

Unit 3:Syllabus

Greedy strategy:

- Principle, control abstraction, time analysis of control abstraction,
- Knapsack problem
- Scheduling algorithms-Job scheduling
- Activity selection problem

Dynamic Programming:

- Principle, control abstraction, time analysis of control abstraction
- Binomial coefficients
- OBST
- 0/1 knapsack
- Chain Matrix multiplication

Unit Outcomes

CO3: Decide and apply algorithmic strategies to solve given problem

CO4: Find optimal solution by applying various methods

Greedy Strategy

Greedy Algorithms

Similar to dynamic programming, but **simpler** approach

- Also used for optimization problems

Idea: When we have a choice to make, make the one that looks best **right now**

- Make a **locally** optimal choice in hope of getting a globally optimal solution

Greedy algorithms don't always yield an optimal solution

Makes the choice that looks best at the moment in order to get optimal solution.

Greedy Algorithms

Greedy Advantages

Greedy algorithms have several advantages over other algorithmic approaches:

Simplicity: Greedy algorithms are often easier to describe and code up than other algorithms.

Efficiency: Greedy algorithms can often be implemented more efficiently than other algorithms.

Greedy Algorithms

Greedy algorithms have several drawbacks:

Hard to design: Once you have found the right greedy approach, designing greedy algorithms can be easy. However, finding the right approach can be hard.

Hard to verify: Showing a greedy algorithm is correct often requires a nuanced argument.

Greedy Algorithms

if the problem has the following properties:

1. Greedy Choice Property:-If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.
2. Optimal Substructure:-If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

Greedy algorithms have the following five components –

- A candidate set – A solution is created from this set.

- A selection function – Used to choose the best candidate to be added to the solution.
- A feasibility function – Used to determine whether a candidate can be used to contribute to the solution.
- An objective function – Used to assign a value to a solution or a partial solution.
- A solution function – Used to indicate whether a complete solution has been reached.

Greedy Approach

- Construct an optimal solution in sequence of choices.
 - Ex. Min. spanning tree, shortest path
- A feasible solution
 - Solution for which the optimization function satisfies the problem constraints.
- An optimal solution
 - A feasible solution for which the optimization function has the best possible value.

Control Abstraction

Algorithm Greedy (a, n)

begin

 solution = nil

 for i = 1 to n do

 begin

 x = Select (a)

 if Feasible (solution, x) then

 solution = Union (solution, x)

 end

 return solution

Fractional Knapsack Problem

fill up the knapsack such that

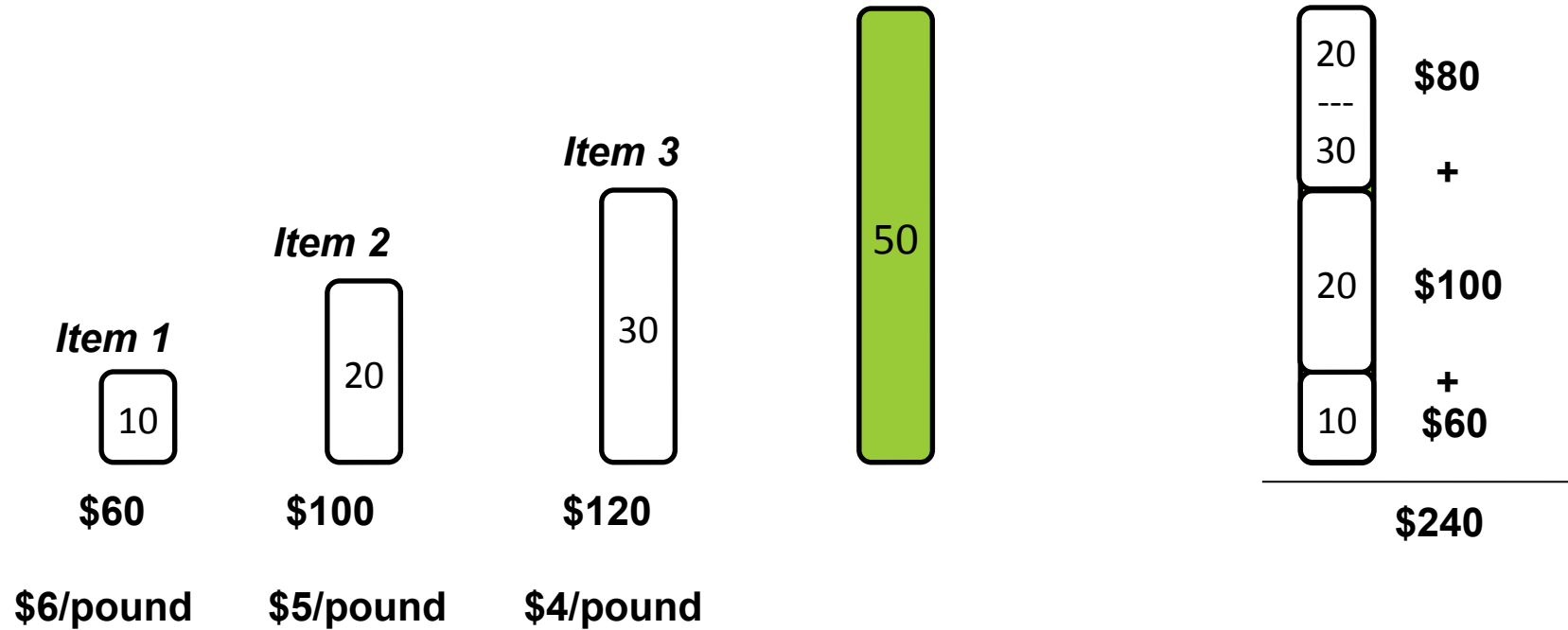
$\sum_{i=1}^n x_i p_i$ is maximum subject to the condition

$$\sum_{i=1}^n x_i w_i \leq M$$

where, $0 \leq x_i \leq 1$

Fractional Knapsack - Example

E.g.:



Knapsack Problem: Algorithm

Knapsack-Greedy ($w[]$, $p[]$, M)

begin

for $i = 1$ to n do $x[i] = 0$ // initialize

weight = 0, profit = 0

while (weight \leq M) do

begin

i = next object with highest
profit/weight ratio

if (weight + $w[i]$ \leq M) then

begin

$x[i] = 1$

weight = weight + $w[i]$

profit = profit + $p[i]$

else

$x[i] = (M - \text{weight}) / w[i]$

profit = profit + $p[i] * x[i]$

weight = M

end

end

return $x[]$ // solution

end

Knapsack Problem

Q. Find feasible solutions for the following knapsack instance :

Let $n = 5$, $M = 100$,

$w = \{10, 20, 30, 40, 50\}$,

$P = \{20, 30, 66, 40, 60\}$

Ans:

- 1) Increasing order of weight*
- 2) Decreasing order of profit*
- 3) Decreasing order of profit per weight ratio*

Job sequencing with deadlines

- We are given a set of n jobs.
- Associated with job i is an integer deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- For any job i , profit p_i is earned iff the job is completed by its deadline.
- In order to complete a job one has to process the job on a machine for one unit of time.
- Only one machine is available for processing jobs.
- **Job sequencing rule:**
 - In the job sequencing with deadlines algorithm, profit p_i is earned iff job i is completed by its deadline.

Job sequencing with deadlines (cont..)

Feasible solution

- A subset J of jobs such that each job in this subset can be completed by its deadline.
- The value of a feasible solution is the sum of the profits of the jobs in J .

Optimal solution

- A feasible solution having a set of jobs completed within their deadline giving maximum profit.

Brute force approach: Obtain all feasible solutions for the following jobs. Let $n=4$, profits $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and deadlines $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.

All feasible solutions:

1	100
2	10
3	15
4	27
2 - 1	110
1 - 3 or 3 - 1	115
4 - 1	127
2 - 3	25
4 - 3	42

Optimal solution:

Job sequence (4,1)

Profit 127

Greedy approach

- Arrange jobs in **the decreasing order of profits.**



- Obtain optimal solutions for the following jobs:

Sorted profits = $(p_1, p_4, p_3, p_2) = (100, 27, 15, 10)$

Deadlines $d = (d_1, d_4, d_3, d_2) = (2, 1, 2, 1)$



Job sequencing with deadlines

Algorithm Greedy_Job (d, J, n)

begin

$J = \{ 1 \}$

 for $i = 2$ to n do

 if (all jobs in $J \cup \{ i \}$ can be completed
 by their deadlines)

 then $J = J \cup \{ i \}$

 end for

End

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

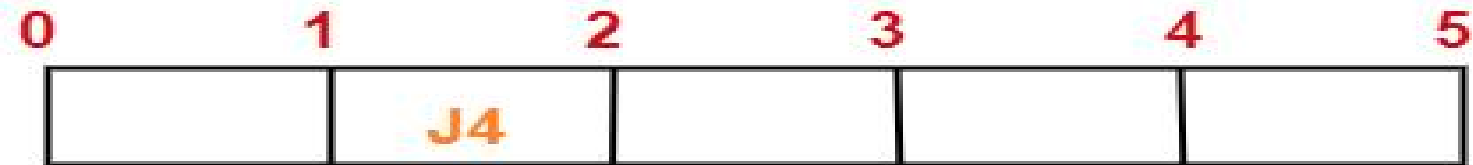
Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



Gantt Chart

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



Step-04:

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

Analysis of Job Sequence Algorithm

- The time complexity of this algorithm can be measured using two parameters
 - total number of jobs n
 - number of jobs k included in the solution J
- Time complexity = $O(kn)$
- But $k = n$, so the worst-case computing time is $O(n^2)$.
- The algorithm requires space for arrays $d[]$ and $J[]$, with few temporary variables.

Activity-selection problem

- When a **common resource** is to be used **exclusively** by various activities, then how the activities should be scheduled is an important issue.
- Let a set of n activities $A = \{A_1, A_2, \dots, A_n\}$ wants to share a common resource exclusively (one at a time).
- **Activity-Selection Problem** is to select a subset of **maximum** size having mutually compatible activities.
- Two activities A_1 and A_2 are said to be **compatible** if $ST_1 \geq FT_2$ or $ST_2 \geq FT_1$.

Activity-selection problem

- Consider the set of activities given in the ascending order of finish time. Find maximum size subset of mutually compatible activities.

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

- Assume that all the activities are arranged according to their **finish time in the ascending order**.

Activity-selection problem

A possible solution would be:

- Step 1: Sort the given activities in ascending order according to their finishing time.
- The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Activity-selection problem: Example (cont..)

: Select the first activity from sorted array **act[]** and add it to the **sol[]** array, thus

. : Repeat the steps 4 and 5 for the remaining activities in **act[]**.

: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to **sol[]**.

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Activity-selection problem: Example (cont..)

: Select the next activity in $act[]$

For the data given in the above table,

A. Select activity . Since the start time of is greater than the finish time of (i.e. $s(a3) > f(a2)$), we add to the solution set. Thus .

B. Select . Since $s(a4) < f(a3)$, it is not added to the solution set.

C. Select . Since $s(a5) > f(a3)$, gets added to solution set. Thus

D. Select . Since $s(a1) < f(a5)$, is not added to the solution set.

E. Select . is added to the solution set since $s(a6) > f(a5)$.

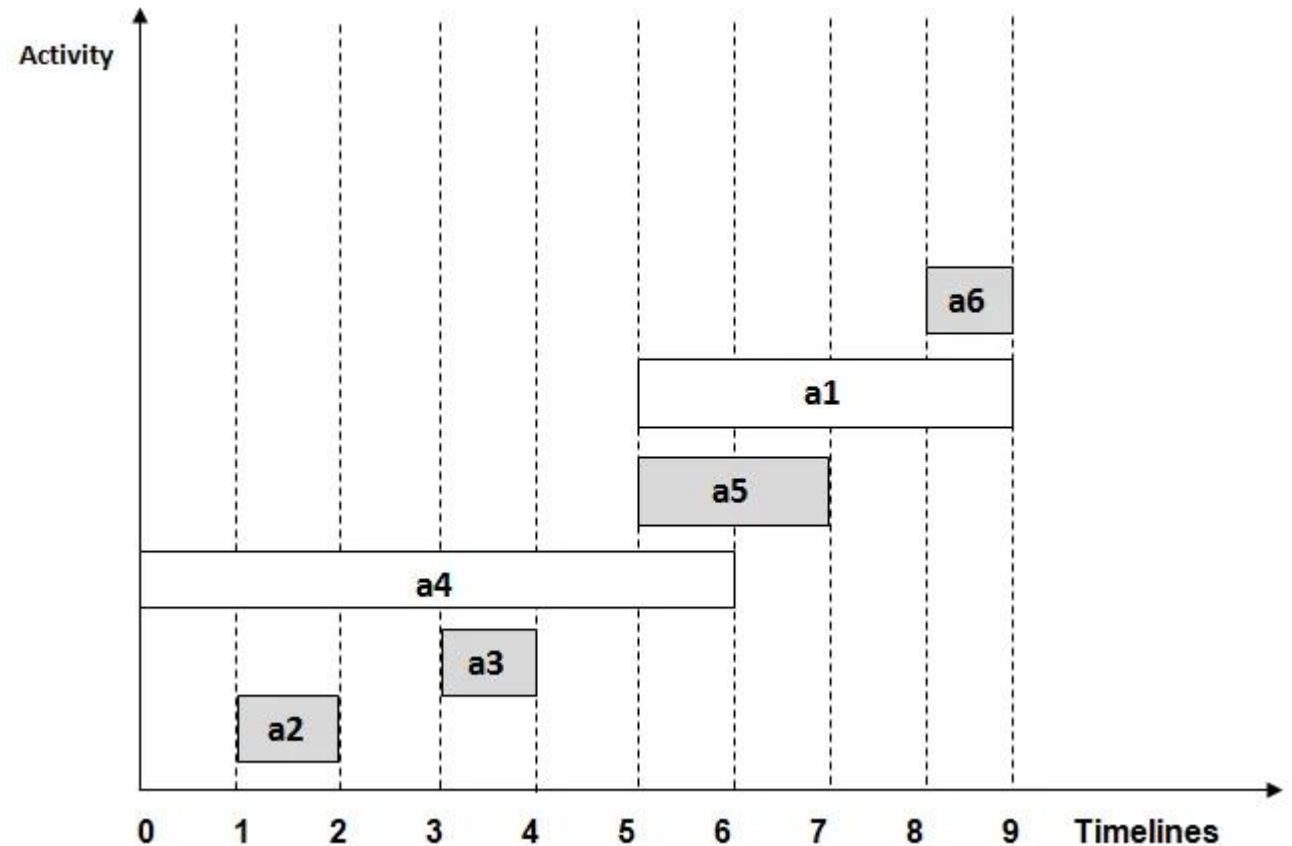
Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Thus .

Activity-selection problem: Example (cont..)

: At last, print the array `sol[]`
Hence, the execution schedule of maximum
number of non-conflicting activities will be:

In the given diagram, the selected activities have
been highlighted in grey.



Dynamic Programming

Dynamic Programming

- Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again
- Dynamic Programming is meant in "**a series of choices**".
- The word **dynamic** conveys the idea that choices may depend on the current state, rather than being decided ahead of time.
- For ex. Programming of the **radio station**

Recursion v/s Dynamic Programming

- Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.
- But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.
- That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

Generic Problem Structure

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Elements of Dynamic Programming

1. Optimal substructure
2. Overlapping sub problems
3. Memorization

Difference between Greedy and Dynamic Programming

- In the greedy method
 - Only one decision sequence is ever generated.
- In dynamic programming
 - Many decision sequences may be generated.

Binomial Coefficients

Pascal's Triangle

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

0th row

1st row

2nd row

3rd row

4th row

5th row

➤ $c(n, k)$

➤ Represents the number of combinations of k values which can be selected from a set of n values.

➤ Represent the values in Pascal's triangle.

➤ Ex. Binomial theorem for $n = 4$

$$(x + y)^4 = x^4 + 4x^3 y + 6x^2 y^2 + 4xy^3 + y^4$$

➤ $c(n, k) = c(n - 1, k - 1) + c(n - 1, k)$

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient $C(n, k)$. For example, your function should return 6 for $n = 4$ and $k = 2$, and it should return 10 for $n = 5$ and $k = 2$.

1) Optimal Substructure


The value of $C(n, k)$ can be recursively calculated using the following standard formula for Binomial Coefficients.

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

$$C(n, 0) = C(n, n) = 1$$

$$\binom{n}{k} = \frac{n!}{k! (n-k)!}$$

FORMULA FOR BINOMIAL COEFFICIENTS


$$\begin{aligned} C(7,3) &= \binom{7}{3} = \frac{7!}{3!(7-3)!} = \frac{7!}{3! \cdot 4!} \\ &= \frac{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{3 \cdot 2 \cdot 1 \cdot 4 \cdot 3 \cdot 2 \cdot 1} \\ &= \frac{5040}{144} = 35 \end{aligned}$$

©Study.com

$$\begin{array}{ccccccc} & & & & 1 & & & & \\ & & & & 1 & & 1 & & \\ & & & 1 & & 2 & & 1 & \\ & & 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \end{array}$$

e.g. $\binom{5}{3} = \frac{5!}{3! \times 2!}$

Binomial Coefficients

Pascal's Triangle

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

0th row

1st row

2nd row

3rd row

4th row

5th row

➤ For each of the first k rows, $(k - 1)$ elements are calculated by adding elements in $(k - 1)^{\text{th}}$ row.

➤ For $(k + 1)$ to n rows, $(k - 1)$ values are calculated by addition for each row.

➤ Number of additions for $c(n, k)$

$$\begin{aligned}
 & \sum_{i=1}^k (k - 1) + \sum_{i=k+1}^n (k - 1) \\
 &= (k(k - 1)/2) + k(n - k) \\
 &\in \theta(nk)
 \end{aligned}$$

Binomial Coefficient

INPUT: N and K

OUTPUT: C [N, K] which is nC_k

Algorithm bc (n, k)

```
{  
  for i = 0 to n  
    {  
      for j = 0 to k  
        if (j = 0 or i = j)  
          then C[i][j] = 1;  
          else C[i][j] = C[i-1, j-1] + C[i-1, j];  
        }  
      return C[n, k];  
    }
```


0-1 KNAPSACK PROBLEM

$$\begin{array}{ll}\text{maximize} & \sum_{1 \leq i \leq n} p_i x_i \\ \text{subject to} & \sum_{1 \leq i \leq n} w_i x_i \leq M \\ \text{and} & x_i = 0/1, 1 \leq i \leq n\end{array}$$

Applications

- Fractional KNAPSACK PROBLEM
 - Object = liquid
 - Ex = petrol, diesel, milk, chemical, medicine
- 0-1 KNAPSACK PROBLEM
 - Object = mobile, chair, duster, box, pen, pencil

Item	Weight (kg)	Value (\$)
Mirror	2	3
Silver nugget	3	4
Painting	4	5
Vase	5	6

Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$$

$$T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$$

Finding T(1,1):-

We have,

- $i = 1$
- $j = 1$
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$$

$$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$$

$$T(1,1) = T(0,1) \{ \text{Ignore } T(0,-1) \}$$

$$T(1,1) = 0$$

Finding $T(1,2)$ -

We have,

- $i = 1$
- $j = 2$
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$$

$$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$$

$$T(1,2) = \max \{ 0, 3+0 \}$$

$$T(1,2) = 3$$

0-1 KNAPSACK Algorithm

- S_{i+1} can be computed by **merging and purging** the states in S_i and S_{i+1} together, using the **dominance rule**:
- If S_{i+1} contains two pairs (p_a, w_a) and (p_b, w_b) where **$p_a < p_b$ and $w_a > w_b$** , then the (p_a, w_a) is dominated by (p_b, w_b) pair. Hence pair (p_a, w_a) is **discarded**.
- Purge (**discard**) all pairs (p, w) with **$w > M$** , because the knapsack capacity M is exceeded.

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Example: Generate the sets S^i , $0 \leq i \leq 3$, for the following knapsack instance: $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$ and $M = 6$. Also find an optimal solution.

$$S^0 = \{(0, 0)\}$$

S is obtained by adding $(p_1, w_1) = (1, 2)$ to each pair of S^0 .

$$S = \{(1, 2)\}$$

S^1 is obtained by merging and purging S^0 and S .

$$S^1 = \{(0, 0), (1, 2)\}$$

S is obtained by adding $(p_2, w_2) = (2, 3)$ to each pair of S^1 .

$$S = \{(2, 3), (3, 5)\}$$

S^2 is obtained by merging and purging S^1 and S .

$$S^2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$$

S is obtained by adding $(p_3, w_3) = (5, 4)$ to each pair of S^2 .

$$S = \{(5, 4), (6, 6), (7, 7), (8, 9)\}$$

S^3 is obtained by merging and purging S^2 and S .

$$S^3 = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$$

Pair $(5, 4)$ get purged here by dominance rule. Also pairs $(7, 7)$ and $(8, 9)$ get purged because $w > M$.

Last pair in S^3 is $(p, w) = (6, 6) \notin S^2$, hence $x_3 = 1$. But $(p_3, w_3) = (5, 4)$

$$\text{Hence } (p, w) = (6 - 5, 6 - 4) = (1, 2).$$

Because $(1, 2) \in S^2$ and $(1, 2) \in S^1$, set $x_2 = 0$

Because $(1, 2) \notin S^0$; set $x_1 = 1$. Hence an optimal solution for the given knapsack problem is $(x_1, x_2, x_3) = (1, 0, 1)$.

Chained Matrix Multiplication

➤ Given a chain $A_1 A_2 A_3 \dots A_n$ of n matrices, find the product

$A_1 A_2 A_3 \dots A_n$.

➤ Parenthesization :

(1)	$((A_1, A_2), A_3), A_4)$
$A_4)$	(2) $((A_1 (A_2 A_3$
$A_4)$	(3) $(A_1 ((A_2 A_3$
$A_4))$	(4) $(A_1 (A_2 (A_3$
$A_4))$	(5) $((A_1 A_2) (A_3$

How we parenthesize a chain of matrices affects cost of evaluating the multiplication of all matrices.

Chained Matrix multiplication (cont..)

➤ For example, consider matrices

A_1 of size 5×3

A_2 of size 3×4

A_3 of size 4×6

A_4 of size 6×5

➤ Different parenthesizing gives different number of multiplications :

- | | | | |
|-----|-------------------------|---------------------------------------|-----|
| (1) | $(A_1 ((A_2 A_3) A_4))$ | takes $(3.4.6) + (3.6.5) + (5.3.5) =$ | 237 |
| | multiplications. | | |
| (2) | $(A_1 (A_2 (A_3 A_4)))$ | takes | |
| | 255 multiplications. | | |
| (3) | $((A_1 A_2) (A_3 A_4))$ | takes | |
| | 280 multiplications. | | |
| (4) | $((A_1 A_2) A_3) A_4$ | takes | |
| | 330 multiplications. | | |
| (5) | $(A_1 (A_2 A_3) A_4)$ | takes | |
| | 312 multiplications. | | |

Chained Matrix multiplication (cont..)

- To find the best way to parenthesize the chain of matrices to minimize the number of scalar multiplications. This can be done by dividing the problem into subproblems and then finding the optimal solution to the subproblem.
- Suppose we have matrix-chain $A_1 \dots A_n$. This chain is divided into subproblems $A_1 \dots A_k$ and $A_{k+1} \dots A_n$. But the problem is to find the value of k . We can find k by looking at the optimal solution of each of the subproblems.

References

1. Parag Himanshu Dave, Himanshu Bhalchandra Dave, Design and Analysis of Algorithms, Pearson Education, ISBN 81-7758-595-9
2. Horowitz and Sahani, "Fundamentals of Computer Algorithms", University Press, ISBN: 978 81 7371 6126, 81 7371 61262
3. Gilles Brassard, Paul Bratley, Fundamentals of Algorithmics , PHI, ISBN 978-81-203-1131-2
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms , MIT Press; ISBN 978-0-262-03384-8

Thank You...

Any Queries ?