1. Consider the problem statement: *sum the numbers from* \( a \) *to* \( b \) (*inclusive*). Assuming \( a \) is less than or equal to \( b \) at the start, does this recursive function compute the correct result?

```c
int sum(int a, int b)
{
    if (a == b)
        return 0;
    else
        return a + sum(a+1, b);
}
```

(a) Yes  
(b) No

2. Consider the problem statement: *sum the numbers from* \( a \) *to* \( b \) (*inclusive*). Assuming \( a \) is less than or equal to \( b \) at the start, does this recursive function compute the correct result?

```c
int sum(int a, int b)
{
    if (a == b)
        return a;
    else
        return a + sum(a+1, b);
}
```

(a) No  
(b) Yes

3. Consider the problem statement: *sum the numbers from* \( a \) *to* \( b \) (*inclusive*). Assuming \( a \) is less than or equal to \( b \) at the start, does this recursive function compute the correct result?

```c
int sum(int a, int b)
{
    if (a == b)
        return b;
    else
        return b + sum(a+1, b);
}
```

(a) No  
(b) Yes

4. Consider the problem statement: *sum the numbers from* \( a \) *to* \( b \) (*inclusive*). Assuming \( a \) is less than or equal to \( b \), does this recursive function compute the correct result?

```c
int sum(int a, int b)
{
    if (a == b)
        return b;
    else
        return a + sum(a+1, b);
}
```

(a) Yes  
(b) No

5. Consider the problem statement: *sum the numbers from* \( a \) *to* \( b \) (*inclusive*). Assuming \( a \) is less than or equal to \( b \), does this recursive function compute the correct result?
int sum(int a,int b)
{
    if (a == b)
        return 1;
    else
        return a + sum(a+1,b);
}

(a) No
(b) Yes

6. Consider the problem statement: sum the numbers from a to b (inclusive). Assuming a is less than or equal to b, does this recursive function compute the correct result?

int sum(int a,int b)
{
    if (a > b)
        return 0;
    else
        return a + sum(a+1,b);
}

(a) No
(b) Yes

7. Consider the problem statement: sum the numbers from a to b (inclusive). Assuming a is less than or equal to b, does this recursive function compute the correct result?

int sum(int a,int b)
{
    if (a == b)
        return a;
    else
        return b + sum(a,b-1);
}

(a) No
(b) Yes

8. Consider the problem statement: sum the numbers from a to b (inclusive). Assuming a is less than or equal to b, does this recursive function compute the correct result?

int sum(int a,int b)
{
    if (a == b)
        return a;
    else
        return a + sum(a,b-1);
}

(a) Yes
(b) No

9. Consider the problem statement: sum the numbers from a to b (inclusive). Assuming a is less than or equal to b, does this recursive function compute the correct result?

int sum(int a,int b)
{
    if (a > b)
        return 0;
    else
        return b + sum(a,b-1);
}

(a) Yes
(b) No
10. Consider the problem statement: *sum the numbers from* a *to b (inclusive)*. Assuming a is less than or equal to b, does this recursive function compute the correct result?

```c
int sum(int a, int b)
{
    if (a > b)
        return b;
    else
        return b + sum(a, b-1);
}
```

(a) Yes
(b) No

11. Consider the problem statement: shuffle two lists, creating a third list. Will the following implementation shuffle correctly? Assume the lists sent to shuffle hold, at least, two items each.

```c
Node *shuffle(Node *a, Node *b)
{
    if (a == 0)
        return b;
    else
    {
        rest = shuffle(b, tail(a));
        return join(head(a), rest);
    }
}
```

(a) sometimes
(b) never
(c) always

12. Consider the problem statement: shuffle two lists, creating a third list. Will the following implementation shuffle correctly? Assume the lists sent to shuffle hold, at least, two items each.

```c
Node *shuffle(Node *a, Node *b)
{
    if (a == 0)
        return 0;
    else
    {
        rest = shuffle(b, tail(a));
        return join(head(a), rest);
    }
}
```

(a) never
(b) sometimes
(c) always

13. Consider the problem statement: shuffle two lists, creating a third list. Will the following implementation shuffle correctly? Assume the lists sent to shuffle hold, at least, two items each.

```c
Node *shuffle(Node *a, Node *b)
{
    if (a == 0)
        return b;
    else
    {
        rest = shuffle(tail(b), tail(a));
        return join(head(a), rest);
    }
}
```

(a) sometimes
(b) always
14. Consider the problem statement: shuffle two lists, creating a third list. Will the following implementation shuffle correctly? Assume the lists sent to shuffle hold, at least, two items each.

Node *shuffle(Node a, Node b)
{
    if (a == 0)
        return b;
    else
    {
        rest = shuffle(tail(a), tail(b));
        return join(head(a), join(head(b), rest));
    }
}

(a) never  
(b) always  
(c) sometimes

15. Consider the problem statement: shuffle two lists, creating a third list. Will the following implementation shuffle correctly? Assume the lists sent to shuffle hold, at least, two items each.

Node *shuffle(Node a, Node b)
{
    if (a == 0)
        return b;
    else if (b == 0)
        return a;
    else
    {
        rest = shuffle(tail(a), tail(b));
        return join(head(a), join(head(b), rest));
    }
}

(a) never  
(b) always  
(c) sometimes

Concept: recognizing patterns

16. What pattern does the following function implement?

Node *f(Node *a)
{
    if (a == 0)
        return 0;
    else if (isX(head(a)))
        return join(head(a), f(tail(a)));
    else
        return f(tail(a));
}

(a) the counting pattern  
(b) the search pattern  
(c) the extreme pattern  
(d) the filter pattern

17. What pattern does the following function implement?

Node *f(Node *a)
{
    if (a == 0)
        return 1;
    else if (isX(head(a)))
```c
return head(a) * f(tail(a));
else
    return f(tail(a));
}
```

18. What pattern does the following function implement?

```c
Node *f(Node *a)
{
    return g(head(a), tail(a));
}
Node *g(Node *e, Node *b)
{
    if (b == 0)
        return e;
    else if (isLT(head(b), e))
        return g(head(b), tail(b));
    else
        return g(e, tail(b));
}
```

(a) the extreme pattern
(b) the filter pattern
(c) the filtered-accumulate pattern
(d) the accumulate pattern

19. What pattern does the following function implement?

```c
Node *f(Node *a, Node *b)
{
    if (b == 0)
        return 0;
    else if (isSame(head(b), a))
        return 1;
    else
        return f(a, tail(b));
}
```

(a) the filter pattern
(b) the extreme pattern
(c) the search pattern
(d) the filtered-counting pattern

Concept: understanding recurrences

20. Consider this recurrence:

- \( f(a, b) \) is 1 if \( b \) is zero
- \( f(a, b) \) is \( a \cdot f(a, b - 1) \) otherwise

The recurrence is implemented with:

(a) one function with an if-else
(b) one function with an if-else if-else if-else
(c) one function with an if-else if-else
(d) two functions, one with an if-else if-else
(e) two functions, one with an if-else

21. Consider this recurrence:
f(n) is 0 if n is 0
f(n) is 1 if n is 1
f(n) is f(n - 1) + f(n - 2) otherwise

The recurrence is implemented with:
(a) two functions, one with an if-else
(b) one function with an if-else
(c) two functions, one with an if-else if-else
(d) one function with an if-else if-else
(e) one function with an if-else if-else if-else

22. Consider this recurrence:
g(a, b, n) is a if n is 0
g(a, b, n) is g(b, a + b, n - 1) otherwise

The recurrence is implemented with:
(a) one function with an if-else
(b) one function with an if-else if-else
(c) two functions, one with an if-else
(d) one function with an if-else if-else if-else
(e) two functions, one with an if-else if-else

23. Consider this recurrence:
f(t, a, b) is t if b is zero
f(t, a, b) is f(t, a * a, b / 2) if b is even
f(t, a, b) is f(t * a, a, b - 1) otherwise

The recurrence is implemented with:
(a) one function with an if-else if-else if-else
(b) two functions, one with an if-else
(c) one function with an if-else if-else
(d) one function with an if-else
(e) two functions, one with an if-else if-else

24. Consider this recurrence:
f(a, b) is 1 if b is zero
f(a, b) is f(a * a, b / 2) if b is even
f(a, b) is a * f(a, b - 1) otherwise

The recurrence is implemented with:
(a) one function with an if-else
(b) one function with an if-else if-else if-else
(c) two functions, one with an if-else
(d) one function with an if-else if-else
(e) two functions, one with an if-else if-else

25. Consider this recurrence:
g(t, a, b) is t if b is zero
g(t, a, b) is g(t, a * a, b / 2) if b is even
g(t, a, b) is g(t * a, a, b - 1) otherwise

The recurrence is implemented with:
(a) one function with an if-else if-else
(b) two functions, one with an if-else
(c) one function with an if-else if-else if-else
(d) two functions, one with an if-else if-else
(e) one function with an if-else
26. Consider this recurrence:

\[
g(t,a,b) = \begin{cases} 
    b & \text{if } a \text{ is the empty list} \\
    a & \text{if } b \text{ is the empty list} \\
    \text{head}(a) \text{ joined to } g(1,tail(a),b) & \text{if } t \text{ is 0} \\
    \text{head}(b) \text{ joined to } g(0,a,tail(b)) & \text{otherwise}
\end{cases}
\]

The recurrence is implemented with:

(a) one function with an if-else if-else if-else
(b) one function with an if-else
(c) two functions, one with an if-else
(d) one function with an if-else if-else
(e) two functions, one with an if-else if-else

27. Consider this recurrence:

\[
f(t,a,b) = \begin{cases} 
    t & \text{if } b \text{ is zero} \\
    f(a \times t,a,b - 1) & \text{otherwise}
\end{cases}
\]

The recurrence is implemented with:

(a) two functions, one with an if-else if-else
(b) one function with an if-else
(c) two functions, one with an if-else
(d) one function with an if-else if-else if-else
(e) one function with an if-else if-else

28. Consider this recurrence:

\[
f(a) = g(\text{head}(a),\text{tail}(a)) \\
g(b,c) = \begin{cases} 
    b & \text{if } c \text{ is the empty list} \\
    g(\text{head}(c),\text{tail}(c)) & \text{if } \text{is}(\text{head}(c),b) \\
    g(b,tail(c)) & \text{otherwise}
\end{cases}
\]

The recurrence is implemented with:

(a) one function with an if-else if-else
(b) two functions, one with an if-else
(c) one function with an if-else
(d) one function with an if-else if-else if-else
(e) two functions, one with an if-else if-else

Concept: implementing recurrences

29. (2 pts) Implement this recurrence as a C function. The return type is int:

\[
f(a,b) = \begin{cases} 
    1 & \text{if } b \text{ is zero} \\
    a \ast f(a,b - 1) & \text{otherwise}
\end{cases}
\]
30. (2 pts) Implement this recurrence as a C function. The return type is `int`:

\[
\begin{align*}
    f(t, a, b) &= t \text{ if } b \text{ is zero} \\
    f(t, a, b) &= f(a \times t, a, b - 1) \text{ otherwise}
\end{align*}
\]

31. (2 pts) Implement this recurrence as a C function. The return type is `int`:

\[
\begin{align*}
    f(n) &= 0 \text{ if } n \text{ is 0} \\
    f(n) &= 1 \text{ if } n \text{ is 1} \\
    f(n) &= f(n - 1) + f(n - 2) \text{ otherwise}
\end{align*}
\]

32. (2 pts) Implement this recurrence as a C function. The return type is `int`:

\[
\begin{align*}
    g(a, b, n) &= a \text{ if } n \text{ is 0} \\
    g(a, b, n) &= g(b, a + b, n - 1) \text{ otherwise}
\end{align*}
\]
33. (2 pts) Implement this recurrence as a C function. The return type is \textbf{int}: 

\begin{align*}
f(a,b) & \text{ is 1 if } b \text{ is zero} \\
f(a,b) & \text{ is } f(a \times a, b / 2) \text{ if } b \text{ is even} \\
f(a,b) & \text{ is } a \times f(a,b - 1) \text{ otherwise}
\end{align*}

34. (2 pts) Implement this recurrence as a C function. The return type is \textbf{int}: 

\begin{align*}
g(t,a,b) & \text{ is } t \text{ if } b \text{ is zero} \\
g(t,a,b) & \text{ is } g(t,a \times a, b / 2) \text{ if } b \text{ is even} \\
g(t,a,b) & \text{ is } g(t \times a,a,b - 1) \text{ otherwise}
\end{align*}

35. (2 pts) Implement this recurrence as a C function. The return type of \( g \) is \textbf{Node *}: 

\begin{align*}
g(t,a,b) & \text{ is } b \text{ if } a \text{ is the empty list} \\
g(t,a,b) & \text{ is } a \text{ if } b \text{ is the empty list} \\
g(t,a,b) & \text{ is} \\
\quad & \text{ head(a) joined to } g(1,\text{tail(a)},b) \text{ if } t \text{ is 0} \\
g(t,a,b) & \text{ is} \\
\quad & \text{ head(b) joined to } g(0,a,\text{tail(b)}) \text{ otherwise}
\end{align*}
36. (3 pts) Implement this recurrence as two C functions. The return type of $f$, $g$, and $head$ is `int`; the return type of $tail$ is `Node *`:

```c
f(a) is g(head(a), tail(a))
g(b, c) is b if c is the empty list
  g(b, c) is
    g(head(c), tail(c)) if isGT(head(c), b)
g(b, c) is g(b, tail(c)) otherwise
```

37. (3 pts) Implement this recurrence as two C functions. The return type of $f$, $g$, and $head$ is `int`; the return type of $tail$ is `Node *`:

```c
f(a) is g(head(a), tail(a))
g(b, c) is b if c is the empty list
  g(b, c) is f(c) if isLT(head(c), b)
g(b, c) is g(b, tail(c)) otherwise
```