Problem 1 — Fibonacci Super Fast (Matrix Form + Master Theorem)

We use the matrix form:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

To compute the power, we use repeated squaring. Each step changes n to n/2.

The recurrence is:

$$T(n) = T(n/2) + 1$$

Using Master Theorem:

a = 1
b = 2

So:

$$T(n) = \Theta(\log_2 n)$$

This means Fibonacci can be computed in log time.

In [1]:
```python
def mat_mul(A, B):
    return [
        [
            A[0][0]*B[0][0] + A[0][1]*B[1][0],
            A[0][0]*B[0][1] + A[0][1]*B[1][1]
        ],
        [
            A[1][0]*B[0][0] + A[1][1]*B[1][0],
            A[1][0]*B[0][1] + A[1][1]*B[1][1]
        ]
    ]

def mat_pow(A, n):
    result = [[1,0],[0,1]]
    while n > 0:
        if n % 2 == 1:
            result = mat_mul(result, A)
        A = mat_mul(A, A)
        n = n // 2
    return result

def fib(n):
    if n == 0:
        return 0
    A = [[1,1],[1,0]]
    P = mat_pow(A, n-1)
    return P[0][0]
```

```
print("F(10) =", fib(10))
```

```
F(10) = 55
```

Problem 2 — Edit Distance

We compute edit distance between: SATURDAY and SUNDAY

We use DP table

Rules:

delete = +1
insert = +1
replace = +1
same letter = 0

In [3]:
```
A = "SATURDAY"
B = "SUNDAY"
m = len(A)
n = len(B)

dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(m+1):
    dp[i][0] = i

for j in range(n+1):
    dp[0][j] = j

for i in range(1, m+1):
    for j in range(1, n+1):
        if A[i-1] == B[j-1]:
            cost = 0
        else:
            cost = 1
        dp[i][j] = min(
            dp[i-1][j] + 1,
            dp[i][j-1] + 1,
            dp[i-1][j-1] + cost
        )

for row in dp:
    print(row)

print("Edit distance =", dp[m][n])
```

```
[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 1, 2, 3, 3, 4]
[3, 2, 2, 2, 3, 4, 4]
[4, 3, 2, 3, 3, 4, 5]
[5, 4, 3, 3, 4, 4, 5]
[6, 5, 4, 4, 3, 4, 5]
[7, 6, 5, 5, 4, 3, 4]
[8, 7, 6, 6, 5, 4, 3]
Edit distance = 3
```

The edit distance is 3.

Why not greedy

Greedy does not always give the best result.

Because choosing the best item first can block better combination later.

Dynamic programming checks all possibilities.

So DP is correct.

Solve example

In [5]:
```python
weights = [10,20,30]
values = [60,100,120]
W = 50
n = len(weights)

dp = [[0]*(W+1) for _ in range(n+1)]

for i in range(1, n+1):
    for w in range(W+1):
        dp[i][w] = dp[i-1][w]
        if w >= weights[i-1]:
            dp[i][w] = max(
                dp[i][w],
                dp[i-1][w-weights[i-1]] + values[i-1]
            )

print("Max value =", dp[n][W])
```

```
Max value = 220
```

Space optimization to O(W)

In [6]:
```python
dp = [0]*(W+1)
for i in range(n):
    for w in range(W, weights[i]-1, -1):
        dp[w] = max(
            dp[w],
            dp[w-weights[i]] + values[i]
        )
```

```
print("Max value =", dp[W])
```

```
Max value = 220
```

In [ ]: