exercise 3.

Problem 1 : (Fibonacci Super Fast !) 1. compute Fibonacci with the 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. This can also be expressed as:

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

Use Master Theorem to discuss the complexity of this decomposition and show, explain why time complexity is $\log_2(n)$

Solve : Fibonacci recurrence: $F(0)=0$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$

Matrix relation.

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

test: $n=1$: $M^1(1,0)^T = (1,1)^T \Rightarrow F_2 = 1, F_1 = 1$

$n=3$: $M^3 = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$, $M^3(1,0)^T = (3,2)^T \Rightarrow F_4 = 3, F_3 = 2$

Instead of multiplying M repeatedly n times $O(n)$, we compute M^n by repeated squaring, reducing the exponent by half each step.

$\Rightarrow$ To compute M^n efficiently, we use repeated squaring so the exponent is halved each recursion.

At each step, solve one subproblem of size $\frac{n}{2}$ and do one constant - time 2x2 matrix multiplication :

$$T(n) = T(\tfrac{n}{2}) + O(1)$$

Master Theorem parameters

$a=1$ one subproblem

$b=2$ size becomes $\frac{n}{2}$

$f(n) = O(1)$ constant work.

compute : $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Since $f(n) = O(1) = O(n^{\log_b a})$, this is Case 2, so

$T(n) = O(\log n) = O(\log_2 n)$

Having n until it becomes 1 takes $\log_2 n$ steps. Each step costs $O(1)$.

Matrix fast exponentiation computes $F(n)$ in $O(\log_2 n)$ time.

Problem 2. Levenshtein (Edit) Distance

Three operations. ① Insert ② Remove ③ Replace.

Source : s1 = "SATURDAY" (length 8).

Target : s2 = "SUNDAY" (length 6).

Base cases:

dp[0][j] = j (transform empty string to first j chars of SATURDAY → j insertions)

dp[i][0] = i (transform first i chars of SUNDAY to empty string → i deletions)

Recurrence

If s2[i] == s1[j] (last characters are same)
  dp[i][j] = dp[i-1][j-1]
if s2[i] ≠ s1[j]:
  dp[i][j] = 1 + min(
    dp[i][j-1], ← insert (move left in table)
    dp[i-1][j], ← remove (move up in table
    dp[i-1][j-1] ← replace (move diagonal)
)

row by row, left to right.

For each cell dp[i][j] : if characters match → copy diagonal (dp[i-1][j-1]); if not → min of three neighbors + 1.

Row "S" (i=1) Row "U" (i=2) ... Row "Y" (i=6)

DP table.

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

Operation 1 – Delete "A" : SATURDAY → STURDAY
2 – Delete "T" : STURDAY → SURDAY
3 → Replace "R" with "N" : SURDAY → SUNDAY

Edit Distance ("SATURDAY", "SUNDAY") = 3
3 operations : Delete "A", Delete "T", Replace "R" → "N"
Time complexity : O(m·n) = O(48)
space : O(m·n)

calculate :
Row "S" (i=1) :
dp[1][1] : S vs S → match → dp[0][0] = 0
dp[1][2] = S vs A → no match → min (dp[0,2], dp[1][1], dp[0][1]) + 1
= min (2, 0, 1) + 1 = 1
dp[1][3] = S vs T → min (dp[0][3], dp[1][2], dp[0][2]) + 1
= min (3, 1, 2) + 1 = 2
dp[1][4] : S vs U → min (3, 2, 3) + 1 = 3
dp[1][5] : S vs R → min (4, 3, 4) + 1 = 4
dp[1][6] : S vs D → min (5, 4, 5) + 1 = 5
dp[1][7] : S vs A → min (6, 5, 6) + 1 = 6
dp[1][8] : S vs Y → min (7, 6, 7) + 1 = 7
Row "U" (i=2) :
dp[2][1] : U vs S → min (dp[1][1], dp[2][0], dp[1][0]) + 1
= min (0, 2, 1) + 1 = 1
dp[2][2] = U vs A → min (1, 1, 0) + 1 = 1
dp[2][3] : U vs T → min (1, 1, 1) + 1 = 2
dp[2][4] : U vs U → match → dp[1][3] = 2
dp[2][5] : U vs R → min (dp[1][5], dp[2][4], dp[1][4]) + 1
= min (4, 2, 3) + 1 = 3
dp[2][6] = U vs D → min (5, 3, 4) + 1 = 4
dp[2][7] : U vs A → min (6, 4, 5) + 1 = 5
dp[2][8] = U vs Y → min (7, 5, 6) + 1 = 6

Row "N" (i=3):

dp[3][1]: N vs S → min(1,3,1)+1=2
dp[3][2] : N vs A → min (1,2,1)+1=2
dp[3][3] : N vs T → min (2,2,1)+1=2
dp[3][4] : N vs U → min (2,2,2)+1=3
dp[3][5] : N vs R → min (3,3,2)+1=3
dp[3][6] : N vs D → min (4,3,3)+1 = 4
dp[3][7] : N vs A → min (5,4,4)+1=5
dp[3][8] : N vs Y → min (6,5,5)+1=6

Row "D" (i=4):

dp[4][1] : D vs S → min (2,4,2)+1=3
dp [4][2] : D vs A → min (2,3,2)+1 =3
dp [4][3] : D vs T → min (2,3,2)+1 = 3
dp [4][4] : D vs U → min (3,3,2)+1=3
dp [4][5] : D vs R → min (3,3,3)+1=4
dp[4][6] : D vs D → match ! → dp[3][5] = 3.
dp [4][7] : D vs A → min (dp[3][7], dp[4][6], dp[3][6])+1
= min (5,3,4)+1=4
dp[4][8] : D vs Y → min (6,4,5)+1=5.

Row "A" (i=5):

dp [5][1] : A vs S. → min(3,5,3)+1=4
dp [5][2] : A vs A → match → dp[4][1]=3
dp [5][3] : A vs T → min (3,3,3)+1=4
dp[5][4] : A vs U → min (3,4,3)+1 = 4
dp [5][5] : A vs R → min (4,4,3)+1=4
dp[5][6] : A vs D → min (3,4,4)+1=4
dp [5][7] : A vs A → match → dp[4][6] =3
dp [5][8] : A vs Y → min (dp[4][8], dp[5][7], dp[4][7])+1
= min (5,3,4)+1=4

Row "Y" (i=6):

dp[6][1] : Y vs S ⇒ min(4, 6, 4) + 1 = 5

dp[6][2] : Y vs A → min(3, 5, 4) + 1 = 4

dp[6][3] : Y vs T → min(4, 4, 3) + 1 = 4

dp[6][4] : Y vs U → min(4, 4, 4) + 1 = 5

dp[6][5] : Y vs R → min(4, 5, 4) + 1 = 5

dp[6][6] : Y vs D → min(4, 5, 4) + 1 = 5

dp[6][7] : Y vs A → min(3, 5, 4) + 1 = 4

dp[6][8] : Y vs Y → match → dp[5][7] = 3

Problem3. (0/1 knapsack Algorithm!)

1. Why is knapsack not greedy Algo, why dynamical programming?

key difference: ① Fractional knapsack allows taking fractions
→ greedy works.

② 0/1 knapsack forbids fractions
→ greedy can fail.

Counterexample (fail) (∴ greedy does NOT always work for 0/1 knapsack)

capacity $W=10$.

Item	W	V	V/W
A	6	7	1.17
B	5	5	1.00
C	5	5	1.00

Greedy picks A → remaining 4 → cannot pick B or C
→ value: 7.

Optimal: picks B+C → weight 10 → value 10.

Greedy gives 7, but optimal is 10, Greedy fails!

DP is suitable because 0/1 knapsack problem has two properties.

① Optimal substructure: The optimal solution for capacity W can be built from optimal solutions to smaller capacities. If item i is in the optimal solution, then the remaining items form an optimal solution for capacity $W-w_i$. (Slide 7, 8)

②. Overlapping subproblems: A naive recursive solution recomputes the same subproblems many times. DP stores results in a table to avoid redundant work.

overlapping subproblems ⇔ memoization/tabulation

Dynamical Programming ⇔ optimal substructures.

2. Solve the knapsack Algorithm for the course example.

Define the problem eg: has capacity $W=8$ with 4 items.

item(i)	weight(w_i)	value(v_i)
1	2	3
2	3	4
3	4	5
4	5	6

maximize total value without exceeding weight capacity 8.

set up DP recurrence

Let $ks[i][w]$ = the maximum value achievable using the first i items with capacity w.

Base case: $ks[0][w] = 0$ for all w (no items → no value)

$ks[i][w] = ks[i-1][w]$, if $w_i > w$ (item too heavy)

$ks[i][w] = \max(ks[i-1][w], ks[i-1][w-w_i] + v_i)$, if $w_i \le w$

Fill the table row by row.

Row 0 (no items): all zeros.

Row 1 (item 1: $w_1=2, v_1=3$)

Row 2 (item 2: $w_2=3, v_2=4$)

Row 3 (item 3: $w_3=4, v_3=5$)

Row 4 (item 4: $w_4=5, v_4=6$)

DP table

i\w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

Backtrack to find selected items.

$ks[4][8] = 10 \neq ks[3][8] = 9 \rightarrow$ value changed
$\rightarrow$ item 4 Taken (remaining : $w = 8-5 = 3$)

$ks[3][3] = 4 = ks[2][3] = 4 \rightarrow$ same value
$\rightarrow$ item 3 NOT taken

$ks[2][3] = 4 \neq ks[1][3] = 3 \rightarrow$ value changed.
$\rightarrow$ item 2 Taken (remaining : $w = 3-3 = 0$)
$w = 0$ done.

So Optimal value : $ks[4][8] = 10$.
selected item : Item 2 ($w=3, v=4$) + Item 4 ($w=5, v=6$)

Total weight : $3+5 = 8 \leq 8$
Total value : $4+6 = 10$.

Time complexity : $O(nW)$
In this example. $O(4 \times 8) = O(32)$

Space complexity : $O(nW)$
In this example $5 \times 9 = 45$ cells.

Row 1 (item 1: $W_1=2$, $V_1=3$):

$w=0$: capacity 0, can't take anything $\Rightarrow 0$

$w=1$: $w_1=2>1$, item too heavy $\Rightarrow$ ks[0][1] = 0

$w=2$: $w_1=2\le 2 \Rightarrow$ max(ks[0][2], ks[0][0] + 3) = max(0,3) = 3

$w=3$: max(ks[0][3], ks[0][1] + 3) = max(0,3) = 3

$w=4$: max(0, ks[0][2] + 3) = max(0,3) = 3

$w=5$ to $w=8$: same logic $\Rightarrow$ all 3

Row 2 (item 2: $W_2=3$, $V_2=4$):

$w=0,1$: $w_2=3>w$, too heavy $\Rightarrow$ copy from row 1: 0, 0

$w=2$: $w_2=3>2$, too heavy $\Rightarrow$ ks[1][2] = 3

$w=3$: $w_2=3\le 3 \Rightarrow$ max(ks[1][3], ks[1][0] + 4) = max(3,4) = 4

$w=4$: max(ks[1][4], ks[1][1] + 4) = max(3,4) = 4

$w=5$: max(ks[1][5], ks[1][2] + 4) = max(3, 3+4) = max(3,7) = 7

$w=6$: max(ks[1][6], ks[1][3] + 4) = max(3, 7) = 7

$w=7$: max(ks[1][7], ks[1][4] + 4) = max(3,7) = 7

$w=8$: max(ks[1][8], ks[1][5] + 4) = max(3,7) = 7

Row 3 (item 3: $W_3=4$, $V_3=5$):

$w=0,1,2,3$: $W_4=4>w$, too heavy $\Rightarrow$ copy: 0, 0, 3, 4

$w=4$: max(ks[2][4], ks[2][0] + 5) = max(4,5) = 5

$w=5$: max(ks[2][5], ks[2][1] + 5) = max(7, 5) = 7

$w=6$: max(ks[2][6], ks[2][2] + 5) = max(7, 3+5) = max(7,8) = 8

$w=7$: max(ks[2][7], ks[2][3] + 5) = max(7, 4+5) = max(7,9) = 9

$w=8$: max(ks[2][8], ks[2][4] + 5) = max(7, 4+5) = max(7,9) = 9

Row 4 (item 4: $W_4=5$, $V_4=6$):

$w=0,1,2,3,4$: $w_4=5>w$, too heavy $\Rightarrow$ copy: 0, 0, 3, 4, 5

$w=5$: max(ks[3][5], ks[3][0] + 6) = max(7,6) = 7

$w=6$: max(ks[3][6], ks[3][1] + 6) = max(8, 6) = 8

$w=7$: max(ks[3][7], ks[3][2] + 6) = max(9, 3+6) = max(9,9) = 9

$w=8$: max(ks[3][8], ks[3][3] + 6) = max(9, 4+6) = max(9,10) = 10

3. Can you get space complexity to $O(W)$?

Yes. $dp[i][w]$ depends only on row $i-1$, so we can use a 1D array $dp[w]$.

update w from W down to w_i (right-to-left) to avoid using item i more than once.

Initialize $dp[0..W] = 0$
For each item i:
  For $w = W$ down to w_i:
    $dp[w] = max(dp[w], dp[w-w_i] + v_i)$

So. Time Complexity $O(nW)$
Space Complexity $O(W)$

Trade-off: 1D DP usually cannot directly backtrack chosen item without extra tracking.

	Time	Space
Optimized 1D DP.	$O(nW)$	$O(W)$
standard 2D DP	$O(nW)$	$O(nW)$

Time complexity stays the same
space drops from $O(nW)$ to $O(W)$ since we only keep one row.