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 $(a[1], a[2], a[3])$:

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

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 \( value(a,c) \) that computes \( \sum_{i=1}^{a[0]} a[i]c[i] \), and \( count(a) \) that computes \( \sum_{i=1}^{a[0]} a[i] \).

**Solution:**

\[ \text{value}(a,c) \]

\[
\text{sum} \leftarrow 0
\]

\[
l \leftarrow a[0]
\]

for \( i \leftarrow 1 \) to \( l \)

\[
\text{sum} \leftarrow \text{sum} + a[i]c[i]
\]

return \( \text{sum} \)

\[ \text{count}(a) \]

\[
um \leftarrow 0
\]

\[
l \leftarrow a[0]
\]

for \( i \leftarrow 1 \) to \( l \)

\[
um \leftarrow \text{num} + a[i]
\]

return \( \text{num} \)

(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 \( a \).

**Solution:**

\[ \text{next}(a,c,M,d) \]

\[
l \leftarrow a[0]
\]

if \( l < d \) not a leaf, so go to first in next level

\[
\text{then } a[0] \leftarrow l + 1
\]

\[
a[l + 1] \leftarrow 0
\]

return

for \( i \leftarrow l \) downto 1 \( \triangleright \) find where we can increment

\[
\text{if } a[i] < M/c[i]
\]

\[
\text{then } a[i] \leftarrow a[i] + 1
\]

\[
a[0] \leftarrow i
\]

return \( a[0] \leftarrow 0 \triangleright \) back to root

(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, if’s \((-,-,\ldots,-)\).

**Solution:**

\[ \text{skip}(a,c,M,d) \]

\[
l \leftarrow a[0] - 1 \triangleright \text{start with previous level}
\]

\[
a[0] \leftarrow l
\]

for \( i \leftarrow l \) downto 1 \( \triangleright \) find where we can increment

\[
\text{if } a[i] < M/c[i]
\]

\[
\text{then } a[i] \leftarrow a[i] + 1
\]

\[
a[0] \leftarrow i + 1
\]

return \( a[0] \leftarrow 0 \triangleright \) back to root
(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.

$$a \leftarrow (1, 0, \ldots, 0)$$
$$\text{best} \leftarrow M$$
$$\text{checked} \leftarrow 0$$

while $a[0] > 0$ (not at root)

$$\text{checked} \leftarrow \text{checked} + 1$$

if $\text{value}(a, c) > M$ or $\text{count}(a) > \text{best}$

then $\text{skip}(a, c, d, M)$

continue

if $a[0] = d$ (a leaf)

then if $\text{value}(a, c) = M$ and $\text{count}(a) < \text{best}$

then $\text{best} \leftarrow \text{count}(a, c)$

$\text{best}_a \leftarrow a$

$\text{next}(a, c, d, M)$

return $\text{best}_a$

**Problem 2: Fibonacci Revisited**

Consider the following algorithm, as seen in class:

$$\text{fib}(a, b, n)$$

while $n > 1$

$$b \leftarrow a + b$$

$$a \leftarrow b - a$$

$$n \leftarrow n - 1$$

return $b$

$$\text{fib}(n)$$

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 $F^n_{2,2}$ 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 $\text{pow}$ (for power).

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

if $i > 0$
  then if $i$ is even
    then return $\text{square}(\text{pow}(F, i/2))$
    else return $F \times \text{pow}(F, i - 1)$
  else return 1

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

**Solution:** The power is halved when its even. So we can’t have more than $O(\log n)$ such events. In addition, every time the power is odd, the next power will be even. Therefore, we at most double the number of multiplications. This is still $O(\log n)$ matrix multiplications.

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

**Solution:** Since each matrix multiplication involves a constant number of scalar multiplications, and all numbers have $O(n)$ bits, the total running time of this algorithm is $O(n^\alpha \log n)$, which is asymptotically better than $n^2$ since $\alpha < 2$, and any power of $n$ dominates any power of $\log n$.

**Problem 3**
Do problems 2.1, 2.2, and 2.15 in the book.

**Solution to Problem 2.1:**
We can start with the minimum as $min = \infty$ and the maximum as $max = -\infty$. Then, we start comparing pairs of numbers, say $a[i]$ and $a[i + 1]$. With one comparison, we can determine which of the two is smaller and which is larger. We then compare the smaller one to $min$ and the larger one to $max$, and thus update our minimum and maximum. This is at most 3 comparisons per pair of numbers, for a total of at most $3n/2$ comparisons.

```
min ← ∞
max ← −∞
▷ assume for simplicity that we have an even number of elements 2n
for i ← 1 to n
  if a[2i − 1] < a[2i]
    then if a[2i − 1] < min
       then min ← a[2i − 1]
    if a[2i] > max
       then max ← a[2i]
  else ▷ do the reverse comparisons
```
Solution to Problem 2.2
I will write a function that increments the counter by 1. One is recursive and one is iterative. To increment from \((0, \ldots, 0)\) to \((n_1, \ldots, n_d)\) we can call the function \(\prod_{i=1}^d (n_i + 1) - 1\) times.

\[
\text{incRecursive}(a, n, d) \begin{array}{l}
\text{if } d > 0 \text{ and } a[d] = n[d] \\
\quad \text{then } a[d] \leftarrow 0 \\
\quad \text{incRecursive}(a, n, d - 1) \\
\text{else if } d > 0 \\
\quad a[d] = a[d] + 1
\end{array}
\]

\[
\text{incIterative}(a, n, d) \begin{array}{l}
\text{while } d > 0 \text{ and } a[d] = n[d] \\
\quad a[d] \leftarrow 0 \\
\quad d \leftarrow d - 1 \\
\text{if } d > 0 \\
\quad \text{then } a[d] = a[d] + 1
\end{array}
\]

Solution to Problem 2.15
This is very similar to the game of Nim. We can first set \(A(1, 1) = 0\) because the first player will lose on a \(1 \times 1\) grid. Then, we have the following recurrence:

\[
A(i, j) = \begin{cases} 
0 & A(i - 1, j) = A(i, j - 1) = A(i - 1, j - 1) = 1 \\
1 & \text{otherwise}
\end{cases}
\]

with special care done to the first row and first column because not all neighbors exist.