \leftarrow 1$
  $X[1] \leftarrow 2$
  AChelper(X,n)
```

```
AChelper($X[1..k], n$):
  if $A[k] = n$
      return k
  best $\leftarrow \infty$
  for $i \leftarrow 1$ to $k$
      for $j \leftarrow 1$ to $k$
          next $\leftarrow A[i] + A[j]$
          if next $> A[k]$
              append next to $X$
              best $\leftarrow$ min(best, AChelper(X,n))
              remove last element from $X$
      return best
```

Runtime: this algorithm makes a quadratic number of recursive calls per level, and (since 1 and 2 are in the chain) at least two of them are as large as the Fibonacci recurrence. As such, this is VERY exponential.

Correctness of AChelper, by induction on $x = n - A[k]$, the "gap" between our last element and target value n:

Base case: if $A[k] = n$ (so $x = 0$), our algorithm correctly returns k, the length of our chain.

Inductive hypothesis: if $n - A[k] < x$, assume the algorithm will correctly determine the right answer.

Inductive step: Consider $x = n - A[k]$. Our algorithm tries every possible next term in the sequence, by iterating over the sum of any two numbers already there. For each, we appending it to the list, and recursing with that value as $A[k+1]$. Since every $A[k+1] > A[k]$, we get $n - A[k+1] < n - A[k] = x$, so by our inductive hypothesis, the algorithm can correctly determine the best possible remainder of the addition chain. Our own sequence must have some number next, and since we are generating and trying every possibility and updating the smallest accordingly, we will find the best one in this search and return it.

Runtime: This algorithm makes k^2 recursive calls. It is worth nothing that at least one is generated by adding $A[k] + 1$, so it is only 1 larger (or x, our gap, is only one smaller), and another will be adding $A[k]+2$, since we know 1 and 2 are in every addition chain at position 0 and 1. That means this is worse than the Fibonacci recurrence, as those are only two of the of the k^2 calls, so this is very very exponential.

■

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.

Solution: Let's try backtracking: if $X[k] = Y[n]$ we can match $X[k]$ to $Y[n]$ or not, and make two recursive calls to try to match $X[1...k-1]$ to $Y[1..n-1]$ in one and $X[1...k]$ to $Y[1..n-1]$ in the other. We return true if either works. If $X[k] \neq Y[n]$, then we can recurse and try to match $X[1...k]$ to $Y[1..n-1]$, since $X[k]$ can't find a match yet. Our base cases are if either X or Y is ever empty - if $k=0$, then we've matched all of the subsequence, and if $n=0$ when k is still positive, then we've failed to match. This yields the following solution:

```
Subsequence($X[1..k], Y[1..n]$):
  if $k = 0$
      return true
  if $n = 0$
      return false
  if $X[k] = Y[n]$
      return Subsequence($X[1..k-1], Y[1..n-1]$) or Subsequence($X[1..k], Y[1..n-1]$)
  else
      return Subsequence($X[1..k], Y[1..n-1]$)
```

Proof of correctness: Induction on n. If $n=0$ and $k=0$, we will return true, as the empty subsequence is a subset of any other. If $k \geq 1$ and $n=0$, we will correctly fail because a positive length subsequence cannot be found in an empty sequence.

Inductive hypothesis: assume the algorithm works correctly on inputs where Y is of length $n-1$ or less.

Inductive step: Consider an input of size n. We consider the last element in Y: Either it matches to the last element of X, or it does not. We try both options, and recurse at most twice on a prefix of Y of length $n-1$, so by our induction hypothesis those calls are answered correctly. Since we try both options, if X is a prefix, one of these two calls much find it.

Runtime: In the worst case, this takes an input where Y has length n and makes two recursive calls where Y has length $n-1$, so the recurrence can be written as: $T(n) \leq 2T(n-1) + O(1)$, which gives exponential time in the worst case: $T(n) = O(2^n)$, as it is the same recurrence as subset sum.

■

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.)