

Dynamic Programming

UNIT 4

CI43/CY43

Steps for Solving DP Problems

1. Define subproblems
 2. Write down the recurrence that relates subproblems
 3. Recognize and solve the base cases
- Each step is very important!

Dynamic Programming

- Dynamic programming is a very powerful, general tool for solving optimization problems.
- Once understood it is relatively easy to apply, but many people have trouble understanding it.

Greedy Algorithms

- Greedy algorithms focus on making the best local choice at each decision point.
- For example, a natural way to compute a shortest path from x to y might be to walk out of x , repeatedly following the cheapest edge until we get to y . WRONG!
- In the absence of a correctness proof greedy algorithms are very likely to fail.

Problem:

Let's consider the calculation of **Fibonacci** numbers:

$$F(n) = F(n-2) + F(n-1)$$

with seed values $F(1) = 1, F(2) = 1$

or $F(0) = 0, F(1) = 1$

What would a series look like:

0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, ...

Recursive Algorithm:

```
Fib(n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    Return Fib(n-1)+Fib(n-2)
}
```

Recursive Algorithm:

Fib(n)

{

if (n == 0)

return 0;

if (n == 1)

return 1;

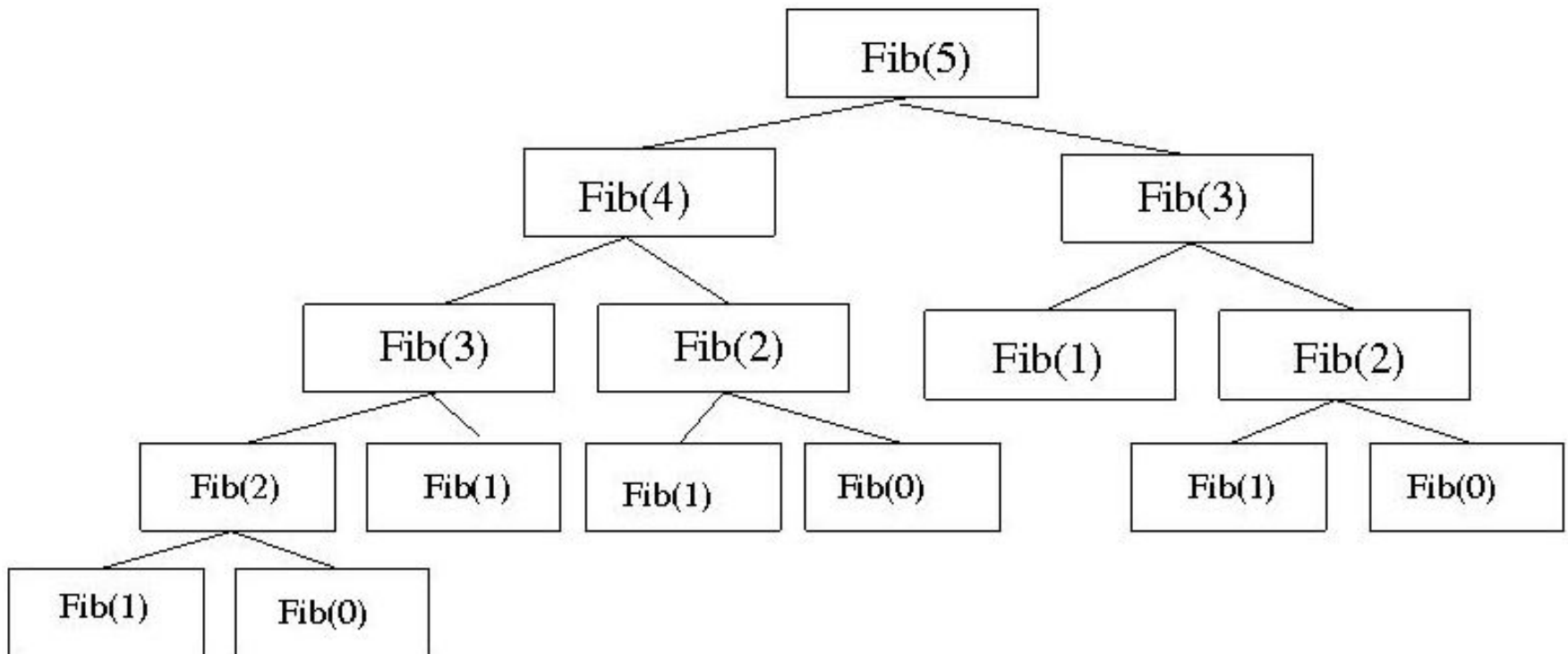
Return Fib(n-1)+Fib(n-2)

}

It has a serious issue!

Recursion tree

What's the problem?



Memoization:

```
Fib(n)
{
    if (n == 0)
        return M[0];

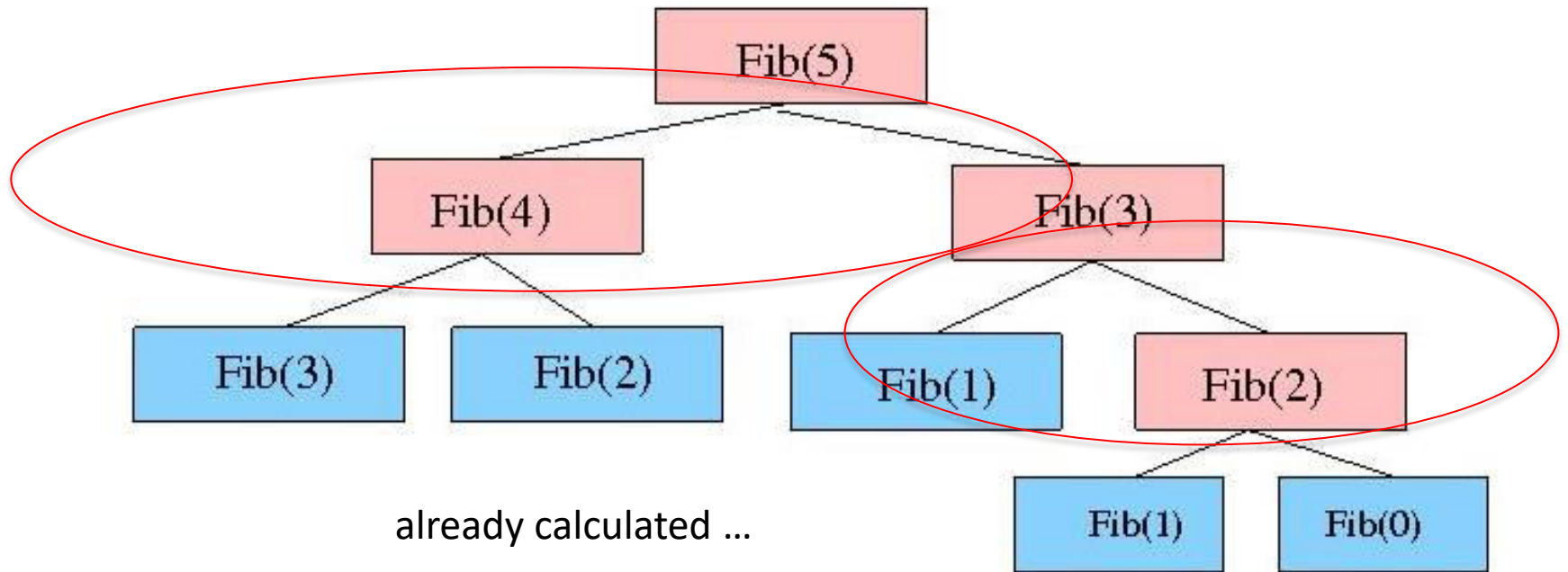
    if (n == 1)
        return M[1];

    if (Fib(n-2) is not already calculated)
        call Fib(n-2);

    if (Fib(n-1) is already calculated)
        call Fib(n-1);

    //Store the  $n^{\text{th}}$  Fibonacci no. in memory & use previous results.
    M[n] = M[n-1] + M[n-2]

    Return M[n];
}
```



Dynamic programming

- Main approach: recursive, holds answers to a sub problem in a table, can be used without recomputing.
- Can be formulated both via recursion and saving results in a table (*memoization*). Typically, we first formulate the recursive solution and then turn it into recursion plus dynamic programming via *memoization* or bottom-up.
- "*programming*" as in tabular not programming code

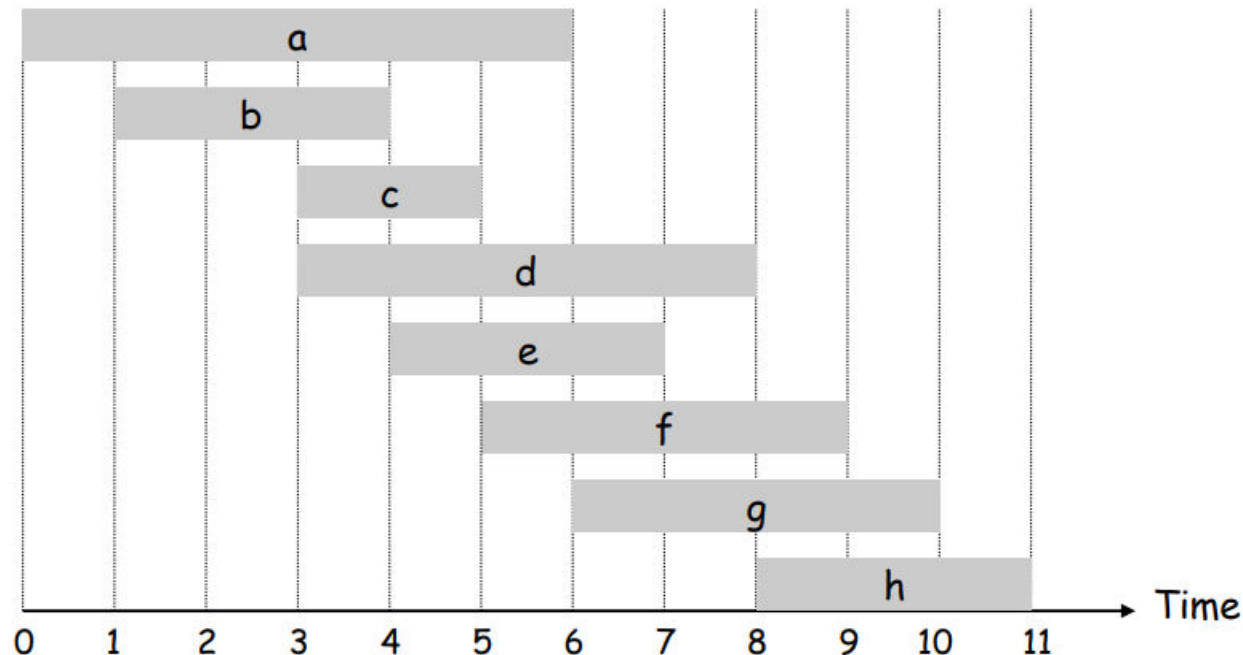
Steps for Solving DP Problems

- Define subproblems
- Write down the recurrence that relates subproblems
- Recognize and solve the base cases
- Each step is very important.

Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j , finishes at f_j , and has weight or value v_j .
- Two jobs **compatible** if they don't overlap.
- Goal: find maximum **weight** subset of mutually compatible jobs.

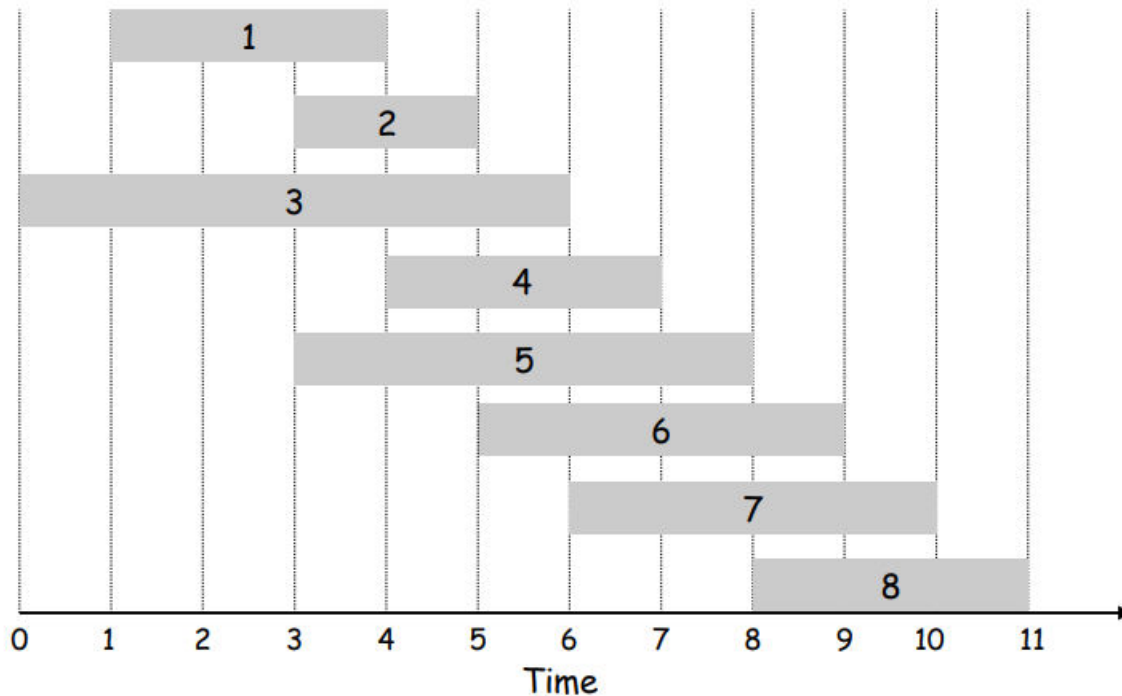


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \dots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j .

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



j	p(j)
0	-
1	0
2	0
3	0
4	1
5	0
6	2
7	3
8	5

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing - "traceback"

```
Run M-Compute-Opt(n)
```

```
Run Find-Solution(n)
```

```
Find-Solution(j) {
```

```
    if (j = 0)
```

```
        output nothing
```

```
    else if ( $v_j + \text{OPT}[p(j)] > \text{OPT}[j-1]$ )
```

```
        print j
```

```
        Find-Solution(p(j))
```

```
    else
```

```
        Find-Solution(j-1)
```

```
}
```

the condition
determining the
max when
computing
OPT[]

the relevant
sub-problem

- # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted Interval Scheduling Problem

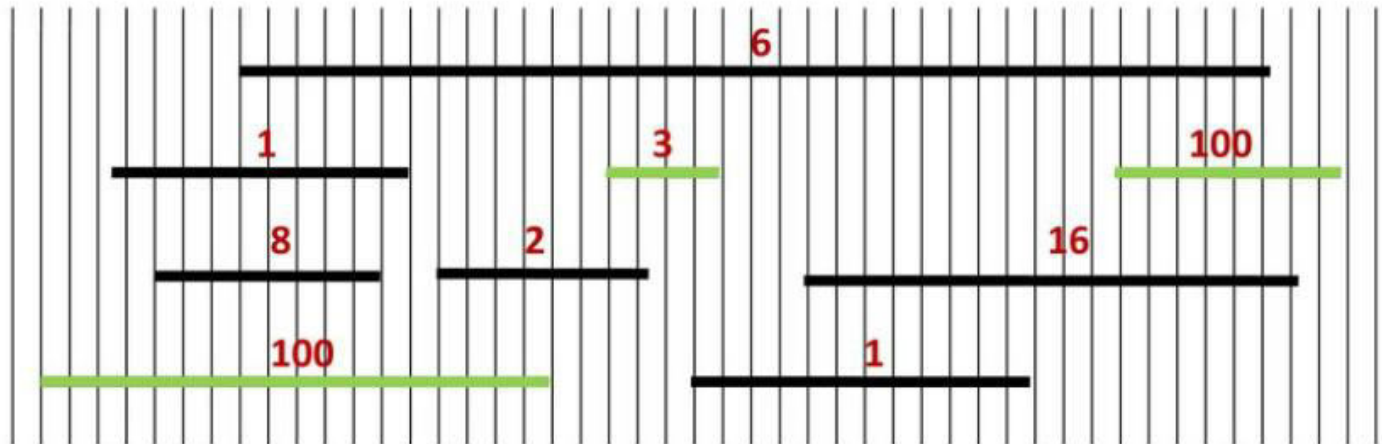
Input: A list of intervals, each with a:

- Start time s_i
- Finish time f_i
- Weight w_i

Output: A subset of *non-overlapping* intervals

Goal: The largest possible weight sum

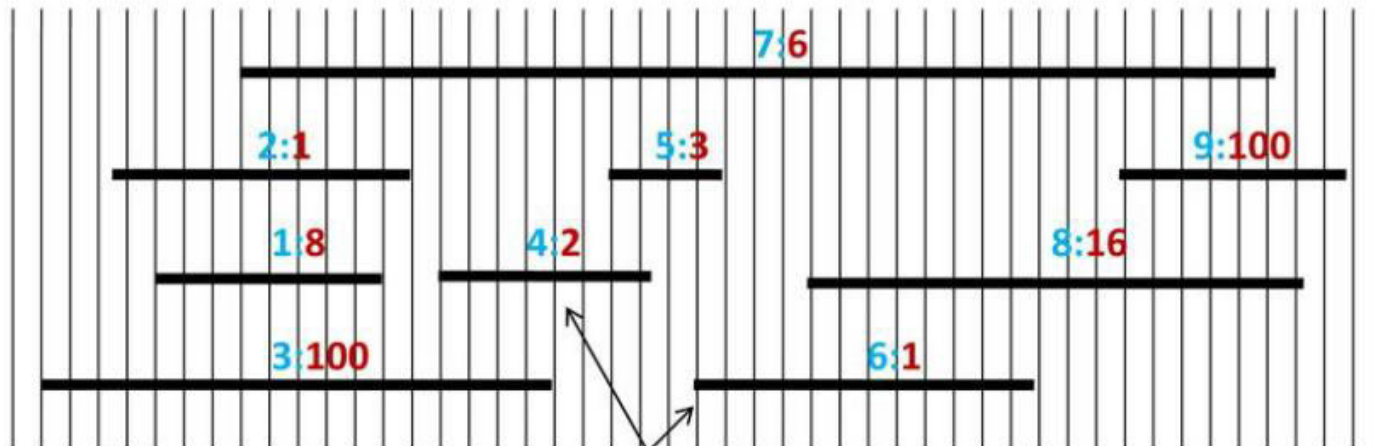
Weighted Interval Scheduling



Optimal weight: $100 + 3 + 100 = 203$

Weighted Interval Scheduling

Note: we are numbering from 1, not 0
(It will be less confusing later on.)



$$p(6) = 4$$

$$p(8) = 5$$

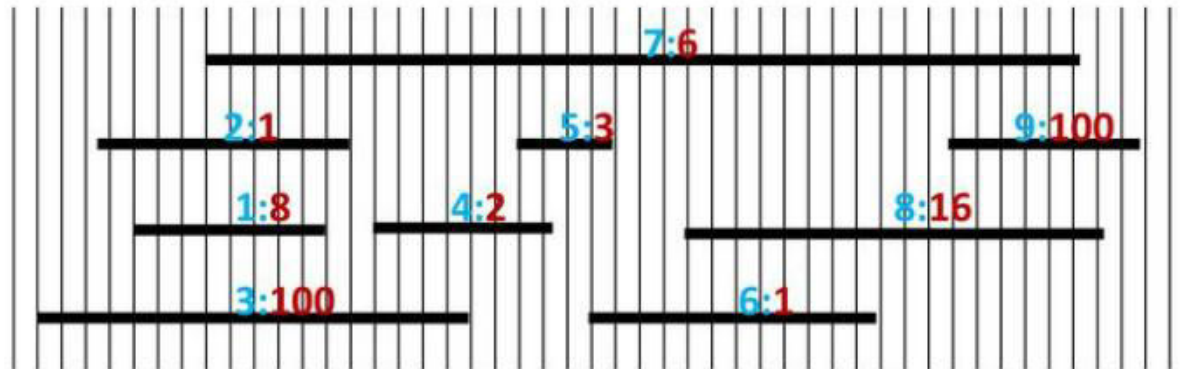
What is $p(9)$?

$$p(7) = 0$$

Undefined

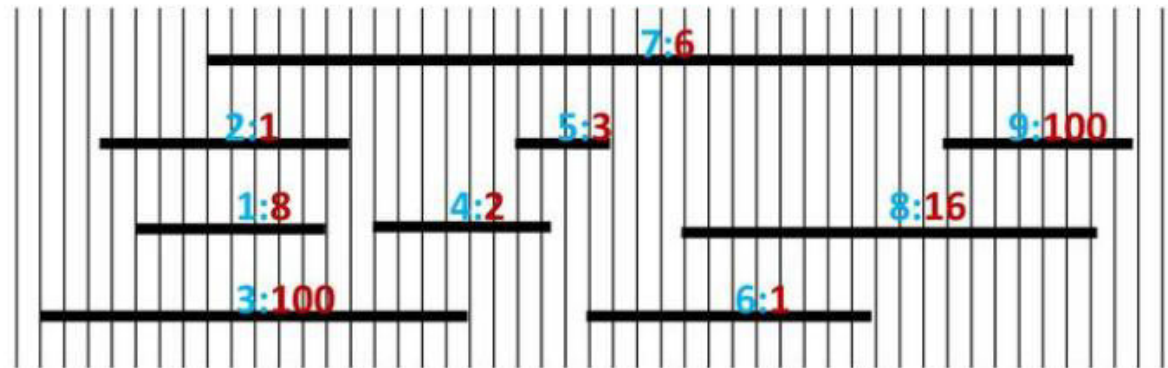
What is $p(9)$?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5
- f) 6
- g) 7
- h) 8



What is $p(9)$?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5
- f) 6
- g) 7
- h) 8



Case 1: interval n is used

If we **know** interval n is used, what else do we need to do?

$p(n)$ is the last interval we can still use
(if $j > p(n)$, then $f_j > s_n$)

Claim: $opt(n) = opt(p(n)) + w_n$

Case 1: interval n is used

Claim: $opt(n) = opt(p(n)) + w_n$

Must have $opt(n) \geq opt(p(n)) + w_n$

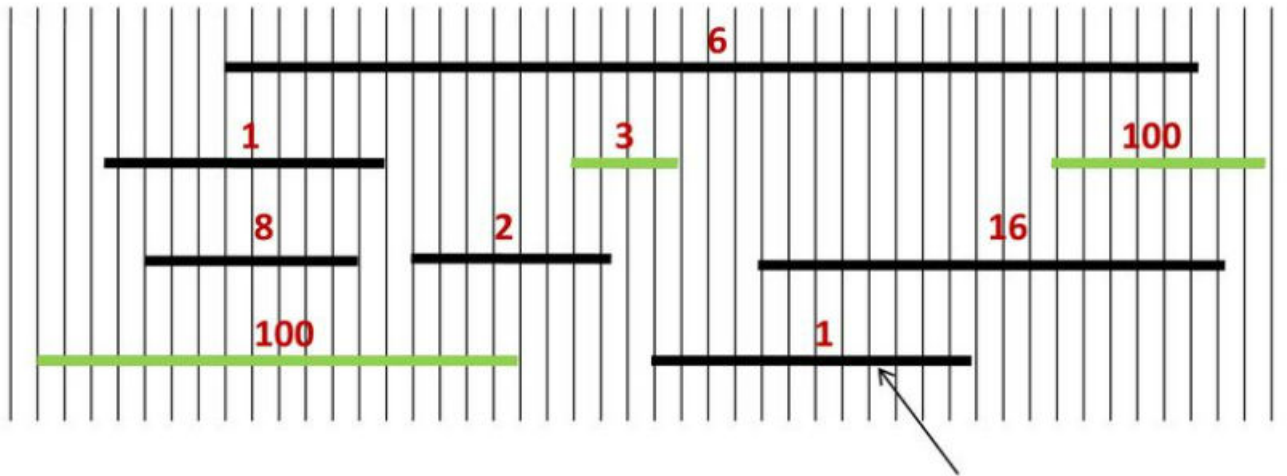
- Take solution for $opt(p(n))$
- Add interval n (must be possible)
- Resulting total: $opt(p(n)) + w_n$

Results comes from
the optimal sub-
structure property

Must have $opt(n) \leq opt(p(n)) + w_n$

- Remove interval n from optimal
- New weight: $opt(n) - w_n$
- Picked from $\{1, 2, \dots, p(n)\}$
- ...Hence must be $\leq opt(p(n))$
- Result: $opt(n) - w_n \leq opt(p(n))$

CASE 1: ILLUSTRATED



$$\text{opt}(9) = 100 + 3 + 100 = 203, p(9) = 6$$

$$\text{opt}(6) = 100 + 3 = 103$$

$$\text{opt}(9) = \text{opt}(6) + w_9$$

Case 1: Recap

If interval n is in the optimal solution:
then $\text{opt}(n) = \text{opt}(p(n)) + w_n$

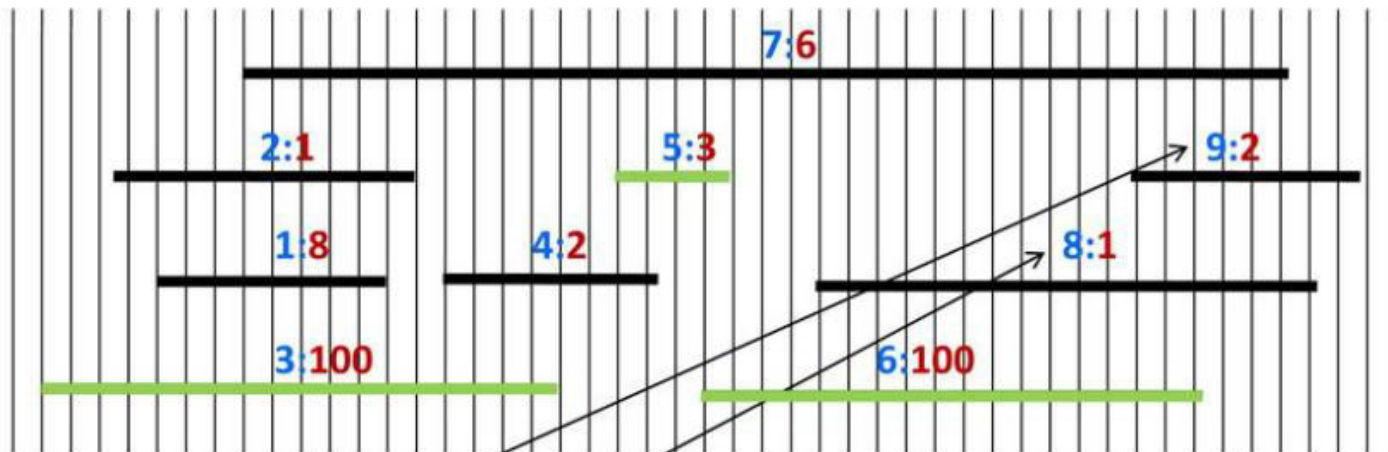
Case 2: Item n is not used

If we *know* the interval item is not used, what else do we need to do?

Claim: $opt(n) = opt(n-1)$

If item n is not used, then $opt(n)$ and $opt(n-1)$ must correspond to the same solution score

CASE 2: ILLUSTRATED



$$\text{opt}(9) = 100 + 100 + 3 = 203$$

The optimal solution clearly doesn't use interval 9

$$\text{opt}(8) = 100 + 100 + 3 = 203$$

$$\text{opt}(9) = \text{opt}(8)$$

So we get the same solution if we remove 9, considering only intervals 1 to 8

Summing Up

If n is in the optimal solution: $opt(n) = opt(p(n)) + w_n$

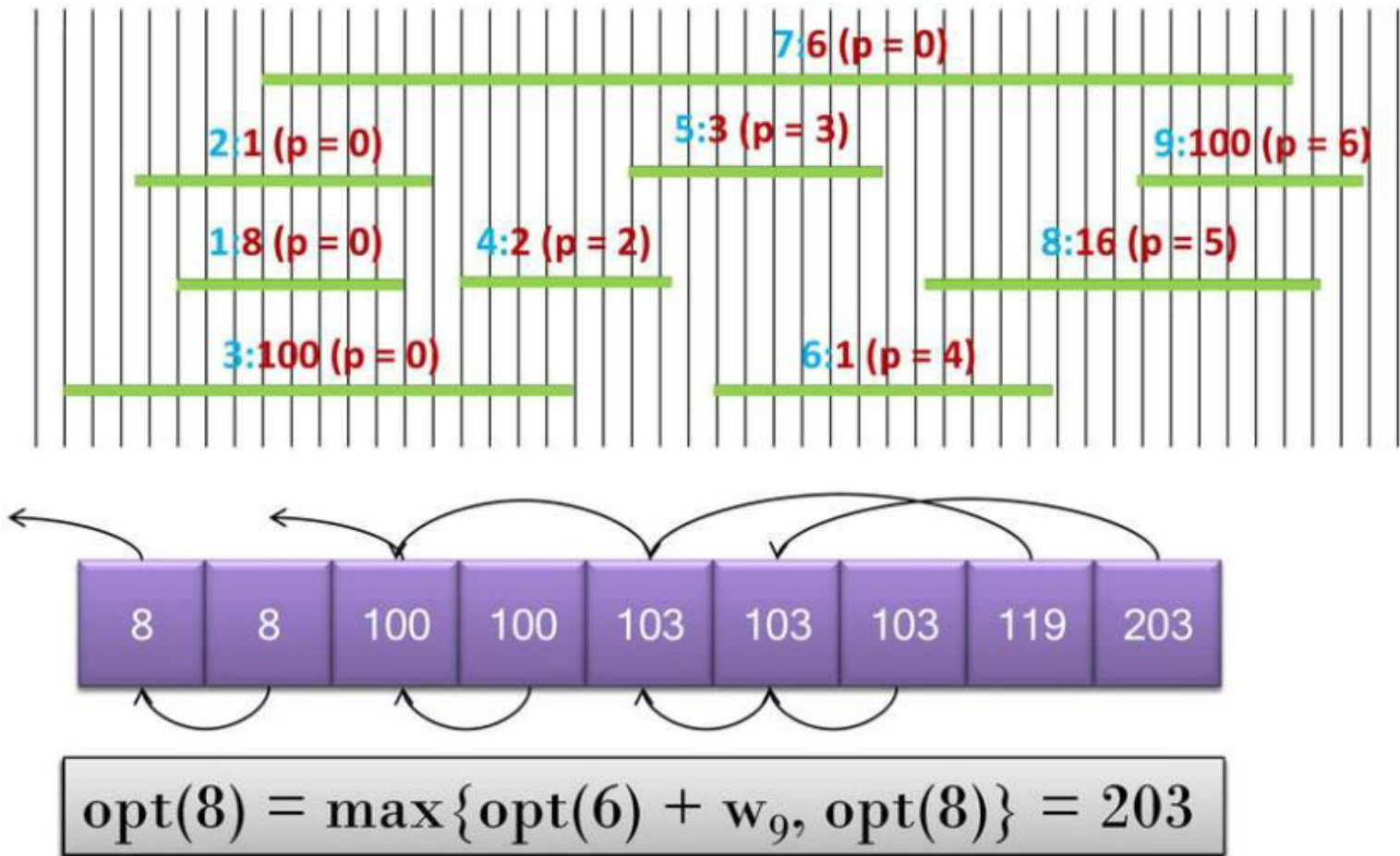
If n is *not* in the optimal solution: $opt(n) = opt(n-1)$

$$opt(n) = \max \left\{ \begin{array}{l} opt(p(n)) + w_n \\ opt(n-1) \end{array} \right.$$

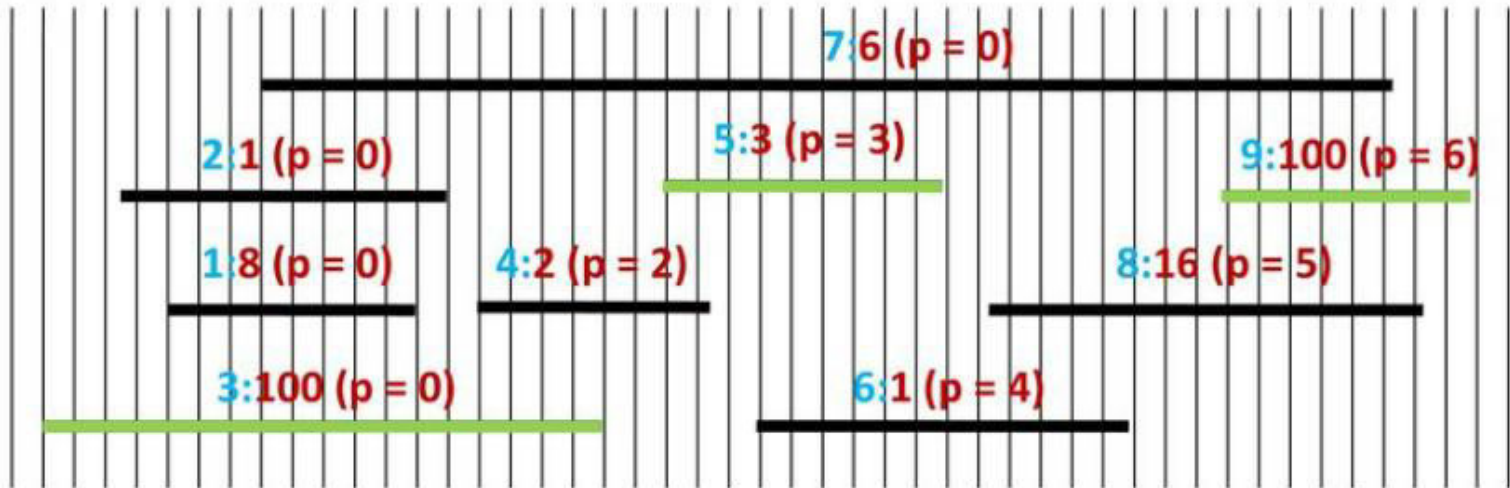
Either we use item n , or we don't.

$$opt(0) = 0$$

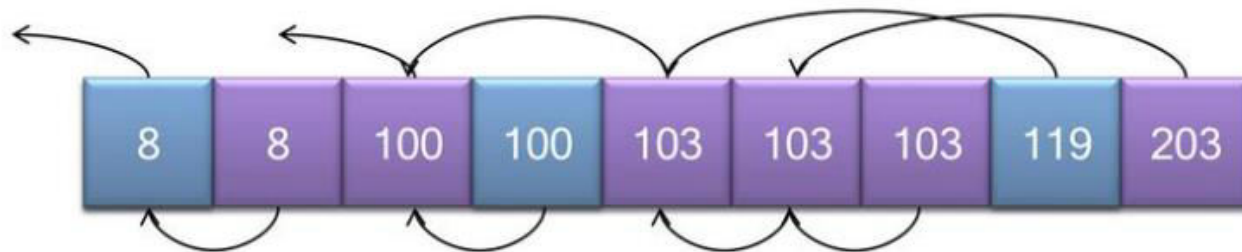
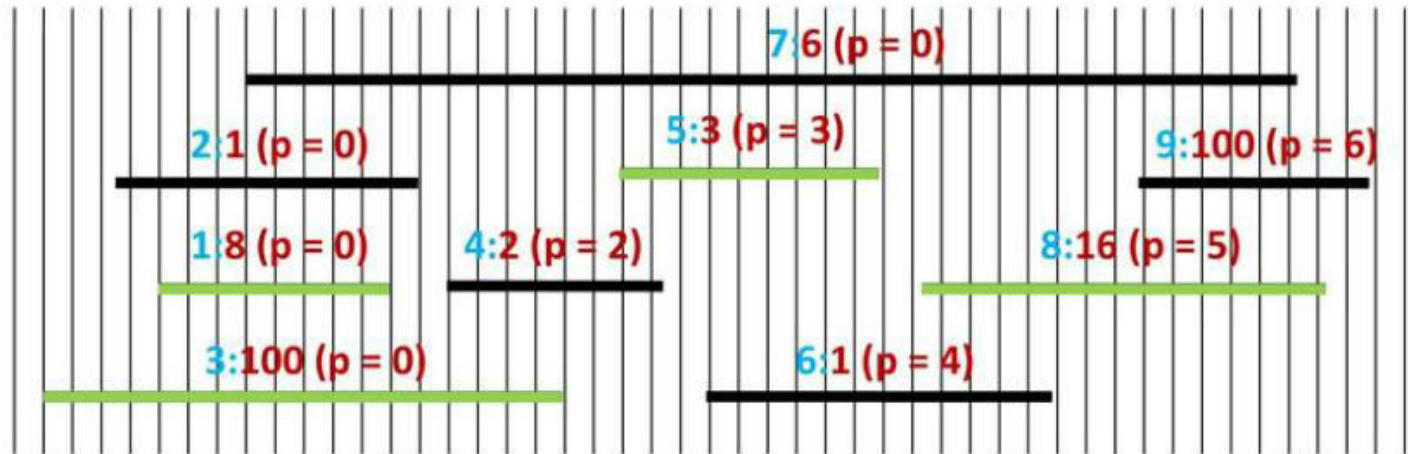
Execution



Trace-back

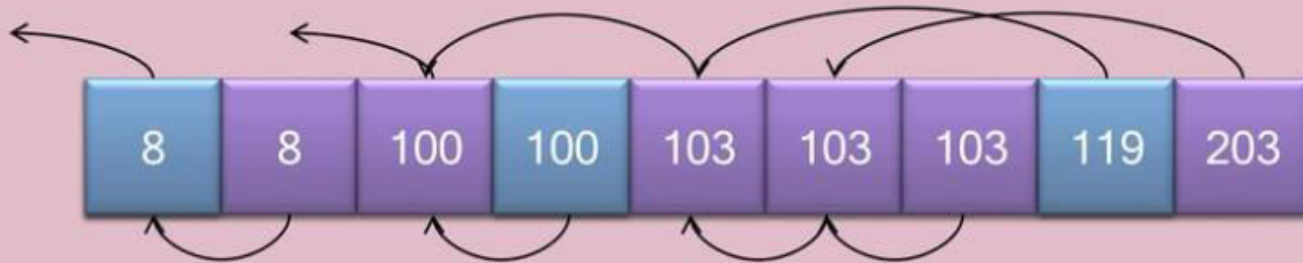


Maintains Solutions



As we continue, we track up to n different solutions
-- more than a greedy algorithm, less than brute force

Runtime



We need to fill in values into n boxes

Each box requires $O(1)$ time to fill
(if we know $p()$)

Total runtime: $O(n)$

1-dimensional DP Problem

- ▶ Problem: given n , find the number of different ways to write n as the sum of 1, 3, 4
- ▶ Example: for $n = 5$, the answer is 6

$$\begin{aligned} 5 &= 1 + 1 + 1 + 1 + 1 \\ &= 1 + 1 + 3 \\ &= 1 + 3 + 1 \\ &= 3 + 1 + 1 \\ &= 1 + 4 \\ &= 4 + 1 \end{aligned}$$

1-dimensional DP Problem

- ▶ Define subproblems
 - Let D_n be the number of ways to write n as the sum of 1, 3, 4
- ▶ Find the recurrence
 - Consider one possible solution $n = x_1 + x_2 + \cdots + x_m$
 - If $x_m = 1$, the rest of the terms must sum to $n - 1$
 - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
 - Take other cases into account ($x_m = 3, x_m = 4$)

1-dimensional DP Problem

- ▶ Recurrence is then

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- ▶ Solve the base cases

- $D_0 = 1$
- $D_n = 0$ for all negative n
- Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$

- ▶ We're basically done!

1-dimensional DP Problem

```
D[0] = D[1] = D[2] = 1; D[3] = 2;  
for(i = 4; i <= n; i++)  
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

- Very short!

Extension: Solving this for huge n ,
 $n \approx 10^{12}$!

We have $D(n) = D(n-1) + D(n-3) + D(n-4)$,

$$D(0) = D(1) = D(2) = 1 \quad n \geq 4$$

$$D(3) = 2.$$

We can write

$$\begin{bmatrix} D(n) \\ D(n-1) \\ D(n-2) \\ D(n-3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix}$$

$$\text{Let } A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\therefore \begin{bmatrix} D(n) \\ D(n-1) \\ D(n-2) \\ D(n-3) \end{bmatrix} = A \begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} \quad \text{--- (1)}$$

We can write again

$$\begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} = A \begin{bmatrix} D(n-2) \\ D(n-3) \\ D(n-4) \\ D(n-5) \end{bmatrix}$$

∴ (1) becomes

$$\begin{bmatrix} D(n) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} = A^2 \begin{bmatrix} D(n-2) \\ D(n-3) \\ D(n-4) \\ D(n-5) \end{bmatrix} = \dots$$

$$= A^{n-3} \begin{bmatrix} D(3) \\ D(2) \\ D(1) \\ D(0) \end{bmatrix}$$

Need to evaluate A^{n-3}

Evaluate (A^k)

- Step 1: If $k=1$ return A^k
- Step 2: Compute $B = A^{\lfloor k/2 \rfloor}$
- Step 3: If k is even return B^2
else return $B^2 \cdot A$

Total cost: $O(\log n)$, ~~less~~ &
Logarithmic time

KNAPSACK PROBLEM - RECURSIVE BACKTRACKING AND DYNAMIC PROGRAMMING

Knapsack Problem

- A variation of a *bin packing* problem
- Similar to fair teams problem from recursion assignment
- You have a set of items
- Each item has a weight and a value
- You have a knapsack with a weight limit
- Goal: Maximize the **value** of the items you put in the knapsack without exceeding the weight limit

The 0-1 knapsack problem

- A thief breaks into a house, carrying a knapsack...
 - He can carry up to 25 pounds of loot
 - He has to choose which of N items to steal
 - Each item has some weight and some value
 - “0-1” because each item is stolen (1) or not stolen (0)
 - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds

The 0-1 knapsack problem

- A greedy algorithm does not find an optimal solution
- A dynamic programming algorithm works well.

The 0-1 knapsack problem

- This is similar to, but not identical to, the coins problem
 - In the coins problem, we had to make an *exact* amount of change
 - In the 0-1 knapsack problem, we can't *exceed* the weight limit, but the optimal solution may be *less* than the weight limit
 - The dynamic programming solution is similar to that of the coins problem

Knapsack Example

- Items:

Item Number	Weight of Item	Value of Item	Value per unit Weight
1	1	6	6.0
2	2	11	5.5
3	4	1	0.25
4	4	12	3.0
5	6	19	3.167
6	7	12	1.714

- Weight Limit = 8

- One greedy solution: Take the highest ratio item that will fit: (1, 6), (2, 11), and (4, 12)
- Total value = $6 + 11 + 12 = 29$
- Clicker 3 - Is this optimal? A. No B. Yes

Knapsack - Recursive Backtracking

```
private static int knapsack(ArrayList<Item> items,
    int current, int capacity) {

    int result = 0;
    if (current < items.size()) {
        // don't use item
        int withoutItem
            = knapsack(items, current + 1, capacity);
        int withItem = 0;
        // if current item will fit, try it
        Item currentItem = items.get(current);
        if (currentItem.weight <= capacity) {
            withItem += currentItem.value;
            withItem += knapsack(items, current + 1,
                capacity - currentItem.weight);
        }
        result = Math.max(withoutItem, withItem);
    }
    return result;
}
```

Knapsack - Dynamic Programming

- Recursive backtracking starts with max capacity and makes choice for items:
choices are:
 - take the item if it fits
 - don't take the item
- Dynamic Programming, start with simpler problems
- Reduce number of items available
- ... AND Reduce weight limit on knapsack
- Creates a 2d array of possibilities

Knapsack - Optimal Function

- $\text{OptimalSolution}(\text{items}, \text{weight})$ is best solution given a subset of items and a weight limit
- 2 options:
- OptimalSolution does not select i^{th} item
 - select best solution for items 1 to $i - 1$ with weight limit of w
- OptimalSolution selects i^{th} item
 - New weight limit = $w - \text{weight of } i^{\text{th}} \text{ item}$
 - select best solution for items 1 to $i - 1$ with new weight limit

Knapsack Optimal Function

- $\text{OptimalSolution}(\text{items}, \text{weight limit}) =$

0 if 0 items

$\text{OptimalSolution}(\text{items} - 1, \text{weight})$ if weight of i^{th} item is greater than allowed weight
 $w_i > w$ (In others i^{th} item doesn't fit)

max of ($\text{OptimalSolution}(\text{items} - 1, w)$,
value of i^{th} item +
 $\text{OptimalSolution}(\text{items} - 1, w - w_i)$)

Knapsack - Algorithm

Example 1:

- Create a 2d array to store value of best option given subset of items and possible weights

Item Number	Weight of Item	Value of Item
1	1	6
2	2	11
3	4	1
4	4	12
5	6	19
6	7	12

- In our example 0 to 6 items and weight limits of 0 to 8
- Fill in table using OptimalSolution Function

Knapsack Algorithm

Given N items and WeightLimit

Create Matrix M with N + 1 rows and WeightLimit + 1 columns

For weight = 0 to WeightLimit

$M[0, w] = 0$

For item = 1 to N

 for weight = 1 to WeightLimit

 if(weight of ith item > weight)

$M[\text{item}, \text{weight}] = M[\text{item} - 1, \text{weight}]$

 else

$M[\text{item}, \text{weight}] = \max \text{ of}$

$M[\text{item} - 1, \text{weight}] \text{ AND}$


 value of item + $M[\text{item} - 1, \text{weight} - \text{weight of item}]$

Knapsack - Completed Table

items / weight	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{1} [1, 6]	0	6	6	6	6	6	6	6	6
{1,2} [2, 11]	0	6	11	17	17	17	17	17	17
{1, 2, 3} [4, 1]	0	6	11	17	17	17	17	18	18
{1, 2, 3, 4} [4, 12]	0	6	11	17	17	18	23	29	29
{1, 2, 3, 4, 5} [6, 19]	0	6	11	17	17	18	23	29	30
{1, 2, 3, 4, 5, 6} [7, 12]	0	6	11	17	17	18	23	29	30

Knapsack - Items to Take

items / weight	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{1} [1, 6]	0	6	6	6	6	6	6	6	6
{1,2} [2, 11]	0	6	11	17	17	17	17	17	17
{1, 2, 3} [4, 1]	0	6	11	17	17	17	17	17	17
{1, 2, 3, 4} [4, 12]	0	6	11	17	17	18	23	29	29
{1, 2, 3, 4, 5} [6, 19]	0	6	11	17	17	18	23	29	30
{1, 2, 3, 4, 5, 6} [7, 12]	0	6	11	17	17	18	23	29	30



Knapsack Problem: Running Time

Running time. $\Theta(n W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Knapsack is NP-hard.

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% (or any other desired factor) of optimum.

Example 2:

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

$W = 11$

Item	Value	Weight	V/W
1	1	1	1
2	6	2	3
3	18	5	3.60
4	22	6	3.66
5	28	7	4

Greedy: repeatedly add item with maximum ratio v_i / w_i .

Ex: { 5, 2, 1 } achieves only value = 35 \Rightarrow greedy not optimal.

Knapsack Algorithm

$W + 1$ →

$n + 1$
↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	34	40

Note: In the original image, green boxes highlight the optimal path (0,0) → (1,0) → (1,1) → (3,18) → (11,40). Pink boxes highlight the value 6 at (2,6) and 7 at (7,7). Arrows show the path from (1,0) to (1,1), (1,1) to (3,18), and (3,18) to (11,40).

OPT: { 4, 3 }
value = 22 + 18 = 40

$W = 11$

```

if ( $w_i > w$ )
    OPT[i, w] = OPT[i-1, w]
else
    OPT[i, w] = max{OPT[i-1, w],  $v_i + \text{OPT}[i-1, w - w_i]$ }
    
```

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Comments

- Dynamic programming relies on working “from the bottom up” and saving the results of solving simpler problems
 - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky

Comments

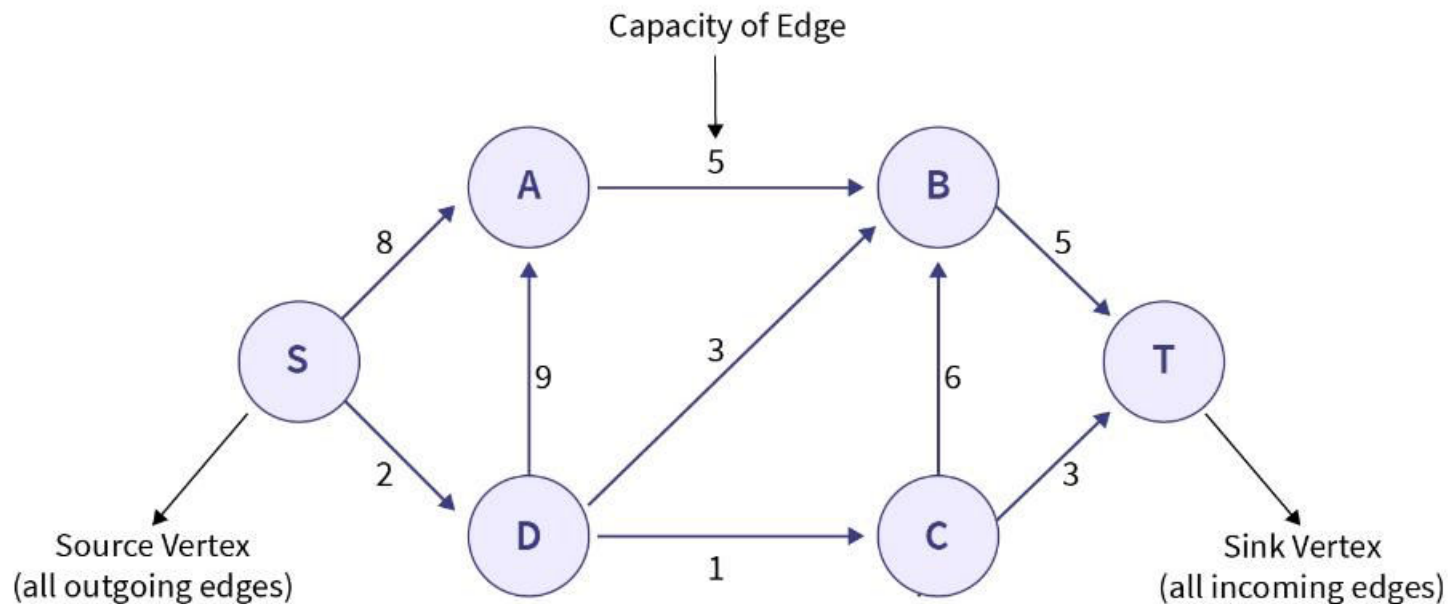
- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
 - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential.

Maximum Flow Network

Flow Network

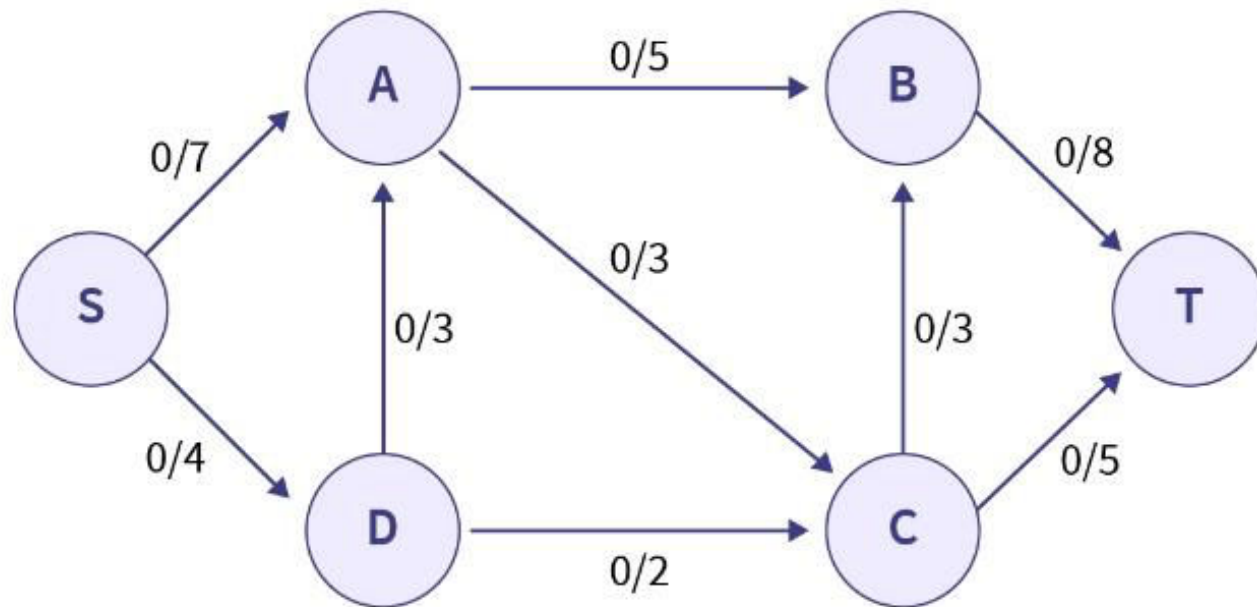
- In graph theory, a flow network is defined as directed graph $G=(V,E)$ constrained with a function c , which bounds each edge e with a non-negative integer value which is known as *capacity* of the edge e with two additional vertices defined as source S and sink T .

- As shown in the flow network given below, a source vertex has all outgoing edges and no incoming edges, more formally we can say $In_degree[source]=0$ and sink vertex has all incoming edges and no outgoing edge more formally $out_degree[sink]=0$.



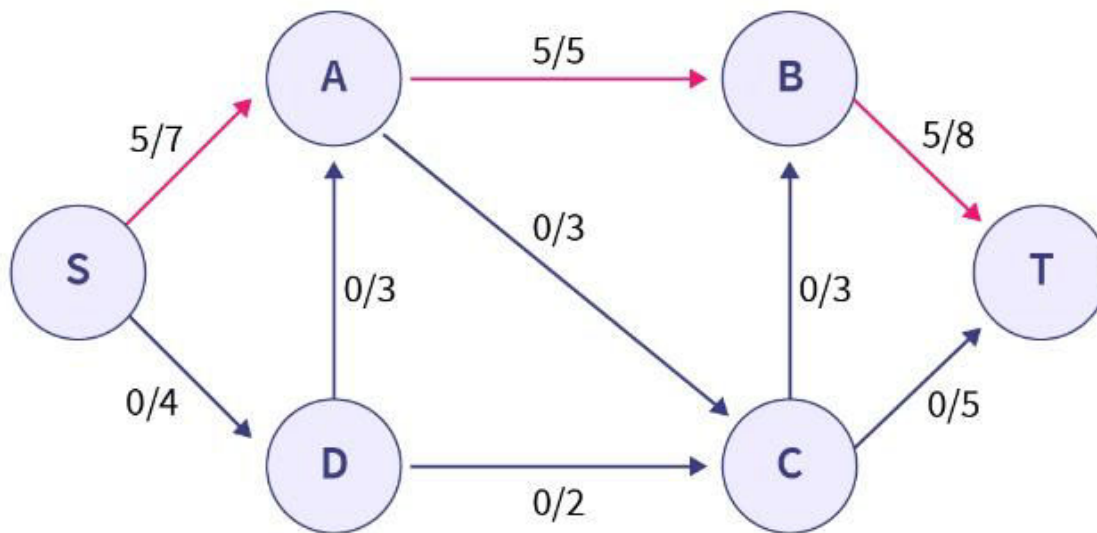
Maximum Flow Problem Introduction

- The underlying image represents a flow network, where the first value of each edge represents the amount of the flow through it (which is initially set to 0) and the second value represents the capacity of the edge.



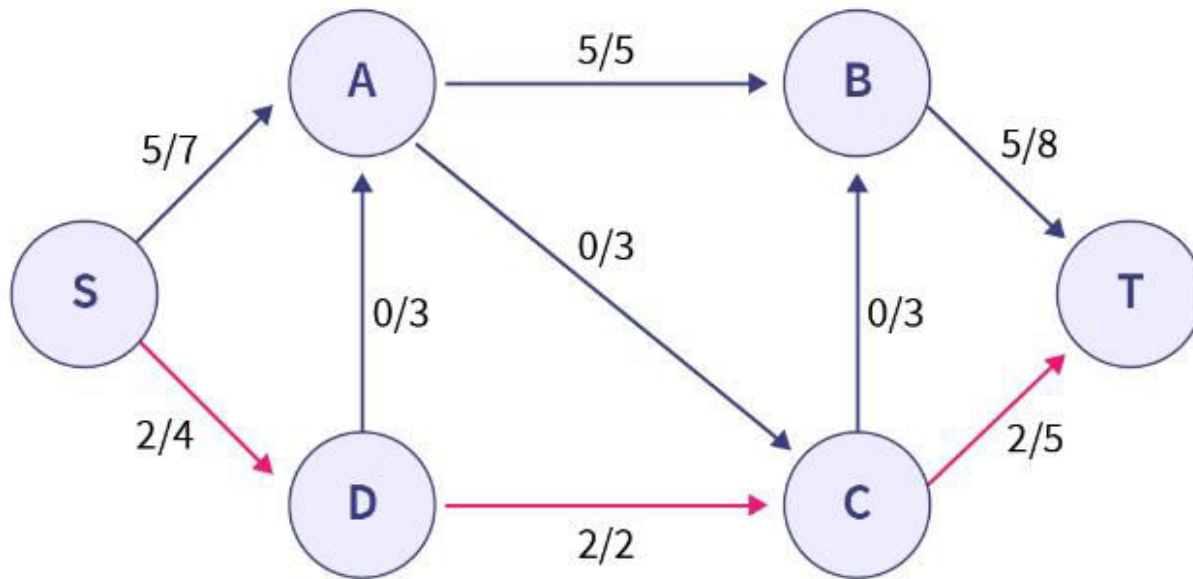
Use the given network to demonstrate how the Ford-Fulkerson method works.

- We can see that the initial flow of all the paths is 0. Now we will search for an augmenting path in the network.
- One such path is $s \rightarrow A \rightarrow B \rightarrow t$ with residual capacities as 7, 5, and 8. Their minimum is 5, so as per the Ford-Fulkerson method we will increase a flow of 5 along the path.
- Residual Capacity = Original Capacity - Flow



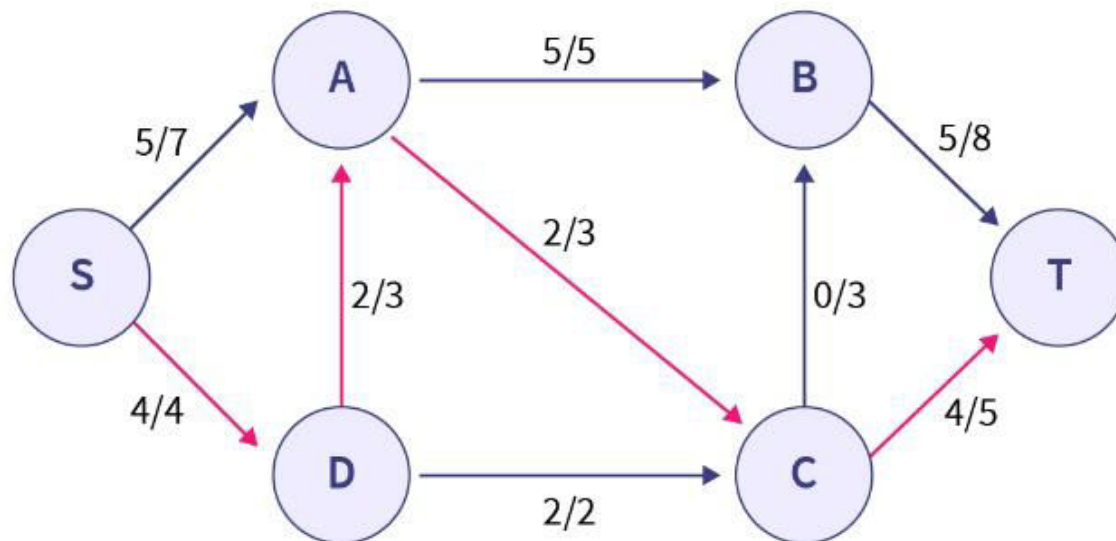
Maximum
Flow=Maximum Flow+5

- Now, we will check for other possible augmenting paths, one such path can be $s \rightarrow D \rightarrow C \rightarrow t$ with residual capacities as 4, 2, and 5 of which 2 is the minimum. So we will increase a flow of 2 along the



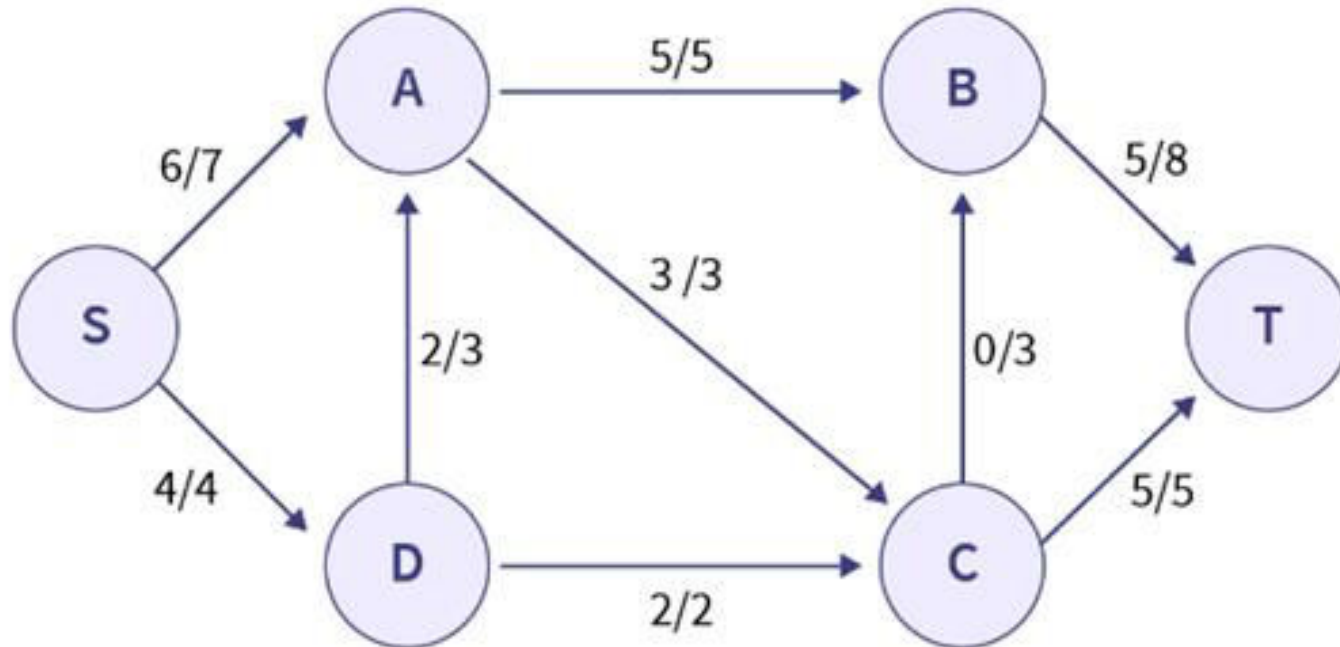
Maximum Flow=5+2

- Check for other possible augmenting paths, one such path can be $s \rightarrow D \rightarrow A \rightarrow C \rightarrow t$ with residual capacities as 2, 3, 3 and 3 of which 2 is the minimum. So we will increase a flow of 2 along the path.



Maximum Flow = $5 + 2 + 2$

Check for other possible augmenting paths, one such path can be $s \rightarrow A \rightarrow C \rightarrow t$ with residual capacities as 2, 1 and 1 of which 1 is the minimum. So we will increase a flow of 1 along the path.



$$\text{Maximum Flow} = 5 + 2 + 2 + 1 = 10$$