Fundamental Algorithm Techniques

Problem Set #3

Aibek Zhazykbek

October 25, 2025

Problem 1: Fibonacci Super Fast!

Question

1. Compute Fibonacci numbers using the matrix relation:

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

2. Alternatively, use decomposition:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \left(\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \right)^{n/2} \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

3. Use the Master Theorem to discuss the complexity and explain why it is $O(\log_2 n)$.

Answer and Discussion

Fibonacci numbers can be expressed through matrix exponentiation:

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

The computation of M^n can be done using **binary exponentiation (exponentiation by squaring)**.

The recurrence relation for the time complexity is:

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

According to the Master Theorem:

$$T(n) = O(\log n).$$

Therefore, computing F_n by this method requires logarithmic time.

Python Implementation

```python
def mat_mult(A, B):
    return (
        A[0]*B[0] + A[1]*B[2],
        A[0]*B[1] + A[1]*B[3],
        A[2]*B[0] + A[3]*B[2],
        A[2]*B[1] + A[3]*B[3]
    )

def mat_pow(M, n):
    result = (1,0,0,1)
    base = M
    while n > 0:
        if n & 1:
            result = mat_mult(result, base)
        base = mat_mult(base, base)
        n >>= 1
    return result

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

Listing 1: Fibonacci using Matrix Exponentiation

Experimental Verification

Fibonacci Growth and Time Complexity

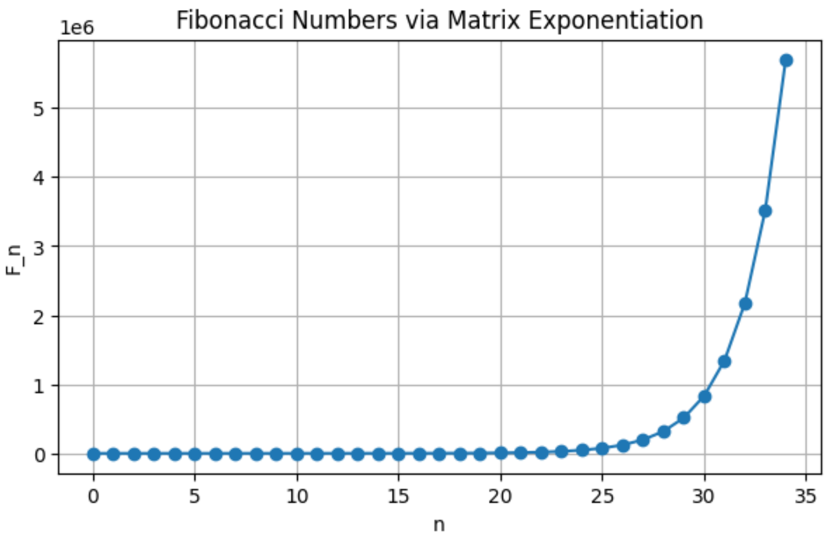

Figure 1: Fibonacci sequence values using matrix exponentiation.

Conclusion: Matrix exponentiation provides a dramatic improvement over naive recursion ($O(2^n)$), achieving only $O(\log n)$ time complexity.

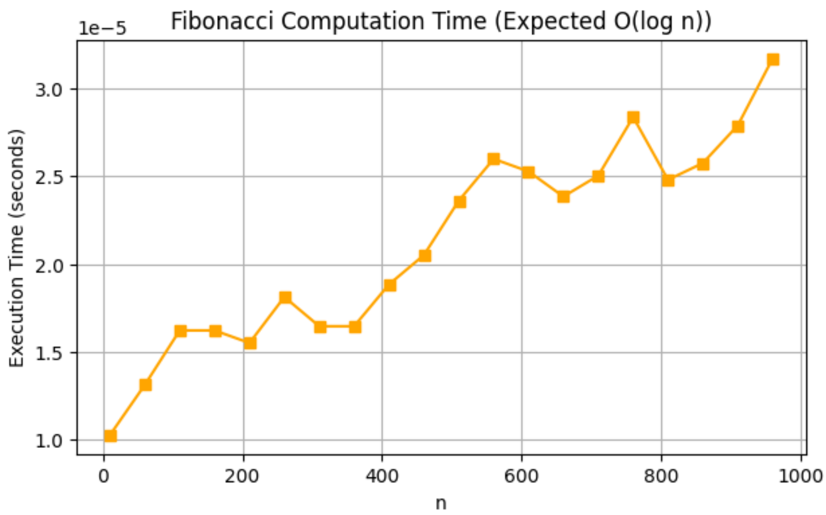

Figure 2: Execution time showing logarithmic growth ($O(\log n)$).

Problem 2: 0/1 Knapsack Algorithm

Questions

1. Why is Knapsack not a greedy algorithm, and why do we use dynamic programming?
2. Solve the Knapsack problem for the course example.
3. Can we reduce the space complexity to $O(W)$?

Answers

- **Not greedy:** A greedy approach (picking the best value/weight ratio) fails in cases where smaller, lower-ratio items together produce a better result.
- **Dynamic Programming:** Knapsack has optimal substructure and overlapping subproblems, so DP ensures optimal results.
- **Space optimization:** By reusing a single 1D DP array, space can be reduced to $O(W)$ while maintaining the same time complexity $O(nW)$.

Mathematical Model

$$dp[i][w] = \begin{cases} dp[i-1][w], & w_i > w, \\ \max(dp[i-1][w], dp[i-1][w-w_i] + v_i), & \text{otherwise.} \end{cases}$$

Python Implementation

```python
def knapsack_dp(values, weights, W):
    n = len(values)
    dp = [[0]*(W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(W+1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w],
                               dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    w = W
    chosen = []
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i-1][w]:
            chosen.append(i-1)
            w -= weights[i-1]
    chosen.reverse()
    return dp[n][W], chosen
```

Listing 2: 0/1 Knapsack Algorithm in Python

Results and Visualization

Example input:

$$\text{Values: } [60, 100, 120], \quad \text{Weights: } [10, 20, 30], \quad W = 50.$$

Output:

$$\text{Max Value} = 220, \quad \text{Chosen Items} = [1, 2].$$

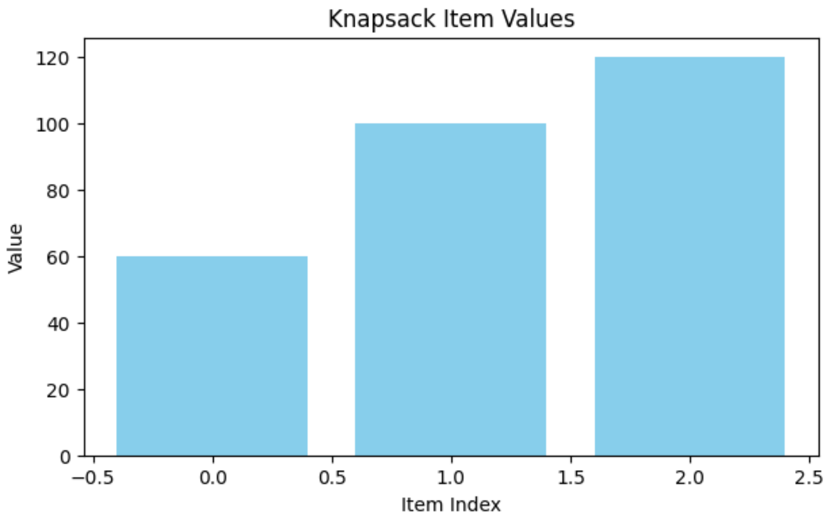

Figure 3: Item value distribution for Knapsack example.

Complexity:

$$O(nW) \text{ time, } O(W) \text{ space (optimized).}$$

Problem 3: Neuro Computing

Questions

1. Generate 100 random binary vectors of length N.
2. Define and analyze similarity functions:
$$sim(x, y) = \frac{x \cdot y}{\|x\|_1 \|y\|_1}, \quad Jacc(x, y) = \frac{|x \cap y|}{|x \cup y|}.$$
3. Observe Gaussian-like distributions and explain why.
4. For sparse binary vectors ($N = 2000$, $w = 5$), how many possible vectors exist?
5. Define a notion of capacity for these vectors.

Answers and Discussion

For random binary vectors, both similarity measures follow approximately Gaussian distributions due to the **Central Limit Theorem**. As N increases:

- The variance decreases (distributions become narrower);
- Mean of $sim(x, y)$ approaches 0 ($O(1/N)$ scaling);
- Jaccard mean stabilizes near $\approx 1/3$ for $p = 0.5$.

Python Code for Similarity Experiment

```python
def generate_binary_vectors(num_vectors, N, p=0.5):
    return (np.random.rand(num_vectors, N) < p).astype(int)

def sim_func(x, y):
    return np.dot(x, y) / (np.sum(x)*np.sum(y) + 1e-10)

def jaccard(x, y):
    inter = np.sum(np.logical_and(x, y))
    union = np.sum(np.logical_or(x, y))
    return inter / (union + 1e-10)
```

Listing 3: Neuro Computing Simulation

Experimental Graphs

Sparse Vector Capacity

The number of possible sparse binary vectors:
$$\binom{2000}{5} = 2.65 \times 10^{14}.$$

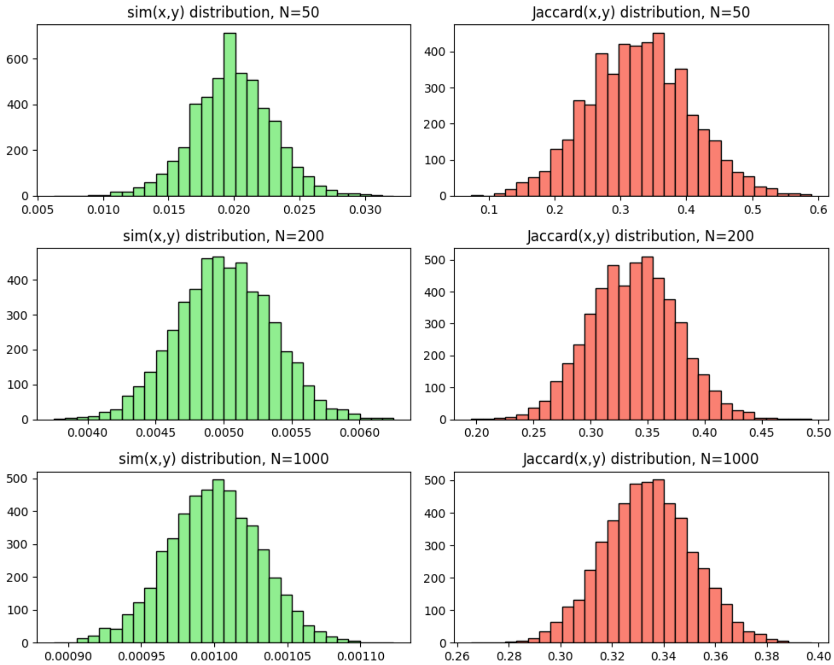

Figure 4: Distribution of sim(x,y) and Jaccard(x,y) for $N = 50, 200, 1000$.

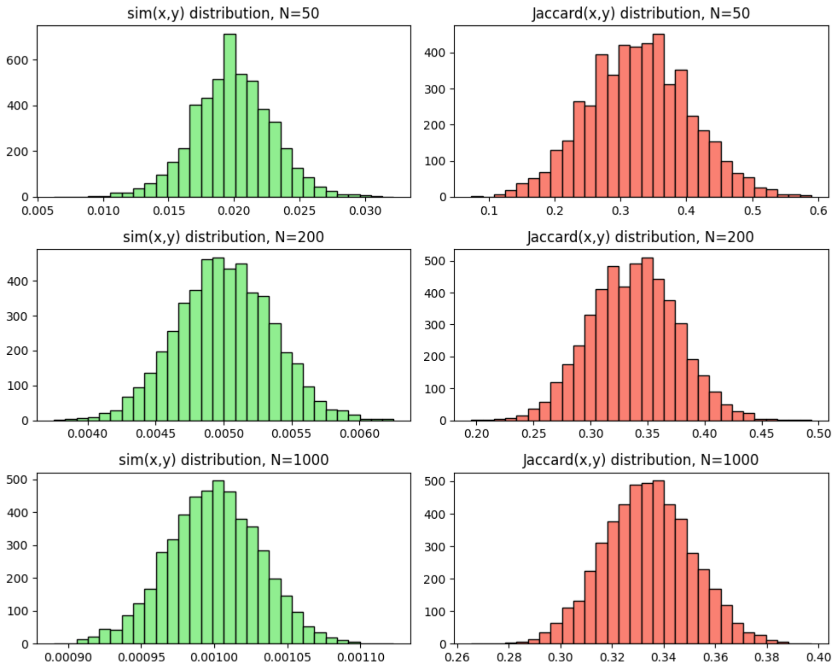

Figure 5: Sparse binary vector similarity (N=2000, w=5).

Information capacity (entropy):

$$H = \log_2 \binom{2000}{5} \approx 48.1 \text{ bits.}$$

Thus, each sparse vector carries ≈ 48 bits of unique information.

Summary of Experimental Results

N	sim_mean	sim_std	jacc_mean	jacc_std
50	0.020	0.003	0.331	0.078
200	0.005	0.0004	0.337	0.040
1000	0.000	0.00005	0.333	0.017

Final Conclusion

- Matrix exponentiation reduces Fibonacci computation to $O(\log n)$.
- Dynamic programming ensures optimal 0/1 Knapsack solutions in $O(nW)$ time.
- Binary vector similarities follow Gaussian distributions for large N.
- Sparse high-dimensional vectors have huge combinatorial capacity and are ideal for neural associative systems.

Thanks for your attention Mr.Jair !