Dynamic Programming

Dr. Waheed Ahmed

Farrukh Salim Shaikh

Dynamic Programming

- Big idea, hard, yet simple.
- Powerful algorithmic design technique.
- Large class of seemingly exponential problems have a polynomial solution (“only”) via DP
- Particularly for optimization problems (min / max).

- DP ≈ “controlled brute force”
- * DP ≈ recursion + re-use

Dynamic Programming (History)

- Richard E. Bellman (1920-1984)

- Richard Bellman received the IEEE Medal of Honor, 1979. “Bellman . . . explained that he invented the name ‘dynamid programming’ to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who ‘had a pathological fear and hatred of the term, research’. He settled on the term ‘dynamic programming’ because it would be difficult to give a ‘pejorative meaning’ and because ‘it was something not even a Congressman could object to’ ” [John Rust 2006]

Fibonacci Numbers

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

Naive Algorithm

follow recursive definition

```
fib(n):
    if $n \le 2$: return $f = 1$
    else: return $f = \text{fib}(n-1) + \text{fib}(n-2)$
```

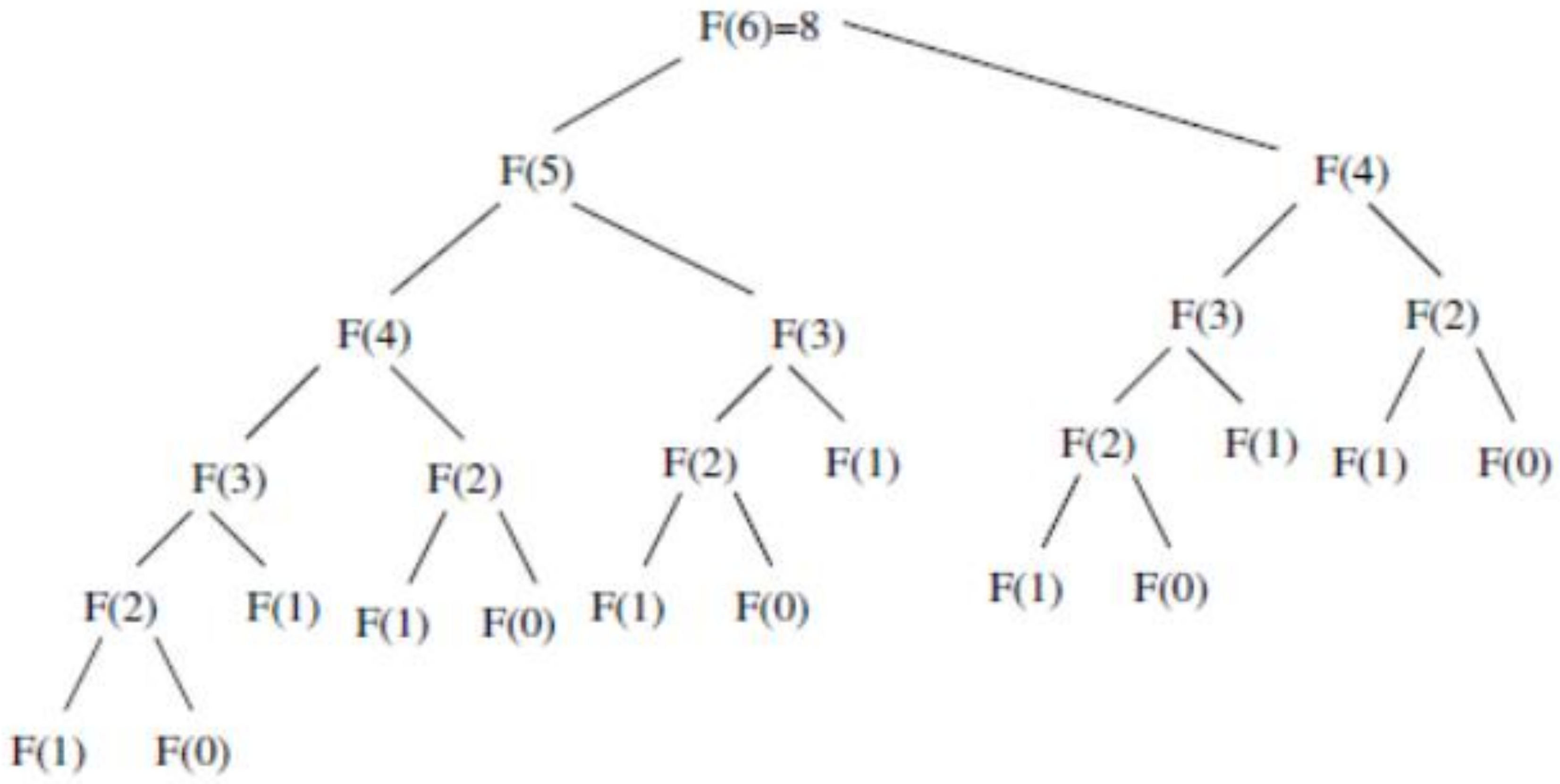
$$\implies T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$$

$$\geq 2T(n-2) + O(1) \geq 2^{n/2}$$

<u>EXPONENTIAL — BAD!</u>

Fibonacci Numbers

$F_0 = 0$ and $F_1 = 1$ $\quad\quad F_n = F_{n-1} + F_{n-2}$



Dynamic Programming

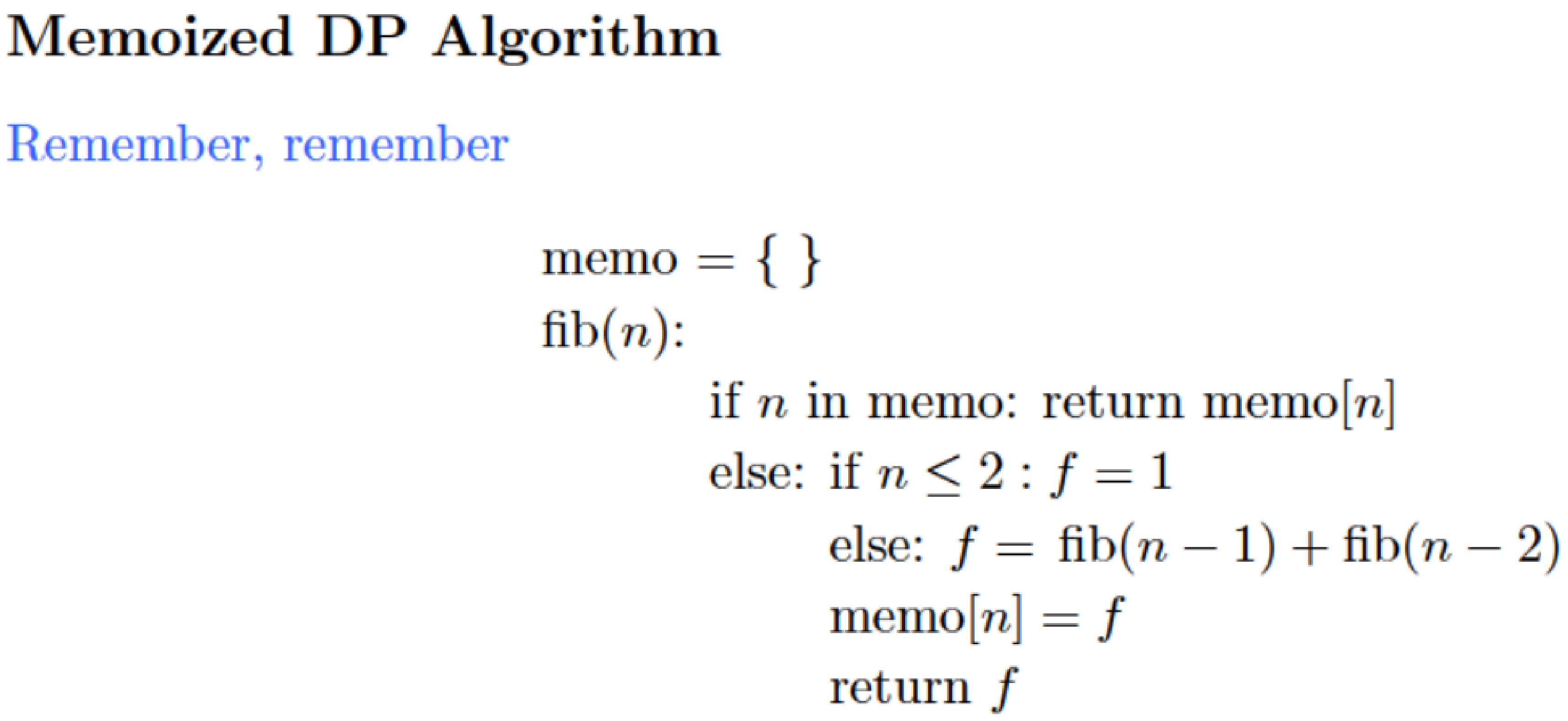
Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)

Fibonacci Numbers



Memoized DP Algorithm

Remember, remember

```
memo = { }
fib($n$):
        if $n$ in memo: return memo[$n$]
        else: if $n \leq 2 : f = 1$
              else: $f =$ fib($n-1$) + fib($n-2$)
              memo[$n$] = $f$
              return $f$
```

Fibonacci Numbers

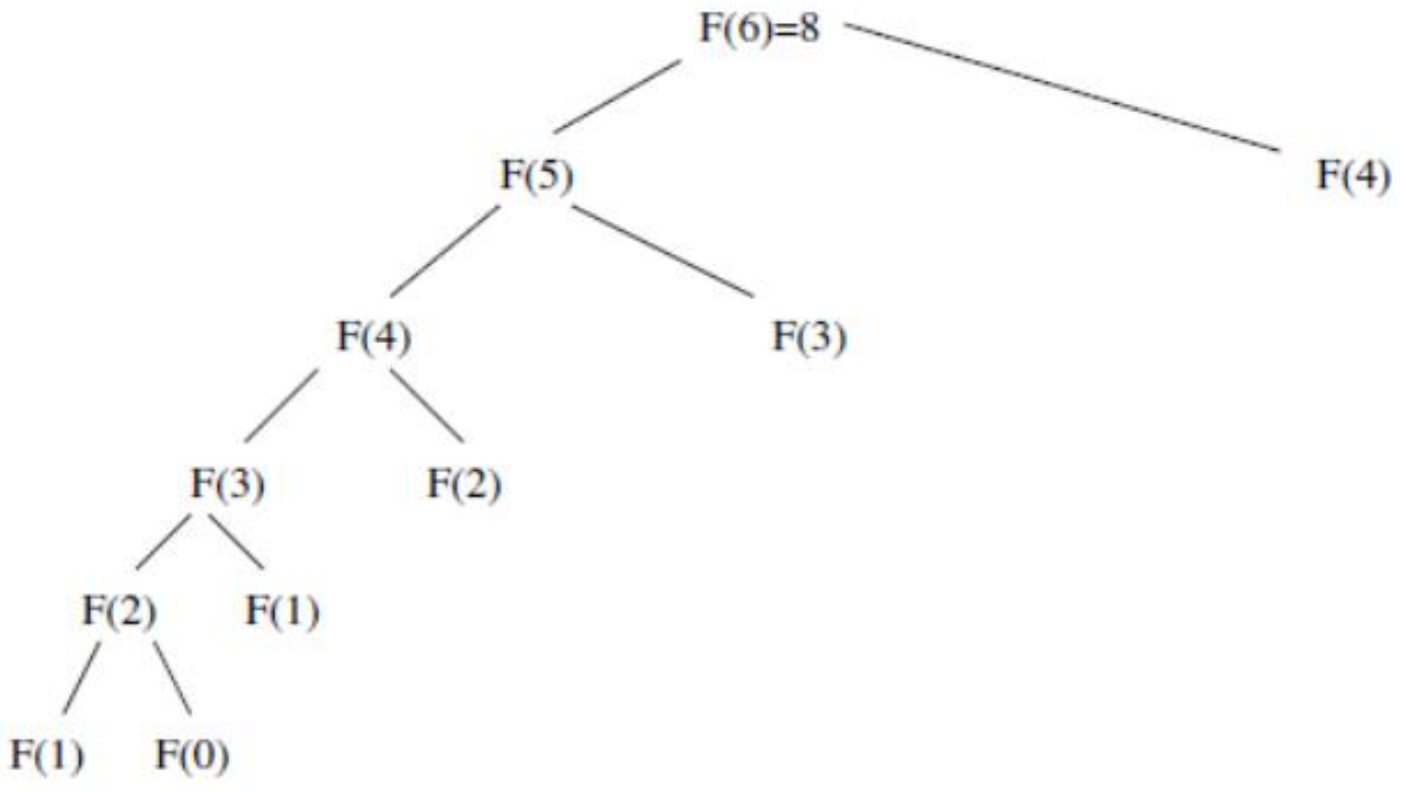
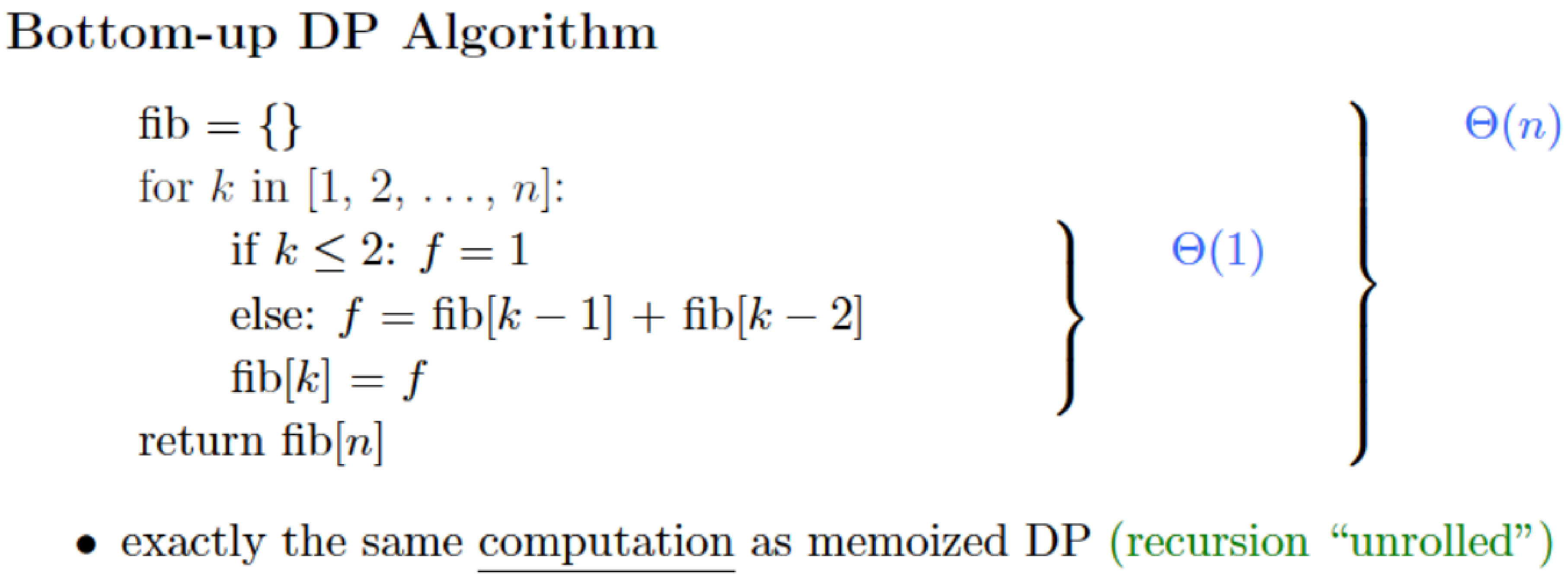


Figure 8.2: The Fibonacci computation tree when caching values

- fib(k) only recurses for the first time called, ∀k
- only n non-memoized calls:

❑ POLYNOMIAL — GOOD!

Bottom-UP DP



Bottom-up DP Algorithm

```
fib = {}                                                    } $\Theta(n)$
for $k$ in [1, 2, . . . , $n$]:
        if $k \le 2$: $f = 1$                              } $\Theta(1)$
        else: $f$ = fib[$k - 1$] + fib[$k - 2$]
        fib[$k$] = $f$
return fib[$n$]
```

- exactly the same <u>computation</u> as memoized DP (recursion "unrolled")

Different algorithm approaches

1. **Brute-force.** Try all possibilities until a satisfactory solution is found
2. **Divide and Conquer.** Divide problem into smaller independent sub-problems and then recursively solves these sub-problems to build solution.
3. **Greedy Algorithm.** It is used to find the best solution by taking best sub-solution at every step.
4. **Dynamic programming algorithm .** Break problem into smaller overlapping sub-problems. Sub-problems that are repeated are solved only once and result is stored (which is called memoization) and this stored result is used next time rather than recomputing solution for another same sub-problem.

Optimization Problems

- For most optimization problems you want to find, not just ***a*** solution, but the ***best*** solution.
- A ***greedy algorithm*** sometimes works well for optimization problems. It works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences.
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

Greedy algorithms

- Make choices one-at-a-time.
- Never look back.
- Hope for the best.

Greedy Approach

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - 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 Approach

- Does **greedy-choice** property lead us to optimize solution?
- **Greedy algorithms** mostly (but not **always**) fail to find the globally **optimal solution** because they usually **do** not operate exhaustively on all the data.

- In general, when are greedy algorithms a good idea?
- When the problem exhibits especially **nice** optimal substructure.
  - i.e. **locally optimal choice** resulted in a **globally optimal solution**

Coin Change Problem

Coin Change Problem

- Goal: Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

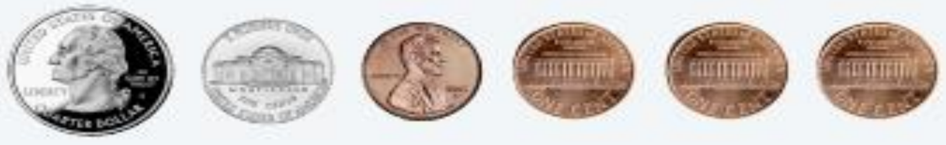
- A greedy algorithm to do this would be:
  At each step, take the largest possible bill or coin that does not overshoot.

- **For US money, the greedy algorithm always gives the optimum solution**

Coin Change Problem

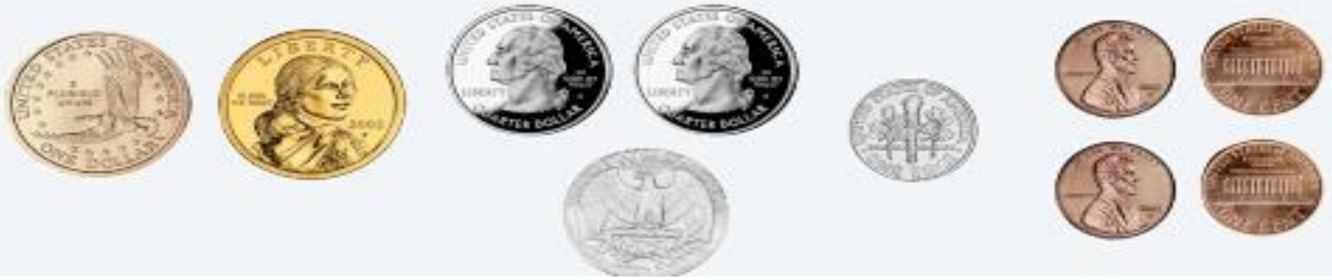
- Goal: Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex. 34¢.



Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. $2.89.



Coin Change Problem

```
CASHIERS-ALGORITHM ($x$, $c_1$, $c_2$, …, $c_n$)
SORT $n$ coin denominations so that $c_1 < c_2 < … < c_n$
$S \leftarrow \phi$        ⟵ set of coins selected
WHILE $x > 0$
    $k \leftarrow$ largest coin denomination $c_k$ such that $c_k \leq x$
    IF no such $k$, RETURN "no solution"
    ELSE
        $x \leftarrow x - c_k$
        $S \leftarrow S \cup \{ k \}$
RETURN $S$
```

Activity Selection problem: Greedy Approach

An Activity Selection Problem (Conference Scheduling Problem)

- **Input: A set of activities S = $\{a_1, \ldots, a_n\}$**
- Each activity has start time and a finish time
  - $a_i = (s_i, f_i)$
- Two activities are compatible if and only if their interval does not overlap
- **Output: a maximum-size subset of mutually compatible activities**

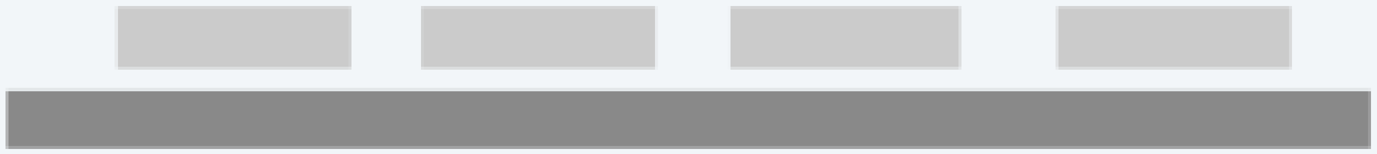
Activity Selection problem: Greedy Approach

- Greedy template: Consider job in some natural order
- Take each job provided it’s compatible with the ones already taken.

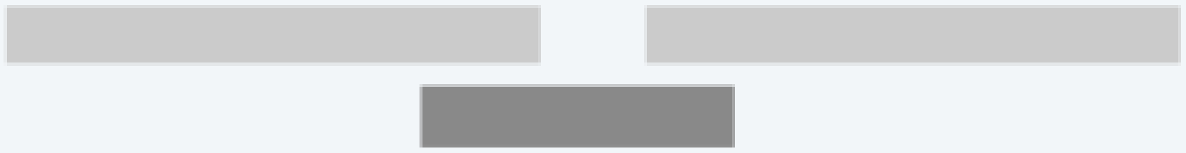
- [Earliest start time] Consider jobs in ascending order of sj.
- [Earliest finish time] Consider jobs in ascending order of gj.
- [Shortest interval] Consider jobs in ascending order of fj – sj.

Activity Selection problem: Greedy Approach

counterexample for earliest start time



counterexample for shortest interval



Activity Selection problem: Greedy Approach

Here are a set of start and finish times

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

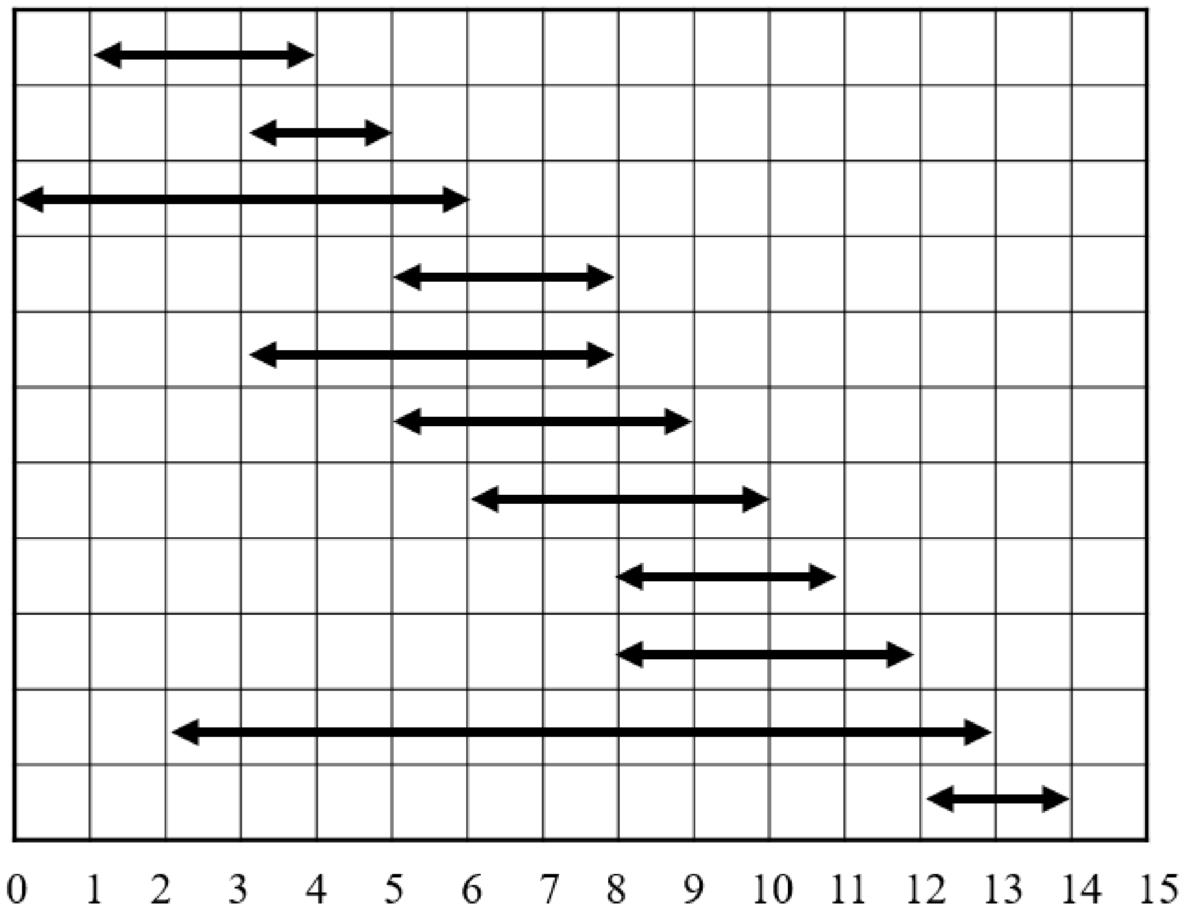
What is the maximum number of activities that can be completed?

Activity Selection problem: Greedy Approach

- Greedy approach is :
    - Select the activity with the earliest finish
    - Eliminate the activities that could not be scheduled
    - Repeat!

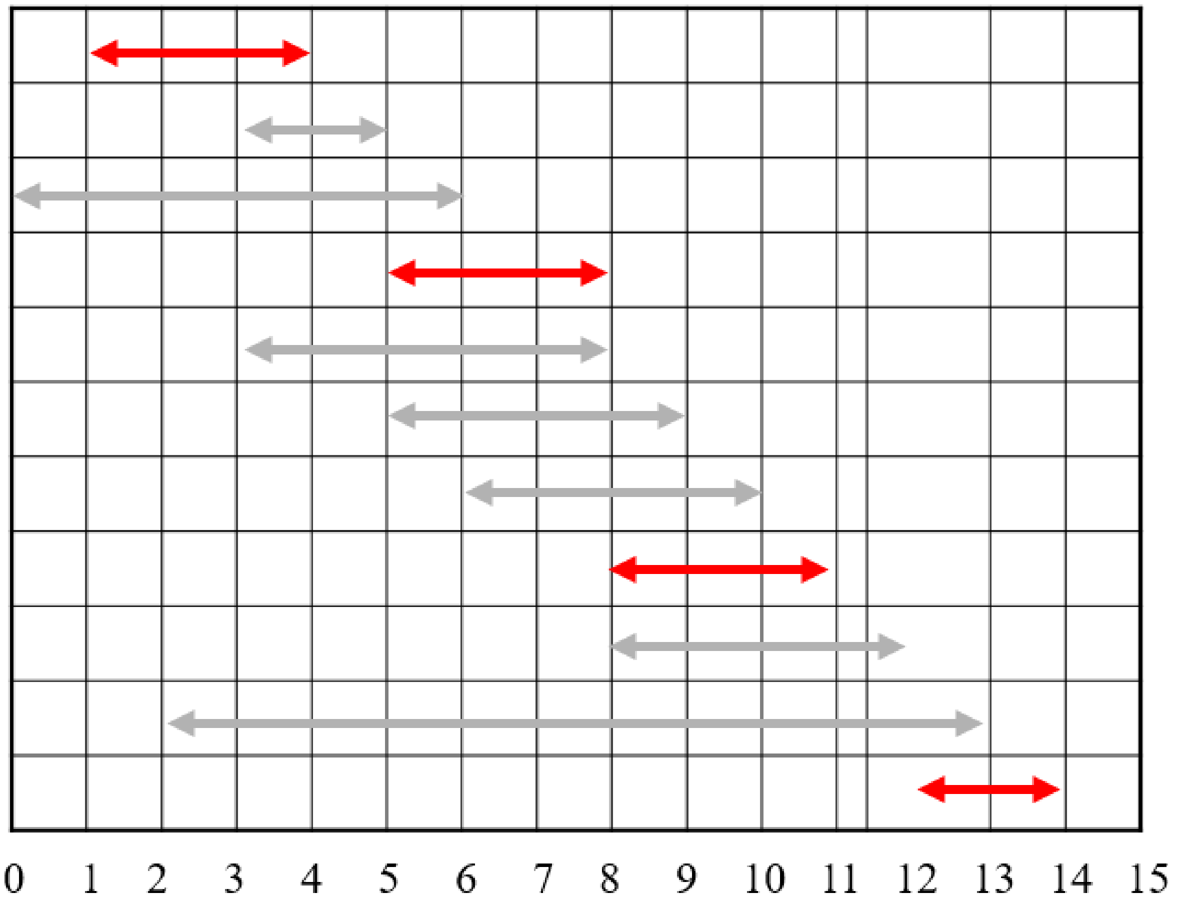
Activity Selection problem: Greedy Approach

- All activities :



Activity Selection problem: Greedy Approach

- Selected activities:



Activity Selection problem: Greedy Approach

- Greedy approach is good for activity selection problem.
    - Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled
    - The greedy choice is the one that maximizes the amount of unscheduled time remaining

Activity Selection problem: Greedy Approach

```
EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish time so that $f_1 \le f_2 \le \ldots \le f_n$
$A \leftarrow \phi$   $\longleftarrow$ set of jobs selected
FOR $j = 1$ TO $n$
    IF job $j$ is compatible with $A$
        $A \leftarrow A \cup \{j\}$
RETURN $A$
```

At least it's fast

- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise, O(nlog(n)) if you have to sort them first.

Knapsack problem

Knapsack Problem

- Greedy approach is also good for fractional knapsack problem but not for 0/1 knapsack.
- **Fractional knapsack** In which you can take fraction of item if you want
- **0/1 knapsack** In which you can only take complete item or leave it but you cannot take fraction of it

0/1 Knapsack problem: two versions

Capacity: **10**

Item:	🥑	🧲	🍇	🐨	🚙
Weight:	**6**	**2**	**4**	**3**	**11**
Value:	**20**	**8**	**14**	**13**	**35**

UNBOUNDED KNAPSACK

We have infinite copies of all the items. What's the most valuable way to fill the knapsack?

🧲 🧲 🐨 🐨

Total weight: **2 + 2 + 3 + 3 = 10**

Total value: **8 + 8 + 13 + 13 = 42**

0/1 KNAPSACK

We have only one copy of each item. What's the most valuable way to fill the knapsack?

🧲 🍇 🐨

Total weight: **2 + 4 + 3 = 9**

Total value: **8 + 14 + 13 = 35**

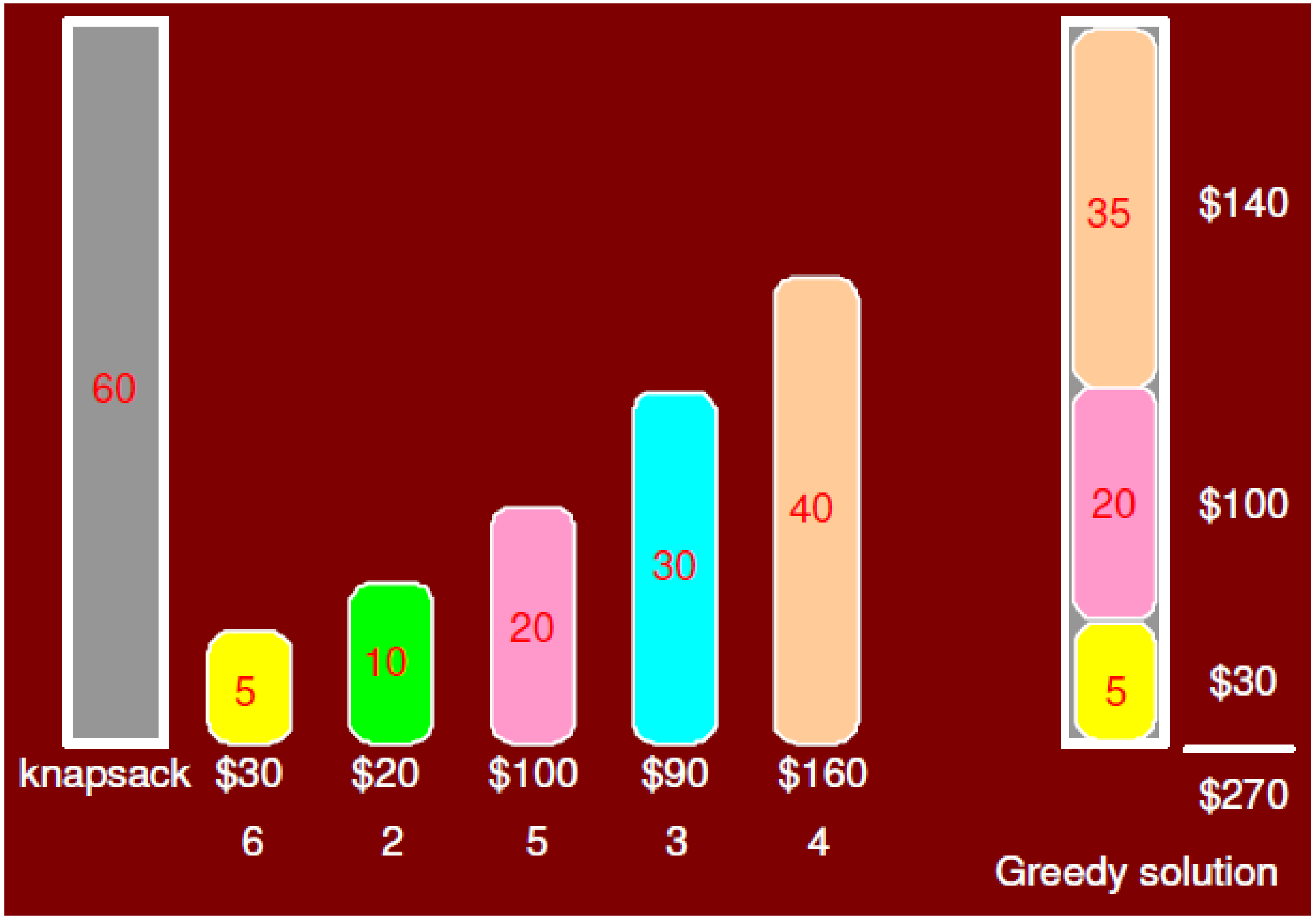
Knapsack Problem

- ❖ Suppose “i” item has value “v(i)” and weight “w(i)”. Capacity of knapsack (bag) is W. Then greedy approach would be :
- Sort in decreasing order of value/weight i-e v(i)/w(i)
- Now start selecting items till you fill the bag.

- ❖ In fractional knapsack, you utilize complete capacity W because you can take fraction of items but in 0/1 knapsack, you may or may not.

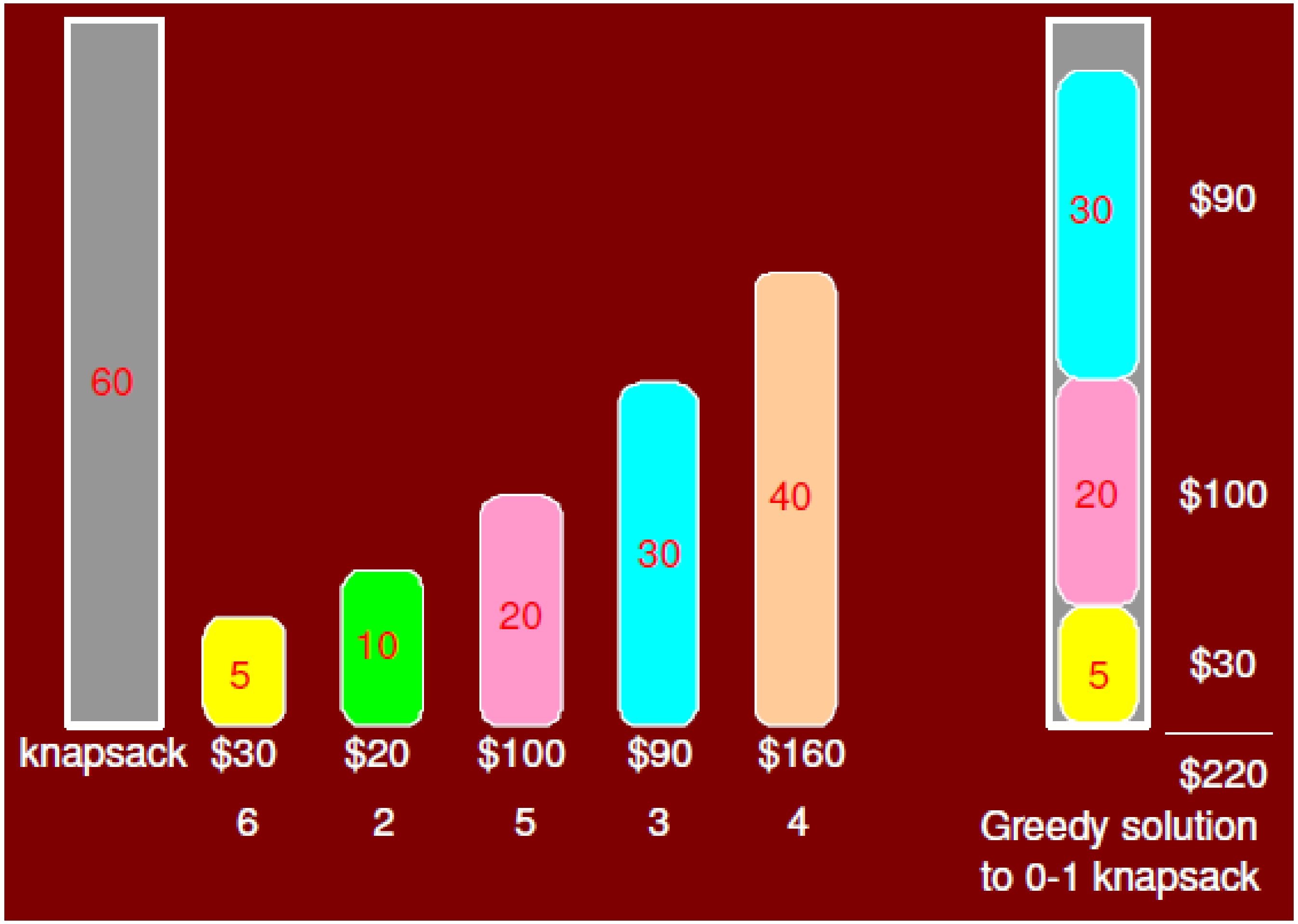
Knapsack Problem : Greedy approach

Greedy/optimal solution to fractional knapsack is :



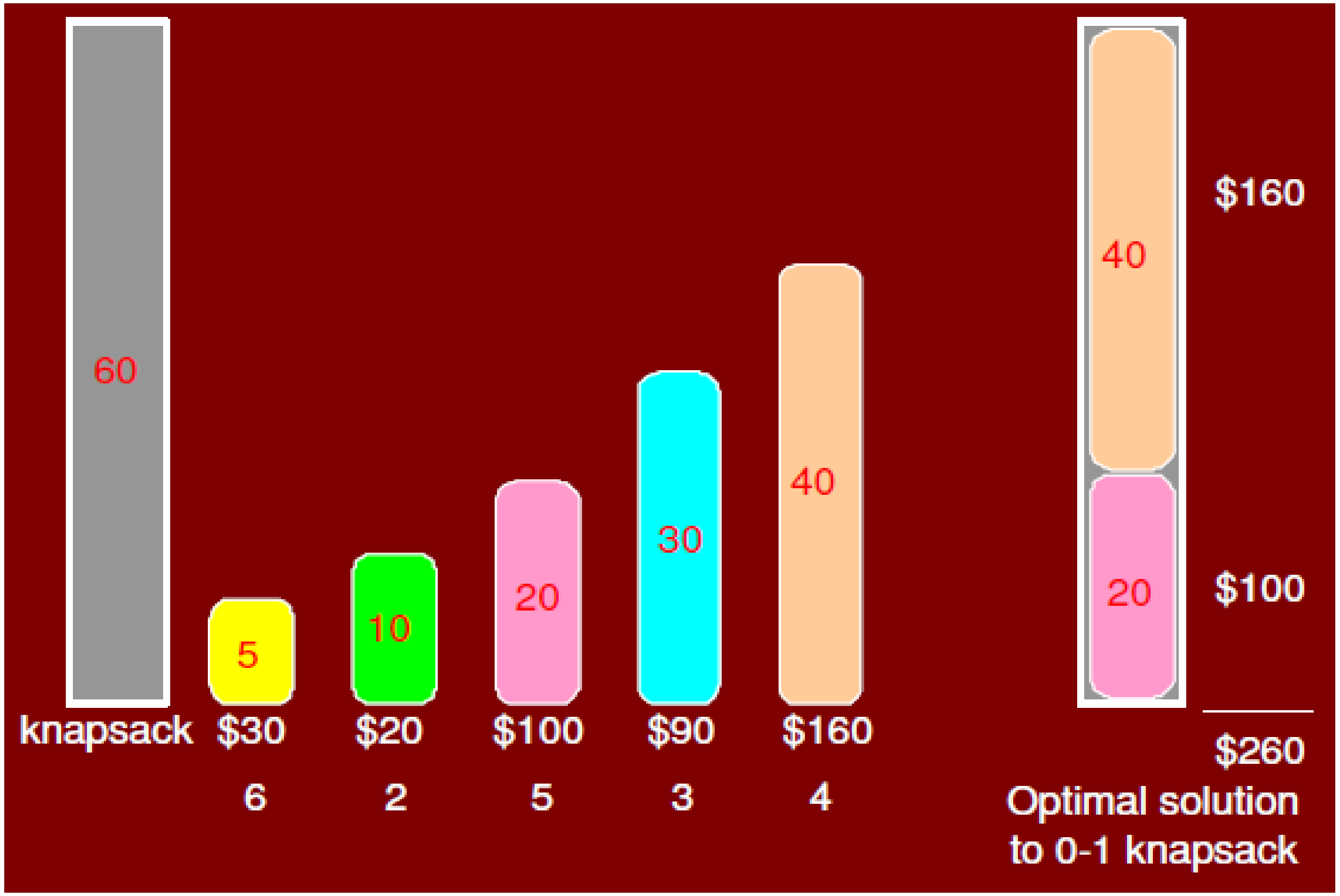
Knapsack Problem : Greedy approach

- Greedy solution to 0/1 knapsack :



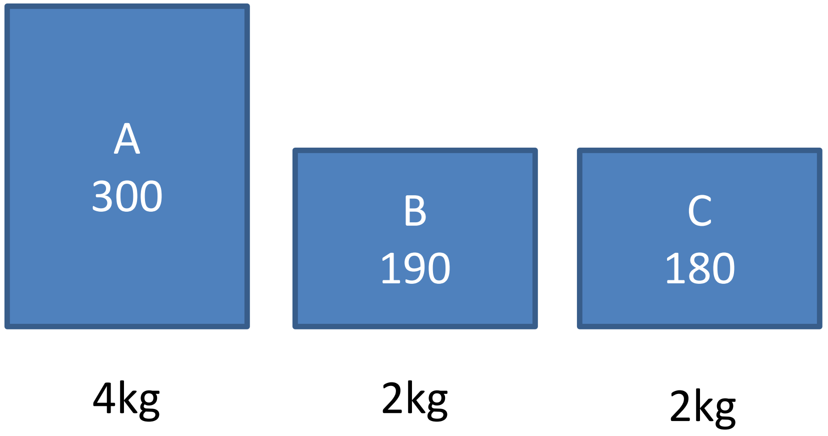
Knapsack Problem : Greedy approach

- Optimal/non-greedy solution to 0/1 knapsack :



0/1 Knapsack problem

- Given a set ‘S’ of ‘n’ items,
- such that each item i has a positive value v_i and positive weight w_i
- The goal is to find maximum-benefit subset that does not exceed the given weight W.



Maximum weight:
W = 4kg

Optima Solution:
Item B and C

Benefit:
370

Developing a Recursive Solution

- Let OPT be an optimal solution
- Note that presence of item i in OPT, does not forbid any other item j.
- Item j (last one) either belongs to OPT, or it doesn't.
  - If $j \in$ OPT:
    - Optimal solution contains 'j',
    - plus optimal solution of other j – 1 items,
    - But with a reduced maximum weight of $W - w_j$
  - If $j \notin$ OPT:
    - Optimal solution if for j – 1 items,
    - with maximum allowed weight W remain unchanged.

Developing a Recursive Solution

- $w_j > W \Rightarrow j \notin OPT$ ------------------------------ **Leave object**
  - OPT(j,w) = OPT(j – 1, w)

- Otherwise, j is either $\in$ OPT or $\notin$ OPT
- If $j \in$ OPT: ---**Take object**
  - OPT(j,w) = v_j + OPT(j - 1, W - w_j)
- If $j \notin$ OPT: ---**Leave object**
  - OPT(j,w) = OPT(j - 1, w)

- OPT(j,w) = MAX(v_j + OPT(j - 1, W - w_j), OPT(j - 1, w))

Developing a Recursive Solution

- $OPT(j,w) = MAX(v_j + OPT(j - 1, W - w_j), OPT(j - 1, w))$

- $$OPT\ (j,w) = \begin{cases} OPT\ (j - 1, w) & \text{if } w_j > w \\ MAX(v_j + OPT(j - 1, w - w_j), \\ OPT\ (j - 1, w)\ ) & \text{otherwise} \end{cases}$$

- j is in optimal solution iff.

  $$v_j + OPT(j - 1, w - w_j) \geq OPT\ (j - 1, w)$$

A recursive algorithm

```
Knapsack(j,w)
  If $j = 0$ or $w = 0$
        return 0
  else if $w_j > w$
        return Knapsack($j – 1$, w)
  else
        return MAX(vj + Knapsack($j – 1$, $w – w_j$),
                               Knapsack($j – 1$, w))
```

- The initial call is Knapsack(n, W)

A recursive algorithm

```
M-Knapsack(j,w)
  If $j = 0$ or $w = 0$
        return 0
  else if M[j,w] is empty
        $M[j,w] \leftarrow MAX(v_j + Knapsack(j-1, w-w_j),$
                               $Knapsack(j-1, w))$
return M[j,w]
```

- This is an example of pseudo-polynomial problem, since it depends on another parameter W that is independent of the problem size.

Dynamic Programming Algorithm

```
Knapsack(j,w)
 for i ← 0 to n          M[i,0] ← 0
 for w ← 0 to W          M[0,w] ← 0

 for j ← 1 to n
         for w ← 0 to W
                 if $v_j + M[j – 1, w – w_j] \geq M[j – 1, w]$
                         $M[j,w] = v_j + M[j – 1, w – wj]$
                 else
                         $M[j,w] = M[j – 1, w))$
return M[n,W]
```

- Complexity = O(nW) (Right?)

This is an example of **pseudo-polynomial** problem, since it depends on another parameter W that is independent of the problem size.

Example

- Let W = 9
- wi = {2, 3, 4, 5}
- vi = {3, 4, 5, 7}

- M[j,w] ← MAX($\mathbf{v_j}$ **+ M[j – 1, w –** $\mathbf{w_j}$**],** M[j – 1, w])

			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4										
5	4	3										
4	3	2										
3	2	1										
		0										

Example

- Let W = 9
- wi = {2, 3, 4, 5}
- vi = {3, 4, 5, 7}

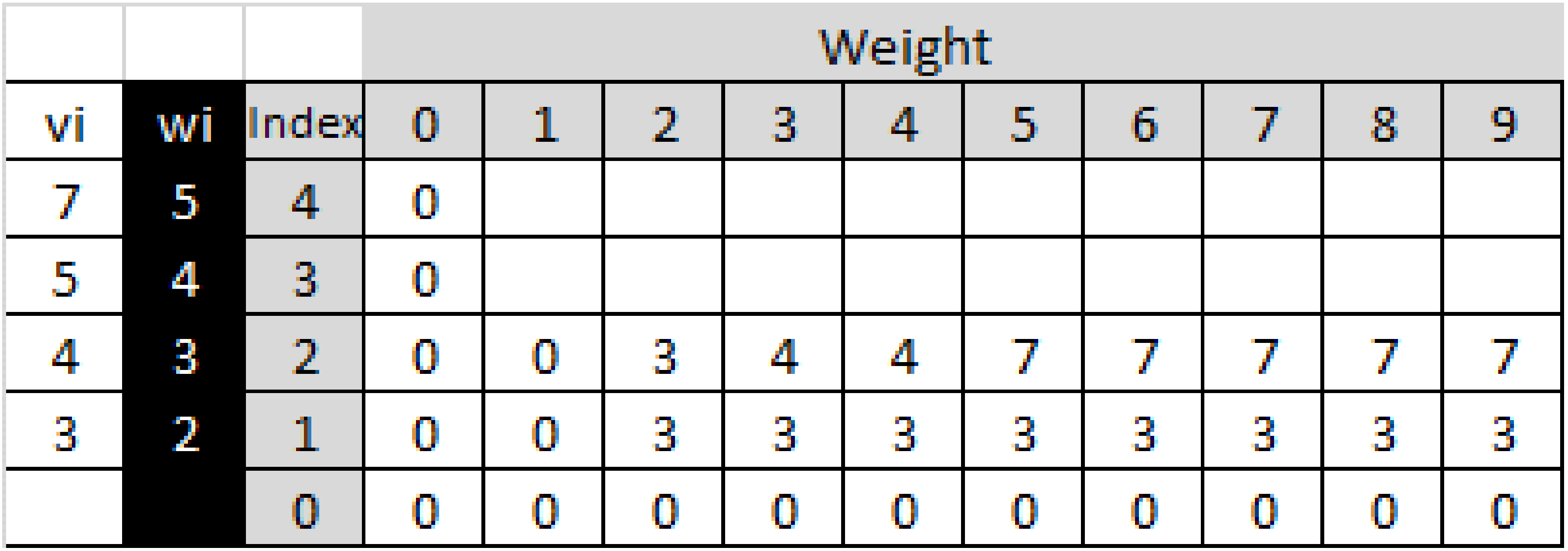
- M[j,w] ← MAX($\mathbf{v_j}$ **+ M[j – 1, w –** $\mathbf{w_j}$**],** M[j – 1, w])

			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4	0									
5	4	3	0									
4	3	2	0	0	3							
3	2	1	0	0	3	3	3	3	3	3	3	3
		0	0	0	0	0	0	0	0	0	0	0

Example

- Let W = 9
- wi = {2, 3, 4, 5}
- vi = {3, 4, 5, 7}

- $M[j,w] \leftarrow \text{MAX}(\mathbf{v_j + M[j-1, w-w_j]}, M[j-1, w])$



			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4	0									
5	4	3	0									
4	3	2	0	0	3	4	4	7	7	7	7	7
3	2	1	0	0	3	3	3	3	3	3	3	3
		0	0	0	0	0	0	0	0	0	0	0

Example

- Let W = 9
- wi = {2, 3, 4, 5}
- vi = {3, 4, 5, 7}

- M[j,w] ← MAX($\mathbf{v_j}$ **+ M[j – 1, w –** $\mathbf{w_j}$**],** M[j – 1, w])

			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4	0	0	3	4	5	7	8	10	11	12
5	4	3	0	0	3	4	5	7	8	9	9	12
4	3	2	0	0	3	4	4	7	7	7	7	7
3	2	1	0	0	3	3	3	3	3	3	3	3
		0	0	0	0	0	0	0	0	0	0	0

Example (contd…)

			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4	0	0	3	4	5	7	8	10	11	12
5	4	3	0	0	3	4	5	7	8	9	9	12
4	3	2	0	0	3	4	4	7	7	7	7	7
3	2	1	0	0	3	3	3	3	3	3	3	3
		0	0	0	0	0	0	0	0	0	0	0

- More than one optimal solutions might be possible, but DP provides only one optimal solution at a time

Complete DP algorithm

```
Knapsack(j,w)
 for i ← 0 to n          M[i,0] ← 0    **s[i] ← 0**
 for w ← 0 to W          M[0,w] ← 0

 for j ← 1 to n
         for w ← 0 to W
                 if vj + M[j – 1, w – wj] ≥ M[j – 1, w]
                         M[j,w] = M[j – 1, w – wj]
                         **s[j] = 1**
                 else
                         M[j,w] = M[j – 1, w))
return M[n,W]
```

Complete DP algorithm

```
Knapsack-Find-Solution
CW = W
for (i = n downto 1)
    if (s[i,CW] == 1)
        output i
        CW = CW – w[i]
```

Complete DP algorithm - II

```
Knapsack(j,w)
 for i ← 0 to n                    M[i,0] ← 0          **s[i] ← 0**
 for w ← 0 to W   M[0,w] ← 0

 for j ← 1 to n
          for w ← 0 to W
                   if vj + M[j – 1, w – wj] ≥ M[j – 1, w]
                        M[j,w] = M[j – 1, w – wj]
                        **s[j] = 1**
                   else
                        M[j,w] = M[j – 1, w))
**CW = W**
**for (i = n downto 1)**
          **if (s[i,CW] == 1)**
                   **output i**
                   **CW = CW – w[i]**
**return M[n,W]**
```

Longest Common Subsequence

- Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.
- For example: if X = {A,B,C,B,D,A,B} and Y = {B,D,C,A,B,A}, the sequence {B,C,A} is a common subsequence of both X and Y.
- The sequence {B,C,A} is not a longest common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence {B,C,B,A}, which is also common to both X and Y, has length 4.
- The sequence {B,C,B,A} is an LCS of X and Y, as is the sequence {B,D,A,B}, since X and Y have no common subsequence of length 5 or greater.
- Applications include Biological applications that often need to compare the DNA of two (or more) different organisms. We can say that two DNA strands are similar if one is a substring of the other.

Longest Common Subsequence

- A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
- In LCS, we have to find the Longest Common Subsequence that is in the same relative order.
- String of length n has 2^n different possible subsequences.

Longest Common Subsequence

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Recurrence Relation for LCS

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

- Case 1: Base Case
- Case 2: If the two alphabets match, add 1 with the value of the diagonal
- Case 3: If the two alphabets do not match, select value from top or left (which ever is greater)

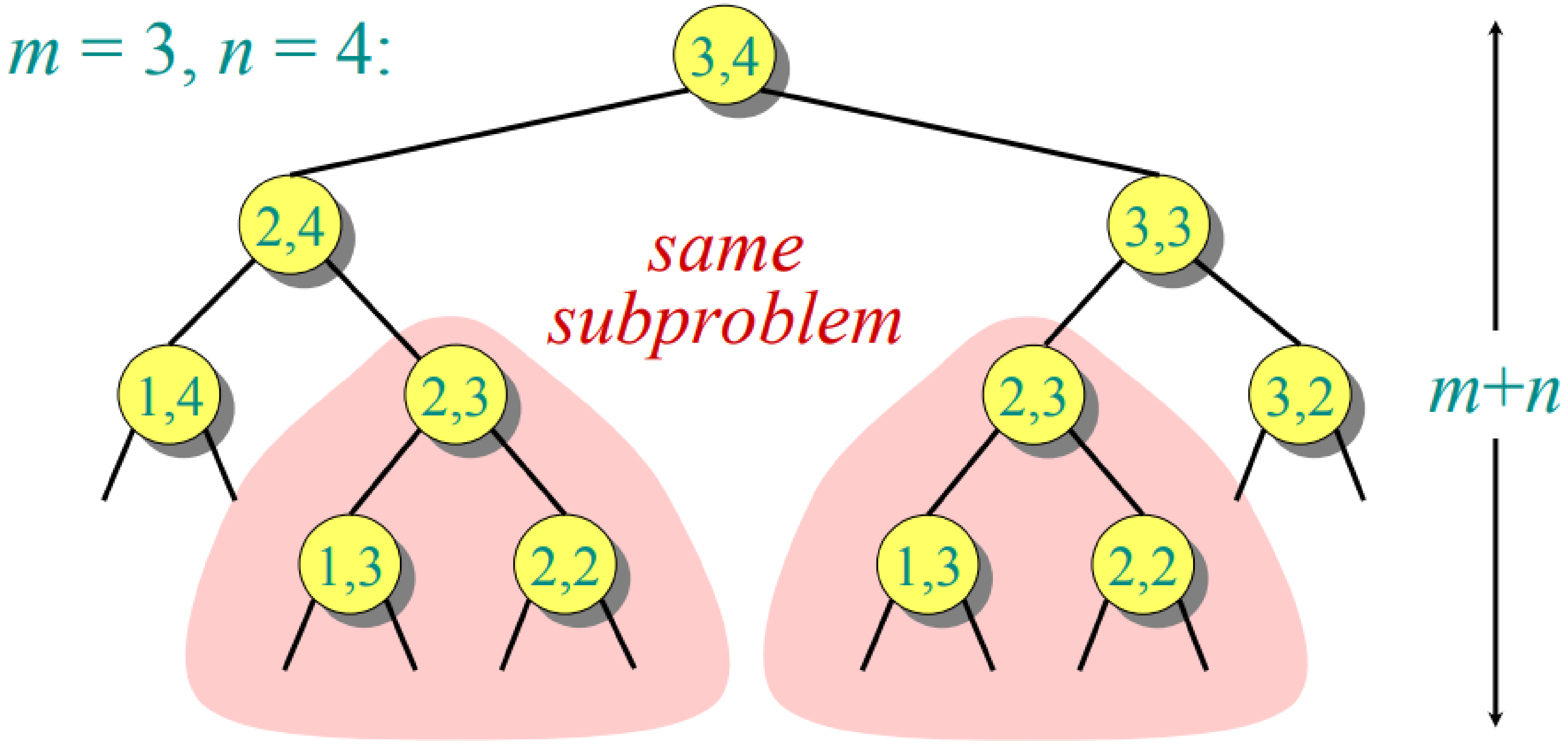
Algorithm for LCS

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

- Complexity = O(nm) (Right?)

```
PRINT-LCS($b, X, i, j$)
if $i == 0$ or $j == 0$
    return
if $b[i, j] == $ "↖"
    PRINT-LCS($b, X, i-1, j-1$)
    print $x_i$
elseif $b[i, j] == $ "↑"
    PRINT-LCS($b, X, i-1, j$)
else PRINT-LCS($b, X, i, j-1$)
```

Recursion Tree

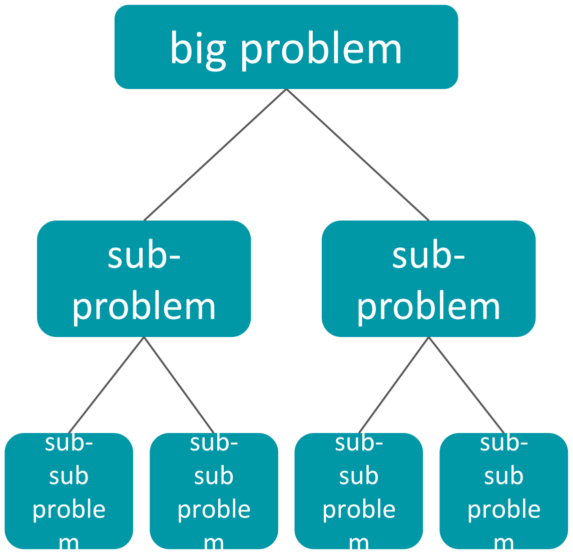
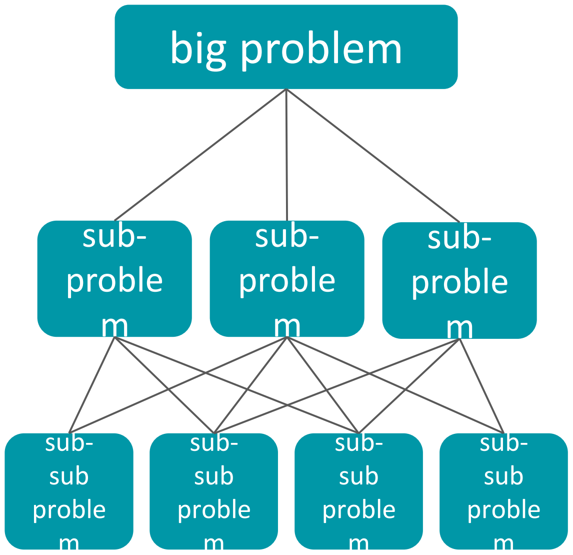
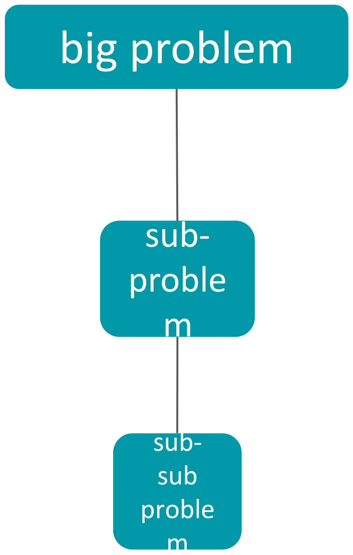


Example # 2

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Running Time and Memory O(mn)
Output: Priden

D&C vs. DP vs. GREEDY

DIVIDE-AND-CONQUER	DYNAMIC PROGRAMMING	GREEDY
 big problem sub-problem sub-problem sub-sub problem sub-sub problem sub-sub problem sub-sub problem	 big problem sub-problem sub-problem sub-problem sub-sub problem sub-sub problem sub-sub problem sub-sub problem	

Steps for DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

Aside – Optimal substructure for Shortest Path

- Sub-path of a shortest path is also a shortest path.

Matrix Chain Multiplication

Matrix-chain Multiplication

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- Matrix multiplication is associative. So parenthesizing does not change result. There are 5 possible ways:
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Matrix Multiplication

A 2 * **3**

2	3	4
5	6	7

B **3** * 4

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

A * B 2 * 4

2*2 + 3*1 + 4*2			

❖**Number of multiplication operations performed while multiplying two matrices**

(# Rows of 1st matrix) x (# columns of 1st matrix) x (# Columns of 2nd matrix)

OR

(# Rows of 1st matrix) x (# rows of 2nd matrix) x (# Columns of 2nd matrix)

<u>For example:</u>

<u>No. of multiplication operations when A and B are multiplied = **2 x 3 x 4** = 24 operations</u>

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)

Result: Matrix $C_{p \times r}$ resulting from the product $A \cdot B$

```
MATRIX-MULTIPLY($A_{p \times q}$ , $B_{q \times r}$)
1.    for $i \leftarrow 1$ to $p$
2.        for $j \leftarrow 1$ to $r$
3.            $C[i, j] \leftarrow 0$
4.            for $k \leftarrow 1$ to $q$
5.                $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6.    return $C$
```

Total Number of Multiplications = p*q*r

Matrix-chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 = 2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 = 50{,}000$ scalar multiplications

Total: 75,000

Matrix-chain Multiplication ...contd

- Matrix-chain multiplication problem
  - Given a chain A_1, A_2, …, A_n of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1A_2…A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Matrix chain multiplication: Dynamic Programming

- DP breaks the problem into subproblems whose solutions can be combined to solve the global subproblem.

A_1	A_2	A_3	A_4	=	$A_{1..4}$
4*5	5*2	2*8	8*7		4*7

- Chain of matrices = A = {A_1,A_2,A_3,A_4}   ; index starts from 1
- Dimensions of matrices = P = {4,5,2,8,7}   ; index starts from 0

- Let $A_{i..j}$ be the result of matrices i through j.
- It is easy to see that $A_{i..j}$ is a p_{i-1} * p_j matrix.

Matrix chain multiplication: Dynamic Programming

- Let $1 \le i < j \le n$
- Let m[i,j] is minimum number of multiplications from i to j, using recursive formulations as:
- If i = j, then m[i,i] = 0 //Only one matrix, diagonal entries
- For **$i < j$**, we need to find the product $A_{i..j}$
- This can be split by k, **$i \le k < j$**, so final product: $A_{i..k}$ ***** $A_{k+1..j}$

Matrix chain multiplication: Dynamic Programming

- The optimum time to compute $A_{i..k}$ is m[i,k] and optimum time for $A_{k+1..j}$ is m[k+1,j]

- Since $A_{i..k}$ is a $p_{i-1} * p_k$ matrix and $A_{k+1..j}$ is $p_{\mathbf{k}} * p_j$ matrix, the number of multiplications will be $p_{i-1} * p_k$ * pj.

m[i, i] = 0

m[i, j] = $\min_{i \leq k < j}$ (m[i, k] + m[k+1, j] + $p_{i-1} * p_k$ * pj)

Matrix chain multiplication: Dynamic Programming

We do not want to calculate m entries recursively. So how should we proceed? We will fill m along the diagonals. Here is how. Set all $m[i, i] = 0$ using the base condition. Compute cost for multiplication of a sequence of 2 matrices. These are $m[1, 2], m[2, 3], m[3, 4], \ldots, m[n-1, n]$. $m[1, 2]$, for example is

$$m[1, 2] = m[1, 1] + m[2, 2] + p_0 \cdot p_1 \cdot p_2$$

For example, for m for product of 5 matrices at this stage would be:

$m[1, 1]$	$\leftarrow m[1, 2]$ $\downarrow$			
	$m[2, 2]$	$\leftarrow m[2, 3]$ $\downarrow$		
		$m[3, 3]$	$\leftarrow m[3, 4]$ $\downarrow$	
			$m[4, 4]$	$\leftarrow m[4, 5]$ $\downarrow$
				$m[5, 5]$

$$m[i, j] = \min_{i \leq k < j} \left(m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \right)$$

Matrix chain multiplication: Dynamic Programming

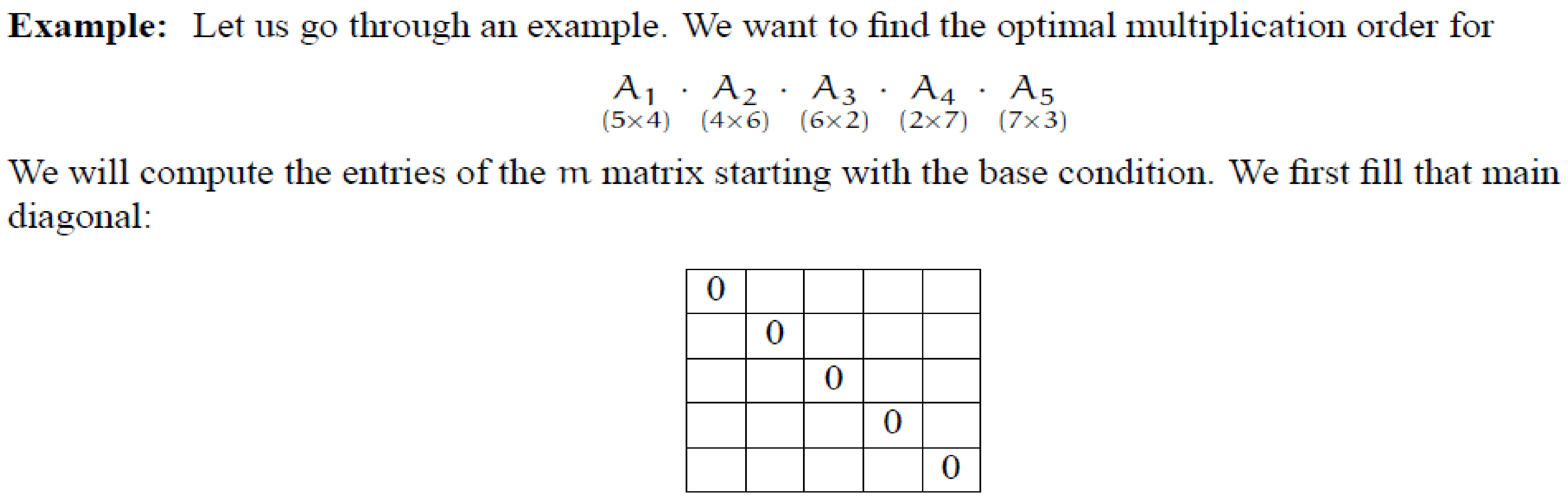
Next, we compute cost of multiplication for sequences of three matrices. These are $m[1,3], m[2,4], m[3,5], \ldots, m[n-2,n]$. $m[1,3]$, for example is

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 \end{cases}$$

We repeat the process for sequences of four, five and higher number of matrices. The final result will end up in $m[1,n]$.

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$$

Matrix chain multiplication: Dynamic Programming



Example: Let us go through an example. We want to find the optimal multiplication order for

$$\underset{(5\times4)}{A_1} \cdot \underset{(4\times6)}{A_2} \cdot \underset{(6\times2)}{A_3} \cdot \underset{(2\times7)}{A_4} \cdot \underset{(7\times3)}{A_5}$$

We will compute the entries of the m matrix starting with the base condition. We first fill that main diagonal:

0				
	0			
		0		
			0	
				0

- Here A = { A1,A2,A3,A4,A5} ; starts from index 1
- and P = {5,4,6,2,7,3}; starts from index 0

Matrix chain multiplication: Dynamic Programming

Next, we compute the entries in the first super diagonal, i.e., the diagonal above the main diagonal:

$$m[1,2] = m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2 = 0 + 0 + 5 \cdot 4 \cdot 6 = 120$$
$$m[2,3] = m[2,2] + m[3,3] + p_1 \cdot p_2 \cdot p_3 = 0 + 0 + 4 \cdot 6 \cdot 2 = 48$$
$$m[3,4] = m[3,3] + m[4,4] + p_2 \cdot p_3 \cdot p_4 = 0 + 0 + 6 \cdot 2 \cdot 7 = 84$$
$$m[4,5] = m[4,4] + m[5,5] + p_3 \cdot p_4 \cdot p_5 = 0 + 0 + 2 \cdot 7 \cdot 3 = 42$$

The matrix m now looks as follows:

0	120			
	0	48		
		0	84	
			0	42
				0

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$$

Matrix chain multiplication: Dynamic Programming

We now proceed to the second super diagonal. This time, however, we will need to try two possible values for k. For example, there are two possible splits for computing $m[1,3]$; we will choose the split that yields the minimum:

$$m[1,3] = m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 == 0 + 48 + 5 \cdot 4 \cdot 2 = 88$$
$$m[1,3] = m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 = 120 + 0 + 5 \cdot 6 \cdot 2 = 180$$
$$\text{the minimum } m[1,3] = 88 \text{ occurs with } k = 1$$

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j)$$

Matrix chain multiplication: Dynamic Programming

Similarly, for $m[2,4]$ and $m[3,5]$:

$$m[2,4] = m[2,2] + m[3,4] + p_1 \cdot p_2 \cdot p_4 = 0 + 84 + 4 \cdot 6 \cdot 7 = 252$$
$$m[2,4] = m[2,3] + m[4,4] + p_1 \cdot p_3 \cdot p_4 = 48 + 0 + 4 \cdot 2 \cdot 7 = 104$$
$$\text{minimum } m[2,4] = 104 \text{ at } k = 3$$

$$m[3,5] = m[3,3] + m[4,5] + p_2 \cdot p_3 \cdot p_5 = 0 + 42 + 6 \cdot 2 \cdot 3 = 78$$
$$m[3,5] = m[3,4] + m[5,5] + p_2 \cdot p_4 \cdot p_5 = 84 + 0 + 6 \cdot 7 \cdot 3 = 210$$
$$\text{minimum } m[3,5] = 78 \text{ at } k = 3$$

$$m[i,j] = \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right)$$

Matrix chain multiplication: Dynamic Programming

With the second super diagonal computed, the m matrix looks as follow:

0	120	88		
	0	48	104	
		0	84	78
			0	42
				0

Matrix chain multiplication: Dynamic Programming

We repeat the process for the remaining diagonals. However, the number of possible splits (values of k) increases:

$$m[1,4] = m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 104 + 5 \cdot 4 \cdot 7 = 244$$
$$m[1,4] = m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 120 + 84 + 5 \cdot 6 \cdot 7 = 414$$
$$m[1,4] = m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 = 88 + 0 + 5 \cdot 2 \cdot 7 = 158$$
$$\text{minimum } m[1,4] = 158 \text{ at } k = 3$$

$$m[2,5] = m[2,2] + m[3,5] + p_1 \cdot p_2 \cdot p_5 = 0 + 78 + 4 \cdot 6 \cdot 3 = 150$$
$$m[2,5] = m[2,3] + m[4,5] + p_1 \cdot p_3 \cdot p_5 = 48 + 42 + 4 \cdot 2 \cdot 3 = 114$$
$$m[2,5] = m[2,4] + m[5,5] + p_1 \cdot p_4 \cdot p_5 = 104 + 0 + 4 \cdot 7 \cdot 3 = 188$$
$$\text{minimum } m[2,5] = 114 \text{ at } k = 3$$

$$m[i,j] = \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$$

Matrix chain multiplication: Dynamic Programming

0	120	88	158	
	0	48	104	114
		0	84	78
			0	42
				0

That leaves the $m[1,5]$ which can now be computed:

$$\begin{aligned}
m[1,5] &= m[1,1] + m[2,5] + p_0 \cdot p_1 \cdot p_5 = 0 + 114 + 5 \cdot 4 \cdot 3 = 174 \\
m[1,5] &= m[1,2] + m[3,5] + p_0 \cdot p_2 \cdot p_5 = 120 + 78 + 5 \cdot 6 \cdot 3 = 288 \\
m[1,5] &= m[1,3] + m[4,5] + p_0 \cdot p_3 \cdot p_5 = 88 + 42 + 5 \cdot 2 \cdot 3 = 160 \\
m[1,5] &= m[1,4] + m[5,5] + p_0 \cdot p_4 \cdot p_5 = 158 + 0 + 5 \cdot 7 \cdot 3 = 263 \\
&\text{minimum } m[1,5] = 160 \text{ at } k = 3
\end{aligned}$$

$$m[i,j] = \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right)$$

Matrix chain multiplication: Dynamic Programming

Matrix "m"

We thus have the final cost matrix.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

- **Matrix "s"**

and the split k values that led to a minimum m[i, j] value

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

Matrix chain multiplication: Dynamic Programming

- Matrix “m” top right value is minimum cost for multiplying five matrices and Matrix ”s” is used to put parenthesis or order in which they will be multiplied to get that minimum cost.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

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

Based on the computation, the minimum cost for multiplying the five matrices is 160 and the optimal order for multiplication is

$$((A_1(A_2A_3))(A_4A_5))$$

Matrix chain multiplication: Dynamic Programming

- **<u>How to put parenthesis by “s” matrix :</u>**
- To find order of parenthesis, start from top right value s[1,5]. At s[1,5], we have k=3 where row value is A1 and column value is A5 so this means that divide A1,A2,A3,A4,A5 into (A1A2A3)(A4A5).
- Now it has been divided into two parts. First part is from 1 to 3 and second from 4 to 5. So look for s[1,3] and s[4,5].
- At s[1,3], we have k=1 where row value is A1 and column value is A3 so this means that divide A1,A2,A3 into (A1)(A2A3) so the complete becomes (A1(A2A3))(A4A5)
- At s[4,5], we have k=4 where row value is A4 and column value is A5 so this means that divide A4,A5 into (A4)(A5) which wont make any effect so the complete becomes (A1(A2A3))(A4A5)

- **Matrix “s”**

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

Algorithm to Compute Optimal Cost

Input: Array $p[0\ldots n]$ containing matrix dimensions and n
Result: Minimum-cost table m and split table s

```
MATRIX-CHAIN-ORDER($p[\ ]$, $n$)
    for $i \leftarrow 1$ to $n$
        $m[i, i] \leftarrow 0$

    for $l \leftarrow 2$ to $n$
        for $i \leftarrow 1$ to $n$-$l$+1
            $j \leftarrow i+l-1$
            $m[i, j] \leftarrow \infty$

            for $k \leftarrow i$ to $j$-1
                $q \leftarrow m[i, k] + m[k+1, j] + p[i-1]\, p[k]\, p[j]$
                if $q < m[i, j]$
                    $m[i, j] \leftarrow q$
                    $s[i, j] \leftarrow k$
    return $m$ and $s$
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Matrix chain multiplication: Dynamic Programming

The s matrix stores the values k. The s matrix can be used to extracting the order in which matrices are to be multiplied. Here is the algorithm that caries out the matrix multiplication to compute $A_{i..j}$:

```
MULTIPLY$(i, j)$
  if $(i = j)$
     then return $A[i]$
     else $k \leftarrow s[i, j]$
          $X \leftarrow$ MULTIPLY$(i, k)$
          $Y \leftarrow$ MULTIPLY$(k + 1, j)$
          return $X \cdot Y$
```