problem 1 - Fibonacci Super Fast

(1) Matrix formulation

Fibonacci can be computed using matrix exponentiation:

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

Let

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

To compute F_n, we compute A^n using exponentiation by squaring:

If n is even $A^n = (A^{n/2})^2$

If n is odd: $A^n = A \cdot A^{n-1}$

This reduces the number of multiplications dramatically.

(2) complexity using Master Theorem

We compute powers recursively: $T(n) = T(n/2) + O(1)$

Explanation:

We solve one subproblem of size $n/2$

Matrix multiplication is constant time (2x2 matrix)

Using Master Theorem:

$a = 1$

$b = 2$

$f(n) = O(1)$

$n^{\log_b a} = n^0 = 1$

So: $T(n) = O(\log n)$

Therefore Fibonacci via matrix exponentiation runs in $O(\log_2 n)$

Reason: each step halves n.

problem 2 – Edit Distance

strings:

SATURDAY

SUNDAY

Using day dynamic programming table, the minimum edit distance is:

3

Operations:

1. Remove A
2. Replace T with N
3. Remove R

so total edits = 3.

problem 3 – 0/1 knapsack

(1) why not greedy?

Greedy does not always give optimal solution because choosing the item with best value or ratio first may block better combinations.

knapsack requires considering combinations. so dynamic programming is used.

(2) knapsack recurrence

$$dp[i][w] = \begin{cases} dp[i-1][w], & \text{if item too heavy} \\ \max(dp[i-1][w],\ value_i + dp[i-1][w - weight_i]) \end{cases}$$

we fill the table row by row and choose the maximum value

(3) space optimization

we can reduce space to $O(w)$ by using one array and updating from right to left:

```
for each item:
    for w from W down to weight:
        dp[w] = max(dp[w], value + dp[w - weight])
```

Thus space complexity is: $O(W)$