DYNAMIC PROGRAMMING

Instructor
Neelima Gupta
University of Delhi, INDIA
ngupta@cs.du.ac.in

Presentation Edited by Sapna Grover

Weighted Interval Scheduling

- Problem: Given a set of jobs described by

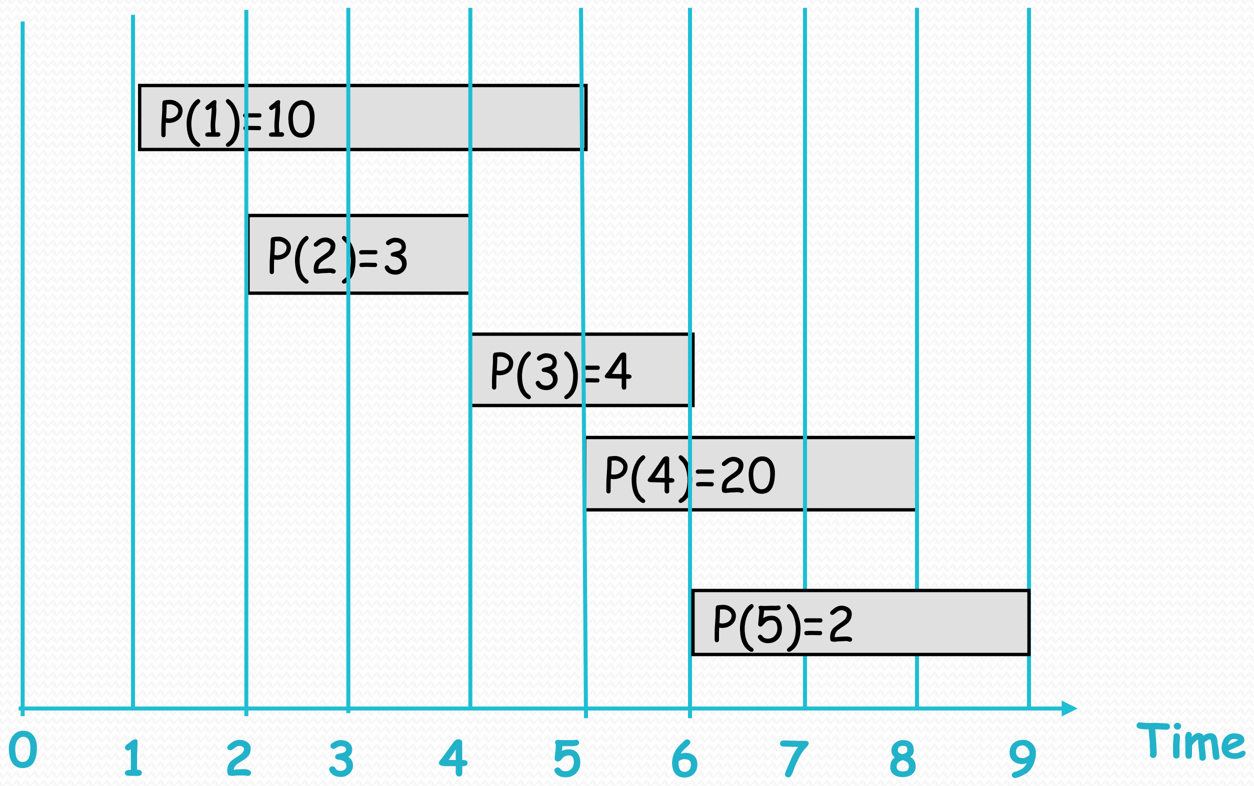
  (s_i , f_i , p_i) where,

  starting time s_i ,

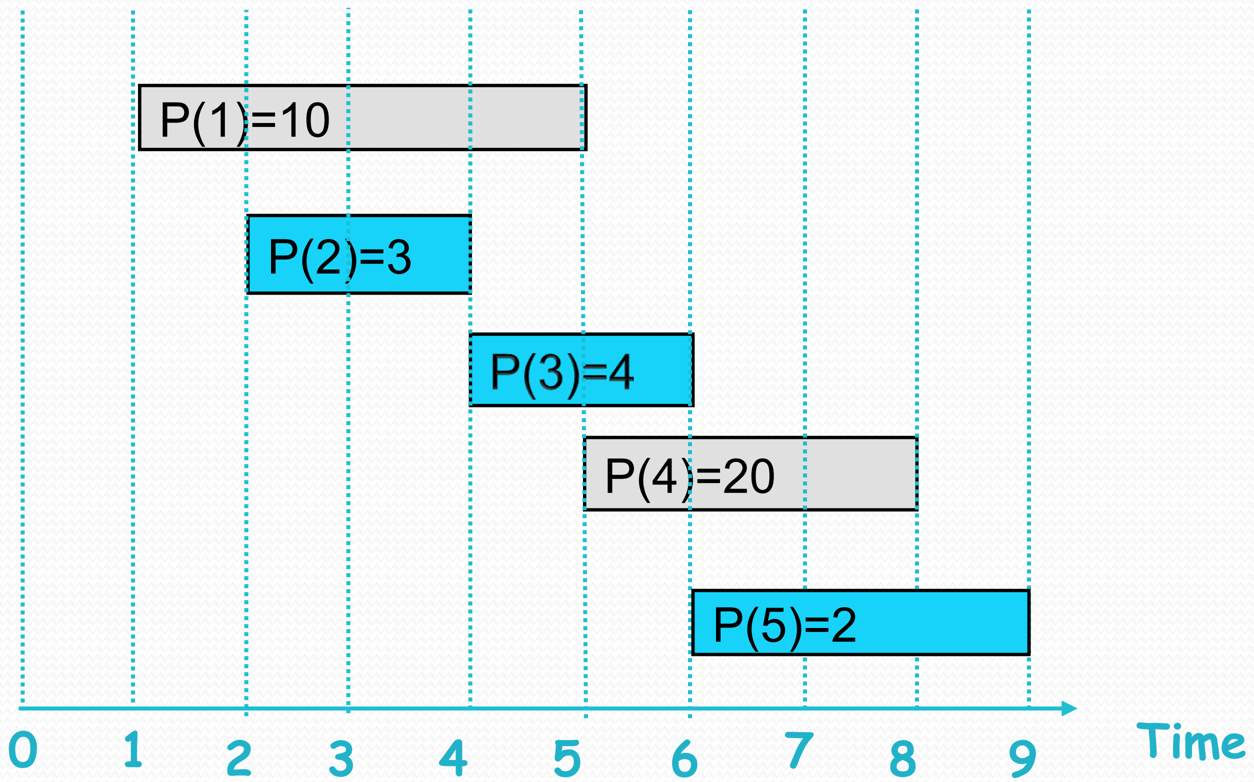
  finishing time f_i, and profit p_i
- Aim: Find an optimal schedule of compatible jobs that makes the maximum profit.
- Two jobs are said to be compatible if one finishes before the other one starts.
- Greedy approach: Choose the job which finishes first…..does not work.

i	S_i	F_i	P_i
2	2	4	3
1	1	5	10
3	4	6	4
4	5	8	20
5	6	9	2

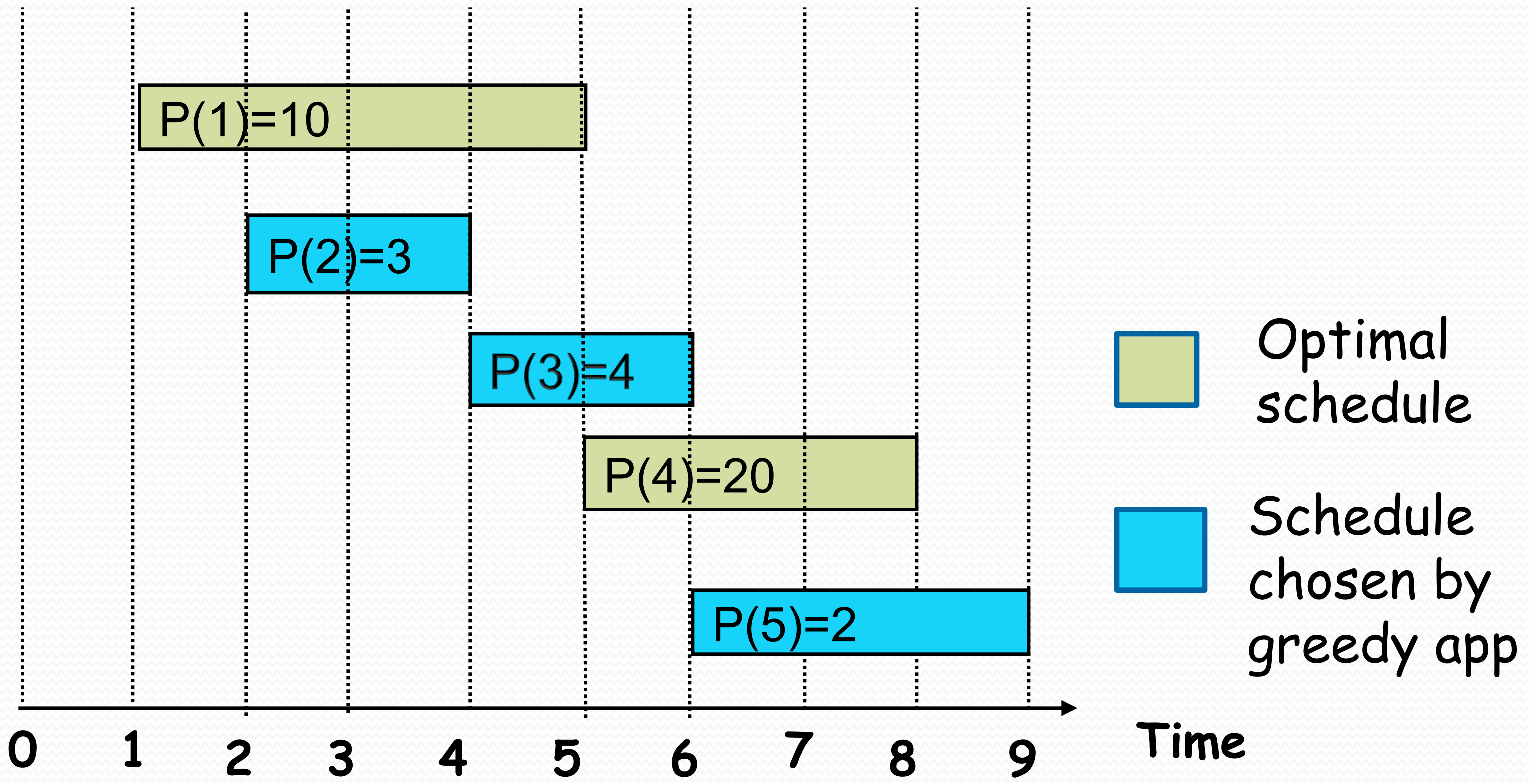
Weighted Interval Scheduling



Greedy Approach



Greedy does not work



Greedy approach takes job 2, 3 and 5 as best schedule and makes profit of 7. While optimal schedule is job 1 and job4 making profit of 30 (10+20). Hence greedy will not work

Recursive Solution

- Order the jobs in increasing order of their finishing times.
- Let m[j]= optimal schedule solution from the first j^{th} jobs,

  p_j=profit of j^{th} job.

  p[j] =largest index i<j , such that interval i and j are disjoint i.e. i is the rightmost interval that ends before j begins or the last interval compatible with j and is before j.

DP Solution for WIS

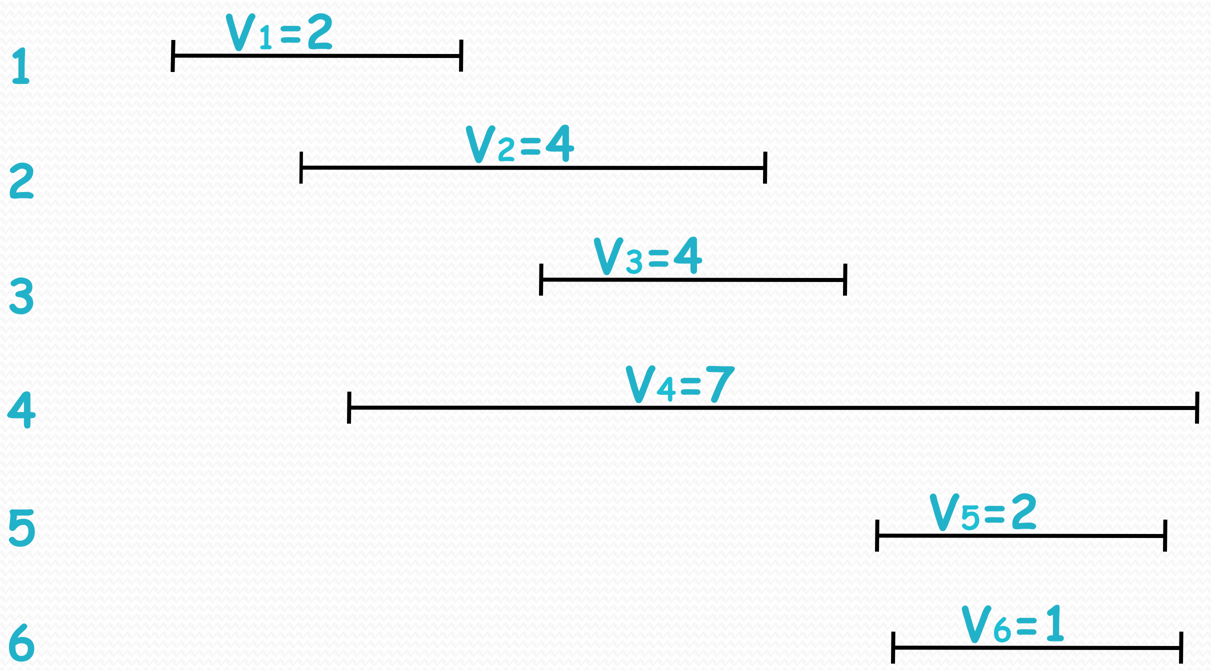
- Either j is in the optimal solution or it is not.
  - If it is, then $m[j] = p_j + m[p(j)]$
  - If it is not, then $m[j] = m[j-1]$
  - Thus, $m[j]= \max(p_j + m[p(j)], m[j-1])$
  - Some of the problems are solved several times leading to exponential time.

An example for Weighted Interval Scheduling problem

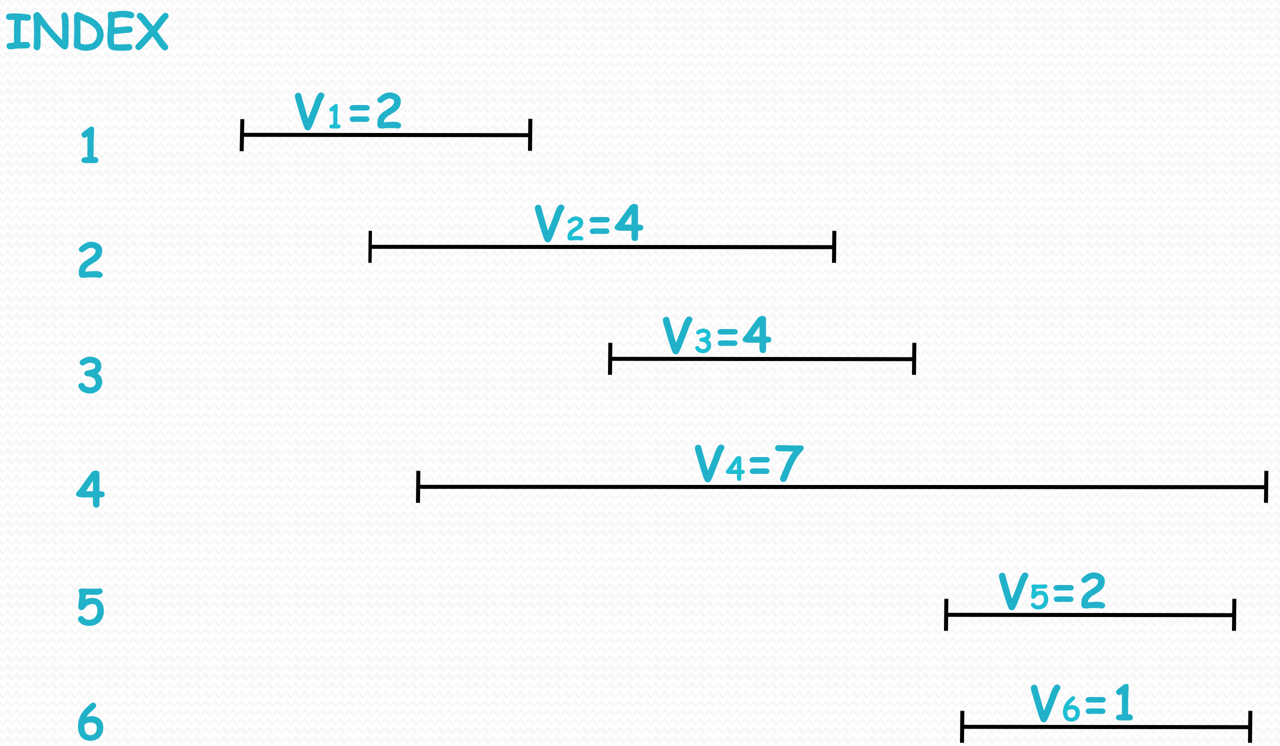
- A set of 6 jobs are given as follows -

Jobs (i)	Start time (s_i)	Finish time (f_i)	Weight (v_i)
1	1	3	2
2	2	5	4
3	3	6	4
4	2	9	7
5	6	8	2
6	6	8	1

INDEX



P[1]=0



P[1]=0

P[2]=0

INDEX

1

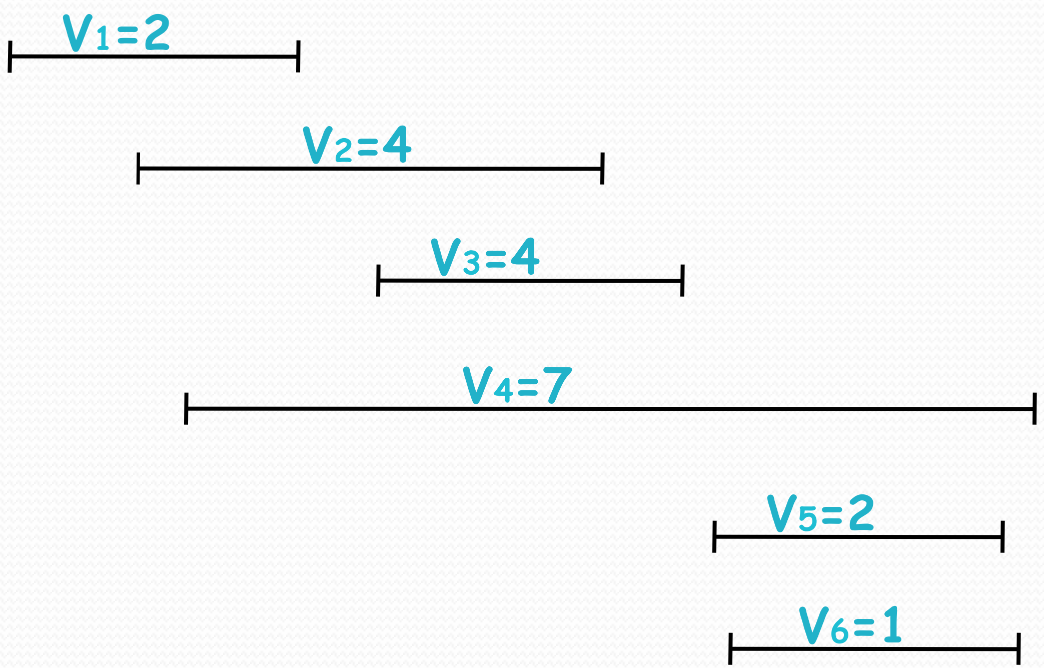
2

3

4

5

6



P[1]=0

P[2]=0

P[3]=1

INDEX

1

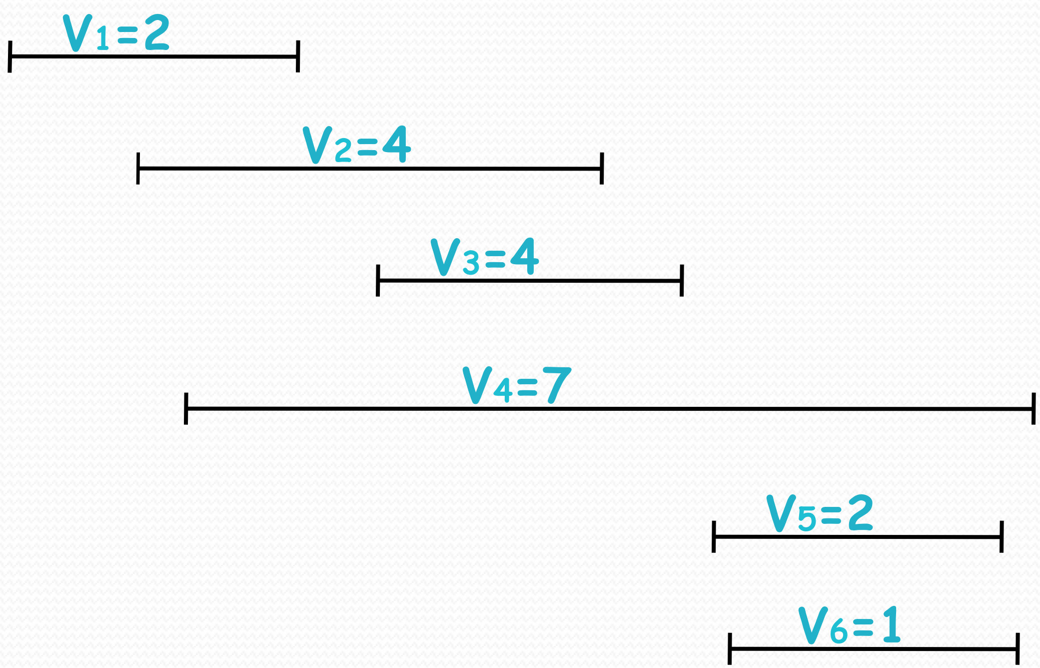
2

3

4

5

6

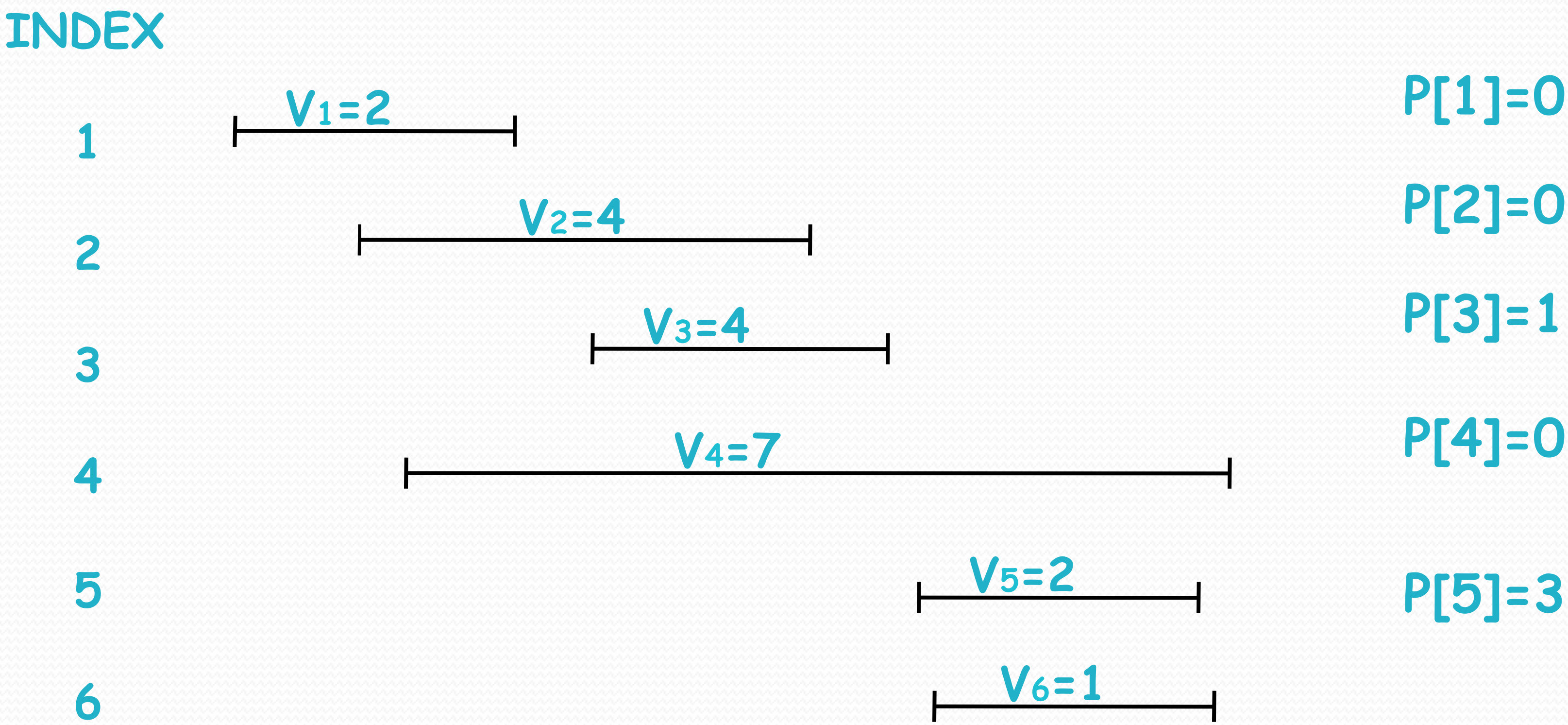


P[1]=0

P[2]=0

P[3]=1

P[4]=0


INDEX
1
2
3
4
5
6
$V_1=2$
$V_2=4$
$V_3=4$
$V_4=7$
$V_5=2$
$V_6=1$
P[1]=0
P[2]=0
P[3]=1
P[4]=0
P[5]=3

INT.
No.

1

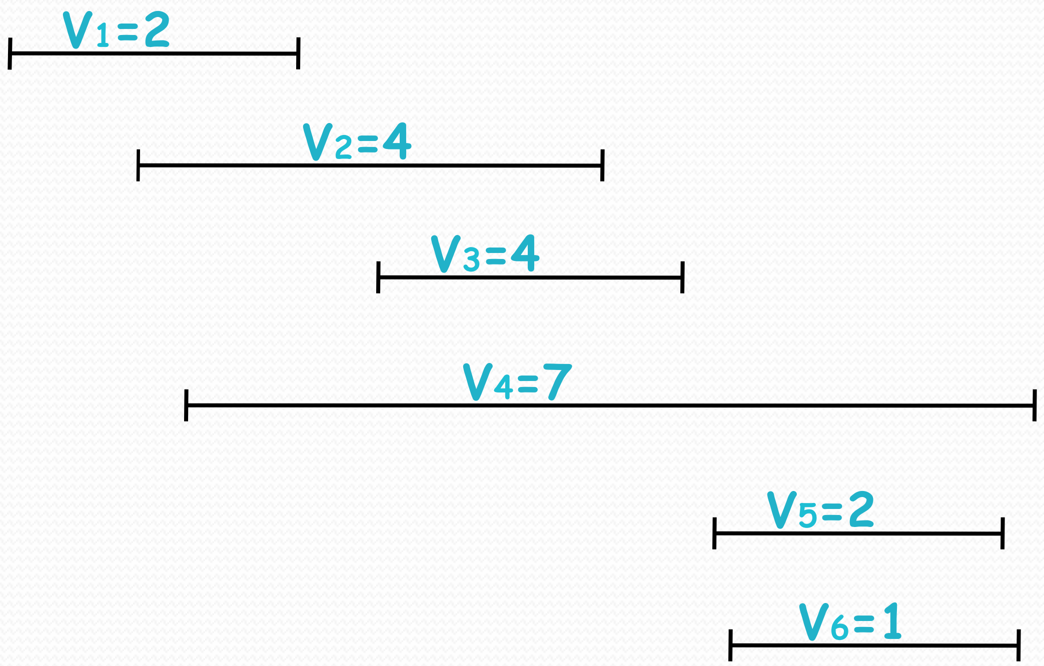
2

3

4

5

6



P[1]=0

P[2]=0

P[3]=1

P[4]=0

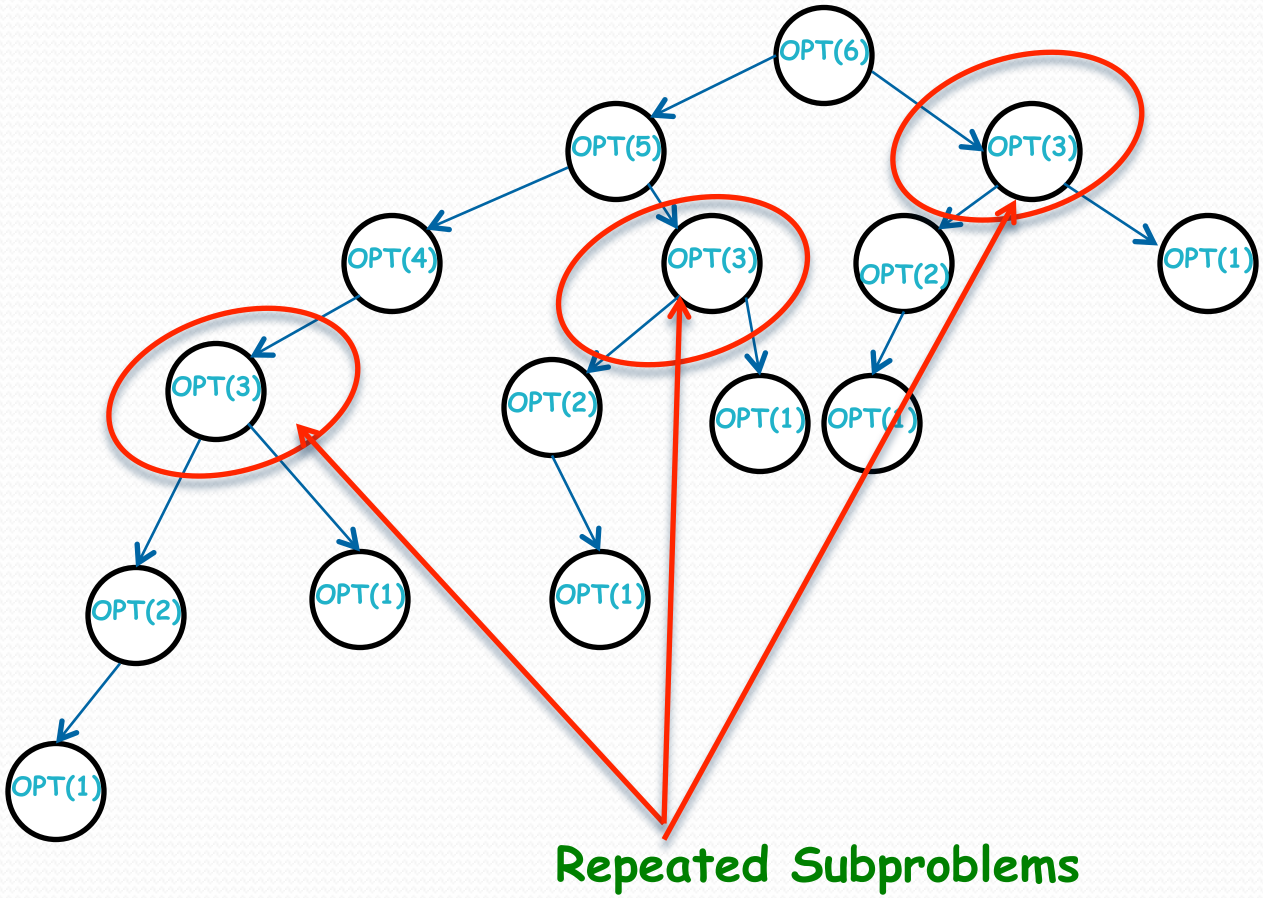
P[5]=3

P[6]=3

Recursive algorithm

```
Compute_Opt(j)
If j=0 then
  Return 0
Else
    m1 = Compute_Opt(j-1)
    m2 = Compute_Opt(p(j))
    If m2 + V_j > m1 then add j to the solution
    return max{m1, m2 + V_j}
Endif
```

Tree of recursion for WIS



Recursion: Time complexity

- Takes exponential time in worst case.
- Some branches are repeated in the tree due to which the total no. of calls made to compute_Opt will grow like Fibonacci numbers.
- In the tree, compute_Opt(3) is called repeatedly .

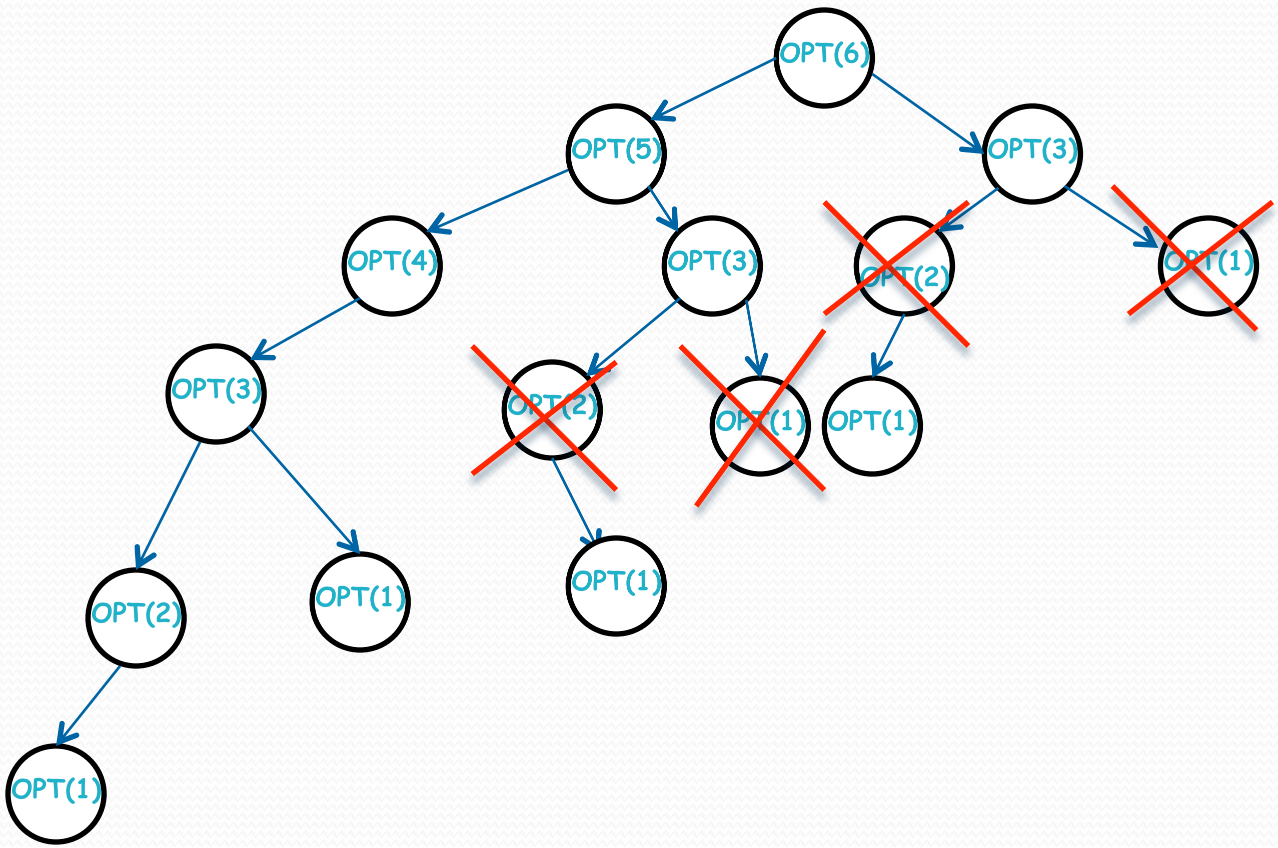
Memoizing the Recursion

Store and Re-USe

Memoization (Store and Reuse in recursion) – a Top Down Approach

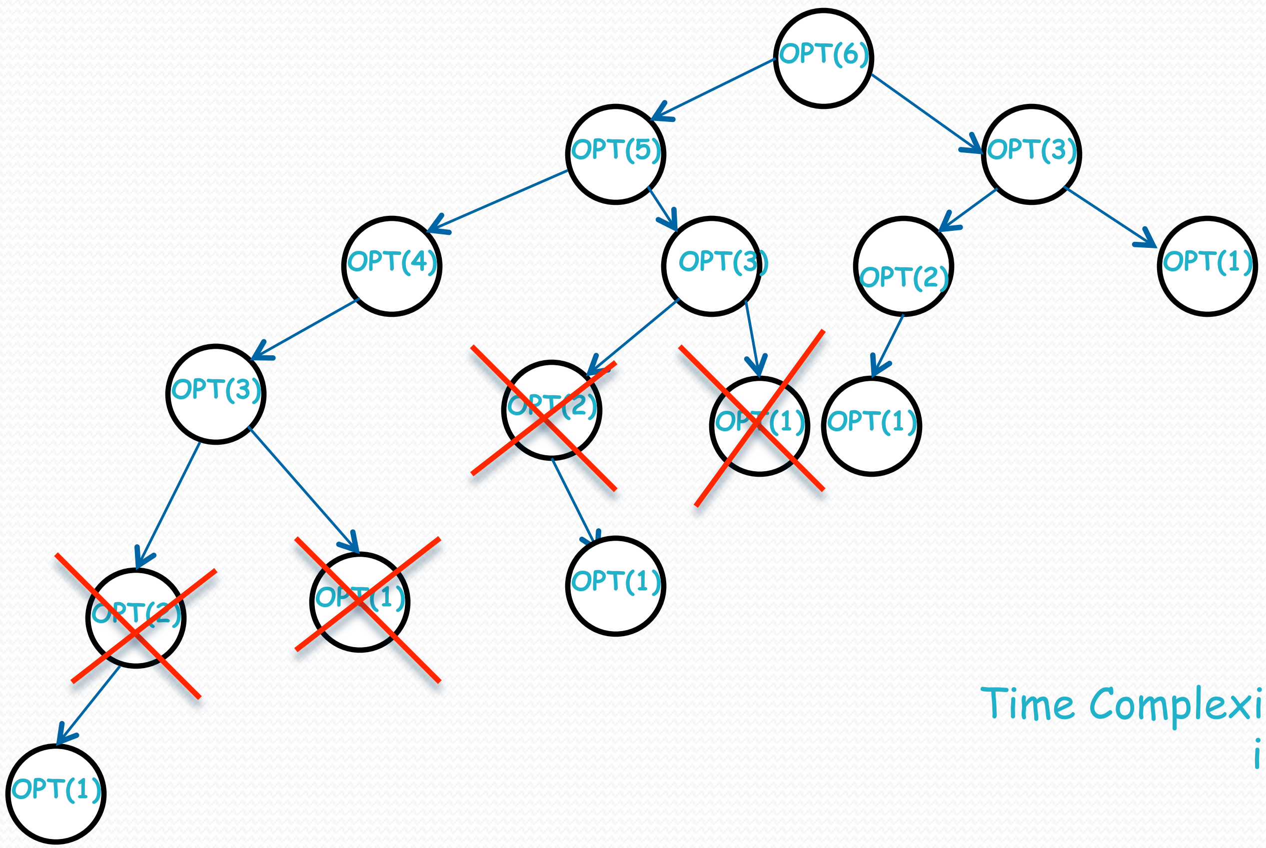
```
M-Compute-Opt(j)
  If j=0 then
     Return 0
  Else if M[j] is not empty then
     Return M[j]                    Reuse
  Else

     M[j]= max ( Vj + M-compute-   opt(p(j)) , M-Compute-
  opt(j-1))
     Return M[j]           Store
  Endif
```

Tree of recursion with Memoization (If recursive call to j-1 is executed before p(j))



It is easy to see that time spent on each call on p(j) is constant as it has been computed earlier and so its simply returns the pre-computed stored value. Clearly the time complexity is O(n)

Tree of recursion with Memoization (If recursive call to p(j) is executed before j-1)



Time Complexity Remains the Same
i.e. O(n)

Computing an Optimal Set of Intervals

- Maintain an additional array S so that S[i] contains an optimal set of intervals among {1,2,.....,i}.
- Additional Space - O(n).
- Alternatively, one can trace through array M to find the set of intervals in an optimal solution as shown in the next slide.

- Additional Time is O(n) in both the cases.

Algorithm to Find Optimal Set of Intervals

```
Find-Solution(j)
If j=0 then
    Output nothing
Else
  If Vj+ M[p(j)] >= M[j-1] then
    Output j together with the result of Find-Solution(p(j))
  Else
     Output the result of Find-Solution(j-1)
  Endif
Endif
```

Iterative Version

```
Iterative-Compute-Opt
  M[0]=0
  For j=1,2,.....,n
       M[j]=max (vj + M[p(j)] ,  M[j-1])
   Endfor
```

Calculating $m[j] = \max\{m[j-1], m[p(j)] + V_j\}$

	0						
m	0	1	2	3	4	5	6

Initially, no job belongs to optimal solution.

Max{0,2+0}

	0	2					
m	0	1	2	3	4	5	6

$V_1 = 2$

If job V_1 selected => m[1] = m[p(1)]+2 = 0+2 = 2

If not selected => m[1] = m[0] = 0

Max. is coming from second component so, Set pa(1) = p(1) =0.

Max{2,4+0}

m	0	1	2	3	4	5	6
	0	2	4				

$V_2 = 4$

If job V_2 selected => m[2] = m[p(2)]+4 = 0+4 = 4

If not selected => m[2] = m[1] = 2

Max. is coming from second component so, Set pa(2) = p(2) =0

Max{4,4+2}

	0	2	4	6			
m	0	1	2	3	4	5	6

$V_3 = 4$

If job V_3 selected => m[3] = m[p(3)]+4 = m[1]+4 = 2+4 = 6

If not selected => m[3] = m[2] = 4

Max. is coming from second component so, Set pa(3) = p(3) =1.

Max{6,7+0}

	0	2	4	6	7		
m	0	1	2	3	4	5	6

V_4 = 7
If job V_4 selected => m[4] = m[p(4)]+4 = m[0]+7 = 0+7 = 7
If not selected => m[4] = m[3] = 6
Max. is coming from second component so, Set pa(4) = p(4) =0.

Max{7,6+2}

	0	1	2	3	4	5	6
m	0	2	4	6	7	8	

$V_5 = 2$

If job V_5 selected => m[5] = m[p(5)]+2 = m[3]+2 = 6+2 = 8

If not selected => m[5] = m[4] = 7

Max. is coming from second component so, Set pa(5) = p(5) =3

Max{8,6+1}

	0	2	4	6	7	8	8
m	0	1	2	3	4	5	6

$V_6 = 1$

If job V_6 selected => m[6] = m[p(6)]+1 = m[3]+1 = 6+1 = 7

If not selected => m[6] =m[5] = 8

Max. is coming from first component so, Set pa(6) = 6 -1 =5

Reconstructing the Solution

Jobs — Input Weights

0 2 4 4 7 2 1

Array M

1	0	2					
2	0	2	4				

Jobs Input Weights

0 2 4 4 7 2 1

Array M

3

0	2	4	6			

4

0	2	4	6	7		

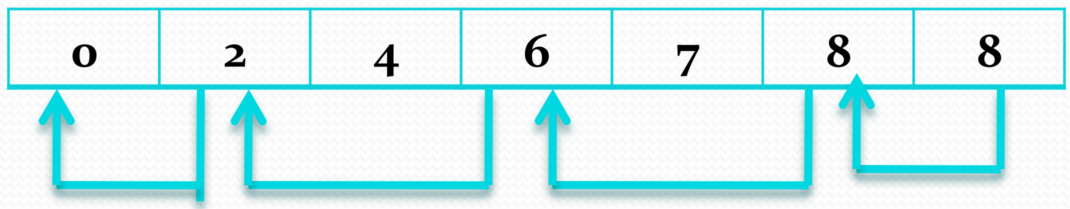
5

0	2	4	6	7	8	

0	2	4	6	7	8	8

Jobs Input Weights

6 0 2 4 4 7 2 1



Following the heads of the pointers, we get intervals {1,3,5} as the solution.

Iterative Version: Running Time

Clearly O(n) , since it runs for n iterations and spends constant time in each iteration.

Principles of DP

- Recursive Solution (Optimal Substructure Property)
- Overlapping Subproblems
- Total Number of Subproblems is polynomial
- A major ingredient of a DP solution is “ordering”.

WIS:

- Optimal Substructure Property : we just exhibited.
- Number of sub-problems: Polynomial (Linear).
- Ordering: Increasing order of finishing times.
- Iterative Version is nothing but the DP solution.

Multi-Way Choices: Matrix Chain Multiplication

Input: Let $A_{1,}$ $A_{2,}$ ………, A_n be n matrices of order ($d_{1,}$ d_2),($d_{2,}$ d_3),…… , ($d_{n,}d_{n+1}$) respectively.

Aim: Determine the order in which the matrices should be multiplied so as to minimize the number of multiplications.

Example: Let $A_{1,}$ $A_{2,}$ A_3 be 3 matrices of order (2 x 3), (3 x 4), (4 x 5) respectively.

($A_{1,}$ A_2) A_3 : (2 x 3 x 4)+ (2 x 4 x 5) = 24+ 40= 64

A_1 (A_2 , A_3): (3 x 4 x 5)+ (2 x 3x 5) = 60 + 30 = 90

So, we see that by changing the evaluation sequence cost of operation changes.

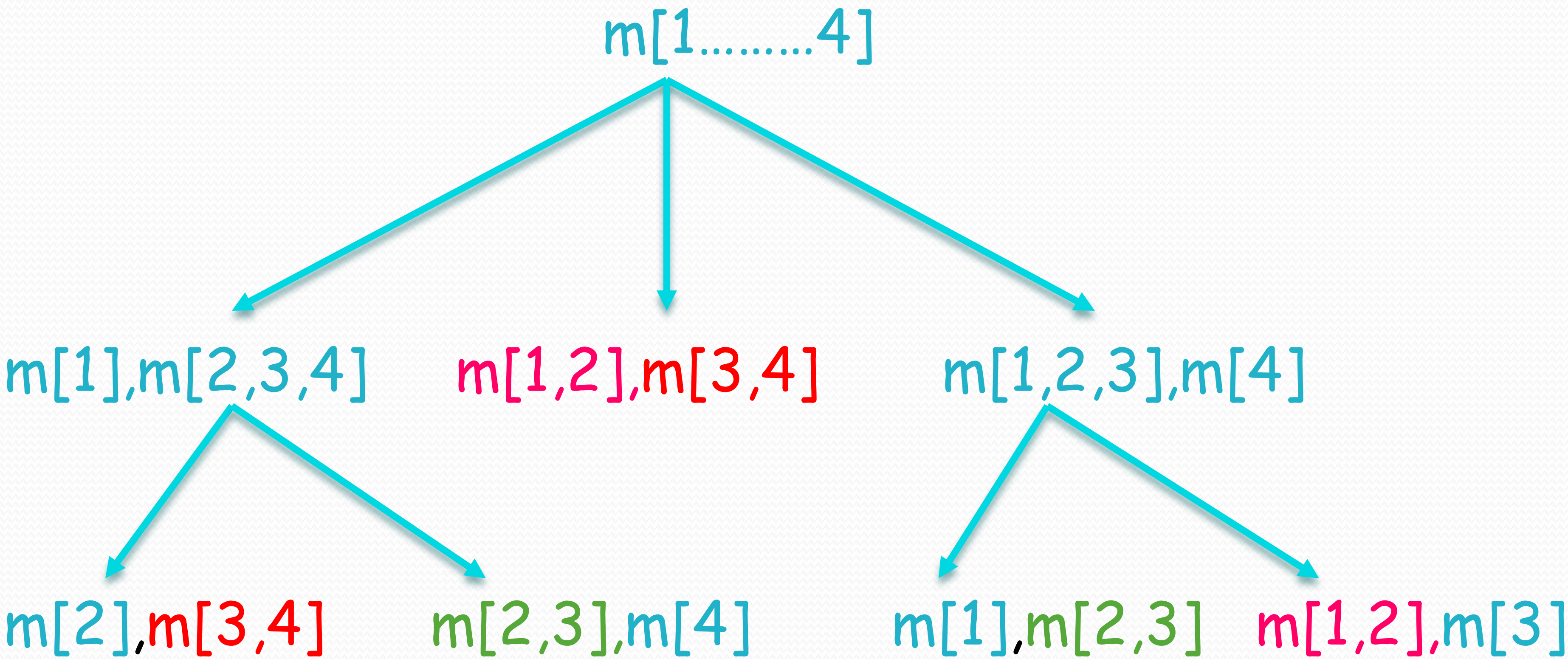
Optimal Substructure (Recursive Solution)

$$m[1 \ldots\ldots n] = \min_{k=i}^{n-1}\{m[1 \ldots\ldots\ldots k] + m[k+1 \ldots\ldots\ldots\ldots\ldots n] + d_1 d_{k+1} d_{n+1}\}$$

In general,

$$m[i \ldots\ldots j] = \min_{k=i}^{j-1}\{m[i \ldots\ldots\ldots k] + m[k+1 \ldots\ldots\ldots\ldots\ldots j] + d_i d_{k+1} d_{j+1}\}$$

Overlapping Subproblems



Number of sub-problems

M[i...............j]   i<=j

n options   n options

Number of subproblems= n^2

Example:

Matrix dimensions:

- A_1 : 3×5
- A_2 : 5×4
- A_3 : 4×2
- A_4 : 2×7
- A_5 : 7×3
- A_6 : 3×8

Problem of parenthesization

	A_1	A_2	A_3	A_4	A_5	A_6
A_1	0	60				
A_2	-	0	40			
A_3	-	-	0	56		
A_4	-	-	-	0	42	
A_5	-	-	-	-	0	168
A_6	-	-	-	-	-	0

$A_1 : 3 \times 5$
$A_2 : 5 \times 4$
$A_3 : 4 \times 2$
$A_4 : 2 \times 7$
$A_5 : 7 \times 3$
$A_6 : 3 \times 8$

$A_1 * A_2 = 3*5*4 = 60$
$A_2 * A_3 = 5*4*2 = 40$
$A_3 * A_4 = 4*2*7 = 56$
$A_4 * A_5 = 2*7*3 = 42$
$A_5 * A_6 = 7*3*8 = 168$

Problem of parenthesization

	A_1	A_2	A_3	A_4	A_5	A_6
A_1	0	60	70			
A_2	-	0	40	110		
A_3	-	-	0	56	66	
A_4	-	-	-	0	42	90
A_5	-	-	-	-	0	168
A_6	-	-	-	-	-	0

A_1 : 3 × 5
A_2 : 5 × 4
A_3 : 4 × 2
A_4 : 2 × 7
A_5 : 7 × 3
A_6 : 3 × 8

- A_1 _A_3= A_1 *A_2 *A_3 = min($(A_1.A_2).A_3$ =60+ 3*4*2 =84 or $A_1.(A_2 .A_3)$=40+ 3*5*2=70) =70
- A_2 _A_4= A_2 *A_3 *A_4 = min($(A_2.A_3).A_4$ =40+ 5*2*7=110 or A2.(A3.A4)=56+5*4*7=196)=110
- A_3 _A_5= A_3 *A_4 *A_5 = min($(A_3.A_4).A_5$ =56+4*7*3=140 or $A_3.(A_4.A_5)$=42+ 4*2*3 =66) = 66
- A_4 _A_6= A_4 *A_5 *A_6 = min((A_4 *A_5)*A_6= 42+2*3*8=90 or A_4 *(A_5 *A_6)=168+ 2*7*8=312)=90

Problem of parenthesization

	A_1	A_2	A_3	A_4	A_5	A_6
A_1	0	60	70 K=1	112 K=3	130 K=3	202 K=5
A_2	-	0	40	110 K=3	112 K=3	218 K=3
A_3	-	-	0	56	66 K=3	154 K=3
A_4	-	-	-	0	42	90 K=5
A_5	-	-	-	-	0	168
A_6	-	-	-	-	-	0

A_1 : 3×5
A_2 : 5×4
A_3 : 4×2
A_4 : 2×7
A_5 : 7×3
A_6 : 3×8

Running Time

$$m[i.......j] = \min_{k=i}^{j-1}\{m[i..........k] + m[k+1...............j] + d_i d_{k+1} d_{j+1}\}$$

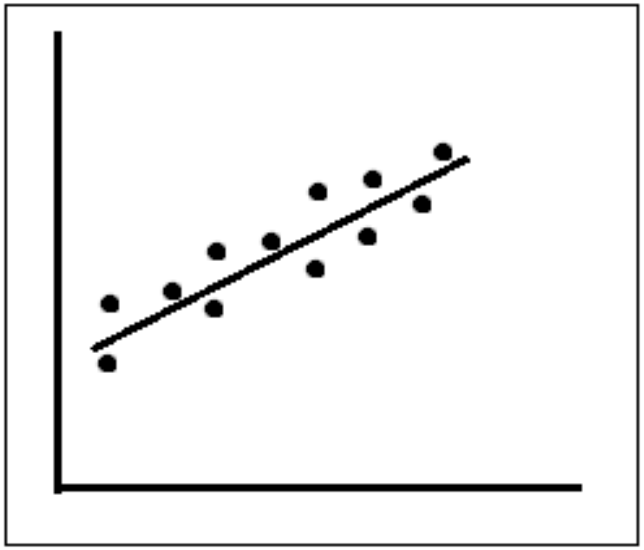
O(n^2) entries each takes O(n) time to compute

The running time of this procedure is O(n^3).

Segmented Least Squares: Another Example of Multi-way choices

Given a set P of n points in a plane denoted by $(x_1,y_1), (x_2,y_2), \ldots,$ (x_n,y_n) and line L defined by the equation $y = ax + b$, error of line L with respect to P is the sum of squares of the distances of these points from L. Aim is to determine a line that minimizes

$$\text{Error}(L, P) = \sum_{i=0}^{n}(y_i - b - ax_i)^2$$



A line of 'best fit'

Line of Best Fit

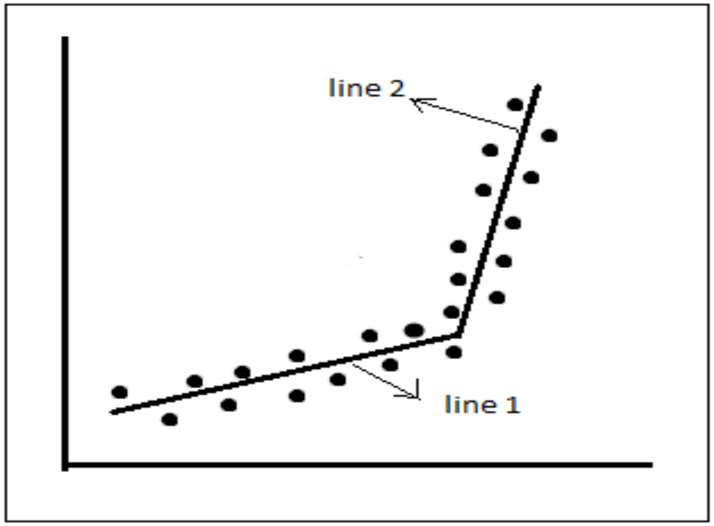
- Line of Best Fit is the one that minimizes this error. This can be computed in O(n) time using the following formula:

$a = (n(\sum x_i y_i) - (\sum y_i \sum x_i)) / (n \sum x_i^2 - (\sum x_i)^2)$

$b=(1/n)(\sum y_i - a \sum x_i)$

- The formula is obtained by differentiating the error wrt a and equating to zero and wrt b and equating to zero

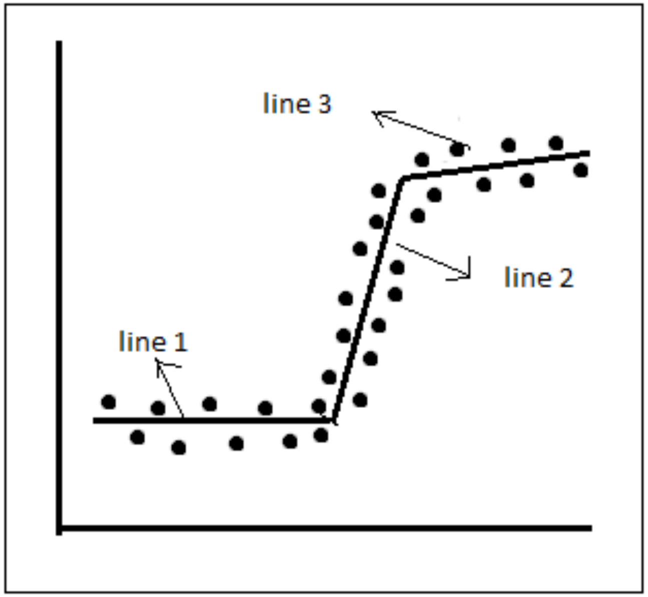
Consider the following scenario: Clearly approximating these points with a single line is not a good idea.



A set of points that lie approximately on two lines

Using two lines clearly gives a better approximation.

(contd.)



A set of points that lie approximately on three lines

In this case, 3 lines will give us a better approximation.

Segmented Least Square Error

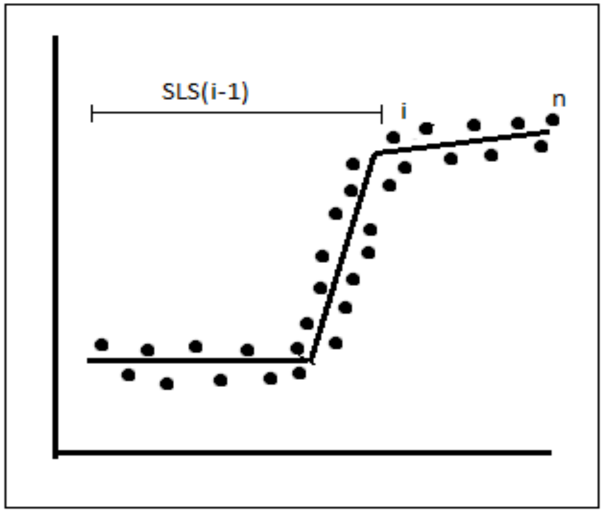
- In all these cases, we are able to tell the number of lines we must use by looking at the points. In general, we don't know this number. That is we don't know what is the minimum number of lines we must use to get a good approximation.

- Thus our aim becomes to minimize the number of lines that minimize the least square error.

Designing the Algorithm

We know that the last point p_n belongs to a single segment in the optimal partition and this segment begins at some point, say p_i .

Thus, if we know the identity of the last segment, then we can recursively solve the problem on the remaining points $p_1 \dots p_{i-1}$ as illustrated below:



Writing the recurrence

- If the last segment in the optimal is p_i , … , $p_{n,}$ then the value of optimal is,

$$SLS(n)= SLS(i-1) + c + e_{in}$$

where *c* is the cost of using the segment and e_{in} is the least square error of this segment.

- But since we don't know i, we'll compute it as follows:

$$SLS(n)= \min_i \{ SLS(i-1) + c + e_{ij} \}$$

- **General recurrence:**

For the sub-problem p_1, … , p_j ,

$$SLS(j)= \min_i \{ SLS(i-1) + c + e_{ij} \}$$

Time Analysis:

- e_{ij} values can be pre-computed in $O(n^3)$ time.
- Additional Time: n entries, each entry computes the minimum of at most n values each of which can be computed in constant time. Thus a total of $O(n^2)$.

Knapsack Problem: Adding a Variable

FRACTIONAL KNAPSACK PROBLEM

Given a set S of n items, with value $\mathbf{v}_i$ and weight $\mathbf{w}_i$ and a knapsack with capacity **W**.

Aim: Pick items with maximum total value but with weight at most W. You may choose fractions of items.

GREEDY APPROACH

Pick the items in the decreasing order of value per unit weight i.e. highest first.

Example

Knapsack Capacity: 50

Item 1	Item 2	Item 3
10	20	30
$v_i = 60$	$v_i = 100$	$v_i = 120$
$v_i/w_i = 6$	$v_i/w_i = 5$	$v_i/w_i = 4$

Example

Knapsack Capacity: 50

Item 2

20

$v_i = 100$

$v_i/w_i = 5$

Item 3

30

$v_i = 120$

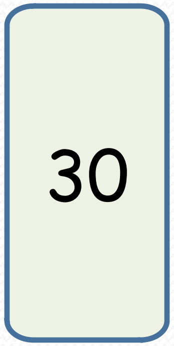
$v_i/w_i = 4$

10

60

Example

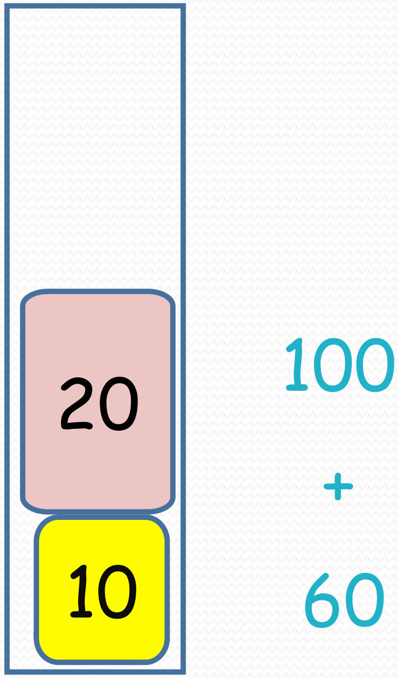
Item 3



$v_i = 120$

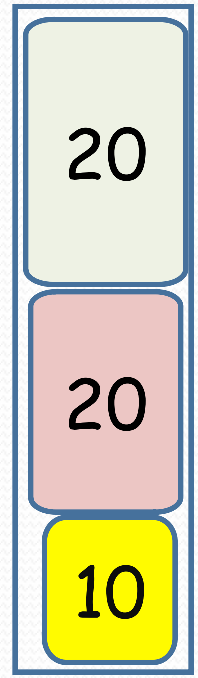
$v_i/w_i = 4$

Knapsack Capacity: 50



Example

Knapsack Capacity: 50



80

+

100

+

60

= 240

0-1 Kanpsack

Example to show that the above greedy approach will not work.

GREEDY APPROACH DOESN'T WORK FOR 0-1 KNAPSACK

-Counter Example

Knapsack Capacity: 50

Item 1	Item 2	Item 3
10	20	30
$v_i = 60$	$v_i = 100$	$v_i = 120$
$v_i/w_i = 6$	$v_i/w_i = 5$	$v_i/w_i = 4$

Counter Example

Knapsack Capacity: 50

Item 2

20

v_i = 100

v_i/w_i = 5

Item 3

30

v_i = 120

v_i/w_i = 4

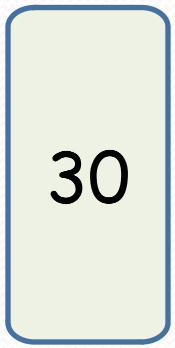
10

60

Counter Example

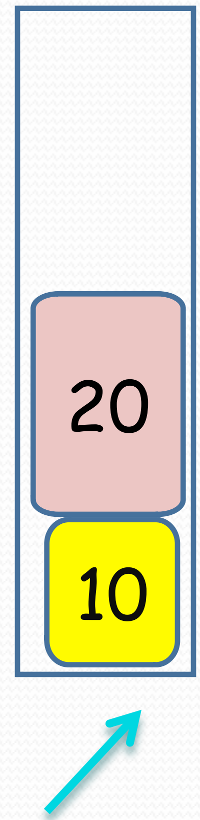
Knapsack Capacity: 50

Item 3



$v_i = 120$

$v_i/w_i = 4$



100

+

60

=160

Suboptimal

DP Solution for 0-1KS : Adding a Variable

- Arrange the elements in any arbitrary order.
- Let OPT(n, W) denote the value of optimal solution with n objects and capacity W.
- Working on similar lines as in WIS and SLS,
- If n does not belong to OPT, then OPT(n, W) = OPT(n-1, W)
- If n belongs to OPT then ?
  - Which subproblem to consider?
  - OPT(n-1, W)?
  - But OPT(n-1, W) denote the optimal solution with knapsack capacity W. If n belongs to OPT then we have reduced capacity in our knapsack for the smaller subproblems. i.e. we need to consider OPT($n - 1$, $W - w_n$)

- If we knew the exact value (say K) of OPT knapsack, then to compute OPT(n, K) we know that for n-1 objects, we have to solve exactly two problems : OPT(n-1, K) and OPT(n -1, K – w_n) and we would solve a linear number of subproblems in all.
- But obviously, we don't know that so we make a guess w for K (i.e. try out all possible values for K) and solve the problem for w. So we add a dimension(/variable) to our problem.
- Similarly while dealing with objects {1 … i} we need to solve OPT(i-1, K) and OPT(i -1, K – w_i).
- Thus in general we need to define OPT(i, w) for every i $\leq$ n and every w $\leq$ W.

DP Solution for 0-1KS : Adding a Variable

- Let m[i,w] be the optimal value obtained when considering objects {1 … i} and filling a knapsack of capacity w
  - $m[0,w] = 0$
  - $m[i,0] = 0$
  - $m[i,w] = m[i-1,w]$ if $w_i > w$
  - $m[i,w] = \max\{m[i-1, w-w_i] + v_i , m[i-1, w]\}$ if $w_i <= w$

Example

- n = 4
- W = 5
- Elements (weight, value):

  (2,3), (3,4), (4,5), (5,6)

(2,3), (3,4), (4,5), (5,6)

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

(2,3), (3,4), (4,5), (5,6)

As $w < w_1$; m[1,1] = m[1-1,1] = m[0,1]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

(2,3), (3,4), (4,5), (5,6)

As $w<w_2$; m[2,1] = m[2-1,1] = m[1,1]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0	0				
3	0					
4	0					

(2,3), (3,4), (4,5), (5,6)

As $w<w_3$; m[3,1] = m[3-1,1] = m[2,1]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0	0				
3	0	0				
4	0					

(2,3), (3,4), (4,5), (5,6)

As $w < w_4$; m[4,1] = m[4-1,1] = m[3,1]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0	0				
3	0	0				
4	0	0				

(2,3), (3,4), (4,5), (5,6)

As $w >= w_1$; m[1,2] = max{m[1-1,2],m[1-1,2-2]+3}
=max{ 0,0+3}

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0	0				
3	0	0				
4	0	0				

(2,3), (3,4), (4,5), (5,6)

As $w < w_2$; m[2,2] = m[2-1,2] = m[1,2]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0	0	3			
3	0	0				
4	0	0				

(2,3), (3,4), (4,5), (5,6)

As $w < w_3$; m[3,2] = m[3-1,2] = m[2,2]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0	0	3			
3	0	0	3			
4	0	0				

(2,3), (3,4), (4,5), (5,6)

As $w < w_4$; $m[4,2] = m[4-1,2] = m[3,2]$

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0	0	3			
3	0	0	3			
4	0	0	3			

(2,3), (3,4), (4,5), (5,6)

As $w >= w_1$; $m[1,3] = \max\{m[1-1,3], m[1-1,3-2]+3\}$

$= \max\{0, 0+3\}$

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0	0	3			
3	0	0	3			
4	0	0	3			

(2,3), (3,4), (4,5), (5,6)

As $w>=w_2$; $m[2,3] = \max\{m[2-1,3],m[2-1,3-3]+4\}$

$=\max\{\ 3,0+4\}$

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0	0	3	4		
3	0	0	3			
4	0	0	3			

(2,3), (3,4), (4,5), (5,6)

As $w < w_3$; m[3,3] = m[3-1,3] = m[2,3]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0	0	3	4		
3	0	0	3	4		
4	0	0	3			

(2,3), (3,4), (4,5), (5,6)

As $w < w_4$; m[4,3] = m[4-1,3] = m[3,3]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0	0	3	4		
3	0	0	3	4		
4	0	0	3	4		

(2,3), (3,4), (4,5), (5,6)

As $w >= w_1$; $m[1,4] = \max\{m[1-1,4], m[1-1,4-2]+3\}$

$= \max\{0, 0+3\}$

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0	0	3	4		
3	0	0	3	4		
4	0	0	3	4		

(2,3), (3,4), (4,5), (5,6)

As $w \geq w_2$; m[2,4] = max{m[2-1,4],m[2-1,4-3]+4}

=max{3,0+4}

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0	0	3	4	4	
3	0	0	3	4		
4	0	0	3	4		

(2,3), (3,4), (4,5), (5,6)

As $w >= w_3$; m[3,4] = max{m[3-1,4],m[3-1,4-4]+5}

=max{ 4,0+5}

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0	0	3	4	4	
3	0	0	3	4	5	
4	0	0	3	4		

(2,3), (3,4), (4,5), (5,6)

As $w < w_4$; m[4,4] = m[4-1,4] = m[3,4]

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0	0	3	4	4	
3	0	0	3	4	5	
4	0	0	3	4	5	

(2,3), (3,4), (4,5), (5,6)

As $w >= w_1$; $m[1,5] = \max\{m[1-1,5], m[1-1,5-2]+3\}$

$= \max\{0, 0+3\}$

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0	0	3	4	5	
4	0	0	3	4	5	

(2,3), (3,4), (4,5), (5,6)

As $w >= w_2$; m[2,5] = max{m[2-1,5],m[2-1,5-3]+4}
=max{ 3,3+4}

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0	0	3	4	5	

(2,3), (3,4), (4,5), (5,6)

As $w \geq w_3$; m[3,5] = max{m[3-1,5],m[3-1,5-4]+5}
=max{ 7,0+5}

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

(2,3), (3,4), (4,5), (5,6)

As $w >= w_4$; m[4,5] = max{m[4-1,5],m[4-1,5-5]+6}
=max{ 7,0+6}

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

(2,3), (3,4), (4,5), (5,6)

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Obtaining a Solution

- As before we backtrack to obtain the optimal solution picking the objects that provided us the maximum profit/value.

- Backtracking gives us {2, 1} as the final solution.

Running Time

- nW cells
- Constant time to compute each cell
- Total Time = O(nW)
- Pseudo-polynomial Algorithm

Pseudo-polynomial algorithm

- An algorithm that runs in time polynomial in the numeric value of the input (which is actually exponential in the size of the input – the number of digits).

- DP solution to 0-1 Knapsack is pseudo-polynomial as it is polynomial in W, the capacity (one of the inputs) of the Knapsack.

- Note that polynomial in 'n' is fine as 'n' is not an input parameter, it is only a symbol we have used for our convenience to denote the number of objects.

Strongly Polynomial Algorithms

- An algorithm is said to be strongly polynomial if its running time is polynomial in the input size and does not depend on any value of the input.

- It will be shown later that 0-1 Knapsack is actually an NP-hard problem and is unlikely to possess a strongly polynomial time solution.

Alternative Definitions

- Pseudo-polynomial Algorithm: An algorithm that runs in time polynomial in the input size when the input is represented as a string of 1's (instead of 0's and 1's) is called a Pseudo-polynomial Algorithm.

- Strongly-polynomial Algorithm: An algorithm that runs in time polynomial in the input size when the input is represented as a string of 0's and 1's is called a Strongly-polynomial Algorithm.

Weakly/Strongly NP hard problems

- An NP hard problem with a known pseudo-polynomial time solution is said to be weakly NP hard...Thus 0-1 knapsack is weakly NP-hard.
- An NP- hard problem for which it has been proved that it cannot admit a pseudo-polynomial solution unless P= NP is said to be strongly NP hard.
- In our next session, we'll studying NP-hard problems. But most of the times, we do not categorize them as weak/strong NP-hard unless we give a pseudo-polynomial algorithm for a problem.

Sequence Alignment

String Similarity

- occurrence
- occurrence

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatch, 3 gaps

- PROBLEM: Given two strings $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_m$
- GOAL: Find an alignment of minimum cost, where δ is the cost of a gap and α is the cost of a mismatch.
- Let OPT(i,j)=min cost of aligning strings $X = x_1 x_2 \ldots x_i$ and $Y = y_1 y_2 \ldots y_j$

4 cases for constructing optimal solution

- Case1: x_i is aligned with y_j in OPT and they match

__x_i
__y_j
match

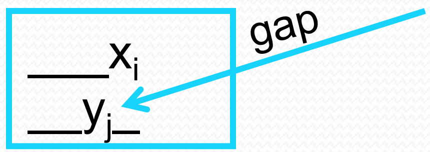
OPT(i,j)=min cost of aligning $x_1x_2.......x_{i-1}$ and $y_1y_2........y_{j-1}$

- Case 2: x_i is aligned with y_j in OPT and they don't match

__x_i
__y_j
mismatch

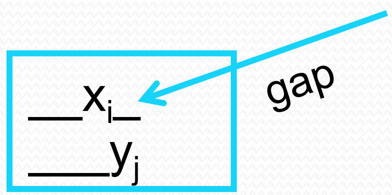
OPT(i,j)=$\alpha_{x_i y_j}$ + min cost of aligning $x_1x_2.......x_{i-1}$ and $y_1y_2........y_{j-1}$

- Case 3: i^{th} position of X is not matched.



$OPT(i,j) = \delta$ + min cost of aligning $x_1\, x_2 \,.\,.\,.\, x_{i-1}$ and $y_1\, y_2 \,.\,.\,.\, y_j$

- Case 4: j^{th} position of Y is not matched.



$OPT(i,j) = \delta$ + min cost of aligning $x_1\, x_2 \,.\,.\,.\, x_i$ and $y_1\, y_2 \,.\,.\,.\, y_{j-1}$

Thus,

$$
OPT(i,j) = \min\{ \\
OPT(i-1,j-1), \\
OPT(i-1,j) + \delta, \\
OPT(i,j-1) + \delta, \\
OPT(i-1,j-1) + \alpha_{x_i y_j} \\
\}
$$

Example

Let X = naem,

Y = name,

$\alpha_{x_i y_j}$:mismatch cost = 1

δ :gap cost = 1

If i and j match,

OPT(i,j)=min{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1}

else

OPT(i,j)=min{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1}

		n	a	m	e
n					
a					
e					
m					

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n					
a					
e					
m					

$OPT(0,j)=0$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n	0				
a	0				
e	0				
m	0				

$OPT(0,j)=0$

$OPT(i,0)=0$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n	0	0			
a	0				
e	0				
m	0				

$$
\begin{aligned}
OPT(1,1)=\min(&OPT(0,0),\\
&OPT(0,1)+1,\\
&OPT(1,0)+1)\\
=&OPT(0,0)\\
=&0
\end{aligned}
$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n	0	0	1		
a	0				
e	0				
m	0				

$$
\begin{aligned}
OPT(1,2)=\min(&OPT(0,1)+1,\\
&OPT(0,2)+1,\\
&OPT(1,1)+1)\\
=&OPT(1,1)+1\\
=&0+1\\
=&1
\end{aligned}
$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n	0	0	1	1	
a	0				
e	0				
m	0				

$$\begin{aligned}OPT(1,3)=\min(&OPT(0,2)+1,\\ &OPT(0,3)+1,\\ &OPT(1,2)+1)\\ =&OPT(0,2)+1\\ =&0+1\\ =&1\end{aligned}$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n	0	0	1	1	1
a	0				
e	0				
m	0				

$OPT(1,4)=\min(OPT(0,3)+1,$
$OPT(0,4)+1,$
$OPT(1,3)+1)$
$=OPT(0,3)+1$
$=0+1$
$=1$

If i and j match,

OPT(i,j)=min{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1}

else

OPT(i,j)=min{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1}

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1			
e	o				
m	o				

OPT(2,1)=min(OPT(1,0)+1,
OPT(1,1)+1,
OPT(2,0)+1)
=OPT(1,0)+1
=0+1
=1

If i and j match,

$$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$$

else

$$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0		
e	o				
m	o				

$$\begin{aligned}OPT(2,2)=\min(&OPT(1,1),\\ &OPT(1,2)+1,\\ &OPT(2,1)+1)\\ =&OPT(1,1)\\ =&0\end{aligned}$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	0	0	0	0	0
n	0	0	1	1	1
a	0	1	0	1	
e	0				
m	0				

$$\begin{aligned} OPT(2,3)=\min(&OPT(1,2)+1,\\ &OPT(1,3)+1,\\ &OPT(2,2)+1)\\ =&OPT(2,2)+1\\ =&0+1\\ =&1 \end{aligned}$$

If i and j match,

$$OPT(i,j)=\min\{OPT(i-1,j-1), OPT(i-1,j)+1, OPT(i,j-1)+1\}$$

else

$$OPT(i,j)=\min\{OPT(i-1,j-1)+1, OPT(i-1,j)+1, OPT(i,j-1)+1\}$$

		n	a	m	e
	0	0	0	0	0
n	0	0	1	1	1
a	0	1	0	1	2
e	0				
m	0				

$$\begin{aligned} OPT(2,4) &= \min(OPT(1,3)+1, \\ &\qquad OPT(1,4)+1, \\ &\qquad OPT(2,3)+1) \\ &= OPT(2,3)+1 \\ &= 1+1 \\ &= 2 \end{aligned}$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1), OPT(i-1,j)+1, OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1, OPT(i-1,j)+1, OPT(i,j-1)+1\}$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1			
m	o				

$OPT(3,1)=\min(OPT(2,0)+1,$
$OPT(2,1)+1,$
$OPT(3,0)+1)$
$=OPT(3,0)+1$
$=0+1$
$=1$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1		
m	o				

$OPT(3,2)=\min(OPT(2,1)+1,$
$OPT(2,2)+1,$
$OPT(3,1)+1)$
$=OPT(2,2)+1$
$=0+1$
$=1$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1	1	
m	o				

$$\begin{aligned}OPT(3,3)=\min(&OPT(2,2)+1,\\ &OPT(2,3)+1,\\ &OPT(3,2)+1)\\ =&OPT(2,2)+1\\ =&0+1\\ =&1\end{aligned}$$

If i and j match,

OPT(i,j)=min{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1}

else

OPT(i,j)=min{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1}

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1	1	1
m	o				

OPT(3,4)=min(OPT(2,3),
OPT(2,4)+1,
OPT(3,3)+1)
=OPT(2,3)
=1

If i and j match,

OPT(i,j)=min{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1}

else

OPT(i,j)=min{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1}

		n	a	m	e
	0	0	0	0	0
n	0	0	1	1	1
a	0	1	0	1	2
e	0	1	1	1	1
m	0	1			

OPT(4,1)=min(OPT(3,0)+1,
OPT(3,1)+1,
OPT(4,0)+1)
=OPT(4,0)+1
=0+1
=1

If i and j match,

$$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$$

else

$$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1	1	1
m	o	1	2		

$$OPT(4,2)=\min(OPT(3,1)+1,\ OPT(3,2)+1,\ OPT(4,1)+1)$$
$$=OPT(4,1)+1$$
$$=1+1$$
$$=2$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1	1	1
m	o	1	2	1	

$OPT(4,3)=\min(OPT(3,2),$
$OPT(3,3)+1,$
$OPT(4,2)+1)$
$=OPT(3,2)$
$=1$

If i and j match,

$$OPT(i,j)=\min\{OPT(i-1,j-1), OPT(i-1,j)+1, OPT(i,j-1)+1\}$$

else

$$OPT(i,j)=\min\{OPT(i-1,j-1)+1, OPT(i-1,j)+1, OPT(i,j-1)+1\}$$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1	1	1
m	o	1	2	1	2

$$\begin{aligned} OPT(4,4) &= \min(OPT(3,3)+1, \\ &\qquad OPT(3,4)+1, \\ &\qquad OPT(4,3)+1) \\ &= OPT(3,3)+1 \\ &= 1+1 \\ &= 2 \end{aligned}$$

If i and j match,

$OPT(i,j)=\min\{OPT(i-1,j-1),OPT(i-1,j)+1,OPT(i,j-1)+1\}$

else

$OPT(i,j)=\min\{OPT(i-1,j-1)+1,OPT(i-1,j)+1,OPT(i,j-1)+1\}$

		n	a	m	e
	o	o	o	o	o
n	o	0	1	1	1
a	o	1	0	1	2
e	o	1	1	1	1
m	o	1	2	1	2

Solution:

- Trace from bottom right
- Diagonal elements having value 0 is the optimal alignment.

So,

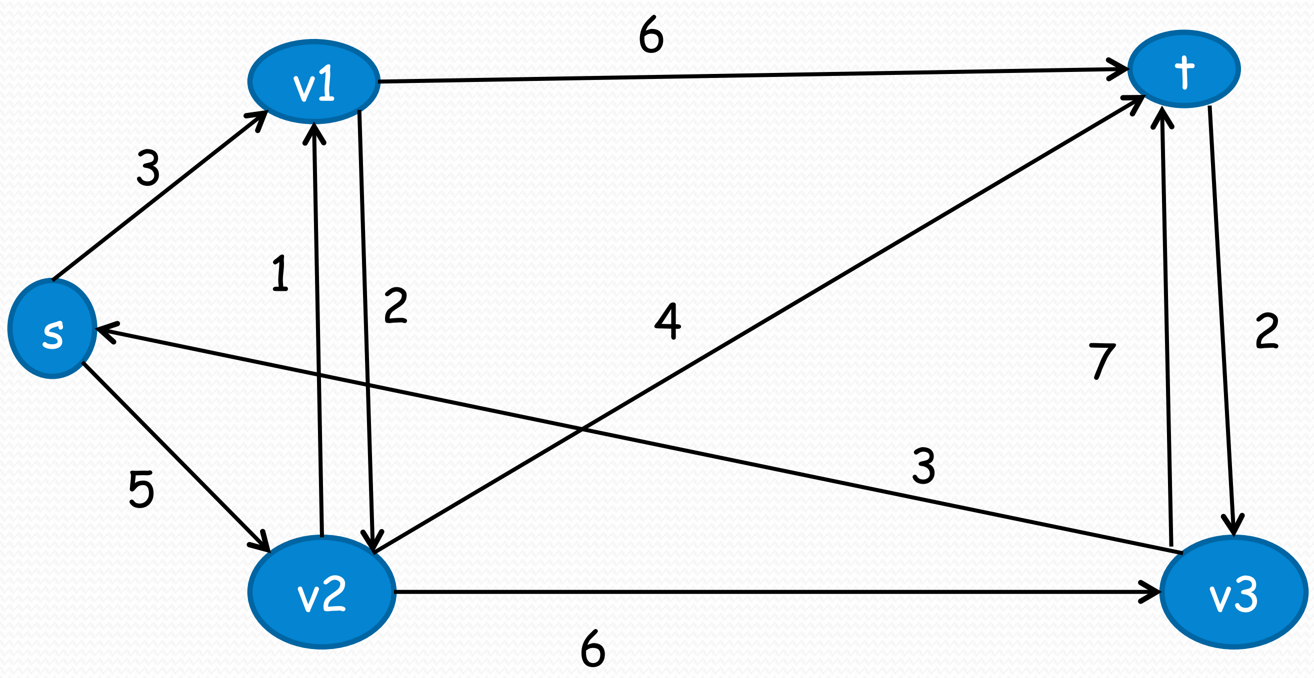
Optimal alignment:

name

naem

SHORTESH PATH PROBLEM: Yet Another Example of Multi-way Choices

The problem: Given a directed graph G with weights on edges, vertices s and t, find a shortest path from s to t.



Optimal Substructure (Recursion)

Apply the same argument as earlier:

Ask the same question as earlier:

How does OPT reach t?

Let w_1, w_2, w_3 are the in-neighbours of t. Then OPT must have taken a route through one of them. We don't know which. So,

we compute shortest path from s to each of w_i and then take the edge wi to t and then take the best amongst all of them..

Optimal Substructure (Recursion) contd..

Alternatively, one could ask

How does OPT start from s?

Let u_1, u_2, u_3 are the out-neighbours of s. Then OPT must have taken a route through one of them. We don't know which. So,

we take the edge s to u_i and compute shortest path from u_i to t for every u_i and then take the best amongst all of them.

Recurrence relation

If OPT(v) denotes the shortest path from v to t, then we are looking for

$OPT(s) = \min(OPT(u_i) + C(s, u_i))$

But this does not work because $OPT(u_i)$ may change in the next iteration (a shorter path using more number of edges). So we also need to add a variable which denotes the length of the path.

Recurrence relation

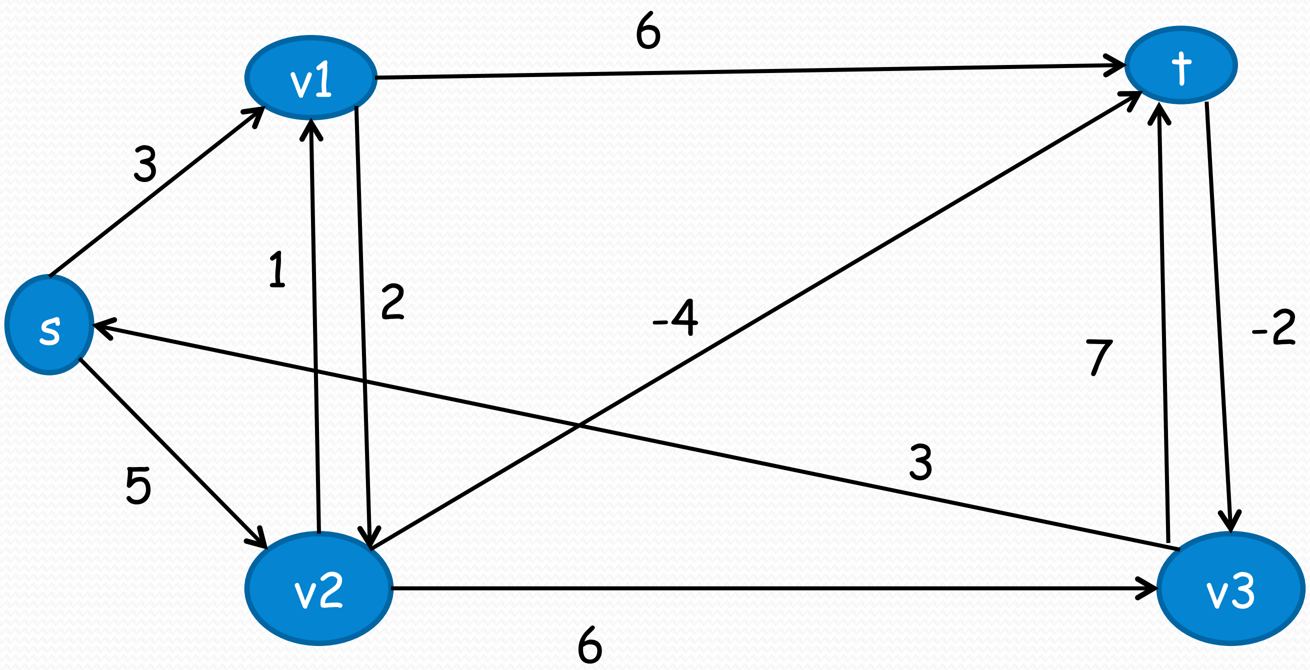
$$OPT(i, v) = \min(OPT(i-1, v), \min_{w \in V}(OPT(i-1, w) + C_{vw}))$$

where

OPT(i,v) denotes the minimum cost of a v-t path using at most i edges.

C_{vw} is the cost of an edge from vertex v to vertex w

Problem: To find shortest path from s to t



	0	1	2	3	4
t	0				
s	∞				
v1	∞				
v2	∞				
v3	∞				

	0	1	2	3	4
t	0	0			
s	∞	∞			
v1	∞				
v2	∞				
v3	∞				

	0	1	2	3	4
t	0	0			
s	∞	∞			
v1	∞	6			
v2	∞				
v3	∞				

	0	1	2	3	4
t	0	0			
s	∞	∞			
v1	∞	6			
v2	∞	-4			
v3	∞				

	0	1	2	3	4
t	0	0			
s	∞	∞			
v1	∞	6			
v2	∞	-4			
v3	∞	7			

	0	1	2	3	4
t	0	0	0		
s	∞	∞			
v1	∞	6			
v2	∞	-4			
v3	∞	7			

	0	1	2	3	4
t	0	0	0		
s	∞	∞	1		
v1	∞	6			
v2	∞	-4			
v3	∞	7			

	0	1	2	3	4
t	0	0	0		
s	∞	∞	1		
v1	∞	6	-2		
v2	∞	-4			
v3	∞	7			

	0	1	2	3	4
t	0	0	0		
s	∞	∞	1		
v1	∞	6	-2		
v2	∞	-4 →	-4		
v3	∞	7			

	0	1	2	3	4
t	0	0	0		
s	∞	∞	1		
v1	∞	6	-2		
v2	∞	-4	-4		
v3	∞	7	7		

	0	1	2	3	4
t	0	0	0 →	0	
s	∞	∞	1		
v1	∞	6	-2		
v2	∞	-4	-4		
v3	∞	7	7		

	0	1	2	3	4
t	0	0	0	0	
s	∞	∞	1	1	
v1	∞	6	-2		
v2	∞	-4	-4		
v3	∞	7	7		

	0	1	2	3	4
t	0	0	0	0	
s	∞	∞	1	1	
v1	∞	6	-2	-2	
v2	∞	-4	-4		
v3	∞	7	7		

	0	1	2	3	4
t	0	0	0	0	
s	∞	∞	1	1	
v1	∞	6	-2	-2	
v2	∞	-4	-4	-4	
v3	∞	7	7		

	0	1	2	3	4
t	0	0	0	0	
s	∞	∞	1	1	
v1	∞	6	-2	-2	
v2	∞	-4	-4	-4	
v3	∞	7	7	4	

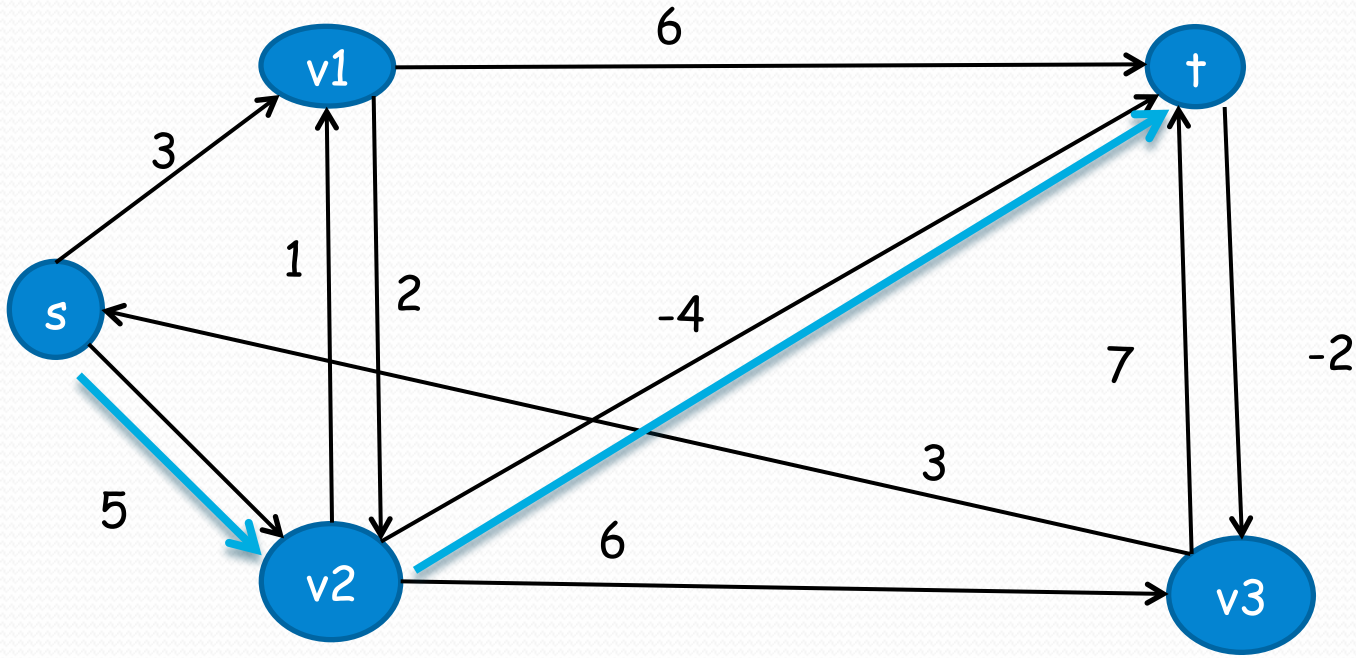
	0	1	2	3	4
t	0	0	0	0 →	0
s	∞	∞	1	1	
v1	∞	6	-2	-2	
v2	∞	-4	-4	-4	
v3	∞	7	7	4	

	0	1	2	3	4
t	0	0	0	0	0
s	∞	∞	1	1 →	1
v1	∞	6	-2	-2	
v2	∞	-4	-4	-4	
v3	∞	7	7	4	

	0	1	2	3	4
t	0	0	0	0	0
s	∞	∞	1	1	1
v1	∞	6	-2	-2 →	-2
v2	∞	-4	-4	-4	
v3	∞	7	7	4	

	0	1	2	3	4
t	0	0	0	0	0
s	∞	∞	1	1	1
v1	∞	6	-2	-2	-2
v2	∞	-4	-4	-4 →	-4
v3	∞	7	7	4	

	0	1	2	3	4
t	0	0	0	0	0
s	∞	∞	1	1	1
v1	∞	6	-2	-2	-2
v2	∞	-4	-4	-4	-4
v3	∞	7	7	4	4



The final s-t path obtained is s-v2-t with 1 as minimum cost .

Acknowledgements

- Nikita Khanna
- Kirti Rani
- Pooja Rani
- Om ji
- Neha Katyal
- Sonam
- Pulkit Ohri
- Garima Jain
- Pooja

Any Questions…..

Thank You!