Problem 1: Exhaustive Coin Change

Given coins $c_1 > c_2 > \ldots > c_d = 1$, assume we have an array $a$, where $a[i]$ denotes the number of coins of type $c_i$. Therefore, $\sum_{i=1}^d a[i]$ is the total number of coins, and $\sum_{i=1}^d a[i]c_i$ is the value.

Given a desired value $M$, an exhaustive strategy is to consider all possible values of $a$, where $a[i]$ varies between 0 and $M/c_i$, as seen in class. For instance, when $M = 30$ and $c_1 = 25$, $c_2 = 15$, and $c_3 = 1$, the possible values of $a$ can be viewed as leaves in a tree.

The preorder traversal of the tree gives internal nodes and leaves of the form $(a[0], a[1], a[2])$:

$(-,-,-), (0,-,-), (0,0,-), (0,0,0), \ldots, (0,0,30), (0,1,-), (0,1,0), \ldots, (0,1,30), (0,2,-), (0,2,0), \ldots, (0,2,30), (1,-,-), \ldots, (1,2,30), (-,-,-)$

Assume $a[0]$ is the level, i.e. this is also the number of values not equal to $'-'$ in $a$. Therefore, if $a[0] = l$, $a[l+1], \ldots, a[d]$ can be ignored.

(a) Given $a$, write two function $\text{value}(a,c)$ that computes $\sum_{i=1}^d a[i]c_i$, and $\text{count}(a)$ that computes $\sum_{i=1}^d a[i].$
(b) Given \( a \), write a function \( \text{next}(a, c, d, M) \), that changes \( a \) to the next node in the preorder traversal of the tree. Do not explicitly build the tree structure, just manipulate the values in \( a \).

(c) Given \( a \), write a function \( \text{skip}(a, c, d, M) \) that changes \( a \) to a node in a lower level that comes next in the preorder traversal of the tree. If this node does not exist, it’s \((-,-,\ldots,-)\).

(d) Implement the exhaustive search by going through all possible values of \( a \), i.e. starting at \((1,0,\ldots,0)\) and using \( \text{next}(a, c, d, M) \); however, if \( \text{value}(a, c) > M \) or \( \text{count}(a) \) is bad, use \( \text{skip}(a, c, d, M) \) instead to bypass some invalid possibilities. Use a large number of \( M \) and keep track of how many nodes you check. Compare this approach to the basic exhaustive method.

\[
a \leftarrow (1,0,\ldots,0)\\
\text{best} \leftarrow M\\
\text{checked} \leftarrow 0\\
\text{while } a[0] > 0 \text{ (not at root)}\\
\quad \text{checked} \leftarrow \text{checked} + 1\\
\quad \text{if } \text{value}(a, c) > M \text{ or } \text{count}(a) > \text{best}\\
\quad \quad \text{then } \text{skip}(a, c, d, M)\\
\quad \quad \quad \text{continue}\\
\quad \text{if } a[0] = d \text{ (a leaf)}\\
\quad \quad \text{then if } \text{value}(a, c) = M \text{ and } \text{count}(a) < \text{best}\\
\quad \quad \quad \text{then } \text{best} \leftarrow \text{count}(a, c)\\
\quad \quad \quad \text{best}_a \leftarrow a\\
\quad \text{next}(a, c, d, M)\\
\text{return best}_a
\]

Problem 2: Fibonacci Revisited

Consider the following algorithm, as seen in class:

\[
\text{fib}(a, b, n)\\
\text{while } n > 1\\
\quad b \leftarrow a + b\\
\quad a \leftarrow b - a\\
\quad n \leftarrow n - 1\\
\text{return } b
\]

\[
\text{fib}(n)\\
\text{return } \text{fib}(1, 1, n)
\]

This algorithm requires \( O(n) \) arithmetic operations. However, since Fibonacci numbers grow fast, it is not reasonable to assume that arithmetic operations take constant time. Addition of \( b \) bit numbers take \( O(b) \) time (standard addition algorithm we do by hand). One can show that the \( n \)th Fibonacci number is \( O(\phi^n) \); therefore, all intermediate results while computing \( \text{fib}(n) \) need \( O(n) \) bits. This makes the above algorithm an \( O(n^2) \) algorithm. We can do better.
Consider the matrix:

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

It is easy to verify that $F_{2,2}^n$ is the $n$th Fibonacci number. This means, we only need to multiply the matrix by itself $n$ times, each matrix multiplication involves 8 multiplications and 4 additions. The bottleneck is multiplication. Assume that we can multiply two $n$ bit numbers is $O(n^\alpha)$ time for $1 < \alpha < 2$. Then the total running time of this algorithm will be $O(n^{1+\alpha})$, not an improvement over the $O(n^2)$ bound.

However, we can use a technique called repeated squaring. Consider the function $pow$ (for power).

$$pow(F, i) \text{ (compute } F^i \text{)}$$

if $i > 0$

then if $i$ is even

then return $\text{square}(pow(F, i/2))$

else return $F \times pow(F, i - 1)$

else return 1

(a) What is the number of multiplications performed by this algorithm using Big-O notation?

(b) Describe a better than $O(n^2)$ algorithm for computing the $n$th Fibonacci number.

**Problem 3**

Do problems 2.1, 2.2, and 2.15 in the book.