COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 10 DYNAMIC PROGRAMMING

ALPTEKİN KÜPÇÜ

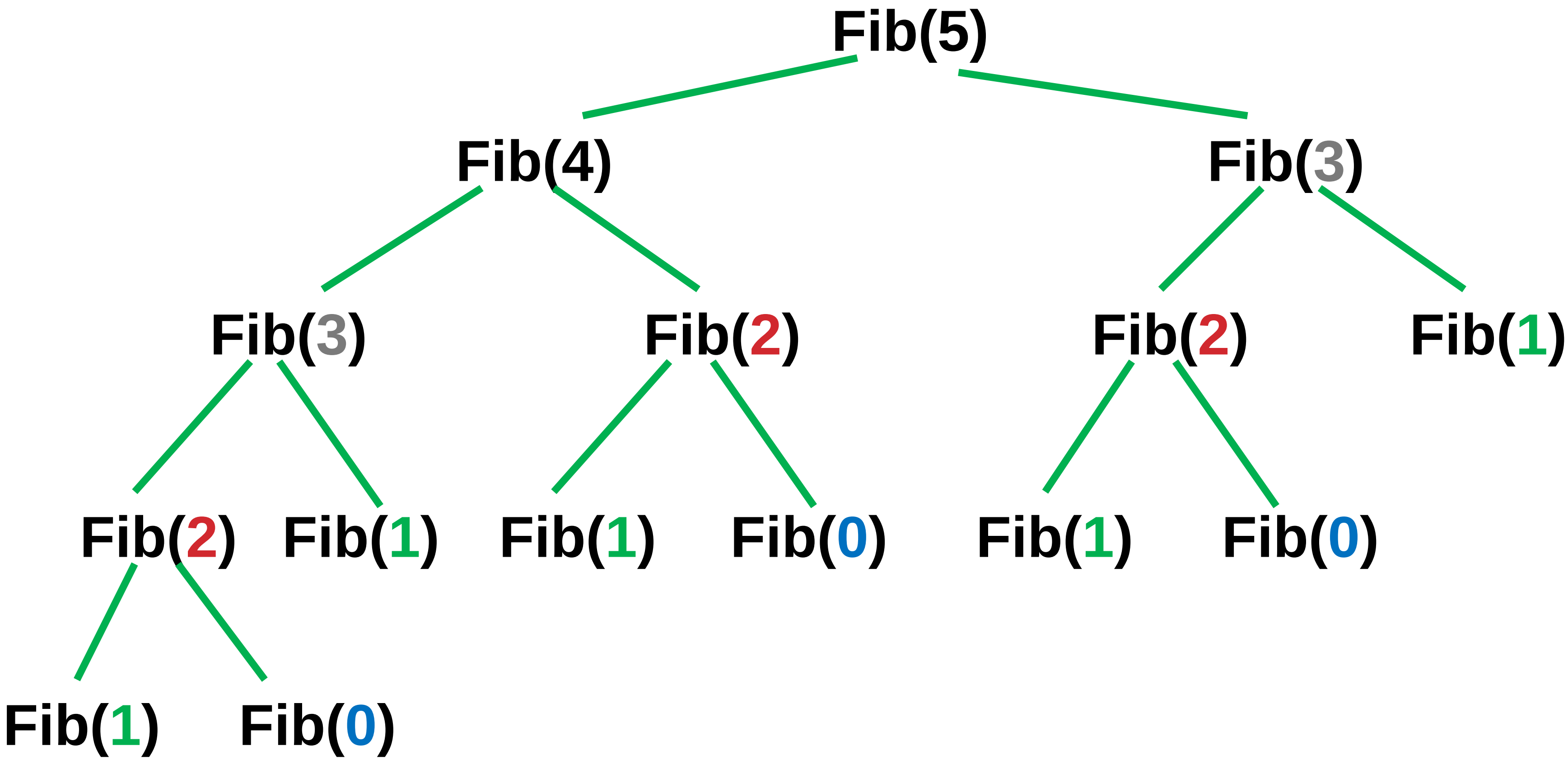
Based on slides of David Luebke, Jennifer Welch, and Cevdet Aykanat

DRAWBACK OF DIVIDE AND CONQUER APPROACH

- **Fibonacci numbers:**
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- **Sequence is 0, 1, 1, 2, 3, 5, 8, 13, …**

- **Obvious recursive algorithm:**
- **Fib(*n*)**
  - if $n = 0$ or $n = 1$ then return n
  - else return (Fib(n-1) + Fib(n-2))

RECURSION TREE FOR FIB(5)



COMPLEXITY OF RECURSIVE FIBONACCI ALGORITHM

- **If all leaves had the same depth, then there would be about 2^n recursive calls.**
- **With careful counting it can be shown that the running time of the recursive Fibonacci algorithm is $\Omega((1.6)^n)$**
  - Exponential!
- **Wasteful approach - repeated work**
  - e.g., Fib(2) is computed three times
- **Instead, compute once, store the result in a table, and access it when needed**

MEMOIZATION

Initialization:

Create a lookup table F

F[0] $\leftarrow$ 0

F[1] $\leftarrow$ 1

F[k] $\leftarrow$ NULL for $n \geq k \geq 2$

Fib(n)

if n = 0 or n = 1 then return F[n]

if F[n-1] = NULL then Fib(n-1)

if F[n-2] = NULL then Fib(n-2)

F[n] $\leftarrow$ F[n-1] + F[n-2]

return F[n]

Computes each F[i] only once

- If already computed, just uses the result
- O(n)

MEMOIZED FIB(5)

	F
0	0
1	1
2	NIL 1
3	NIL 2
4	NIL 3
5	NIL 5

Fib(5) — Sets F[5] = 2+3 = 5
returns 5

Fib(4) — Sets F[4] = 1+2 = 3
returns 3

Fib(3) — Sets F[3] = 1+1 = 2
returns 2

Fib(2) — Sets F[2] = 0+1 = 1
returns 1

GET RID OF THE RECURSION

- **Memoization is very useful**
  - Linear-time algorithm instead of exponential!!
- **Recursion adds overhead**
  - extra time for function calls
  - extra space to store information on the runtime stack about each currently active function call
- **Avoid the recursion overhead by filling in the table entries bottom-up, instead of top-down.**
- **Find which sub-problems rely on which other sub-problems**
  - This leads to an order for computing the sub-problems that respects the dependencies
  - When solving a sub-problem, you must have already solved all the sub-problems on which the current one depends

BOTTOM-UP APPROACH: DYNAMIC PROGRAMMING

- **Dependency-Respecting Order: Find Fibonacci numbers sequentially**
  - $F_0, F_1, F_2, F_3, \ldots$
- **Fib(n)**
  - F[0] ← 0
  - F[1] ← 1
  - for i = 2 to *n* do
    - F[i] ← F[i-1] + F[i-2]
  - return F[*n*]
- **Optimization: save space by only keeping last two numbers computed**

F0 F1 F2 F3 … Fn-2 Fn-1 Fn

MATRIX CHAIN MULTIPLICATION PROBLEM

- **Multiplying non-square matrices:**
  - A is p x r, B is r x s
  - AB is p x s whose $(i,j)^{th}$ entry is $\sum a_{ik}\, b_{kj}$
- **Computing *AB* takes *prs* scalar multiplications and *p(r-1)s* scalar additions (using basic algorithm).**
  - Thus $O(prs)$
- **We have a sequence of matrices to multiply. *What is the best order of multiplication?***
  - Remember, matrix multiplication is associative.

MATRIX CHAIN MULTIPLICATION PROBLEM

- **Input: A sequence (chain) < A_1, A_2, …. A_n > of n matrices**
  - A_i is of size p_{i-1} x p_i
- **Goal: Compute the product $A_1 \cdot A_2 \ldots\ldots \cdot A_n$ optimally**
- **A product of matrices is fully parenthesized if it is**
  - a single matrix
  - Or, the product of two fully parenthesized matrix products surrounded by a pair of parentheses.
- **All parenthesizations yield the same product**
  - $(A_i (A_{i+1} A_{i+2} \ldots\ldots. A_j))$
  - $((A_i A_{i+1} A_{i+2} \ldots\ldots. A_{j-1}) A_j)$
  - $((A_i A_{i+1} A_{i+2} \ldots\ldots. A_k)(A_{k+1} A_{k+2} \ldots\ldots. A_j))$ for $i \leq k \leq j-1$

ORDER MATTERS

- **Example: Compute < A_1 , A_2 , A_3 > where**
  - A_1 : 10×100
  - A_2 : 100×5
  - A_3 : 5×50
- **Best parenthesization?**
- **Two options**
  - ((A_1 A_2) A_3) : 10x100x5 + 10x5x50 = 7500
    - Compute A_{12} = (A_1 A_2) and then A_{12} A_3
  - (A_1 (A_2 A_3)) : 100x5x50 + 10x100x50 = 75000
    - Compute A_{23} = (A_2 A_3) and then A_1 A_{23}
- **First parenthesization yields 10 times faster computation**

BRUTE FORCE SOLUTION

- **Brute force approach:** **exhaustively check all parenthesizations**
- **P(n):** **# of parenthesizations of a sequence of n matrices**
- **We can split sequence between k^{th} and $(k+1)^{st}$ matrices for any $k \in \{1, 2, \ldots, n-1\}$, then parenthesize the two resulting sequences independently**

$$(A_1 A_2 A_3 \ldots A_k)(A_{k+1} A_{k+2} \ldots\ldots A_n)$$

- **We obtain the recurrence**
- **$P(1) = 1$ and $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$**

- **EXPONENTIAL!!**

DYNAMIC PROGRAMMING DESIGN

1. **Characterize the structure of an optimal solution.**
2. **Recursively define the value of an optimal solution based on optimal solution to sub-problems.**
3. **Compute the value of an optimal solution in a bottom-up fashion, respecting dependencies.**
4. **Construct an optimal solution from the information computed in step 3 by remembering the optimal choices you have made along the path.**

STEP 1: CHARACTERIZE

- **Given an optimal parenthesization**
  - $(A_1 A_2 A_3 \ldots. A_k) \cdot (A_{k+1} A_{k+2} A_{k+3} \ldots. A_n)$
- **Parenthesization of the two sub-chains**
  - $A_1 A_2 A_3 \ldots. A_k$
  - $A_{k+1} A_{k+2} A_{k+3} \ldots. A_n$
- **should both be optimal**
- **Thus, a globally-optimal solution contains optimal solutions to sub-problems**
- **Optimal substructure property exists.**

STEP 2: RECURSIVELY DEFINE

- **Step 2: Define the value of an optimal solution recursively in terms of optimal solutions to the sub-problems**
- **Subproblem: The problem of determining the minimum cost of multiplying sub-chain $A_{i..j}$**
  - i.e., finding optimal parenthesization of $A_i A_{i+1} A_{i+2} \ldots A_j$
- $\boldsymbol{m_{ij}}$**: min # of scalar multiplications and additions needed to multiply sub-chain $A_{i..j}$**
  - The value of a (global) optimal solution is $\boldsymbol{m_{1n}}$
  - $\boldsymbol{m_{ii}} = 0$, since subchain $A_{i..i}$ contains just one matrix A_i
    - no multiplication at all
  - $\boldsymbol{m_{ij}} =$ ***? recursively ?***

STEP 2: RECURSIVELY DEFINE

- **For $i < j$, optimal parenthesization splits subchain $A_{i..j}$ as $A_{i..k}$ and $A_{k+1...j}$ where $i \leq k < j$**
  - Optimal cost of computing $A_{i..k}$: m_{ik}
  - Optimal cost of computing $A_{k+1..j}$: $m_{k+1,j}$
  - Cost of multiplying $A_{i..k}A_{k+1..j}$: $p_{i-1} \times p_k \times p_j$
    ($A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is a $p_k \times p_j$ matrix)
- $\Rightarrow$ $m_{ij} = ?$

STEP 2: RECURSIVELY DEFINE

- **For $i < j$, optimal parenthesization splits subchain $\boldsymbol{A_{i..j}}$ as $\boldsymbol{A_{i..k}}$ and $\boldsymbol{A_{k+1...j}}$ where $i \leq k < j$**
  - Optimal cost of computing $\boldsymbol{A_{i..k}}$: m_{ik}
  - Optimal cost of computing $\boldsymbol{A_{k+1..j}}$: $m_{k+1,j}$
  - Cost of multiplying $\boldsymbol{A_{i..k}A_{k+1..j}}$: $p_{i-1} \times p_k \times p_j$
    ($\boldsymbol{A_{i..k}}$ is a $p_{i-1} \times p_k$ matrix and $\boldsymbol{A_{k+1..j}}$ is a $p_k \times p_j$ matrix)
- $\Rightarrow \boldsymbol{m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j}$

STEP 2: RECURSIVELY DEFINE

- **For $i < j$, optimal parenthesization splits subchain $A_{i..j}$ as $A_{i..k}$ and $A_{k+1...j}$ where $i \leq k < j$**
  - Optimal cost of computing $A_{i..k}$: m_{ik}
  - Optimal cost of computing $A_{k+1..j}$: $m_{k+1,j}$
  - Cost of multiplying $A_{i..k}A_{k+1..j}$: $p_{i-1} \times p_k \times p_j$
    ($A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is a $p_k \times p_j$ matrix)
- $\Rightarrow \boldsymbol{m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j}$
- **The equation assumes we know the value of k, but we do not know it.** ***How can we find it?***

STEP 2: RECURSIVELY DEFINE

- $m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j$
- We do not know k, but there are $j - i$ possible values for k
  - $k = i, i+1, i+2, \ldots, j-1$
- Since optimal parenthesization must be one of these k values we need to check them all to find the best

$$m_{ij} = \begin{cases} 0 \text{ if } i = j \\ \underset{i \leq k \leq j-1}{\text{MIN}} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \} \text{ if } i < j \end{cases}$$

OVERLAPPING SUB-PROBLEMS

- **An important observation:**
  - One sub-problem for each choice of i and j satisfying $1 \leq i \leq j \leq n$
  - Total $n + (n-1) + \cdots + 2 + 1 = \frac{1}{2}n(n+1) = \Theta(n^2)$
- **We have relatively few subproblems (definitely not exponential)**
- **We can write a recursive algorithm based on this recurrence.**
- **However, a recursive algorithm may encounter each subproblem many times in different branches of the recursion tree**
- **When there are overlapping sub-problems, a dynamic programming approach performs much better.**

STEP 3: FIND DEPENDENCIES AMONG SUB-PROBLEMS

M:

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

GOAL!

computing the gray square requires the yellow ones: to the left and below.

Computing $M(i,j)$ requires everything in same row to the left:

$M(i,i)$, $M(i,i+1)$, …, $M(i,j-1)$

and everything in same column below:

$M(i,j)$, $M(i+1,j)$,…,$M(j,j)$

STEP 3: IDENTIFY ORDER FOR SOLVING SUB-PROBLEMS

M:

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

GOAL!

Other possible orderings?

STEP 3: COMPUTE THE VALUE

```
for i := 1 to n do                      // initialization
        M[i,i] := 0
for d := 1 to n-1 do                    // go diagonally
        for i := 1 to n-d do            // rows with an entry on d^th diagonal
            j := i + d                  // column of row i on d^th diagonal
            M[i,j] := infinity
            for k := i to j-1 do
                M[i,j] := min(M[i,j], M[i,k]+M[k+1,j]+p_{i-1}p_k p_j)
```

Running time O(n^3)

EXAMPLE

M:

	A	B	C	D
A	0			
B	n/a	0		
C	n/a	n/a	0	
D	n/a	n/a	n/a	0

A: 30x1
B: 1x40
C: 40x10
D: 10x25

Solve the optimal cost of computing

A.B.C.D

EXAMPLE

M:

	A	B	C	D
A	0	1200	700	1400
B	n/a	0	400	650
C	n/a	n/a	0	10000
D	n/a	n/a	n/a	0

A: 30x1
B: 1x40
C: 40x10
D: 10x25

STEP 4: CONSTRUCT

- It's fine to know the cost of the cheapest order, but what *is* that cheapest order?
- When sub-problems are optimized, keep track of the optimal solution (e.g., optimal parenthesis point *k*).
- At the end, call a recursive algorithm to print out the actual order.

STEP 4: CONSTRUCT

```
for i := 1 to n do                      // initialization
        M[i,i] := 0
for d := 1 to n-1 do                    // go diagonally
        for i := 1 to n-d do            // rows with an entry on d^th diagonal
            j := i + d                  // column of row i on d^th diagonal
            M[i,j] := infinity
            for k := i to j-1 do
                x := min(M[i,j], M[i,k]+M[k+1,j]+p_{i-1}p_k p_j)
                if x < M[i,j] then
                        M[i,j] := x
                        S[i,j] := k     // remember optimal sub-solution
```

Running time still O(n^3)

EXAMPLE

M:
S:

	A	B	C	D
A	0	1200_{1}	700_{1}	1400_{1}
B	n/a	0	400_{2}	650_{3}
C	n/a	n/a	0	10000_{3}
D	n/a	n/a	n/a	0

A: 30x1
B: 1x40
C: 40x10
D: 10x25

Initial call Print(S,1,n)

Print(S,i,j)

```
if i = j then
        output "A" + i   // + is string concatenation
else
        k := S[i,j]
        output "(" + Print(S,i,k) + Print(S,k+1,j) + ")"
```

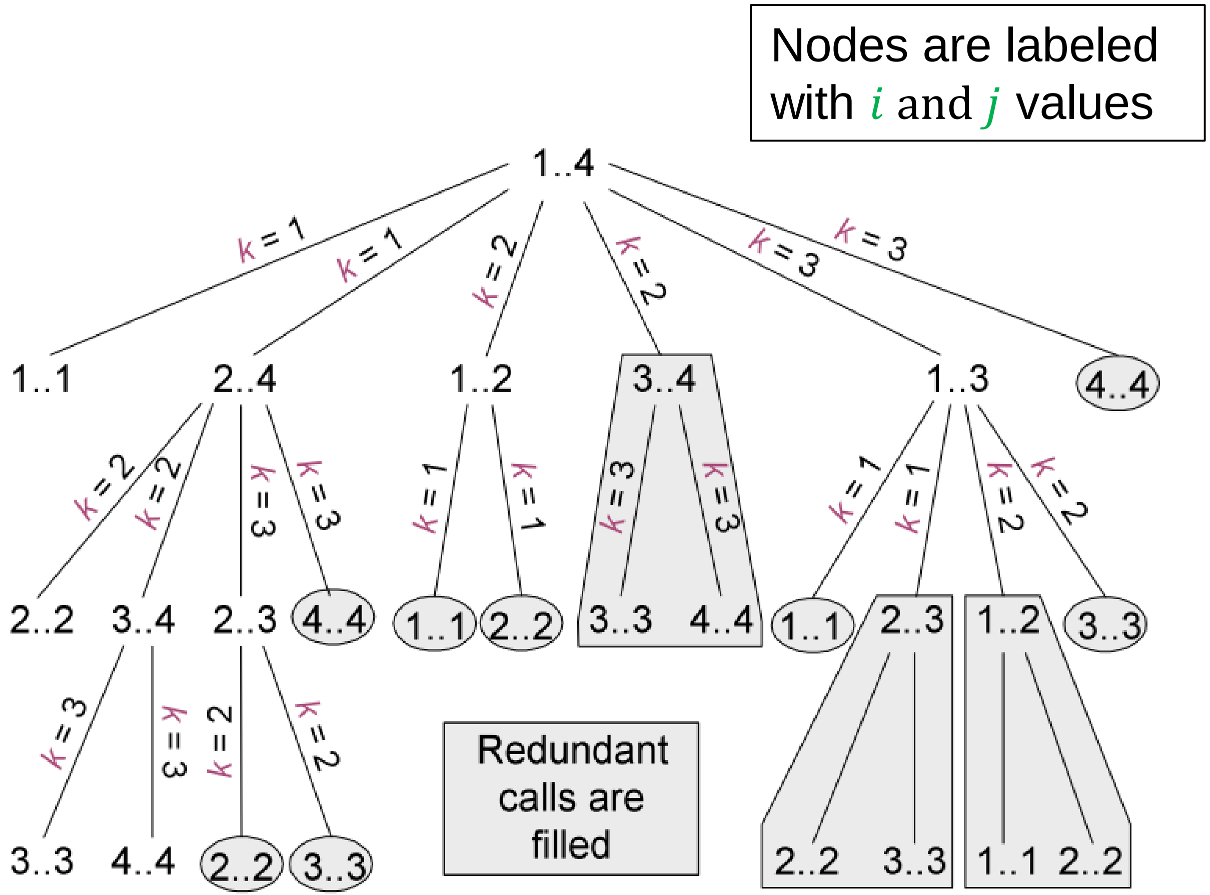
Output:
$(A_1(A_2A_3)A_4)$

Equivalently:
(A(BC)D)

DYNAMIC PROGRAMMING

- **Two key ingredients**
- **Optimal Sub-structure**
  - A problem exhibits optimal sub-structure if an optimal solution to a problem contains within it optimal solutions to sub-problems
    - Example: matrix-chain-multiplication
      - Optimal parenthesization of $A_1 A_2 \dots A_n$ that splits the product between A_k and A_{k+1}, contains within it optimal solutions to the problems of parenthesizing $A_1 A_2 \dots A_k$ and $A_{k+1} A_{k+2} \dots A_n$
- **Overlapping Sub-problems**
  - Total number of distinct sub-problems must be polynomial in the input size.
  - When a recursive algorithm revisits the same problem over and over again we say that the optimization problem has overlapping sub-problems.

RECURSIVE MATRIX CHAIN MULTIPLICATION



DYNAMIC PROGRAMMING

- **Dynamic Programming algorithms typically take advantage of overlapping sub-problems**
  - by solving each problem once
  - then storing the solutions in a table where it can be looked up when needed
  - using constant time per lookup
- **These two properties already exist in Greedy Algorithms. But they require a third property, the greedy choice property, which Dynamic Programming does not require.**

MEMOIZATION VS. DYNAMIC PROGRAMMING

- **Memoization offers the asymptotic efficiency of Dynamic Programming while maintaining top-down strategy**
- **Idea is to memoize the natural, but inefficient, recursive algorithm**
  - Maintains an entry in a table for the solution to each sub-problem
  - Each table entry contains a special value to indicate that the entry has yet to be filled in (e.g., NULL)
  - When the sub-problem is encountered for the first time, its solution is computed and then stored in the table
  - Each subsequent time that the sub-problem is encountered, the value stored in the table is simply looked up and returned
- **Complicated lookups are possible using hashing with sub-problem parameters as key**

MEMOIZATION VS. DYNAMIC PROGRAMMING

- **Matrix-chain multiplication can be solved in $O(n^3)$ time**
  - by either a top-down memoized recursive algorithm
  - or a bottom-up dynamic programming algorithm
- **Both methods exploit the overlapping sub-problems property**
  - There are only $\Theta(n^2)$ different sub-problems in total
  - Both methods compute the solution to each problem once
- **Without memoization, the regular recursive algorithm runs in exponential time since sub-problems are solved repeatedly**
- **Overall, Dynamic Programming may be harder to program compared to memoization**
- **But avoids recursion overheads, and therefore is faster in practice.**

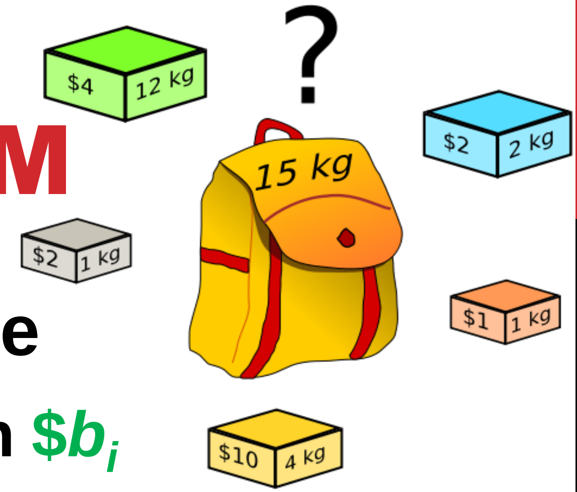
OPTIMAL SUB-STRUCTURE

- **Showing optimal sub-structure property usually follows the following proof-by-contradiction outline:**
  - We show that the solutions to the sub-problems must be optimal for the global solution to be optimal:
  - Suppose that one of the sub-problem solutions is not optimal.
  - Cut it out.
  - Paste in an optimal solution.
  - Get a better solution to the original problem.
  - This contradicts optimality of global solution.

DYNAMIC PROGRAMMING VS. GREEDY ALGORITHMS

- **Dynamic Programming uses optimal sub-structure bottom-up.**
  - First find optimal solutions to sub-problems.
  - Then choose which to use in optimal solution to the problem.
- **Greedy Algorithms work top-down**
  - First make a choice that looks best
  - Then solve the resulting sub-problem.
- **Knapsack:**
  - Fractional Knapsack solvable by a Greedy Algorithm
  - 0-1 Knapsack requires Dynamic Programming
  - Difference: Greedy choice does not guarantee optimal solution in 0-1 Knapsack

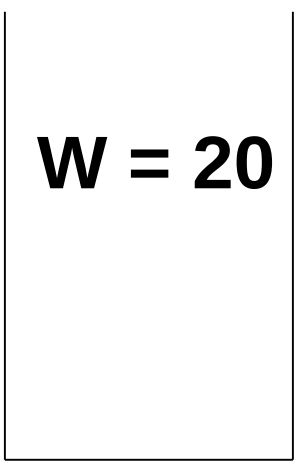
0-1 KNAPSACK PROBLEM



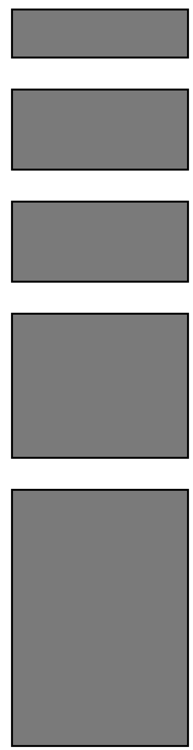
- **There are *n* different items in a store**
- **Item *i* weighs w_i kilograms and is worth $\$b_i$**
- **We can carry up to *W* kilograms in a knapsack**
  - w_i, b_i, W all integers
- **An item must be taken as a whole or left behind.**
- **Problem: What should we take to maximize the total value ?**
- **We have already showed that the 0-1 Knapsack problem exhibits optimal sub-structure property and has many overlapping sub-problems.**

0-1 KNAPSACK PROBLEM: EXAMPLE

Knapsack
Max weight: W = 20



Items



Weight w_i	**Benefit** b_i
2	3
3	4
4	5
5	8
9	10

STEP 2: RECURSIVELY DEFINE

- **Goal:** $$\max \sum_{i \in T} x_i b_i \text{ subject to} \sum_{i \in T} w_i \leq W \ and \ x_i \in \{0,1\}$$
- **The problem is called a *0-1* Knapsack problem, because each item must be entirely accepted or rejected.**
  - x_i values are either 0 or 1
- **Let items be labeled *1..n*, and define S_k = *{items labeled 1, 2, .. k}***
- **Then the problem is picking best haul from S_n, and a sub-problem would be to find an optimal solution for S_k**
  - A valid sub-problem definition
  - Can we describe the final solution S_n in terms of sub-problems S_k ?
  - Unfortunately, we cannot…

STEP 2: RECURSIVELY DEFINE

$w_1 = 2$ $b_1 = 3$	$w_3 = 4$ $b_3 = 5$	$w_4 = 5$ $b_4 = 8$	$w_2 = 3$ $b_2 = 4$	

Optimal S_4: {1,2,3,4}
Total weight: 14
Total benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_3 = 4$ $b_3 = 5$	$w_4 = 5$ $b_4 = 8$	$w_5 = 9$ $b_5 = 10$

Optimal S_5: {1,3,4,5}
Total weight: 20
Total benefit: 26

S_5 S_4

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Max weight: W = 20

Solution for S_4 is not part of the solution for S_5

STEP 2: RECURSIVELY DEFINE

- **Attempt #2:**
  - Add a weight parameter w to sub-problem definition.
  - The sub-problem now is to compute the total benefit of choosing from the first k items with weight limit w: $B[k,w]$
- **The best subset of S_k that has the total weight w either contains item k or not. It is one of the two:**
  - 1) The best subset of S_{k-1} that has total weight w, or
  - 2) The best subset of S_{k-1} that has total weight $w-w_k$ plus the item k

STEP 2: RECURSIVELY DEFINE

- **First case: $w_k > w$. Item k cannot be part of the solution, since if it was, the total weight would be more than w, which is unacceptable.**
  - Thus, we need to fill the same weight using first $k-1$ items.
- **Second case: $w_k \leq w$. Then the item k may or may not be in the optimal solution, and we choose the case with greater benefit value.**

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

0-1 KNAPSACK ALGORITHM

```
for w = 0 to W do
        B[0,w] = 0
for i = 0 to n do
        B[i,0] = 0
        for w = 0 to W do
                if w_i <= w then  // item i can be part of the solution
                        B[i,w] = max( b_i + B[i-1,w- w_i] , B[i-1,w] )
                else              // w_i > w
                        B[i,w] = B[i-1,w]
```

Running Time: $O(nW)$

vs.

Brute Force: EXPONENTIAL

0-1 KNAPSACK PROBLEM: EXAMPLE

Items	Weight w_i	Benefit b_i
	2	3
	3	4
	4	5
	5	6

Knapsack
Max weight: W = 5

W = 5

EXAMPLE

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

for w = 0 to W **do**

B[0,w] = 0

EXAMPLE

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

for i = 0 to n **do**

  B[i,0] = 0

EXAMPLE

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i <= w$ **then** // item i can be part of the solution

B[i,w] = max(b_i + B[i-1,w- w_i] , B[i-1,w])

else // $w_i > w$

B[i,w] = B[i-1,w]

EXAMPLE

Items:

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

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

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

if $w_i <= w$ **then** // item i can be part of the solution

  $B[i,w] = \max(b_i + B[i-1,w-w_i] , B[i-1,w])$

else // $w_i > w$

  $B[i,w] = B[i-1,w]$

EXAMPLE

Items:

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

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

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if w_i <= w then // item i can be part of the solution

B[i,w] = max(b_i + B[i-1,w- w_i] , B[i-1,w])

else // w_i > w

B[i,w] = B[i-1,w]

EXAMPLE

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i \le w$ **then** // item i can be part of the solution

$B[i,w] = \max(b_i + B[i-1,w-w_i], B[i-1,w])$

else // $w_i > w$

$B[i,w] = B[i-1,w]$

EXAMPLE

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i <= w$ **then** // item i can be part of the solution

$B[i,w] = \max(b_i + B[i-1,w- w_i] , B[i-1,w])$

else // $w_i > w$

$B[i,w] = B[i-1,w]$

EXAMPLE

Items:

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

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

i=2

$b_i=4$

$w_i=3$

w=1

$w-w_i=-2$

if w_i <= w **then** // item i can be part of the solution
    B[i,w] = max(b_i + B[i-1,w- w_i] , B[i-1,w])
else // $w_i > w$
    B[i,w] = B[i-1,w]

EXAMPLE

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

if w_i <= w **then** // item i can be part of the solution

    $B[i,w] = \max(b_i + B[i-1, w-w_i], B[i-1,w])$

else // $w_i > w$

    $B[i,w] = B[i-1,w]$

EXAMPLE

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if w_i <= w then // item i can be part of the solution
    B[i,w] = max(b_i + B[i-1,w- w_i] , B[i-1,w])
else // w_i > w
    B[i,w] = B[i-1,w]

EXAMPLE

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

if w_i <= w then // item i can be part of the solution
    B[i,w] = max(b_i + B[i-1,w- w_i] , B[i-1,w])
else // w_i > w
    B[i,w] = B[i-1,w]

EXAMPLE

Items:

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

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

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

if $w_i <= w$ then // item i can be part of the solution
    $B[i,w] = \max(b_i + B[i-1,w-w_i] , B[i-1,w])$
else // $w_i > w$
    $B[i,w] = B[i-1,w]$

EXAMPLE

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

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ then // item i can be part of the solution

$B[i,w] = \max(b_i + B[i-1,w- w_i] , B[i-1,w])$

else // $w_i > w$

$B[i,w] = B[i-1,w]$

EXAMPLE

Items:

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

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

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

if $w_i <= w$ **then** // item i can be part of the solution

$B[i,w] = \max(b_i + B[i-1,w- w_i] , B[i-1,w])$

else // $w_i > w$

$B[i,w] = B[i-1,w]$

EXAMPLE

Items:

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

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

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

if $w_i <= w$ then // item i can be part of the solution
  $B[i,w] = \max(b_i + B[i-1,w- w_i] , B[i-1,w])$
else // $w_i > w$
  $B[i,w] = B[i-1,w]$

EXAMPLE

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

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

Items:

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

if w_i <= w then   // item i can be part of the solution
    $B[i,w] = \max(b_i + B[i-1,w-w_i] , B[i-1,w])$
else   // $w_i > w$
    $B[i,w] = B[i-1,w]$

EXAMPLE

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

$i=3$

$b_i=5$

$w_i=4$

$w=5$

Items:

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

if w_i <= w **then** // item i can be part of the solution
    B[i,w] = max(b_i + B[i-1,w- w_i] , B[i-1,w])
else // w_i > w
    B[i,w] = B[i-1,w]

STEP 4: CONSTRUCT

- **This algorithm only finds the optimal value that can be carried in the knapsack.**
- **To know the items that make this maximum value, an addition to this algorithm is necessary.**
- **Just as what we did for Matrix-Chain Multiplication, we need to remember at each cell what was the sub-problem leading to the best solution (which other cell lead to the best solution).**
- ***Can Dynamic Programming solve all optimization problems?***
  - **NO!!**
    - Check Shortest Path vs. Longest Path problems in graphs