DTS203TC
Design and Analysis of Algorithms

Lecture 10: Dynamic Programming

Dr. Qi Chen
School of AI and Advanced Computing

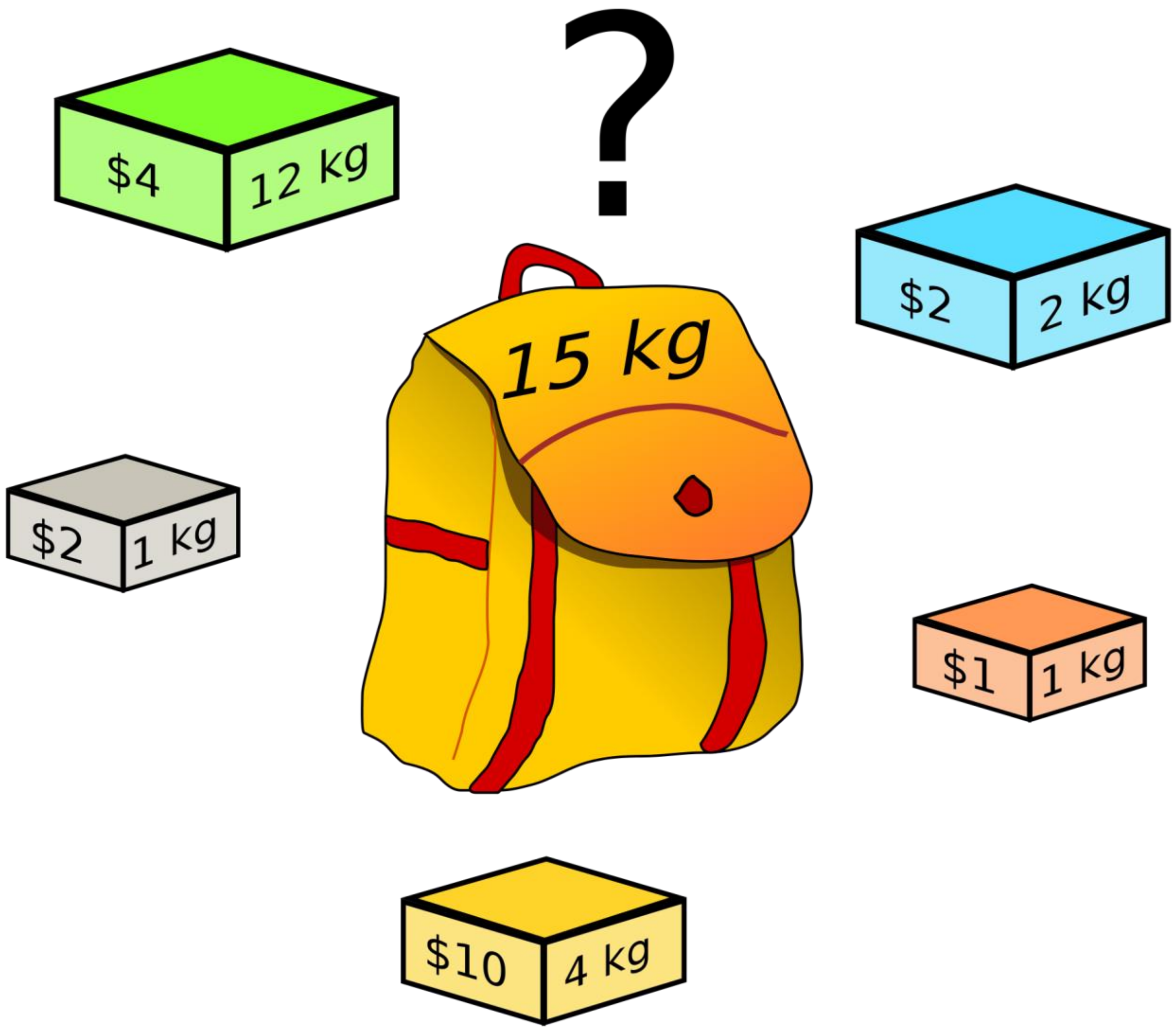
Learning outcome

- Able to apply dynamic programming to solve the 0-1 knapsack problem
- Able to apply dynamic programming to solve the longest common subsequence problem
- Able to implement dynamic programming with Python

How to apply Dynamic Programming?

- Step 1: Identify sub-problems and optimal substructure.
- Step 2: Find a recursive formulation
- Step 3: Use dynamic programming (typically in a bottom-up fashion) to compute the value of an optimal solution
- Step 4: If needed, keep track of some additional info to get the optimal solution

0-1 Knapsack Problem

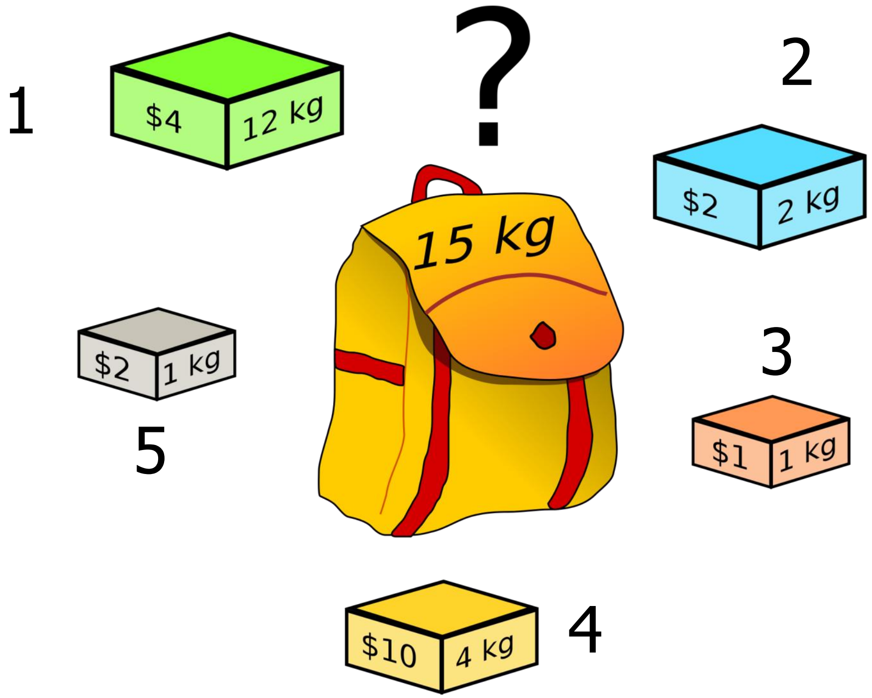

$4
12 kg
?
15 kg
$2
2 kg
$2
1 kg
$1
1 kg
$10
4 kg

The 0-1 Knapsack Problem

- Given: A set of n items, with each item i having
  - w_i – a positive weight
  - v_i – a positive benefit value
- Goal: Choose items with maximum total value but with weight at most W.
- If we are not allowed to take fractional amounts, then this is the 0-1 knapsack problem.

The 0-1 Knapsack Problem

- Given: A set of n items, with each item i having
  - w_i – a positive weight
  - v_i – a positive benefit value
- Goal: Choose items with maximum total value but with weight at most W.



W = 15

Item (i)	1	2	3	4	5
Weight (w)	12	2	1	4	1
Value (v)	4	2	1	10	2

Recursion by Brute-Force algorithm

- Approach:
  - consider all subsets of items and calculate the total weight and value of all subsets.
  - Consider the only subsets whose total weight is smaller than W.
  - From all such subsets, pick the maximum value subset.

- Time complexity?
  - $O(2^n)$

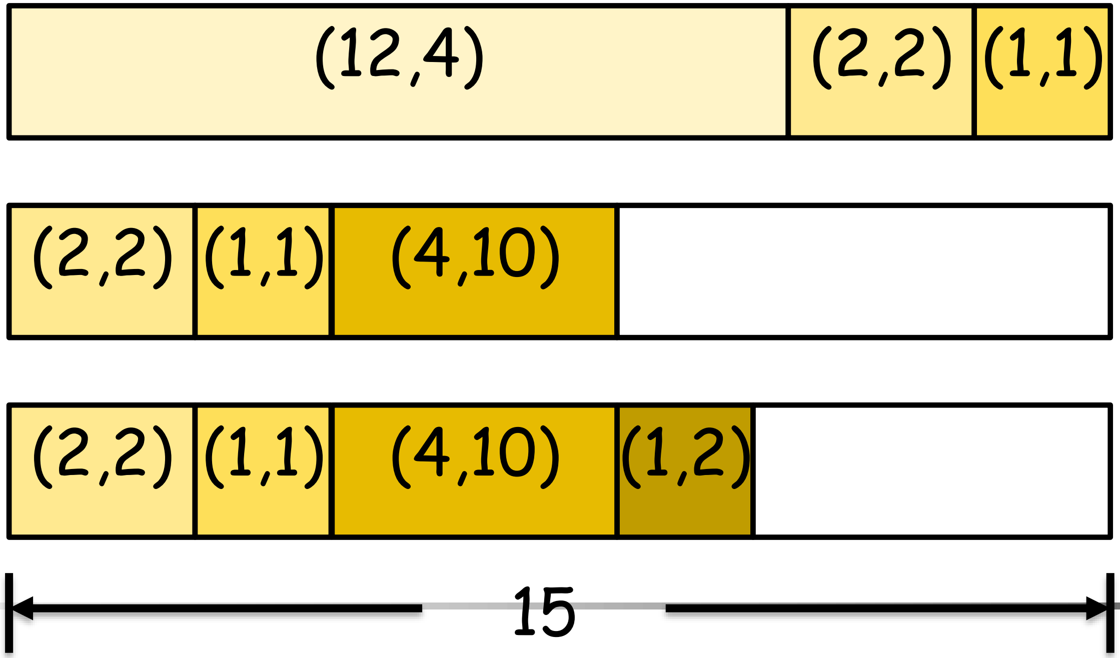
Sub-problems

- We don't need to try all possible choices.

- We can make use of dynamic programming:
  - Case 1: We don't select item i. We select the best of $\{1,2,3,\ldots,i-1\}$ using weight limit w
  - Case 2: We select item i. We select the best of $\{1,2,3,\ldots,i-1\}$ using the new weight limit: $w-w_i$

Solving the sub-problems

- S_k: Set of items numbered 1 to k
- Define f[i,w] = max profit subset of items {1, …, i} with weight limit w
- Consider set S={(12,4),(2,2),(1,1),(4,10),(1,2)} of (weight, value) pairs and total weight W=15.



Max profit for S_3 — f[3,15]=7

Max profit for S_4 — f[4,15]=13

Max profit for S_5 — f[5,15]=15

Solving the subproblem

- S_k: Set of items numbered 1 to k
- Define f[i,w] = max profit subset of items {1, …, i} with weight limit w

Case 1: We don't select item i. We select the best of {1,2,3,…,i-1} using weight limit w

Case 2: We select item i. We select the best of {1,2,3,…,i-1} using the new weight limit: $w-w_i$

$$
f[i, w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ f[i-1, w] & \text{else if } w_i>w \\ \max(f[i-1, w], v_i + f[i-1, w- w_i]) & \text{otherwise} \end{cases}
$$

Knapsack problem

Item (i)	1	2	3	4	5
Weight (w)	12	2	1	4	1
Value (v)	4	2	1	10	2

- Fill up the table.

$$f[i, w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ f[i-1, w] & \text{else if } w_i > w \\ \max(f[i-1, w], v_i + f[i-1, w-w_i]) & \text{otherwise} \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_0 ={}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S_1 ={1}	0															
S_2 ={1,2}	0															
S_3 ={1,2,3}	0															
S_4 ={1,2,3,4}	0															
S_5 ={1,2,3,4,5}	0															

Knapsack problem

Item (i)	1	2	3	4	5
Weight (w)	12	2	1	4	1
Value (v)	4	2	1	10	2

- Fill up the table.

$$
f[i, w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ f[i-1, w] & \text{else if } w_i > w \\ \max(f[i-1, w], v_i + f[i-1, w-w_i]) & \text{otherwise} \end{cases}
$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_0 ={}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S_1 ={1}	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
S_2 ={1,2}	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
S_3 ={1,2,3}	0	1	2	3	3	3	3	3	3	3	3	3	4	5	6	7
S_4 ={1,2,3,4}	0	1	2	3	10	11	12	13	13	13	13	13	13	13	13	13
S_5 ={1,2,3,4,5}	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Item: {2,3,4,5}　　　Max value: 15

Dynamic Programming Implementation

```
def knapsack(W, weight, value, n):
    f = [[None]*(W+1) for x in range(n + 1)]

    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                f[i][w] = 0
            elif weight[i-1] <= w:
                f[i][w] = max(value[i-1]
                          + f[i-1][w-weight[i-1]],
                            f[i-1][w])
            else:
                f[i][w] = f[i-1][w]

    return f[n][W]
```

```
weight = [12, 2, 1, 4, 1]
value = [4, 2, 1, 10, 2]
W = 15
n = len(weight)
print(knapsack(W, weight, value, n))
```

```
15
```

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Knapsack Problem: Running Time

- Running time: $O(nW)$
  - Not polynomial in input size! (W may be large)
  - "Pseudo-polynomial"
  - Decision version of Knapsack is NP-complete. [Chapter 34]

Longest Common Subsequence

Longest Common Subsequence (LCS)

- How similar are these two DNA sequences?

1) AGCCCTAAGGGCTACCTAGCTT
2) GACAGCCTACAAGCGTTAGCTTG

- Pretty similar, has a long common subsequence:
  1) AGCCCTAAGGGCTACCTAGCTT
  2) GACAGCCTACAAGCGTTAGCTTG

AGCCTAAGCTTAGCTT

Longest Common Subsequence (LCS)

- Subsequence:
  - BDFH is a subsequence of ABCDEFGH
- If X and Y are sequences, a common subsequence is a sequence which is a subsequence of both.
  - BDFH is a common subsequence of ABCDEFGH and of ABDFGHI
- A longest common subsequence…
  - is a common subsequence that is longest.
  - The longest common subsequence of ABCDEFGH and ABDFGHI is ABDFGH.
- Has applications to DNA similarity testing.

Sub-problems

- Prefixes:

X	A	C	G	G	T		
Y	A	C	G	C	T	T	A

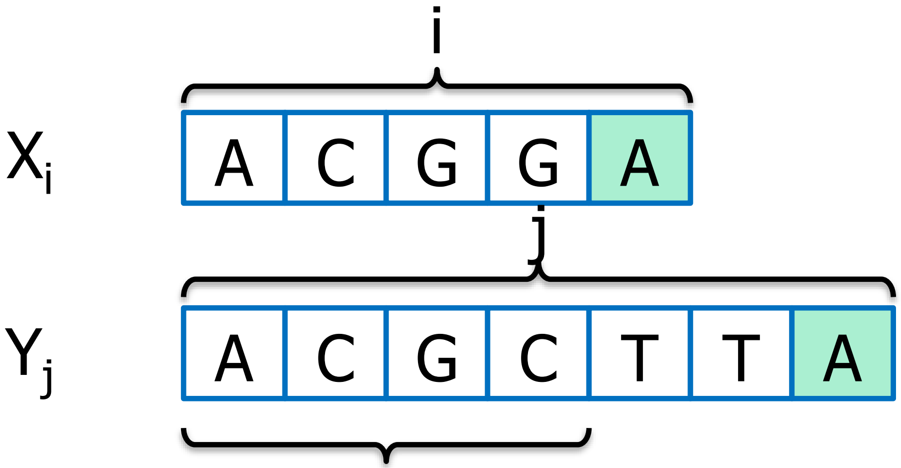
Notation: Denote this prefix *ACGC* by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let $f[i,j]$ = length_of_LCS(X_i, Y_j)
- Finding LCS's prefixes of X and Y.

Sub-problems

Our sub-problems will be finding LCS's of prefixes to X and Y.
Let f[i,j] = length_of_LCS(X_i, Y_j)

- *Case 1:* X[i] = Y[j]



Notation: Denote this prefix *ACGC* by Y_4

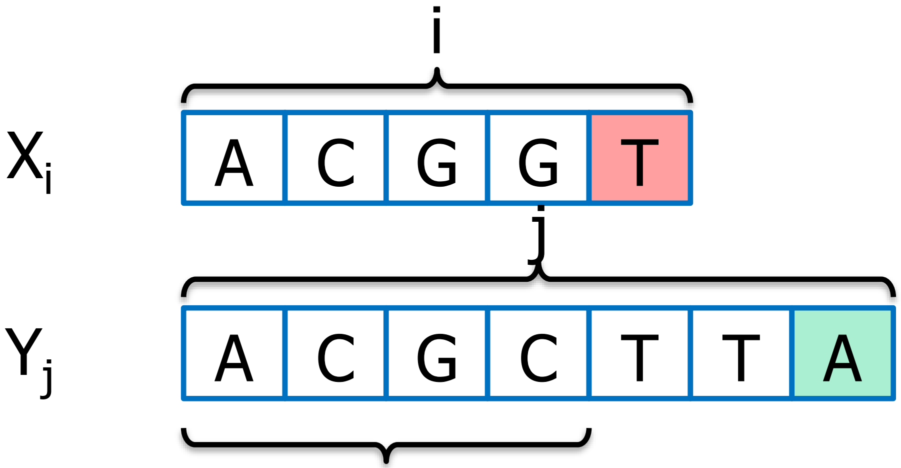
- Then, f[i, j] = 1 + f[i-1,j-1].
  - Because LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) followed by $\boxed{A}$

Sub-problems

Our sub-problems will be finding LCS's of prefixes to X and Y.

Let $f[i,j]$ = length_of_LCS(X_i, Y_j)

- *Case 2: X[i] != Y[j]*



Notation: Denote this prefix *ACGC* by Y_4

- Then, f[i, j] = max(f[i-1, j], f[i,j-1]).
  - either $LCS(X_i, Y_j) = LCS(X_{i-1}, Y_j)$ and $\boxed{T}$ is not involved,
  - Or $LCS(X_i, Y_j) = LCS(X_i, Y_{j-1})$ and $\boxed{A}$ is not involved.

Recursive formulation

Case 0

X_0							
Y_j	A	C	G	C	T	T	A

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

Case 1

X_i	A	C	G	G	A		
Y_j	A	C	G	C	T	T	A

Case 2

X_i	A	C	G	G	T		
Y_j	A	C	G	C	T	T	A

LCS Example

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X: | A | C | G | G | A |

Y: | A | C | T | G |

		A	C	T	G
	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

LCS Example

$$f[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X	A	C	G	G	A
Y	A	C	T	G	

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

LCS Example

$$
f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}
$$

X | A | C | G | G | A

Y | A | C | T | G

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

Once we've filled in,
we can work backwards
to get the LCS

LCS Example

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X	A	C	G	G	A

Y	A	C	T	G

That 3 comes from the 3 above it

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

LCS Example

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X	A	C	G	G	A

Y	A	C	T	G

That 3 comes from the 3 above it

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

LCS Example

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X

A	C	G	G	A

Y

A	C	T	G

A diagonal jump means that we found an element of the LCS!

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

LCS Example

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X | A | C | G | G | A |

Y | A | C | T | G |

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

LCS Example

$$f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}$$

X	A	C	G	G	A

Y	A	C	T	G

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

A diagonal jump means that we found an element of the LCS!

LCS Example

$$
f[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + f[i-1, j-1] & \text{if } X[i]=Y[j] \\ \max(f[i-1, j], f[i,j-1]) & \text{if } X[i]!=Y[j] \end{cases}
$$

X: | A | C | G | G | A |

Y: | A | C | T | G |

This is the LCS: | A | C | G |

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

Dynamic Programming Implementation

```
def LCS(X , Y):
    m = len(X)
    n = len(Y)
    f = [[None]*(n+1) for i in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                f[i][j] = 0
            elif X[i-1] == Y[j-1]:
                f[i][j] = f[i-1][j-1]+1
            else:
                f[i][j] = max(f[i-1][j] , f[i][j-1])

    return f[m][n]
```

Running time: O(nm)

```
X = "ACGGA"
Y = "ACTG"
print ("Length of LCS is ", LCS(X, Y) )
```

```
Length of LCS is  3
```

Exercise

- Write code to recover the actual LCS not just its length.
  - Time complexity?

Exercise: Longest Increasing Subsequence

- Subsequence:
  - [3,6,2,7] is a subsequence of [0,3,1,6,2,7]

- Use Dynamic Programming to find the length of the longest strictly increasing subsequence.
  - [0,1,0,3,2,3] : the longest increasing subsequence is [0,1,2,3], therefore the length is 4

- Find the actual longest increasing subsequence.

Past Exam Paper

Question 3. [15 MARKS]

Given six items with weights and values shown in the table below.

Item	1	2	3	4	5	6
Weight	1	2	4	5	2	1
Value	12	20	15	35	10	9

Assume we have a knapsack that can hold items with a total weight of 10. What is the best solution to the 0-1 knapsack problem? Solve the problem with dynamic programming. **[15 marks]**

Learning outcome

- Able to apply dynamic programming to solve the 0-1 knapsack problem
- Able to apply dynamic programming to solve the longest common subsequence problem
- Able to implement dynamic programming with Python