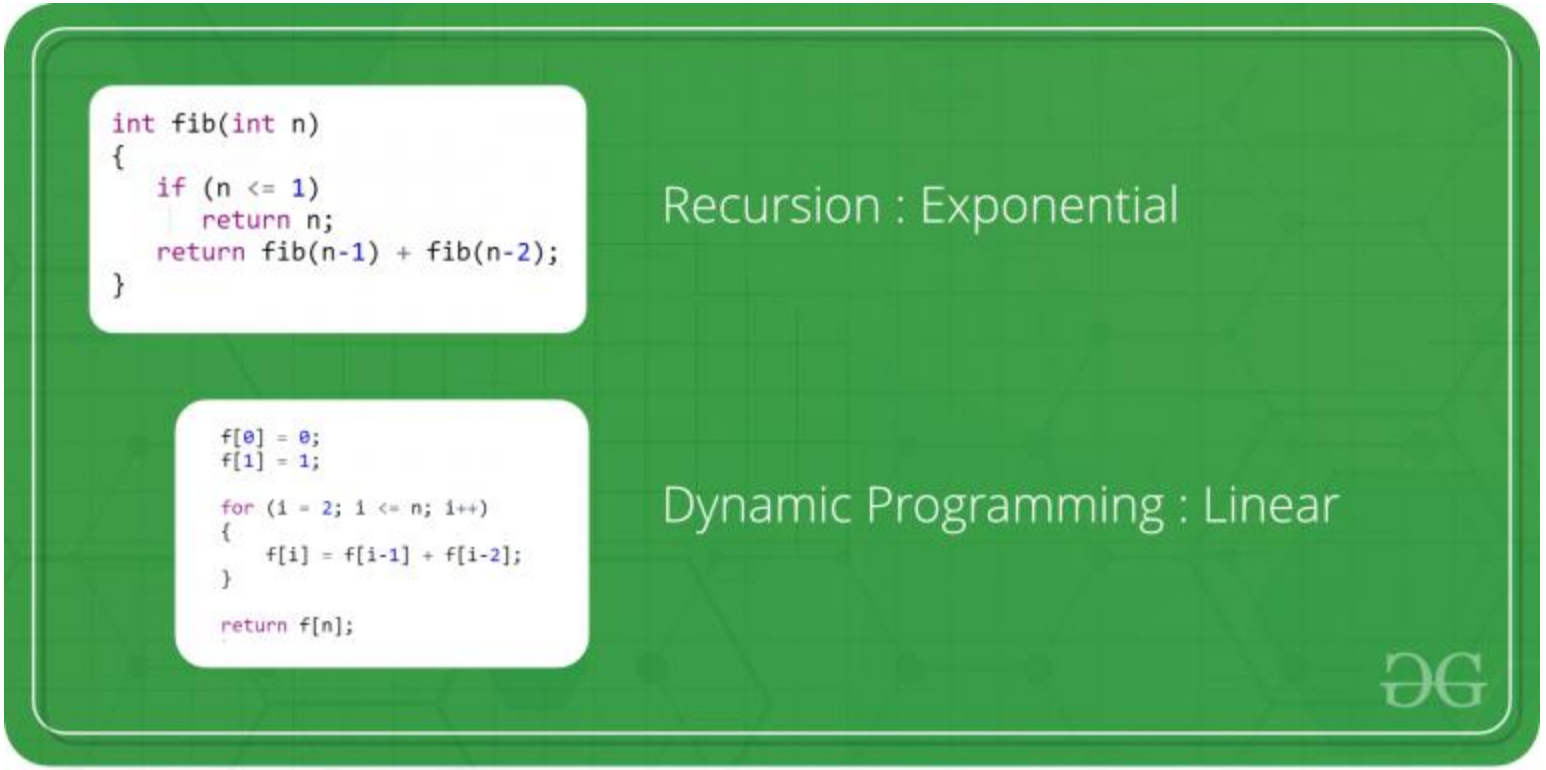
Dynamic Programming

Definition & Problem 1

What is??

- Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub-problems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of sub-problems, time complexity reduces to linear.

With recursion O(2^n), without linear order.



Principle of optimality

- The principle of optimality is the basic principle of dynamic programming, which was developed by Richard Bellman: that an optimal path has the property that whatever the initial conditions and control variables (choices) over some initial period, the control (or decision variables) chosen over the remaining period must be optimal for the remaining problem, with the state resulting from the early decisions taken to be the initial condition.

Difference b/w Greedy and Dynamic

Feasibility:

- In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.
- In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .

Continued…

Optimality:

- In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.
- It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.

Continued…

Recursion:-

A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.

A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.

Continued…

Memorization:

- It is more efficient in terms of memory as it never look back or revise previous choices
- It requires dp table for memorizat memory complexity.

Continued…

Time complexity

- Greedy methods are generally faster. For example, Dijkst shortest path algorithm takes O(ELogV + VLogV) time.
- Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.

Knapsack 0-1 Problem

- The goal is to **maximize the value of a knapsack** that can hold at most W units (i.e. lbs or kg) worth of goods from a list of items I_0, I_1, I_{n-1}.

  - Each item has 2 attributes:
    1) Value – let this be v_i for item I_i
    2) Weight – let this be w_i for item I_i

Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - You can either take it or not.
  - Hence the name Knapsack 0-1 problem.

Knapsack 0-1 Problem

- Brute Force
  - The naïve way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack.
  - We can come up with a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Knapsack 0-1 Problem

- As we did before we are going to solve the problem in terms of sub-problems.
  - So let's try to do that…
- Our first attempt might be to characterize a sub-problem as follows:
  - Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots I_k\}$.
    - What we find is that the optimal subset from the elements $\{I_0, I_1, \ldots I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, \ldots I_k\}$, in any regular pattern.
  - Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let's illustrate that poin

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

- **The maximum weight the knapsack can hold is 20.**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Knapsack 0-1 Problem – Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

- **First case:** $w_k > w$
  - Item k can't be part of the s would be > w, which is unacceptable.

- **Second case:** $w_k \leq w$
  - Then the item k can be in the solution, and we choose the case with greater value.

Knapsack 0-1 Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
  B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
  B[i,0] = 0
}
for i = 1 to n {
  for w = 0 to W {
       if w_i <= w {  //item i can be in the solution
            if v_i + B[i-1,w-w_i] > B[i-1,w]
                 B[i,w] = v_i + B[i-1,w- w_i]
            else
                 B[i,w] = B[i-1,w]
       }
       else B[i,w] = B[i-1,w] // w_i > w
  }
}
```

Knapsack 0-1 Problem

- Let's run our algorithm
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

for i = 1 to n
        B[i,0] = 0
```

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

i = 1
b_i = 3
w_i = 2
w = 1
$w-w_i$ = -1

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

i = 1
$b_i = 3$
$w_i = 2$
w = 2
$w-w_i = 0$

if $w_i <= w$ //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

i = 1
b_i = 3
w_i = 2
w = 3
$w-w_i$ = 1

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

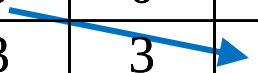
i = 1
b_i = 3
w_i = 2
w = 4
$w-w_i$ = 2

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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					
3	0					
4	0					



i = 1
b_i = 3
w_i = 2
w = 5
$w-w_i$ = 3

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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	0					
4	0					

i = 2
$b_i = 4$
$w_i = 3$
w = 1
$w-w_i = -2$

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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			
3	0					
4	0					

i = 2
$_{bi}$ = 4
$w_i = 3$
w = 2
$w-w_i = -1$

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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		
3	0					
4	0					

i = 2
b_i = 4
w_i = 3
w = 3
$w-w_i$ = 0

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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	
3	0					
4	0					

i = 2
b_i = 4
w_i = 3
w = 4
$w-w_i$ = 1

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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					
4	0					

$i = 2$

$b_i = 4$

$w_i = 3$

w = 5

$w\text{-}w_i = 2$

if $w_i <= w$ //item i can be in the solution

if $b_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = b_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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		
4	0					

i = 3
b_i = 5
w_i = 4
w = 1..3
$w-w_i$ = -3..-1

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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	
4	0					

i = 3
b_i = 5
w_i = 4
w = 4
$w-w_i$ = 0

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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					

$i = 3$
$v_i = 5$
$w_i = 4$
w = 5
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution
    if $b_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = b_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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	

i = 4
b_i = 6
w_i = 5
w = 1..4
$w-w_i$ = -4..-1

if w_i <= w //item i can be in the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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

$i = 4$

$b_i = 6$

$w_i = 5$

w = 5

$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $b_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = b_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (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**

We ' r e D O N E ! !

The max possible value that can be carried in this knapsack is ***Rs.7***

Knapsack 0-1 Algorithm

- This algorithm only finds the max possible value that can be carried in the knapsack
  - The value in B[n,W]

- To know the *items* that make this maximum value, we need to trace back through the table.

Knapsack 0-1 Algorithm Finding the Items

- Let i = n and k = W

  if B [i , k] then ≠ B [i -1, k]

  mark the i^{th} item as in the knapsack

  $i = i-1, k = k-w_i$

  else

  i = i-1 // Assume the i^{th} item is not in the knapsack

  // Could it be in the optimally packed knapsack?

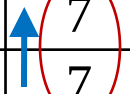
Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

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



i = 4
k = 5
v_i = 6
w_i = 5
B[i,k] = 7
B[i-1,k] = 7

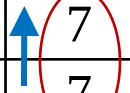
i = n , k = W
while i, k > 0
    if *B [i , k] ≠ B [i-1, k]* then
        mark the i^{th} item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1

Knapsack 0-1 Algorithm Finding the Items

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

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



i = 3
k = 5
v_i = 5
w_i = 4
B[i,k] = 7
B[i-1,k] = 7

i = n , k = W
while i, k > 0
if *B [i , k] ≠ B [i -1, k]* then
mark the i^{th} item as in the knapsack
i = i-1, k = k-w_i
else
i = i-1

Knapsack 0-1 Algorithm Finding the Items

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:
Item 2

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

i = 2
k = 5
$v_i = 4$
$w_i = 3$
B[i,k] = 7
B[i-1,k] = 3
$k - w_i = 2$

i = n , k = W
while i, k > 0
    if *B [i , k] ≠ B [i-1, k]* then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Knapsack 0-1 Algorithm Finding the Items

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:
Item 2
Item 1

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

i = 1
k = 2
$v_i = 3$
$w_i = 2$
B[i,k] = 3
B[i-1,k] = 0
$k - w_i = 0$

i = n , k = W
while i, k > 0
if *B [i , k] ≠ B [i-1, k]* then
mark the i^{th} item as in the knapsack
i = i-1, k = k-w_i
else
i = i-1

Knapsack 0-1 Algorithm Finding the Items

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:
Item 2
Item 1

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

i = 1
k = 2
$v_i = 3$
$w_i = 2$
B[i,k] = 3
B[i-1,k] = 0
$k - w_i = 0$

k = 0, so we're DONE!

The optimal knapsack should contain:
Item 1 and Item 2

Knapsack 0-1 Problem – Run Time

```
for w = 0 to W
    B[0,w] = 0
```

O(*W*)

```
for i = 1 to n
    B[i,0] = 0
```

O(*n*)

```
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

Repeat *n* times

O(*W*)

What is the running time of this algorithm?

O(*n*W*)**

Remember that the brute-force algorithm takes: **O(2^n)**

Bellman-Ford Algorithm for Single Source Shortest Paths

- More general than Dijkstra's algori
  - Edge-weights can be negative
- Detects the existence of negative-weight cycle(s) reachable from s.

Bellman-Ford Algorithm for Single Source Shortest Paths

```
BELMAN-FORD( G, s )
    INIT( G, s )
    for i ←1 to |V|-1 do
        for each edge (u, v) ∈ E do
            RELAX( u, v )
    for each edge ( u, v ) ∈ E do
        if d[v] > d[u]+w(u,v) then
            return FALSE        > neg-weight cycle
    return TRUE
```

Bellman-Ford Algorithm for Single Source Shortest Paths

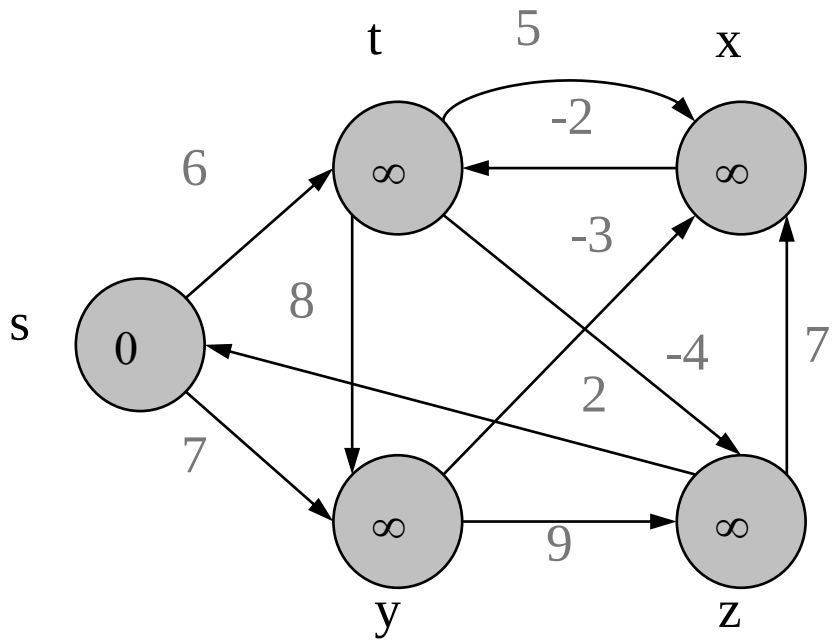
Observe:

- First nested for-loop performs |V|-1 relaxation passes; relax every edge at each pass
- Last for-loop checks the existence of a negative-weight cycle reachable from s

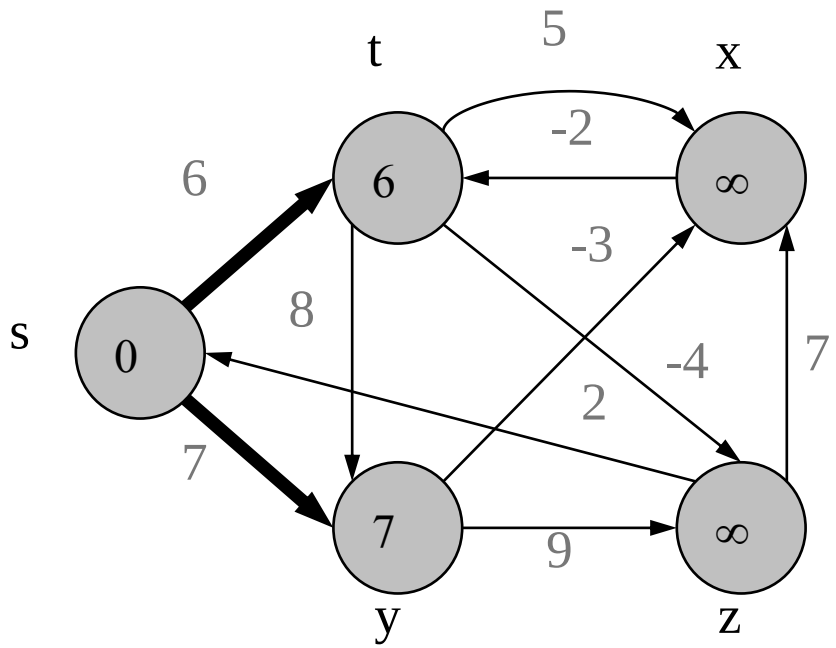
Bellman-Ford Algorithm for Single Source Shortest Paths

- *Running time* = O(VxE)
  Constants are good; it is practical)
- *Example:* Run algorithm on a sample graph with no negative weight cycles.

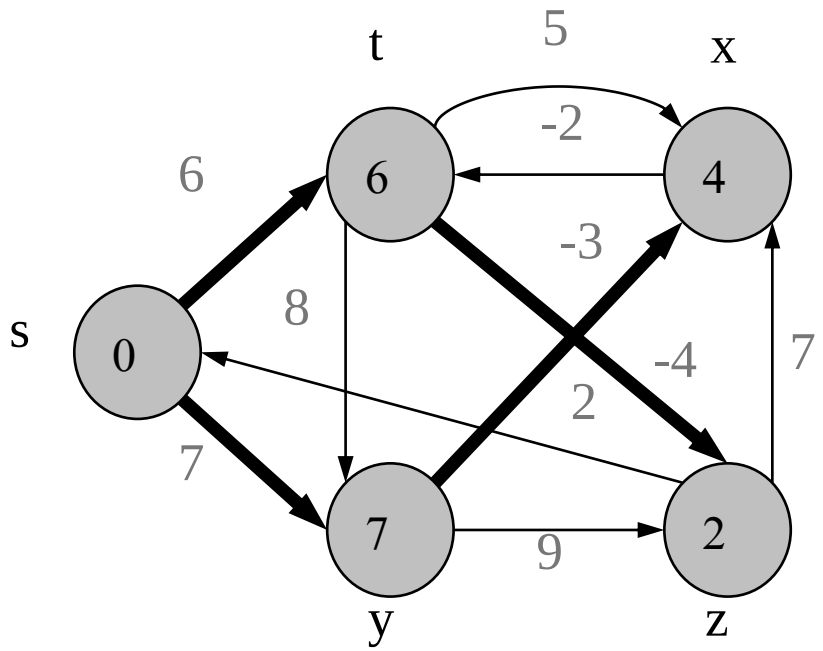
Bellman-Ford Algorithm Example



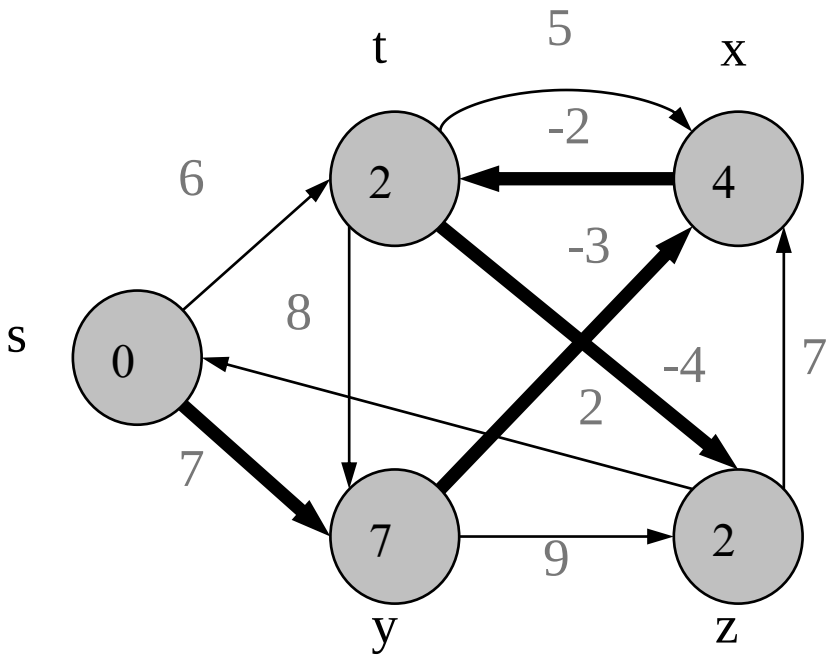
Bellman-Ford Algorithm Example



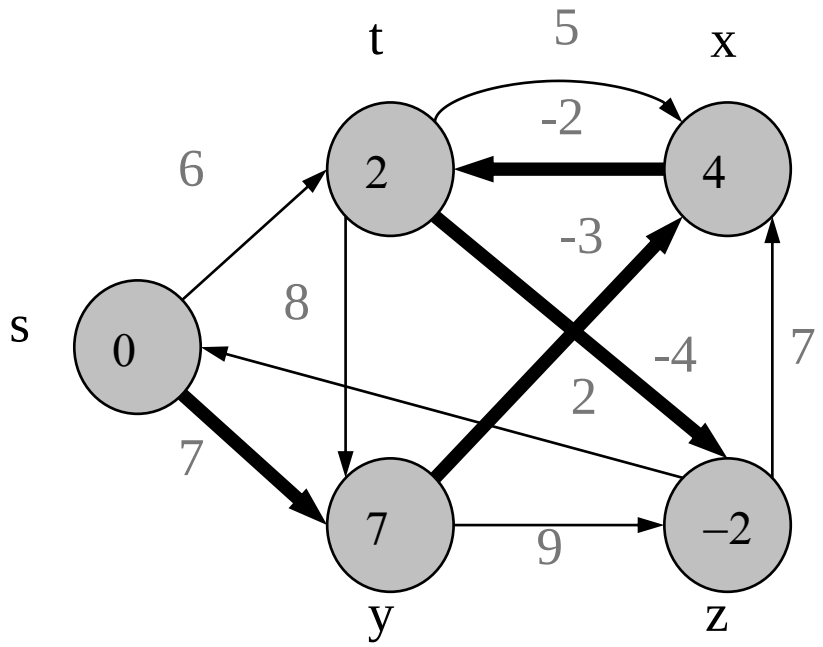
Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example



Floyd-Warshall

We will now investigate a dynamic programming solution that solved the problem in $O(n^3)$ time for a graph with n vertices.

This algorithm is known as the Floyd-Warshall algorithm, but it was apparently described earlier by Roy.

Representation of the Input

We assume that the input is represented by a weight matrix $W = (w_{ij})_{i,j \text{ in } E}$ that is defined by

$w_{ij} = 0$ if $i=j$

$w_{ij} = w(i,j)$ if $i \neq j$ and (i,j) in E

$w_{ij} = \infty$ if $i \neq j$ and (i,j) not in E

Format of the Output

If the graph has n vertices, we return a distance matrix (d_{ij}), where d_{ij} the length of the path from i to j.

Intermediate Vertices

Without loss of generality (i.e., that the vertices of the graph are numbered from 1 to n.

Given a path $p=(v_1, v_2, \ldots, v_m)$ in the graph, we will call the vertices v_k with index $k \in \{2, \ldots, m-1\}$ the intermediate vertices of p.

Key Definition

The key to the Floyd-Warshall algorithm is the following definition:

Let $d_{ij}^{(k)}$ denote the length of the shortest path from i to j such that all intermediate ver

Remark 1

A shortest path does not contain any vertex twice, as this would imply that the path contains a cycle. By assumption, cycles in the graph have a positive weight, so removing the cycle would result in a shorter path, which is impossible.

Remark 2

Consider a shortest path p from i to j such that the intermediate vertices are

- If the vertex k is not an intermediate vertex on p, then $d_{ij}^{(k)} = d_{ij}^{(k-1)}$

If the vertex k is an intermediate vertex on p, then $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

Interestingly, in either case, the subpaths contain merely nodes from $\{1, 2, \ldots, k-1\}$.

Remark 2

Therefore, we can conclude that

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Recursive Formulation

If we do not use intermediate nodes, i.e., when k=0, then

$$d_{ij}^{(0)} = w_{ij}$$

If k>0, then

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

The Floyd-Warshall Algorithm

```
Floyd-Warshall(W)
n = # of rows of W;
D(0) = W;
for k = 1 to n do
    for i = 1 to n do
            for j = 1 to n do
                    d_ij(k) = min{d_ij(k-1) , d_ik(k-1) + d_kj(k-1)};
            od;
    od;
od;
 return D(n);
```

Time and Space Requirements

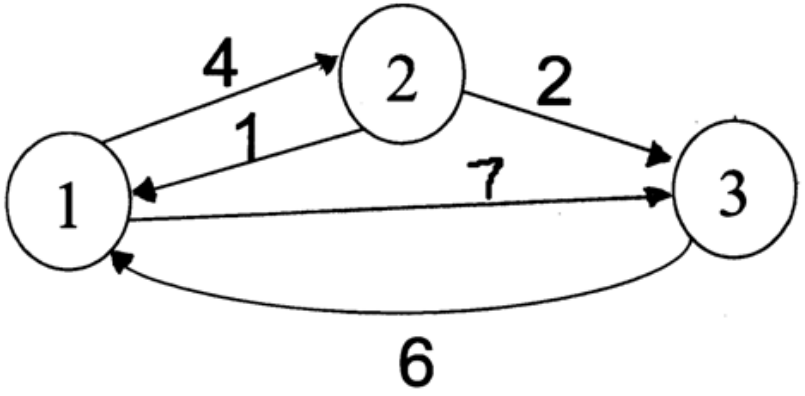
The running time is obviously $O(n^3)$.

However, in this version, the space requirements are high. One can reduce the space from $O(n^3)$ to $O(n^2)$ by using a single array d.

Floyd Warshall Algorithm - Example

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 1:

D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:

D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 3:

Nothing changes.

Floyd Warshall Algorithm

- Looking at this example, we can come up with the following algorithm:
  - Let D store the matrix with the initial graph edge information initially, and update D with the calculated shortest paths.

```
For k=1 to n {
  For i=1 to n {
      For j=1 to n
          D[i,j] = min(D[i,j],D[i,k]+D[k,j])
  }
}
```

  - The final D matrix will store all the shortest paths.