ALGORITHMS DESIGN STRATEGIES

Chapter5

Prepared by: Enas Abu Samra

KEY POINTS OF CHAPTER5

- Classification of Algorithms.
- Brute Force Technique.
- Divide and Conquer Technique.
- Dynamic Programming Technique.
- Longest Common Subsequences Problem.
- Greedy Technique.
- Knapsack Problem.

CLASSIFICATION OF ALGORITHMS

- Getting familiar with different Algorithm designs has become important for IT professionals.
- How to classify / group algorithms?
    - Type of problems solved
    - Design techniques
    - Deterministic vs non-deterministic

CLASSIFICATION OF ALGORITHMS
(TYPE OF PROBLEMS SOLVED)

Problem Types:

- ✓ Searching (Linear search algorithm, Binary search algorithm, …)
- ✓ Sorting (Selection sort algorithm, Insertion sort algorithm, …)
- ✓ Graph/ Network problems

  Shortest path, Traveling salesman.
- ✓ Dealing with sequences:

  Storing, Mapping and analyzing, Aligning
- ✓ and others...

CLASSIFICATION OF ALGORITHMS
(DESIGN TECHNIQUES)

A given problem can be solved using different approaches. Some approaches deliver much more efficient results than others by means of:

- ✓ Usage of Resources
- ✓ Time and Space Complexities
- ✓ Maintainability
- ✓ Security

ALGORITHM DESIGN STRATEGIES

- Brute force (Exhaustive method)
- Divide and conquer (D & C)
- Dynamic programming (DP)
- Greedy approach
- Decrease and conquer
- Transform and conquer
- Backtracking and branch-and-bound

BRUTE FORCE

- Brute Force Algorithms refers to a programming style that **does not** include any shortcuts to improve performance.
- A brute force algorithm blindly iterates an entire domain of possible solutions in search of one or more solutions that satisfy a condition.
- An algorithm that **inefficiently** solves a problem, often by trying every one of a wide range of possible solutions.

BRUTE FORCE

Example:

Break a password (Open a Lock)

- ✓ It is to attempt to break the 3-digit password (each digit either x or o) then brute force may take up to 2^3 → (8) attempts to crack the code.
- ✓ **Disadvantage of Brute-Force algorithms:**

  In many cases not efficient in terms of (time, space complexities)

DIVIDE AND CONQUER

- Recursive decomposition into "smaller" problem instances and solving them all.

✓ **Divide** – The original problem is divided into **independent** sub-problems.

✓ **Conquer -** The sub-problems are solved **recursively**.

✓ **Combine** – The solutions of the sub-problems are combined together to get the solution of the original problem.

DIVIDE AND CONQUER

- **Examples of algorithms based on divide and conquer technique:**

  Binary search, Quick sort, Merge sort, Matrix inversion, Matrix multiplication, ….
- **Advantages**
  - ✓ Solving different problems in less time and thus less complexity
- **Disadvantages**
  - ✓ Sometimes it can become more complicated than a basic iterative approach
  - ✓ Recursive calls use the **stack**, which means more space complexity.
  - ✓ Sometimes more calculations are performed.

DYNAMIC PROGRAMMING (TABULAR METHOD)

- Dynamic Programming (DP) is a bottom-up approach in which all possible small problems are solved and then combined to obtain solutions for bigger problems.

✓ **Divide** – The original problem is divided into **dependent** sub-problems.

✓ **Conquer -** The sub-problems are solved recursively.

✓ **Combine -** The solutions of the sub-problems are combined together to get the solution of the original problem.

DYNAMIC PROGRAMMING (TABULAR METHOD)

- The word "**programming**" in the name of this technique stands for "**planning**" and does not refer to computer programming.
- Dynamic programming is a technique for solving problems with **overlapping** subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only **once** and recording the results in a **table** from which a solution to the original problem can then be obtained.

ELEMENTS OF DYNAMIC PROGRAMMING

- Three elements characterize a dynamic programming algorithm:

1. **Substructure:** Decompose the given problem into smaller subproblems.
2. **Table Structure:** After solving the sub-problems, store the results of the sub-problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem.
3. **Bottom-up Computation:** Using the table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrive at a solution to complete a problem.

DYNAMIC PROGRAMMING EXAMPLES

- **Examples:**
  - ✓ 0/1 Knapsack problem
  - ✓ Largest Common Subsequences (LCS)
  - ✓ All Pair Shortest Path Problem
  - ✓ Time Sharing: Schedule Jobs to maximize CPU Utilization Longest.

OPTIMIZATION PROBLEMS

- Dynamic Programming is the most powerful design technique for solving **optimization problems.**

- **Optimization problem includes:**

✓ Find a solution with the **GLOBAL** optimal value (**minimum or maximum**).

✓ A set of choices must be made to get an optimal solution.

✓ There may be many solutions that return the optimal value: we want to find one of them.

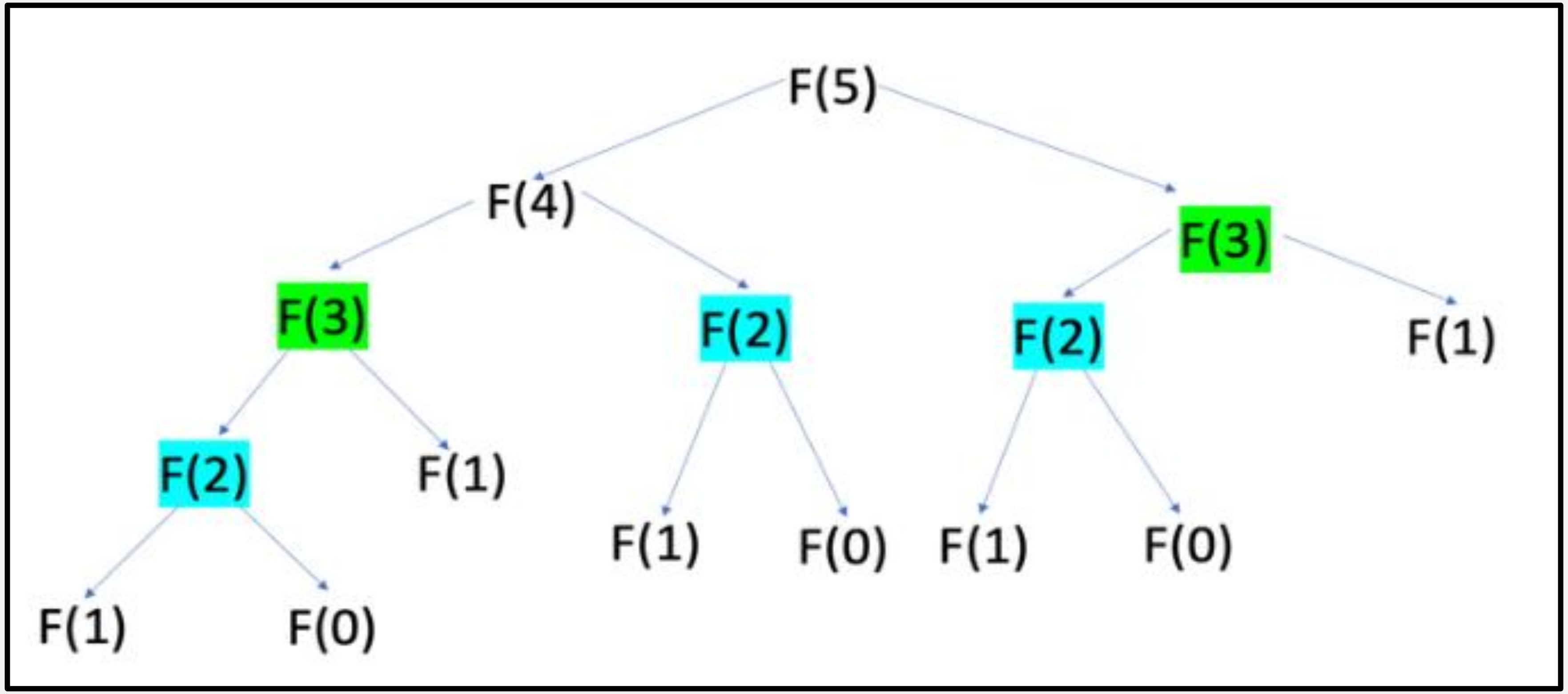
DYNAMIC PROGRAMMING VERSUS DIVIDE AND CONQUER

- **The divide and conquer** algorithm partition the problem into **independent** subproblems, solve the subproblems recursively, and then combine their solution to solve the original problems.
- **Dynamic Programming** is used when the subproblems are **dependent**, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub-problem multiple times.

FIBONACCI NUMBERS PROBLEM

- **Recursion ➔**
  - ✓ **Recurrence case:** F(n) = F(n-1)+ F(n-2)
  - ✓ **Base case:** F(0) = 0, F(1) = 1
- **A divide-and-conquer approach** would **repeatedly** solve the common subproblems.
- **Dynamic programming approach** solves every subproblem **just once** and stores the answer in a table.

FIBONACCI NUMBERS PROBLEM



LONGEST COMMON SUBSEQUENCE (LCS)

- The sequence is: $X = <X_1, X_2, ..., X_n>$
- **Examples:**
  - ✓ Y= <B, D, C, A, B, D, E>
  - ✓ Z = <1, 2, 5>
  - ✓ S = <H>
  - ✓ X = <A, B, C>

Note: A subset of elements in the sequence taken **in order** (but not necessarily consecutive).

- Subsequence of X ➔ <A>, <B>, <C>, <A, B>, <A, C>, <B, C>, <A, B, C>

LONGEST COMMON SUBSEQUENCE (LCS)

- **Example:**

X = <A, B, C, B, D, A, B>

Y = <B, D, C, A, B, A>

➔<B, C, B, A> and <B, D, A, B> are the longest common subsequences of X and Y

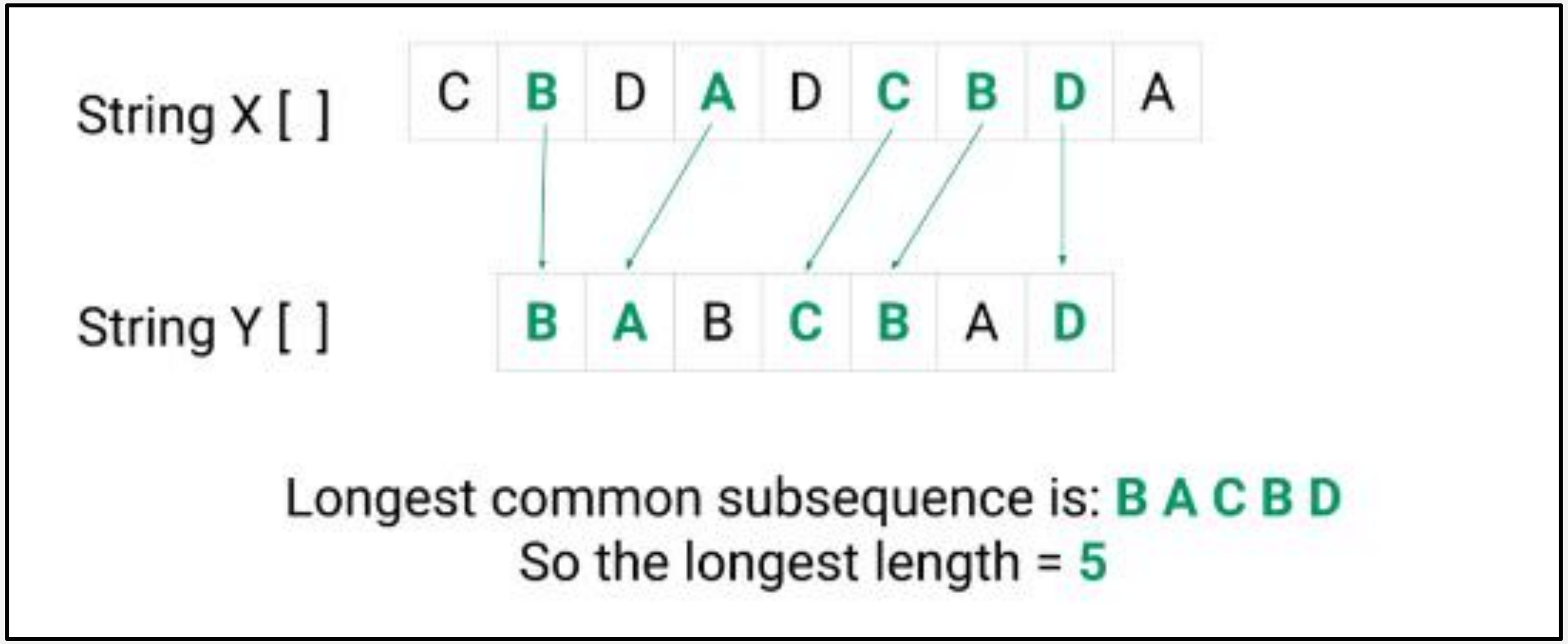
➔ (length = 4)

Note:<(B, C, A> is a subsequence, but is not a LCS of X and Y.

SOLVING LCS USING BRUTE FORCE

- For every subsequence of X, check whether it's a subsequence of Y.
- There are 2^m subsequences of X to check. (m: length of X)
- Each subsequence takes (n) time to check. (n: length of Y)
- **Running time:** $O(n2^m)$
- This technique useful in case the size of the problem was **small**.

SOLVING LCS USING BRUTE FORCE



SOLVING LCS USING DP
(RECURSION FUNCTION)

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0 \ , \\ c[i-1, j-1] + 1 & \textbf{if } i, j > 0 \textbf{ and } x_i = y_j \ , \\ \max(c[i, j-1], c[i-1, j]) & \textbf{if } i, j > 0 \textbf{ and } x_i \neq y_j \ . \end{cases}$$

SOLVING LCS USING DP

- Number of rows = n + 1
- Number of columns = m +1
- Always fill the first row and the first column with **zero**.
- There is no difference in the solution if you swap the X and Y.

- n ➔ size of X, m ➔ size of Y

		0	1	2	3	4	5	6
		Yj	A	D	C	A	B	A
0	Xi	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	D	0						
4	C	0						
5	B	0						
6	A	0						
7	B	0						

SOLVING LCS USING DP

- The complexity is: **O(m*n)**

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

EXAMPLES OF SOLVING LCS USING DP

- Similar ➔ ↖ (value +1)
- Not Similar ➔

If (The value of above >= Adjacent value)

↑ (same value)

Else

← (same value)

X \ Y		0	a 1	s 2	w 3	v 4
	0	0	0	0	0	0
a	1	0	↖ **1**	← 1	← 1	← 1
r	2	0	↑ 1	↑ 1	↑ 1	↑ 1
s	3	0	↑ 1	↖ **2**	← 2	← 2
w	4	0	↑ 1	↑ 2	↖ **3**	← 3
q	5	0	↑ 1	↑ 2	↑ 3	← 3
v	6	0	↑ 1	↑ 2	↑ 3	↖ **4**

EXAMPLES OF SOLVING LCS USING DP

- Similar ➔ ↖ (value +1)

- Not Similar ➔

If (The value of above >= Adjacent value)

↑ (same value)

Else

← (same value)

$a = \langle A, B, C, B, D, A, B \rangle$ $b = \langle B, D, C, A, B, A \rangle$

Longest Common Subsequence: BCBA

	j	0	1	2	3	4	5	6
i		b_j	**B**	D	**C**	A	**B**	**A**
0	a_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	**B**	0	↖1	←1	←1	↑1	↖2	←2
3	**C**	0	↑1	↑1	↖2	←2	↑2	↑2
4	**B**	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	**A**	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

EXAMPLES OF SOLVING LCS USING DP

- Similar ➔ ↖ (value +1)

- Not Similar ➔

If (The value of above >= Adjacent value)

↑ (same value)

Else

← (same value)

		A	B	A	C	A	B	B
	0	0	0	0	0	0	0	0
B	0	↑ 0	↖ 1	← 1	← 1	← 1	↖ 1	↖ 1
A	0	↖ 1	↑ 1	↖ 2	← 2	↖ 2	← 2	← 2
B	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 2	↖ 3	↖ 3
C	0	↑ 1	↑ 2	↑ 2	↖ 3	← 3	← 3	← 3
A	0	↖ 1	↑ 2	↖ 3	↑ 3	↖ 4	← 4	← 4
B	0	↑ 1	↖ 2	↑ 3	↑ 3	↑ 4	↖ 5	↖ 5

EXAMPLES OF SOLVING LCS USING DP

- Similar ➔ ↖ (value +1)

- Not Similar ➔

If (The value of above >= Adjacent value)

↑ (same value)

Else

← (same value)

	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
A	0	↑0	↑0	↑0	↖1	←1	↖1
B	0	↖1	←1	←1	↑1	↖2	←2
C	0	↑1	↑1	↖2	←2	↑2	↑2
B	0	↖1	↑1	↑2	↑2	↖3	←3
D	0	↑1	↖2	↑2	↑2	↑3	↑3
A	0	↑1	↑2	↑2	↖3	↑3	↖4
B	0	↖1	↑2	↑2	↑3	↖4	↑4

GREEDY APPROACH

- A **greedy** algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage in the hope of getting a globally optimal solution.
- In many problems, **a greedy strategy does not always produce/guarantee an optimal solution**, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.
- Also used for optimization and complex problems.
- This algorithm is called **greedy** because when the optimal solution to the smaller instance is provided, the algorithm does not consider the total problem as a whole.

GREEDY APPROACH

- **Examples**

✓ Fraction knapsack problem

✓ Coin-changing problem

✓ Graphs

  - Dijkstra's shortest-path algorithm
  - Prim's minimum-spanning tree algorithm
  - Kruskal's minimum-spanning tree algorithm

KNAPSACK PROBLEM

- A thief robbing a store finds n items: the i-th item is worth $\mathbf{v_i}$ dollars **(profit)** and $\mathbf{w_i}$ pounds **(weight)** (v_i and w_i ➔ integers).
- The thief can only carry **W** pounds in his knapsack, he puts these items in a knapsack to get the **maximum** profit in the knapsack.
- Which items should the thief take to **maximize** the value of his load?

KNAPSACK PROBLEM

- There are **two versions** of the problem:

1. "Fractional knapsack problem"

Items are **divisible**; you can take any fraction of an item.

2. "0-1 knapsack problem"

Items are **indivisible**; you either take an item or not.

FRACTIONAL KNAPSACK PROBLEM

- Items are **divisible**; you can take any fraction of an item.

- **There are basically three approaches to solve the problem:**

  - The first approach is to select the item based on the maximum profit.

  - The second approach is to select the item based on the minimum weight.

  - The third approach is to calculate the ratio of profit/weight. (Here)

- **Time Complexity:** O(n) if items already ordered; else O(nlogn).

FRACTIONAL KNAPSACK PROBLEM

- **Steps:**

1. Compute ratio = P_i / W_i
2. Order the items **descending** based on **ratio**.
3. Choose items so that they **do not exceed** the capacity of knapsack.
4. The overall profit = **sum** (profit of the selected items)

- **Note:** Greedy Strategy is **good** for "Fractional Knapsack Problem"

FRACTIONAL KNAPSACK PROBLEM

Ex1:

Item	Profit	Weight
A	500	40
B	225	25
C	330	30

Consider that the capacity of the knapsack is **W = 55**

Sol.

Item	Profit	Weight	Ratio	Remaining Weight	Overall Profit
A	500	40	12.5	55- 40 = 15	**500**
C	330	30	11	15-15 = 0	**165**
B	225	25	9	0	**0**
					665

Using the greedy technique, item A is selected. the profit is 500. However, we have remaining available weight 15. it's picked from C with a profit 165.

Hence, **the total profit is 500 + 165 =665.**

FRACTIONAL KNAPSACK PROBLEM

Ex2:

Item	Profit	Weight
A	5	1
B	10	3
C	15	5
D	7	4
E	8	1
F	9	3
G	4	2

Consider that the capacity of the knapsack is **W = 15**

Sol.

Item	Profit	Weight	Ratio	Remaining Weight	Overall Profit
E	8	1	8	15 – 1 = 14	**8**
A	5	1	5	14 – 1 = 13	**5**
B	10	3	3.3	13 – 3 = 10	**10**
C	15	5	3	10 – 5 = 5	**15**
F	9	3	3	5 – 3 = 2	**9**
G	4	2	2	2 – 2 = 0	**4**
D	7	4	1.7	0	**0**
					51

Hence, **the total profit = 51**

0/1 KNAPSACK PROBLEM

- Items are **indivisible**; you either take an item or not.
- **There are many approaches to solve the problem:**

  - Greedy Approach. (**Does not** ensure an optimal solution)

  - Dynamic Programming Approach. (Ensure an optimal solution)

  - Other approaches.

SOLVING 0/1 KNAPSACK
USING GREEDY TECHNIQUE (DON'T USE IT)

Ex1: Given the following items:

Item	Profit	Weight	Ratio
A	500	40	12.5
B	225	25	9
C	330	30	11

Consider that the capacity of the knapsack is **W = 55**

Using the Greedy approach, item A is selected.

The profit is 500 and W = 55 – 40 =15. However, the **global** optimal solution of this instance can be achieved by selecting items, B and C, where the total profit is 225 + 330 = 555.

Conclusion: 0-1 Knapsack **cannot** be solved by the greedy technique. The greedy technique **does not** ensure an optimal solution, but in some cases, the greedy technique may give a global optimal solution.

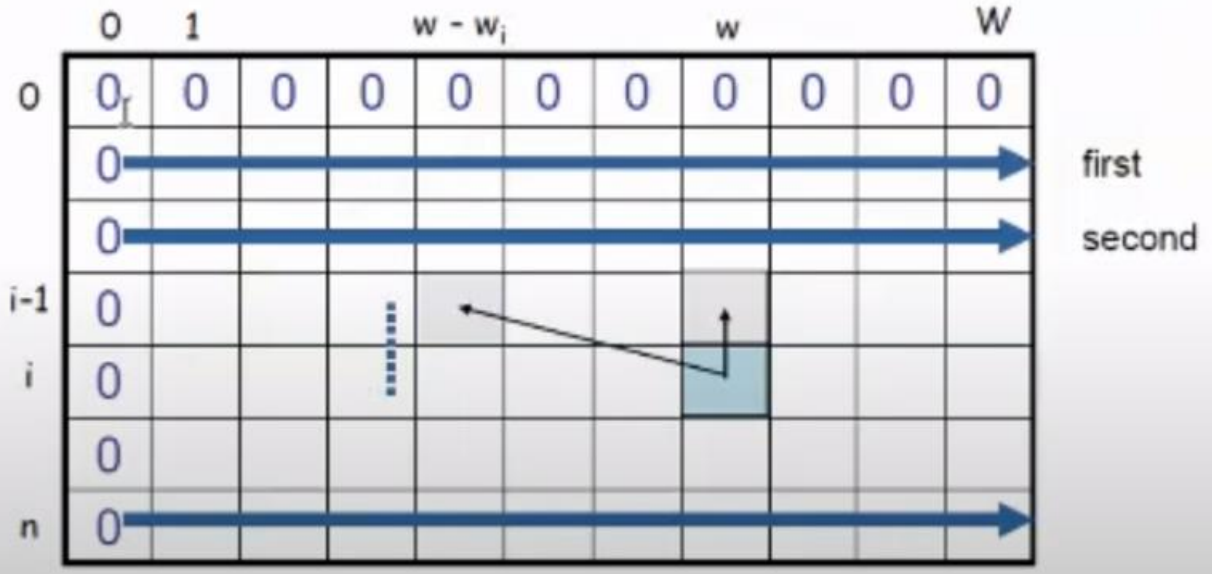
SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

- Fill the table rows using the following rule:

If (j < w[i])

then T[i][j] = T[i-1][j]; #copy from the above cell

$$\text{else } T[i][j] = \max(\underbrace{(p[i] + T[i-1][j-w[i]])}_{\text{Item i was taken}},\ \underbrace{T[i-1][j]}_{\text{Item i was **not** taken}})$$



- Running time: $\Theta(nW)$

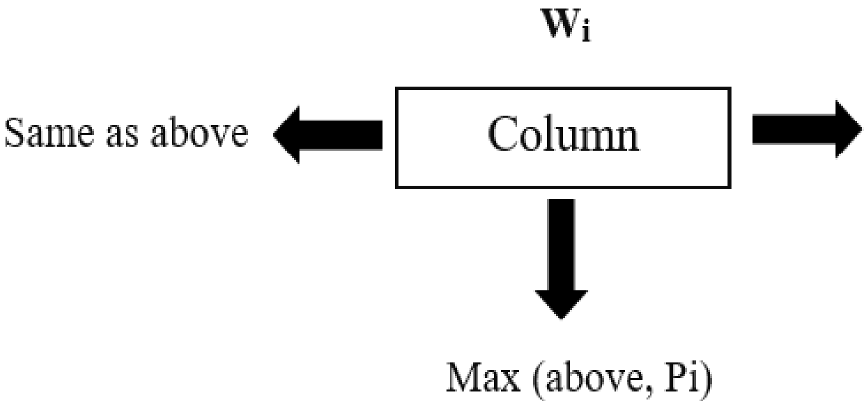
SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Rules:

# of rows = # of items + 1

# of columns = W + 1

Fill the first row and the first column with zero.



1) Value = current weight - W_i
2) Intersection = column(value), previous row
3) $P = P_i + P_{intersection}$
4) Max (above, P)

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

How to choose the item?

Start at the last cell

if (cell == above cell)

ignore this item and move to the previous row.

else

{**choose** this item

$W = W - W_i$

move to (column(W), previous row)

}

Repeat the previous steps until reaching to the first row/column.

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Ex1: Given the following items:

Item	Profit	Weight
A	10	1
B	15	2
C	40	3

Consider that the capacity of the knapsack is **W = 6**

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Sol:

	0	1	2	3	4	5	6
0	**0**	0	0	0	0	0	0
A	0	**10**	10	10	10	10	10
B	0	10	15	**25**	25	25	25
C	0	10	15	40	50	55	**65**

W

6

3 column = 3

1 column = 1

0 column = 0

Profit = 40 + 15 + 10 = 65

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

A value = 2-1 = 1 inter= col(1), prev_row P=10+0 =10 Max(0,10)	**A** value = 3-1 = 2 inter= col(2), prev_row P=10+0 =10 Max(0,10)	**A** value = 4-1 = 3 inter= col(3), prev_row P=10+0 =10 Max(0,10)	**A** value = 5-1 = 4 inter= col(4), prev_row P=10+0 =10 Max(0,10)	**A** value = 6-1 = 5 inter= col(5), prev_row P=10+0 =10 Max(0,10)
	B value = 3-2 = 1 inter= col(1), prev_row P=15+ 10=25 Max(10, 25)	**B** value = 4-2 = 2 inter= col(2), prev_row P=15+ 10=25 Max(10, 25)	**B** value = 5-2 = 3 inter= col(3), prev_row P=15+ 10=25 Max(10, 25)	**B** value = 6-2 = 4 inter= col(4), prev_row P=15+ 10=25 Max(10, 25)
		C value = 4-3 = 1 inter= col(1), prev_row P=40+10 =50 Max(25,50)	**C** value = 5-3 = 2 inter= col(2), prev_row P=40+15 =55 Max(25,55)	**C** value = 6-3 = 3 inter= col(3), prev_row P=40+25 =65 Max(25,65)

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Ex2: Given the following items:

Item	Profit	Weight
A	12	2
B	10	1
C	20	3
D	15	2

Consider that the capacity of the knapsack is **W = 5**

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Sol:

	0	1	2	3	4	5
0	0	**0**	0	0	0	0
A	0	0	**12**	12	12	12
B	0	10	12	**22**	22	22
C	0	10	12	**22**	30	32
D	0	10	15	25	30	**37**

W

5

3 column = 3

2 column = 2

0 column = 0

Profit = 15 + 10 + 12 = 37

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

	A value = 3-2 = 1 inter= col(1), prev_row P=12+0 =12 Max(0,12)	**A** value = 4-2 = 2 inter= col(2), prev_row P=12+0 =12 Max(0,12)	**A** value = 5-2 = 3 inter= col(3), prev_row P=12+0 =12 Max(0,12)
B value = 2-1 = 1 inter= col(1), prev_row P=10+ 0=10 Max(12, 10)	**B** value = 3-1 = 2 inter= col(2), prev_row P=10+ 12=22 Max(12, 22)	**B** value = 4-1 = 3 inter= col(3), prev_row P=10+ 12=22 Max(12, 22)	**B** value = 5-1 = 4 inter= col(4), prev_row P=10+ 12=22 Max(12, 22)

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

C

value = 4-3 = 1

inter= col(1), prev_row

P=20+10 =30

Max(22,30)

C

value = 5-3 = 2

inter= col(2), prev_row

P=20+12 =32

Max(22,32)

D

value = 3-2 = 1

inter= col(1), prev_row

P=15+10 =25

Max(22,25)

D

value = 4-2 = 2

inter= col(2), prev_row

P=15+12 =27

Max(30,27)

D

value = 5-2 = 3

inter= col(3), prev_row

P=15+22 =37

Max(32,37)

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Ex3: Given the following items:

Item	Profit	Weight
A	2	3
B	3	4
C	1	6
D	4	5

Consider that the capacity of the knapsack is **W = 8**

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Sol:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
A	0	0	0	**2**	2	2	2	2	2
B	0	0	0	**2**	3	3	3	5	5
C	0	0	0	**2**	3	3	3	5	5
D	0	0	0	2	3	4	4	5	**6**

<u>W</u>

8

3 column = 3

0 column = 0

Profit = 4 + 2 = 6

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Ex4: Given the following items:

Item	Profit	Weight
A	2	1
B	4	3
C	7	5
D	10	7

Consider that the capacity of the knapsack is **W = 8**

SOLVING 0/1 KNAPSACK USING DYNAMIC PROGRAMMING

Sol:

	0	1	2	3	4	5	6	7	8
0	**0**	0	0	0	0	0	0	0	0
A	0	**2**	2	2	2	2	2	2	2
B	0	**2**	2	4	6	6	6	6	6
C	0	**2**	2	4	6	7	9	9	11
D	0	2	2	4	6	7	9	10	**12**

W

8

1 column = 1

0 column = 0

Profit = 10 + 2 = 12

CONCLUSION

- Creating an algorithm design and choosing the best algorithm design strategy for a particular problem is an art that requires a good understanding of each strategy, the strengths and weak points of applying it to each type of problem, and taking into consideration the environment and constraints for solving the problem.
- **For example**, some strategies solve the problem with less time, but require extra memory, while others may solve it with less space requirements, but they do need more time.

END OF CHAPTER5