

Problem 1

1. Matrix form of Fibonacci The Fibonacci numbers satisfy:

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

So if we denote $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ then $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

2. Alternative decomposition The assignment mentions a decomposition using

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}^{n/2}.$$

This is another matrix related to Fibonacci-like recurrences. The main idea for fast Fibonacci is to compute powers of a 2x2 matrix quickly using exponentiation by squaring.

Time complexity using divide-and-conquer To compute A^n by exponentiation by squaring we use:

$$A^n = \begin{cases} A \cdot (A^{(n-1)/2})^2 & \text{if } n \text{ is odd,} \\ (A^{n/2})^2 & \text{if } n \text{ is even.} \end{cases}$$

This leads to the recurrence for running time (counting matrix multiplications):

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

Here $a = 1$, $b = 2$, and $f(n) = O(1)$. By the Master Theorem (or by simple recurrence solving) this gives

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

Why? Because each recursive call halves the exponent n , and we do one (constant-cost) matrix multiplication per level of recursion. The recursion depth is $O(\log n)$, so total cost is $O(\log n)$. Since matrices are 2×2 , each multiplication is constant time in standard RAM model (or $O(1)$ arithmetic ops), so overall we get $O(\log n)$ arithmetic operations.

(This code is simple to understand; more efficient versions avoid recursion or use iterative exponentiation.)

Problem 2

1. Why knapsack is not greedy Greedy algorithms pick the locally best choice hoping to be globally optimal. For 0/1 Knapsack, the common greedy idea is to pick items with highest value-to-weight ratio. But this fails for 0/1 Knapsack because items are indivisible. A counterexample:

- Capacity $W = 50$.
- Item A: weight 49, value 100 (ratio ≈ 2.04).
- Item B: weight 25, value 60 (ratio 2.4).
- Item C: weight 25, value 60 (ratio 2.4).

Greedy picks B and then cannot pick C (capacity left 25, actually it can pick C; adjust example: use weight 26 for A so greedy picks B then C, but capacity 50 can't take A; better classic examples exist). The key point: greedy can fail; we need dynamic programming to guarantee optimality for 0/1 Knapsack.

2. Dynamic programming solution (example) Let there be n items with weights w_i and values v_i . Capacity is W . Define DP table $dp[i][c] = \max$ value using first i items with capacity c . Recurrence:

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

Base: $dp[0][c] = 0$ for all c .

Small course-like example: Items:

- Item 1: weight 3, value 6
- Item 2: weight 4, value 8
- Item 3: weight 2, value 4

Capacity $W = 7$.

We build table $dp[0..3][0..7]$:

$i \setminus c$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	6	6	6	6	6
2	0	0	0	6	8	8	8	14
3	0	0	4	6	8	10	12	14

Final answer $dp[3][7] = 14$, achieved by picking item 2 ($w=4, v=8$) and item 3 ($w=2, v=4$) and item 1 can't fit together with them; result $8+4+?$ actually $8+6$ is 14 if items 1 and 2 ($3+4=7$) give $6+8=14$ as well. So optimal is 14.

3. Space optimization to $O(W)$ We can store only one array of size $W + 1$. The trick is to iterate items in outer loop and capacities in decreasing order (from W down to w_i). This prevents reuse of the same item twice.

Complexity summary

- Time: $O(nW)$ (where n is number of items).
- Space: can be $O(W)$ with the optimized 1D DP.

Problem 3

1. Generate 100 random binary vectors of length N A random binary vector of length N can be generated by sampling Bernoulli(0.5) for each coordinate. Below is Python code to generate 100 such vectors.

2. Similarity functions Two similarity measures:

$$\text{sim}(x, y) = \frac{x \cdot y}{\|x\|_1 \|y\|_1} = \frac{\sum_{i=1}^N x_i y_i}{(\sum_{i=1}^N x_i)(\sum_{i=1}^N y_i)}.$$

This is like normalized overlap: number of common 1s divided by product of counts (not symmetric scaling like cosine, but a simple normalized dot).

Jaccard:

$$\text{Jacc}(x, y) = \frac{\sum_{i=1}^N x_i y_i}{\sum_{i=1}^N \max(x_i, y_i)} = \frac{|x \cap y|}{|x \cup y|}.$$

Why similarities look Gaussian If each coordinate is an independent Bernoulli random variable (probability p of being 1), then the number of common ones between two vectors X, Y is:

$$S = \sum_{i=1}^N X_i Y_i.$$

Each $X_i Y_i$ is Bernoulli with probability p^2 (if independent), so S is binomial $\text{Bin}(N, p^2)$. For large N , the binomial distribution is approximately Gaussian by the Central Limit Theorem (CLT): mean Np^2 , variance $Np^2(1 - p^2)$. Normalizing these overlaps (as similarity measures) often makes their empirical distribution look like a Gaussian. This is why when you compute pairwise similarities for many random vectors the histogram often looks bell-shaped.

3. Effect of larger N When N increases:

- The mean of the overlap S grows linearly with N : $\mathbb{E}[S] = Np^2$.
- The standard deviation grows like \sqrt{N} : $\sigma = \sqrt{Np^2(1 - p^2)}$.

- After normalizing (e.g., by mean or by N), the distribution becomes more concentrated (by law of large numbers). So relative fluctuations shrink (coefficient of variation decreases like $1/\sqrt{N}$).
- By CLT, the shape becomes more Gaussian as N grows.

So larger N makes the empirical similarity distribution more tightly peaked and more normal-looking.

4. Huge sparse binary vectors: $N = 2000, w = 5$ If vectors have exactly $w = 5$ ones (and the rest zeros), the number of possible distinct vectors is simply the number of ways to choose 5 positions from 2000:

$$\binom{2000}{5}.$$

This is a very large number. Numerically,

$$\binom{2000}{5} = \frac{2000 \cdot 1999 \cdot 1998 \cdot 1997 \cdot 1996}{5!},$$

which is on the order of $\approx 2.6 \times 10^{15}$ (you can compute exact value with Python's `math.comb`).

5. Notion of capacity for these vectors A few intuitive ways to think about capacity:

- **Counting capacity:** The total number of distinct codewords is $\binom{N}{w}$. This is the maximum number of different signals you can represent if you insist on exactly w ones.
- **Collision probability (random pick):** If you pick m random vectors with w ones uniformly, the probability that two of them are equal can be approximated by the birthday paradox: collisions become likely when $m \approx \sqrt{\binom{N}{w}}$.
- **Separation / Hamming distance:** For robust storage or retrieval, you'd like vectors to be well-separated in Hamming distance. Capacity under a minimum Hamming distance constraint can be studied by sphere-packing bounds (like in coding theory). For sparse vectors, minimum overlap means higher separability.
- **Practical capacity:** For an application (e.g., associative memory or hashing), capacity often means how many different patterns can reliably be stored and distinguished given noise. This depends on overlap statistics, acceptable error rates and the retrieval algorithm.