Dynamic Programming

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS. "Programming" here means "planning"

D*ynamic Programming* is a powerful algorithm design technique for solving problems that

- Appear to be exponential but have a poly solution with DP
- In many cases are optimization problems (min/max)
- Defined by or formulated as recurrences with overlapping subproblems
- Optimal solution to a problem contains optimal solutions to subproblems.

Dynamic Programming

- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem.
- Key: Determine structure of optimal solutions

5 Steps to DP

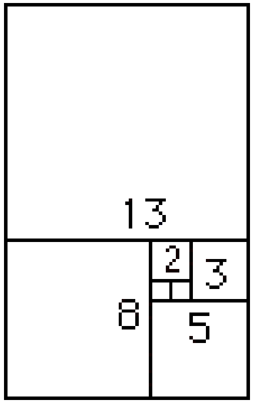
1. Define subproblems
2. Guess part of the solution
3. Relate subproblem solutions
4. Recurse + memoize or Build a DP bottom-up table.
5. Solve original problem

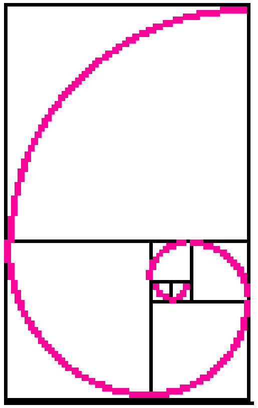
DP Examples

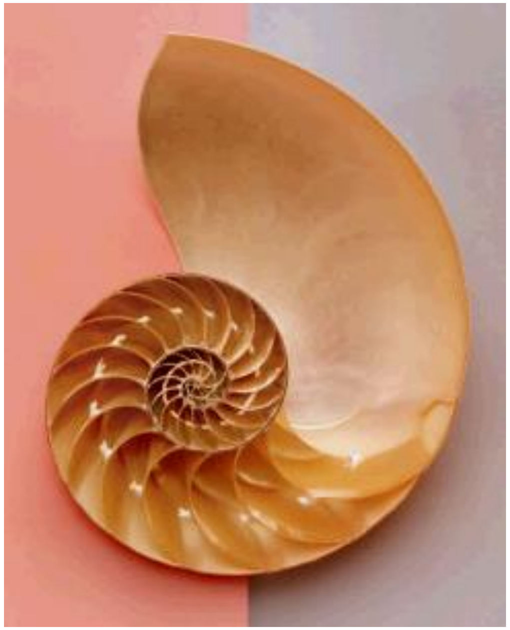
- Fibonacci
- Binomial Coefficients
- Longest Common Subsequence
- Longest Increasing Subsequence
- Knapsack
- Shortest Path
- Chain Matrix Multiplication
- Edit Distance
- Rod Cutting
- Optimal BST

Fibonacci Sequence

- 0,1,1,2,3,5,8,13,21,34,…

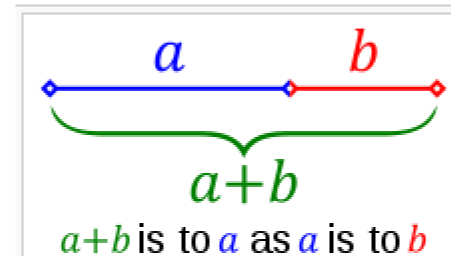






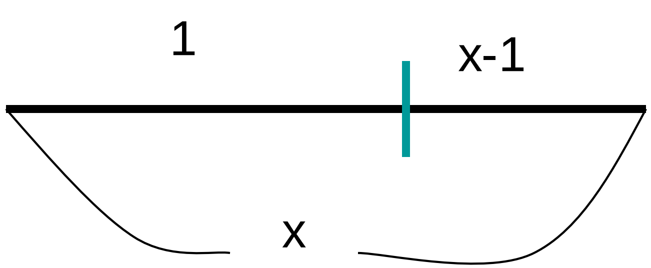
Fibonacci Number and Golden Ratio

0,1,1,2,3,5,8,13,21,34,...



$$\begin{cases} f_n = 0 & \text{if } n = 0 \\ f_n = 1 & \text{if } n = 1 \\ f_n = f_{n-1} + f_{n-2} & \text{if } n \geq 2 \end{cases}$$

$$\lim_{n\to\infty} \frac{f_n}{f_{n-1}} = \frac{1+\sqrt{5}}{2} = \text{Golden Ratio} = \phi = 1.61803..$$



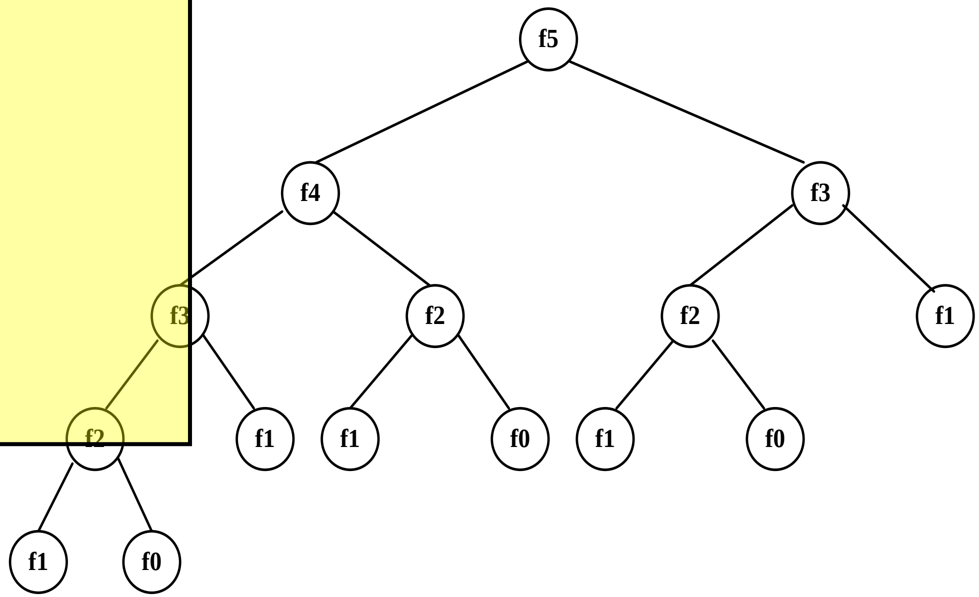
$$\frac{x}{1} = \frac{1}{x-1}$$

$$x^2 - x - 1 = 0$$

$$x = \frac{1+\sqrt{5}}{2}$$

Naive Recursive Algorithm

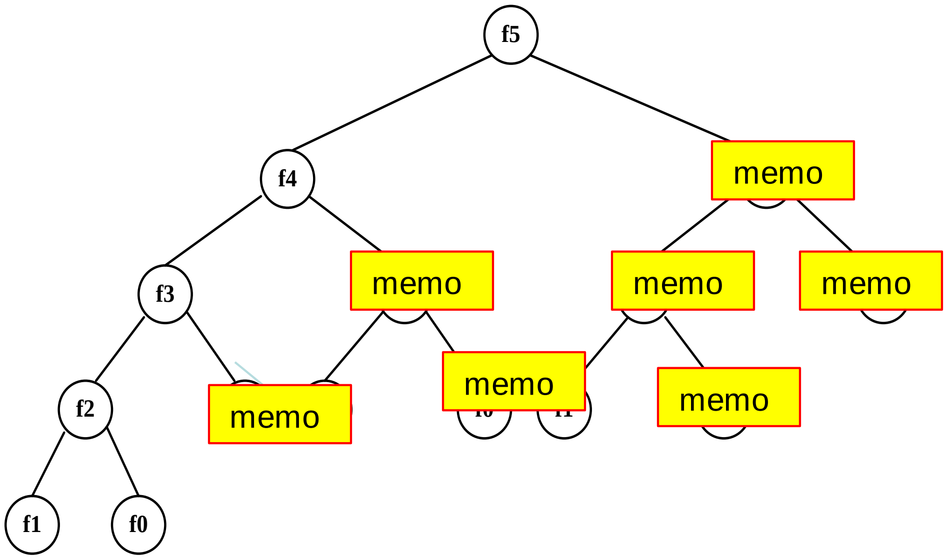
```
fib (n) {
  if (n = 0) {
     return 0;
  } else if (n = 1) {
     return 1;
  } else {
     return fib(n-1) + fib(n-2);
  }
}
```



- Solved by a recursive program
- Much replicated computation is done.
- Running time $\Theta(\phi^n)$ - exponential

Memoized DP Algorithm

```
memo = { }
fib (n) {
   if (n in memo) { return memo[n] }
   if (n <= 1) {
      f =  n;
    } else {
      f =  fib(n-1) + fib(n-2);
   }
   memo[n] = f;
   return f
}
```



- fib(k) only recurses the first time called only n nonmemoized calles
- Memorized calls “free” $\Theta(1)$.
- Time = #subproblems * time/subproblem

  $= n * \Theta(1)$
- Running time $\Theta(n)$ - linear

Bottom-up DP Algorithm

```
fib = { }
fib[0] = 0;
fib[1] = 1;
for k = 2 to n
        fib[k] =  fib[k-1] + fib[k-2];
return fib[n]
```

- Same as memoized DP with recursion “unrolled” into iteration.
- Practically faster since no recursion
- Analysis is more obvious
- Running time $\Theta(n)$ - linear

A Basic Idea of Dynamic Programming

- DP = recursion + memoization
  - Memoize = remember and reuse solutions to subproblems
- Botton-Up Method stores all values in a table

Binomial Coefficient/Combinations C(n, k)

- The number of ways can you select k lottery balls out of n
- The number of way to select a group of 4 from 6 students
- the number of acyclic paths connecting 2 corners of an $k \times (n-k)$ grid
- the coefficient of the $a^k b^{n-k}$ term in the polynomial expansion of $(a + b)^n$

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Binomial Coefficient/Combinations C(n, k)

– The number of way to select a group of 4 from 6 students

$$C(n,k) \equiv \binom{n}{k} \equiv \frac{n!}{k!(n-k)!}$$

$$C(6,4) \equiv \binom{6}{4} \equiv \frac{6!}{4!(6-4)!} = 15$$

C(6,4) number of different groups of 4 students selected from 6

{a, b, c, d}	{a, b, e, f}	{c, d, e, f}
{a, b, c, e}	{a, c, d, e}	{b, d, e, f}
{a, b, c, f}	{a, c, d, f}	{b, c, e, f}
{a, b, d, e}	{a, c, e, f}	{b, c, d, e}
{a, b, f, d}	{a, d, e, f}	{b, c, d, f}

Binomial Coefficient/Combinations C(n, k)

– The number of way to select a group of 4 from 6 students

$$C(n,k) \equiv \binom{n}{k} \equiv \frac{n!}{k!(n-k)!}$$

$$C(6,4) \equiv \binom{6}{4} \equiv \frac{6!}{4!(6-4)!} = 15$$

$$C(6,4) = \mathbf{C(5,3)} + \mathbf{C(5,4)}$$

Six students = {a, b, c, d, e, f}

Groups of 4:

{a, b, c, d}	{a, b, e, f}	{c, d, e, f}
{a, b, c, e}	{a, c, d, e}	{b, d, e, f}
{a, b, c, f}	{a, c, d, f}	{b, c, e, f}
{a, b, d, e}	{a, c, e, f}	{b, c, d, e}
{a, b, f, d}	{a, d, e, f}	{b, c, d, f}

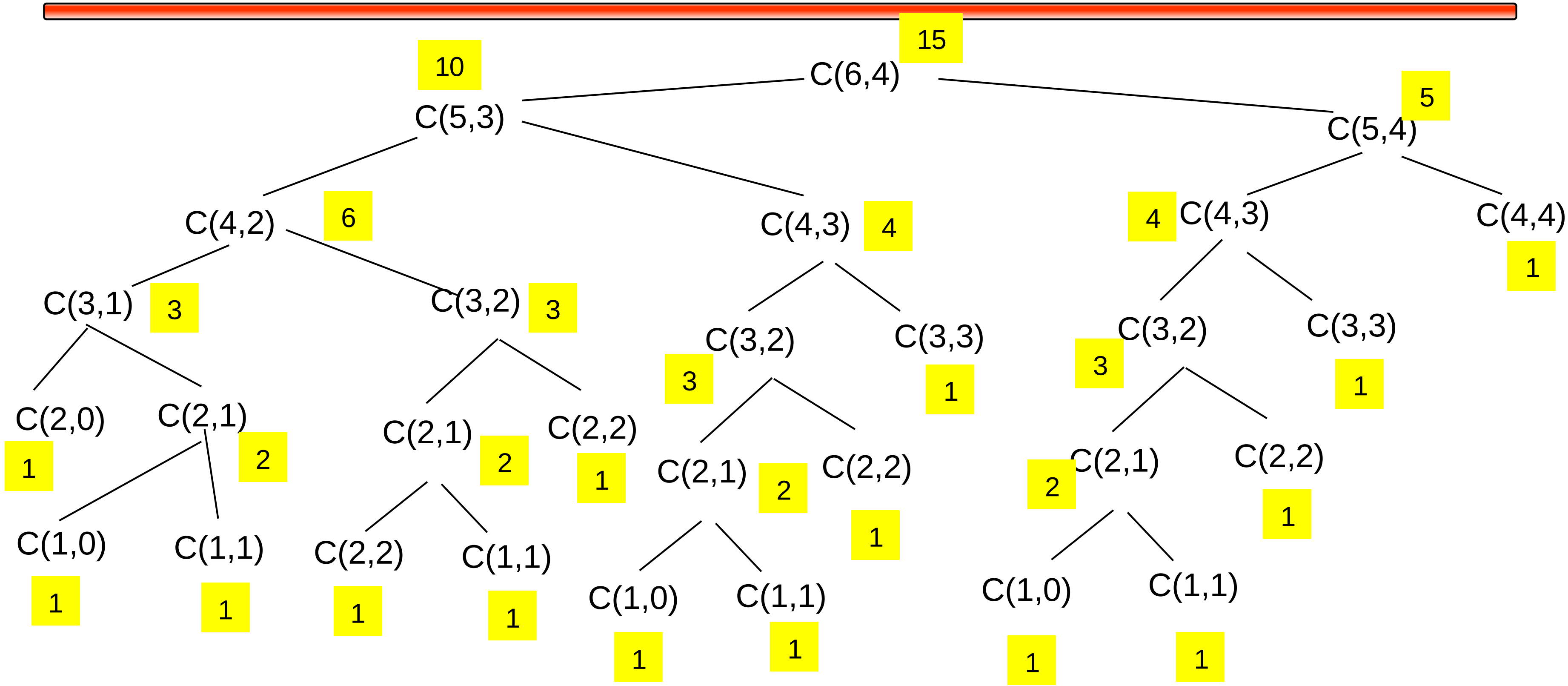
Recursive Relationship

A computationally easier approach makes use of the following recursive relationship

$$\binom{n}{k} \equiv \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

Binomial Coefficient Tree



$T(n, k) = T(n-1,k-1) + T(n-1, k) + 1$

$T(j,0) = 1, \quad T(j,j) = 1$

Example: Combinations

The number of ways to select 6 lottery balls from 49

6 number lottery with 49 balls → 49!/(6!43!) = 13,983,816

49!=608,281,864,034,267,560,872,252,163,321,295,376,887,552,831,379,210,240,000,000,000

Could try to get fancy by canceling terms from numerator & denominator

- can still can end up with individual terms that exceed integer limits

$$\binom{49}{6}$$

$$\binom{48}{5} \quad + \quad \binom{48}{6}$$

$$\binom{47}{4} + \binom{47}{5} \qquad \binom{47}{5} + \binom{47}{6}$$

$$\binom{n}{k} \equiv \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

To select 6 lottery balls out of 49, partition into:

selections that include 1
(must select 5 out of remaining 48)
+
selections that don't include 1
(must select 6 out of remaining 48)

Recursive Combination

could use straight divide & conquer to compute based on this relation

```
/** using divide-and-conquer
 * Calculates n choose k
 *   n the total number to choose from (n > 0)
 *   k the number to choose (0 <= k <= n)
 * */
int CombinationI(int n, int k) {
    if (k == 0 || n == 0 || n == k) {
        return 1;
    }
    else {
        return Combination(n-1, k-1) + CombinationI(n-1, k);
    }
}
```

however, this will take a <u>long</u> time or exceed memory due to redundant work

Recurrence

$T(n, k) = T(n-1,k-1) + T(n-1, k) + 1$

$T(j,0) = 1, \quad T(j,j) = 1$

$T(n, k) = T(n-1,k-1) + T(n-1, k) + 1 < 2\ T(n-1) + 1$

. . . .

$O(2^n)$

Computing a Combination by DP

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

$C(j,0) = 1, \quad C(j,j) = 1$ for $j \geq 0$

	K = 0	1	2	3	4	5	6
N = 0	1						
1	1	1					
2	1		1				
3	1			1			
4	1				1		
5	1					1	
6	1						1

Computing by DP

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

	K = 0	1	2	3	4	5	6
N = 0	1						
1	1	1					
2	1	1+1=2	1				
3	1			1			
4	1				1		
5	1					1	
6	1						1

Computing by DP

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

	K = 0	1	2	3	4	5	6
N = 0	1						
1	1	1					
2	1	2	1				
3	1	3		1			
4	1				1		
5	1					1	
6	1						1

Computing by DP

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

	K = 0	1	2	3	4	5	6
N = 0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1				1		
5	1					1	
6	1						1

Computing by DP

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

	K = 0	1	2	3	4	5	6
N = 0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4			1		
5	1					1	
6	1						1

Computing by DP

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

	K = 0	1	2	3	4	5	6
N = 0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

DP Algorithm for Combinations

```
CombDPI(n,k)
// Computes C(n,k) by DP
// Input: A pair of nonnegative integers n≥k≥0
// Output: the value of C(n,k)
for i ← 0 to n do
     for j ← 0 to min(i, k) do
          if j = 0 or j = i
               C[i, j] ← 1
          else
               C[i, j] ← C[i-1, j-1] + C[i-1, j]

Return C[n.k]
```

Running time: $\Theta(nk)$ k is bounded by n so in the worst case $\Theta(n^2)$

Space efficiency: $\Theta(nk)$ or $\Theta(n^2)$

DP Examples

- Fibonacci
- Binomial Coefficients
- Longest Common Subsequence
- Longest Increasing Subsequence
- Knapsack
- Shortest Path
- Edit Distance
- Rod Cutting
- Optimal BST

Longest Common Subsequence

- Given two sequences x[1..m] and y[1..n]

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  find a maximum length common subsequence (LCS) of X and Y
- Example

  $X = \langle A, B, C, B, D, A, B \rangle$
- Subsequences of X:
  - A subset of elements from the sequence taken in order

  $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Longest Common Subsequence (LCS)

Application: Comparison of two DNA strings

Ex: X= $\langle$A, B, C, B, D, A, B $\rangle$, Y= $\langle$B, D, C, A, B, A$\rangle$

Longest Common Subsequence:

X = A BC**B** **D** A **B**

Y = **B** **D** C A **B** A

$\langle$B, D, B $\rangle$ is a common subsequence with length 3 but is it the longest?

LCS is not unique

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y (length = 4)

- $\langle B, C, A \rangle$ is a CS of X and Y but not the longest

Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are 2^m subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check
  - scan Y for first letter, from there scan for second, and so on
- Running time: $\Theta(n2^m)$

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values bottom-up in a table.
4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

Notations

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ we define the i-th prefix of X, for i = 0, 1, 2, …, m

  $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ or x[1,..,i]

  $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$ or y[1,..,j]

- c[i, j] = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

Making the choice

$X = \langle A, B, D, E \rangle$

$Y = \langle Z, B, E \rangle$

- Choice: include one element into the common sequence (E) and solve the resulting subproblem

$X = \langle A, B, D, G \rangle$

$Y = \langle Z, B, D \rangle$

- Choice: exclude an element from a string and solve the resulting subproblem

A Recursive Solution

Case 1: $x_i = y_j$

$X_i = \langle A, B, D, E \rangle$

$Y_j = \langle Z, B, E \rangle$

$$c[i, j] = c[i-1, j-1] + 1$$

- Append $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}
- Must find a LCS of X_{i-1} and Y_{j-1} $\Rightarrow$ optimal solution to a problem includes optimal solutions to subproblems

A Recursive Solution

Case 2: $x_i \neq y_j$

$X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$c[i, j] = \max \{ c[i-1, j], c[i, j-1] \}$$

– Must solve two problems

- find a LCS of X_{i-1} and Y_j: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
- find a LCS of X_i and Y_{j-1}: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$

- Optimal solution to a problem includes optimal solutions to subproblems

Overlapping Subproblems

- To find a LCS of X and Y
  - we may need to find the LCS between X and Y_{n-1} and that of X_{m-1} and Y
  - Both the above subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}
- Subproblems share subsubproblems

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define c[i,j] to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be c[m,n]

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** x[i]=y[j]: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** x[i] != y[j]
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

LCS Length Algorithm

```
LCS-Length(X, Y)
    m = length(X)  // get the # of symbols in X
    n  = length(Y) // get the # of symbols in Y
    for i = 1 to m
        c[i,0] = 0        // special case: Y_0
    for j = 1 to n
        c[0,j] = 0        // special case: X_0
    for i = 1 to m                          // for all X_i
        for j = 1 to n                      // for all Y_j
            if ( X_i == Y_j )
                c[i,j] = c[i-1,j-1] + 1
            else c[i,j] = max( c[i-1,j], c[i,j-1] )
    return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

X = ABCB

Y = BDCAB

LCS Example (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[4,5]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0				
2	B		0					
3	C		0					
4	B		0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example

ABCB
BDCAB

j		0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example

ABCB
BDCAB

i	Xi	j 0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

if (X_i == Y_j)
  $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example

ABCB
BDCAB

j		0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB

BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

i		j: 0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if (X_i == Y_j)

  c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (13)

ABCB
BDCAB

i	Xi	j: 0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**				

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (15)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(mn)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

Finding LCS

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Finding LCS (2)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1 ←	1	1	1	2
3	C	0	1	1	2 ←	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

Additional Information

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

b & c:

		0	1	2	3		n
		y_j:	A	C	D		F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		↖ c[i,j-1]	↑		
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$

  b[i, j] = “ ”
- Else, if c[i - 1, j] ≥ c[i, j-1]

  b[i, j] = “ ↑ ”

  else

  b[i, j] = “ ← ”

Example

$X = \langle A, B, C, B, D, A \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

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

If $x_i = y_j$

$b[i, j] =$ " ↖ "

Else if $c[i - 1, j] \geq c[i, j-1]$

$b[i, j] =$ " ↑ "

else

$b[i, j] =$ " ← "

		0	1	2	3	4	5	6
	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

Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a "↖" in b[i, j] $\Rightarrow$ $x_i = y_j$ is an element of the LCS

		0	1	2	3	4	5	6
		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

PRINT-LCS(b, X, i, j)

if (i = 0 or j = 0) Running time: $\Theta(m + n)$

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)

}

Initial call: PRINT-LCS(b, X, length[X], length[Y])

Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time. The row being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows

DP Optimization

- Used for **optimization problems**
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is built in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information

Knapsack problem

Given a set of items, each with a weight and a benefit (value), pack a knapsack with a subset of items to achieve the maximum total benefit (value). Total weight that can be carried in the knapsack is no more than some fixed number W.

There are two versions:

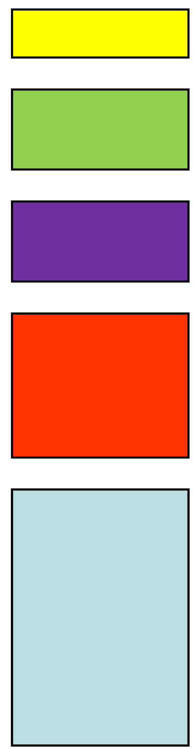
1. "0-1 knapsack problem" use DP
   Items are indivisible: you either take an item or not.

2. "Fractional knapsack problem" Use a Greedy Method
   Items are divisible: you can take any fraction of an item

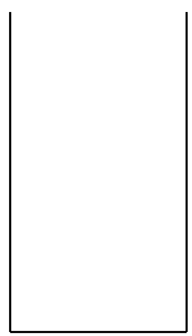
0-1 Knapsack problem:

This is a knapsack
Max weight: $W = 13$

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16



Benefit = 0
Weight = 0

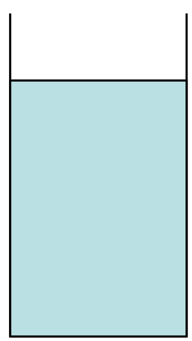


0-1 Knapsack problem:

This is a knapsack
Max weight: $W = 13$

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Benefit = 16
Weight = 10

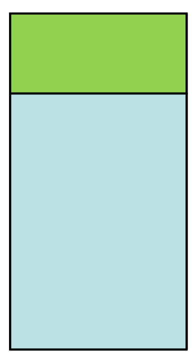


0-1 Knapsack problem:

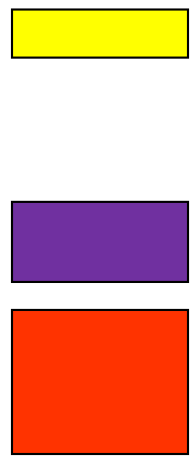
This is a knapsack
Max weight: W = 13

Is this maximum ?

Benefit = 20
Weight = 13



Items



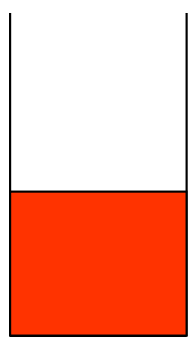
Weight w_i	Benefit value b_i
2	3
3	4
4	5
5	15
10	16

0-1 Knapsack problem:

This is a knapsack
Max weight: W = 13

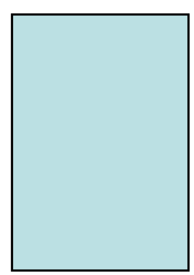
Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Benefit = 15
Weight = 5

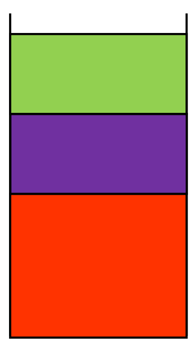


0-1 Knapsack problem:

This is a knapsack
Max weight: W = 13

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Benefit = 24
Weight = 12



0-1 Knapsack Problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total benefit of packed items?

0-1 Knapsack problem

Let S be the set of items represented by the ordered pairs (w_i, b_i) and W be the capacity of the knapsack.

Find a T $\subseteq$ S such that

$$\max \sum_{i \in T} b_i \text{ subject to} \sum_{i \in T} w_i \leq W$$

The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.

0-1 Knapsack Brute-Force

Let's first solve this problem with a straightforward algorithm

- Since there are *n* items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to *W*
- Running time will be $O(n2^n)$

Can we do better?
Yes, with an algorithm based on dynamic programming
We need to carefully identify the subproblems

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that. ….Why???

Defining a Subproblem

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

Max weight: $W = 20$

For S_4: {1, 2, 3, 4}

Total weight: 14;

total benefit: 20

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

For S_5: { 1, 3, 4, 5 }

Total weight: 20

total benefit: 26

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

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: *w*, which will represent the exact weight for each subset of items
- The subproblem then will be to compute B[k,w] which is the maximum benefit for total capacity w and items {1, 2, ..k}

Recursive Formula

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

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 solution, since if it was, the total weight would be > w. So we select the "optimal" using items 1,.., k-1
- **Second case**: $w_k \le w$. Then the item k can be in the solution, and we choose the case with greater value

Recursive Formula for subproblems

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

It means, that the best subset of S_k that has total weight w is one of the two:

Item k is too big to fit in the knapsack with capacity w

Do not use item k: the best subset of S_{k-1} that has total weight w, **or**

Use item k: the best subset of S_{k-1} that has total weight w-w_k plus the item k *with benefit* b_k

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0       // 0 item's
for i = 0 to n
    B[i,0] = 0          // 0 weight
    for w = 1 to W
        if w_i <= w // item i can be part of 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 item i is too big
```

Running time

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

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(nW) pseudo-polynomial

Remember that the brute-force algorithm takes $O(n2^n)$.
Better than Brute force if $W << 2^n$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
S = {(2,3), (3,4), (4,5), (5,6) }

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

for w = 0 to W

$B[0,w] = 0$

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

B[i,0] = 0

Item : (w, b)

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

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

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i <= w$ // item i can be part of 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]** // $w_i > w$

Item : (w, b)

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				

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

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

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Item : (w, b)

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				

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

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

if $w_i <= w$ // item i can be part of 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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

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				

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

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i <= w$ // item i can be part of 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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

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**			

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

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of 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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

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			

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

i=2

b_i=4

w_i=3

w=1

w-w_i=-2

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

Item : (w, b)

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			

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

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

if w_i <= w // item i can be part of 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]** // $w_i > w$

Item : (w, b)

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			

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

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if w_i <= w // item i can be part of 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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

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			

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

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

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

Item : (w, b)

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		

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

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

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

Item : (w, b)

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		

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

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

if $w_i \le w$ // item i can be part of 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]** // $w_i > w$

Item : (w, b)

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		

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

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

if $w_i <= w$ // item i can be part of 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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

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	

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

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

if $w_i <= w$ // item i can be part of 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]   // $w_i > w$

Item : (w, b)

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	

$i=4$

$b_i=6$

$w_i=5$

$w=1..4$

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

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

Item : (w, b)

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**

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

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

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

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- See LCS algorithm for the example how to extract this data from the table we built using “parent pointers”.

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time (Dynamic Programming algorithm vs. naïve algorithm):
  - LCS: **$O(mn)$** vs. **$O(n\, 2^m)$**
  - 0-1 Knapsack problem: **$O(Wn)$** vs. **$O(n2^n)$**

Pseudo-polynomial