问题一——斐波那契超快！

我们使用矩阵恒等式：

[

$$\begin{bmatrix} F_{n+1} \\ \backslash \mathbf{F}_{n} \end{bmatrix}$$

A^n

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

， \quad A=

$$\begin{bmatrix} 1 & 1 \\ \backslash 1 & 0 \end{bmatrix}$$

.]

另外要注意：

[A^2 = \bestart{bmatrix}2&1\\1&1\end{bmatrix} = B，] 因此对偶数（n）： （A^n = （A^2）^{n/2} = B^{n/2}）。

我们通过平方法计算矩阵幂，每一步将指数减半。

在[2]中：

```
def mat_mul_2x2(X, Y):
    # 2x2 matrix multiplication in O(1)
```

```
    return [
        [X[0][0]*Y[0][0] + X[0][1]*Y[1][0], X[0][0]*Y[0][1] + X[0][1]*Y[1][1]],
        [X[1][0]*Y[0][0] + X[1][1]*Y[1][0], X[1][0]*Y[0][1] + X[1][1]*Y[1][1]],
    ]

def mat_pow_2x2(M, n):
    # Exponentiation by squaring: O(log n)
    # Returns M^n for n >= 0
    I = [[1,0],[0,1]]
    result = I
    base = M
    exp = n

    while exp > 0:
        if exp & 1:
            result = mat_mul_2x2(result, base)
        base = mat_mul_2x2(base, base)
        exp >>= 1
    return result
```

在[3]中：

```
A = [[1,1],[1,0]]

def fib_matrix(n):
    """
    Returns F_n using:
    [F_{n+1}; F_n] = A^n [1;0]
    => F_n = (A^n)[1][0]
    """
    if n < 0:
        raise ValueError("n must be >= 0")
    if n == 0:
        return 0
    P = mat_pow_2x2(A, n)
    return P[1][0]
```

在[4]中：

```
B = [[2,1],[1,1]]

# verify A^2 == B
A2 = mat_mul_2x2(A, A)
print("A^2 =", A2, "  B =", B, "  Equal?", A2 == B)
```

```
def fib_via_B(n):
    """
    Uses decomposition with B = A^2.
    If n is even: A^n = B^(n/2)
    If n is odd:  A^n = A * B^((n-1)/2)
    Then F_n = (A^n)[1][0]
    """
    if n < 0:
        raise ValueError("n must be >= 0")
    if n == 0:
        return 0

    if n % 2 == 0:
        P = mat_pow_2x2(B, n//2)
    else:
        P = mat_mul_2x2(A, mat_pow_2x2(B, (n-1)//2))
    return P[1][0]
```

```
A^2 = [[2, 1], [1, 1]]   B = [[2, 1], [1, 1]]   Equal? True
```

在[5]中：

```
def fib_iter(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a+b
    return a

for n in range(0, 21):
    f1 = fib_matrix(n)
    f2 = fib_via_B(n)
    f3 = fib_iter(n)
    assert f1 == f3 == f2
print("All Fibonacci tests passed ✅")
```

```
All Fibonacci tests passed ✅
```

问题2 — 列文斯坦（编辑）距离（表格法）

计算编辑距离：

- **星期六**
- **周日**

作（每次花费1）：插入、删除、替换（同一字符成本为0）。

我们使用动态规划： 设 为将 的第一个字符转换为 的第一个字符的最小编辑。 `dp[i][j] i s j t`

在[9]中：

```
def levenshtein_dp(s, t):
    m, n = len(s), len(t)
    dp = [[0]*(n+1) for _ in range(m+1)]

    # init
    for i in range(m+1):
        dp[i][0] = i
    for j in range(n+1):
        dp[0][j] = j

    # fill
    for i in range(1, m+1):
        for j in range(1, n+1):
            cost = 0 if s[i-1] == t[j-1] else 1
            dp[i][j] = min(
                dp[i-1][j] + 1,      # delete
                dp[i][j-1] + 1,      # insert
                dp[i-1][j-1] + cost  # substitute / match
            )
    return dp

s = "SATURDAY"
t = "SUNDAY"
dp = levenshtein_dp(s, t)
dp[-1][-1]
```

Out[9]: 3

In [10]:

```
def print_dp_table(s, t, dp):
    # header
    header = ["Ø"] + list(t)
    print("     " + "  ".join(f"{c:>2}" for c in header))
```

```
    # rows
    for i in range(len(s)+1):
        row_label = "Ø" if i == 0 else s[i-1]
        row_vals = dp[i]
        print(f"{row_label:>2}  " + "  ".join(f"{v:>2}" for v in row_vals))

print_dp_table(s, t, dp)
```

```
    Ø  S  U  N  D  A  Y
Ø   0  1  2  3  4  5  6
S   1  0  1  2  3  4  5
A   2  1  1  2  3  3  4
T   3  2  2  2  3  4  4
U   4  3  2  3  3  4  5
R   5  4  3  3  4  4  5
D   6  5  4  4  3  4  5
A   7  6  5  5  4  3  4
Y   8  7  6  6  5  4  3
```

Result

The edit distance is the bottom-right cell of the table:

[\text{dist}(\text{"SATURDAY"},\text{"SUNDAY"}) = dp[8][6] = 3.]

So the minimum number of edits required is **3.**

Problem 3 — 0/1 Knapsack Algorithm

We are given items with:

- weight `w[i]`
- value and a knapsack capacity . `v[i] W`

Goal: maximize total value such that total weight ≤ W, and each item can be taken **at most once** (0/1).

Tasks:

1. Why is knapsack not greedy? Why dynamic programming?
2. Solve knapsack for the **course example.**
3. Can we reduce space complexity to **O(W)**?

(1) Why not greedy? Why dynamic programming?

A greedy strategy (e.g., pick highest value, or highest value/weight ratio first) does **not** always produce an optimal solution for **0/1** knapsack, because decisions are discrete: taking one item can prevent taking a better combination of other items.

0/1 knapsack has:

- **Optimal substructure**: optimal solution for capacity W depends on optimal solutions of smaller capacities.
- **Overlapping subproblems**: the same capacities are solved repeatedly.

Therefore, dynamic programming is appropriate.

In [13]:
```
weights = [6, 5, 4]
values  = [30, 26, 20]
W = 10

assert len(weights) == len(values), "weights and values must have same length"
n = len(weights)
n, W
```

Out[13]:
```
(3, 10)
```

In [14]:
```
def knapsack_2d(weights, values, W):
    n = len(weights)
    dp = [[0]*(W+1) for _ in range(n+1)]

    for i in range(1, n+1):
        wi, vi = weights[i-1], values[i-1]
        for cap in range(W+1):
            dp[i][cap] = dp[i-1][cap]              # not take
            if wi <= cap:
```

```
                dp[i][cap] = max(dp[i][cap], dp[i-1][cap-wi] + vi)  # take

    # backtrack chosen items
    chosen = []
    cap = W
    for i in range(n, 0, -1):
        if dp[i][cap] != dp[i-1][cap]:
            chosen.append(i-1)
            cap -= weights[i-1]
    chosen.reverse()
    return dp[n][W], chosen, dp

best_value, chosen_items, dp_table = knapsack_2d(weights, values, W)
best_value, chosen_items
```

Out[14]: (50, [0, 2])

In [16]:
```
def print_dp(dp):
    for i, row in enumerate(dp):
        print(f"i={i:>2}:", row)

print_dp(dp_table)
```

```
i= 0: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
i= 1: [0, 0, 0, 0, 0, 0, 30, 30, 30, 30, 30]
i= 2: [0, 0, 0, 0, 0, 26, 30, 30, 30, 30, 30]
i= 3: [0, 0, 0, 0, 20, 26, 30, 30, 30, 46, 50]
```

In [17]:
```
def knapsack_1d(weights, values, W):
    dp = [0]*(W+1)
    for wi, vi in zip(weights, values):
        for cap in range(W, wi-1, -1):  # backward to keep 0/1 constraint
            dp[cap] = max(dp[cap], dp[cap-wi] + vi)
    return dp[W]

best_value_1d = knapsack_1d(weights, values, W)
best_value_1d
```

Out[17]: 50

In [18]:
```
assert best_value_1d == best_value
print("2D and 1D results match ✅")
```

```
2D and 1D results match ✅
```

Interpretation (course example)

Capacity (W=10). The optimal solution reaches value **50** by choosing items **0 and 2**:

- item 0: weight 6, value 30
- item 2: weight 4, value 20
  Total weight (6+4=10), total value (30+20=50).

A greedy choice by value/weight ratio may pick item 1 first (26/5), then item 2, giving value (26+20=46), which is not optimal. Thus greedy fails for 0/1 knapsack, while DP finds the optimum.

Space optimization: the DP can be reduced from (O(nW)) to (O(W)) by using a 1D array and iterating capacity backwards.

In [ ]: