COP 3530: Computer Science III
Summer 2005

Dynamic Programming

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
CSB 242, (407)823-2790

Course Webpage:
http://www.cs.ucf.edu/courses/cop3530/sum2005

School of Computer Science
University of Central Florida

What Is Dynamic Programming?

- Dynamic programming is an algorithm design technique with a rather interesting history. It was invented in 1957 by prominent U.S. mathematician Richard Bellman as a general method for optimizing multistage decision processes.
  - The word “programming” in the name of this technique stands for “planning” or “a series of choices” and does not refer to computer programming. The word “dynamic” conveys the idea that the choices may depend on the current state, rather than being decided ahead of time.
    - Useful analogy is a pre-programmed radio show with a set play-list as opposed to a call-in radio show where listeners request songs to be played – the call-in show is “dynamically programmed.”
- Originally a tool of applied mathematics designed for optimization problems, in computer science it is considered as a general algorithm design technique which is not limited to optimization problems.

What Is Dynamic Programming? (cont.)

- Dynamic programming is a technique for solving problems with overlapping sub-problems.
- Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub-problems of the same type.
- Rather than solving overlapping sub-problems again and again, dynamic programming solves each of the smaller sub-problems only once and stores the results in a table from which the solution to the original problem can be obtained.
- One of the defining features of dynamic programming is that it is capable of replacing exponential-time computation with a polynomial-time computation.

What Is Dynamic Programming? (cont.)

- Dynamic programming is similar to divide and conquer in the sense that it is based on a recursive division of a problem instance into smaller or simpler problem instances.
- Divide and conquer algorithms often use a top-down resolution method (working from the larger problem down to the smaller problem).
- Dynamic programming algorithms invariably proceed by solving all of the simplest problem instances before combining them into more complicated problem instances in a bottom-up fashion.
- Let's first look at dynamic programming as it is applied to a non-optimization problem.

Computing A Binomial Coefficient

- Computing a binomial coefficient is a basic example of applying dynamic programming to a non-optimization problem.
- Recall that the binomial coefficient, is the number of combinations (subsets) of *k* elements from an *n*-element set ($0 \leq k \leq n$) and is denoted as C(n,k) or $\binom{n}{k}$.
- The name "binomial coefficient" comes from the participation of these numbers in the so-called binomial formula:

$$(a+b)^n = C(n,0)a^n + \cdots + C(n,i)a^{n-i}b^i + \cdots + C(n,n)b^n$$

Computing A Binomial Coefficient (cont.)

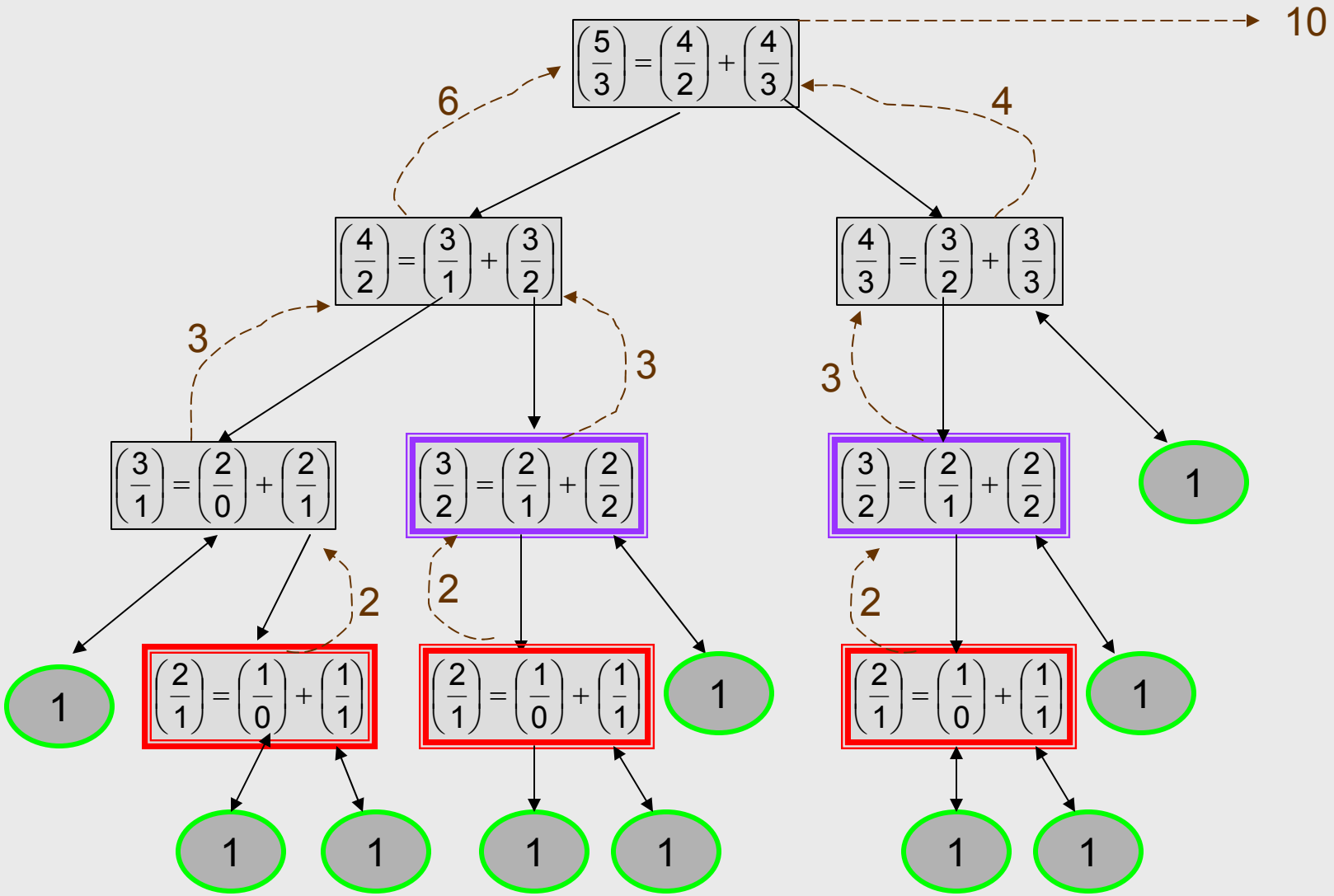
- Of the numerous properties of binomial coefficient, we need to concentrate on only two:

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \end{cases}$$

- Let's consider the case of computing C(5,3) using this recurrence.
- C(5,3) = C(4,2) + C(4,3)

Also expressed as: $\binom{5}{3} = \binom{4}{2} + \binom{4}{3}$

Computing A Binomial Coefficient (cont.)



Computing A Binomial Coefficient (cont.)

- The nature of the recurrence on page 6, which expresses the problem of computing C(n, k) in terms of the smaller and overlapping problems of computing C(n-1, k-1) and C(n-1, k), lends itself to solving using the dynamic programming approach.
- To do this, we'll record the values of the binomial coefficients in a matrix of n+1 rows and k+1 columns, numbered from 0 to n and 0 to k, respectively.
- The dynamic programming algorithm to solve the binomial coefficient problem is given on the next page, followed by an example computing C(5,3).

Computing A Binomial Coefficient (cont.)

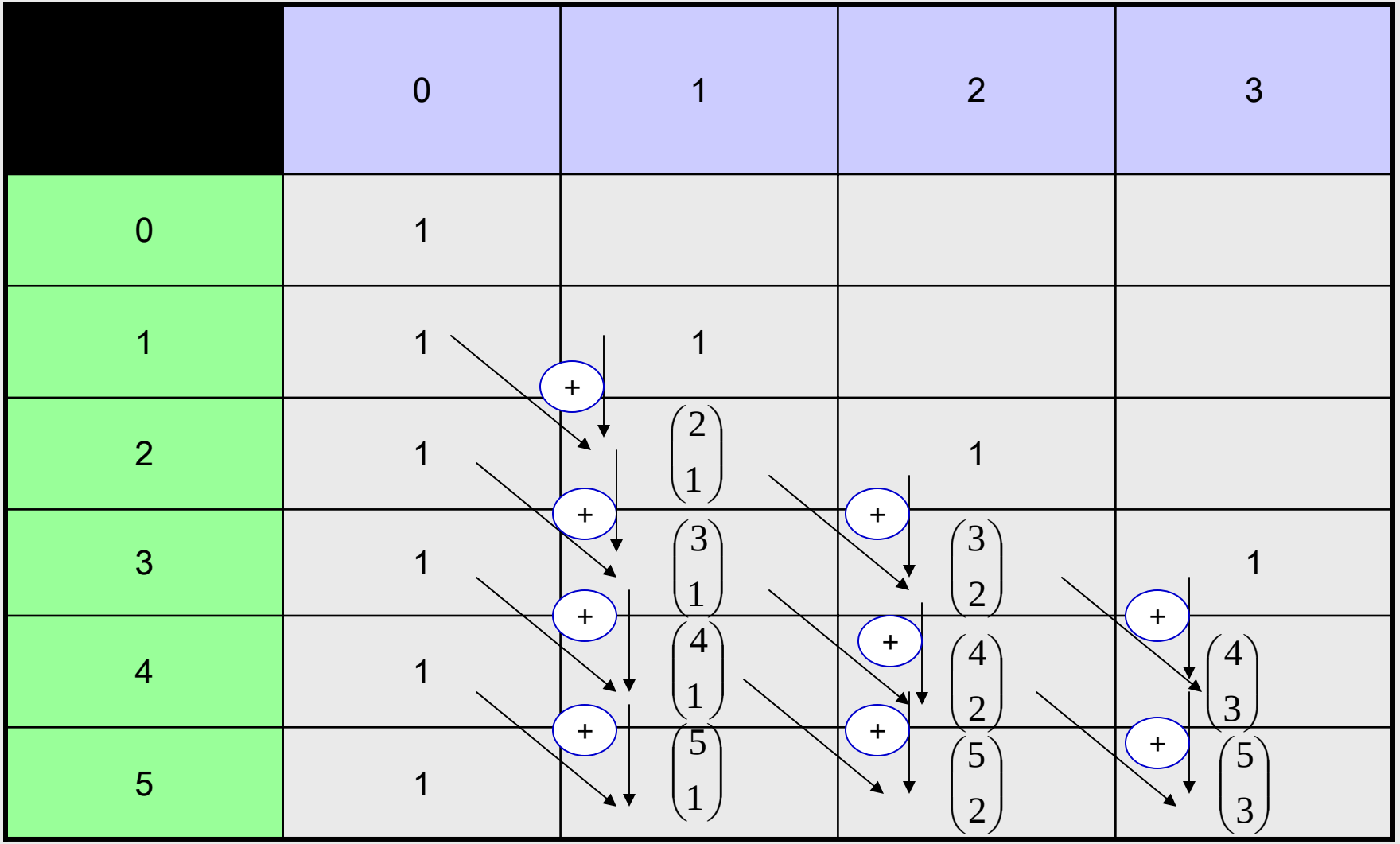
```
Algorithm Binomial
//Computes C(n,k) using dynamic programming
//Input: non-negative integers n ≥ k ≥ 0
//Output: C(n,k)

for i ←0 to n do

   for j ← 0 to min(i, k) do
       if j = 0 or j = k

          c[i,j] ← 1

       else C[i,j] ← C[i-1,j-1] + C[i-1,j]
Return C[n,k]
```

To compute C(n,k), the matrix is filled row by row, starting with row 0 and ending with row *n*. Each row i ($0 \leq i \leq n$) is filled left to right, starting with 1 because C(n,0) = 1. Rows 0 through *k* also end with 1 on the matrix diagonal: C(i, i) = 1 for $0 \leq i \leq k$. The other values in the matrix are computed by adding the contents of the cells in the preceding row and the previous column and in the preceding row and the same column.

Computing A Binomial Coefficient (cont.)

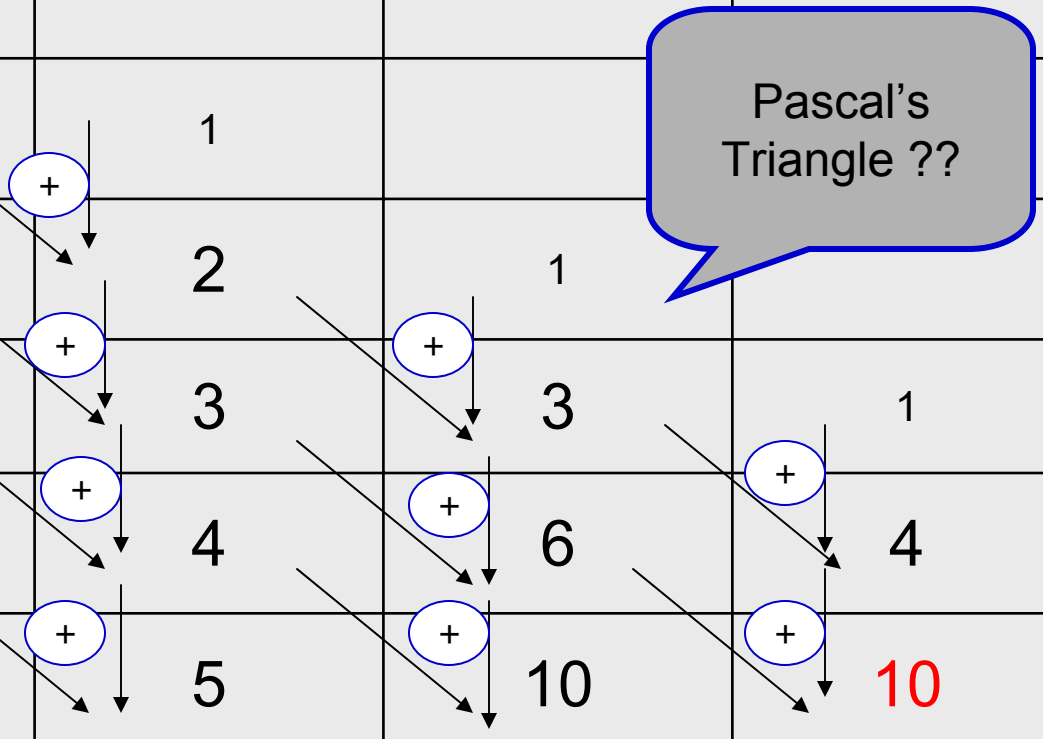
	0	1	2	3
0	1			
1	1	1		
2	1	$\binom{2}{1}$	1	
3	1	$\binom{3}{1}$	$\binom{3}{2}$	1
4	1	$\binom{4}{1}$	$\binom{4}{2}$	$\binom{4}{3}$
5	1	$\binom{5}{1}$	$\binom{5}{2}$	$\binom{5}{3}$

Computing A Binomial Coefficient (cont.)



Computing A Binomial Coefficient (cont.)

	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10



Computing A Binomial Coefficient (cont.)

	0	1	2	3	…	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1			1			
…							
k	1						1
…							
n-1	1					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n	1						$\binom{n}{k}$

For general C(n,k) case

Computing A Binomial Coefficient (cont.)

- What is the time complexity of the dynamic programming binomial coefficient algorithm?
- Obviously, the basic operation is addition, so let *A*(n,k) be the total number of additions made by the algorithm when computing C(n,k). Computing each entry in the matrix requires just one addition.
- The first *k+1* rows of the table form a triangle while the remaining *n-k* rows form a rectangle. This causes us to split the sum expressing *A*(n,k) into two parts:

$$A(n,k)=\sum_{i=1}^{k}\sum_{j=1}^{i-1}1+\sum_{i=k+1}^{n}\sum_{j=1}^{k}1=\sum_{i=1}^{k}(i-1)+\sum_{i=k+1}^{n}k=\frac{(k-1)k}{2}+k(n-k)\in\Theta(nk)$$

Dynamic Programming and Optimization Problems

- The binomial coefficient problem was an example of the application of dynamic programming to a non-optimization problem.
- Dynamic programming is commonly applied to optimization problems. Optimization problems typically wish to find the "best" way of doing something.
- Often the number of different ways of doing that "something" is exponential, so a brute-force search for the best solution is computationally infeasible for all but the smallest problem sizes.
- Dynamic programming comes to the rescue is such situations; if the problem has a certain amount of structure that can be exploited.

Basic Requirements for Dynamic Programming and Optimization Problems

1. Simple Sub-problems: There has to be some way of breaking the global optimization problem into sub-problems, each having a similar structure to the original problem.

2. Sub-problem optimality: An optimal solution to the global problem must be a composition of optimal sub-problem solutions, using a relatively simple combining operation. It must not be possible to find a globally optimal solution that contains sub-optimal sub-problems. (Principle of Optimality)

3. Sub-problem Overlap: Optimal solutions to unrelated sub-problems can contain sub-problems in common. Indeed, such overlap improves the efficiency of a dynamic programming algorithm that stores solutions to sub-problems.

Principle of Optimality

- The Principle of Optimality states that an optimal solution to any instance of an optimization problem is composed of optimal solutions to its sub-instances.
- More often than not, this principle will hold in a optimization problem. (An example of a rare case where the principle of optimality does not hold is in finding the longest simple path in a graph – we'll see this problem later in the term.)
- Although its applicability to a particular problem needs to be checked – it is usually not a principle difficulty in developing a dynamic programming algorithm. The challenge typically lies in figuring out what smaller sub-instances need to be considered and in deriving an equation relation a solution to any instance with solutions to its smaller sub-instances.

The 0-1 Knapsack Problem

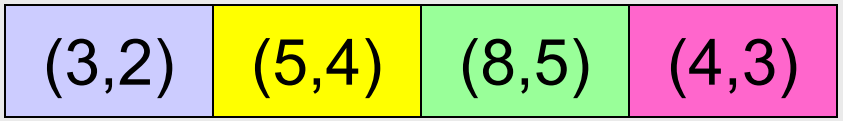
- The 0-1 Knapsack problem consists of a knapsack with a fixed capacity, W (weight or volume), a set of objects, S where each object in S has an associated weight, w_i (or volume) and benefit, b_i. The objective is to maximize the benefit of objects selected to be placed in the knapsack without exceeding the capacity of the knapsack.
- Note that the problem is easily solved in $\Theta(2^n)$ time, by enumerating all subsets of S and selecting the one with the highest benefit from among all those with total weight not exceeding W (brute-force technique).
- As with many dynamic programming problems, one of the hardest parts of designing an algorithm for the 0-1 knapsack problem is to find a nice characterization for sub-problems (so that the three requirements are satisfied).

The 0-1 Knapsack Problem (cont.)

- As an example, let's consider the following 0-1 knapsack problem: Let S = {(3,2), (5,4), (8,5), (4,3), (10,9)} and W = 20. (Let pairs be denoted as (weight, benefit).)
- Approach 1: Number the items in S as 1, 2, …,n and define, for each $k \in$ {1, 2, …, n}, the subset S_k = {items in S labeled 1, 2, …, k}.
  - One way to define sub-problems by using parameter k so that sub-problem k is the best way to fill the knapsack using only items from the set S_k. This would be a valid sub-problem definition, but it is not clear how to define an optimal solution for index k in terms of optimal sub-problem solutions.
  - Unfortunately, this solution won't work. Why?

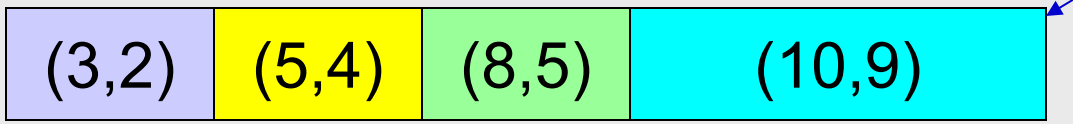
The 0-1 Knapsack Problem (cont.)

- Let S = {(3,2), (5,4), (8,5), (4,3), (10,9)} and W = 20.



Using first four items in S

Weight = 20, Total benefit = 14



Using first five items in S

Weight = 20, Total benefit = 20

The 0-1 Knapsack Problem (cont.)

- The reason that defining the sub-problems only in terms of an index *k* doesn't work is that there is not enough information represented in a sub-problem to help in solving the global optimization problem.
- In other words, we need to get the weights of the objects involved.
- We'll add a second parameter (in addition to *k*), called *w*, to represent the weight.
- Approach 2: Formulate each sub-problem as computing *B*[k, w], which is defined as the maximum total value of a subset of S_k from among all those subsets having total weight exactly equal to *w*. Thus, B[0,w] = 0 for each $w \leq W$.

The 0-1 Knapsack Problem (cont.)

- The general case is:

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

- That is, the best subset of S_k that has a total weight w is either the best subset of S_{k-1} that has total weight w or the best subset of S_{k-1} that has total weight $w - w_k$ plus the item k.

- Since the best subset of S_k that has total weight w must either contain item k or not, one of these two choices must be the right choice. Thus, we have a sub-problem definition that is simple (it involves only 2 parameters), satisfies the sub-problem optimization condition, and it has sub-problems which overlap, for the optimal way of summing exactly w to weight may be used by many sub-problems.

The 0-1 Knapsack Problem (cont.)

- Before looking at the algorithm for the 0-1 Knapsack problem, note one additional item.
- The definition of B[k,w] is built from B[k-1,w] and possibly B[k-1,w-w_k].
- Thus, the algorithm can be implemented using only a single array B, which can be updated in each of a series of iterations indexed by parameter *k*, so that at the end of each iteration B[w] = B[k,w].

The 0-1 Knapsack Problem (cont.)

```
Algorithm 01Knapsack (S, W)

Input: A set S of n items, such that item I has
positive benefit b_i and positive integer weight
w_i; positive integer maximum total weight W

Output: For w = 0,…,W, maximum benefit B[w] of a
subset of S with total weight w.

for w ← 0 to W do

   B[w] ← 0

for k ← 1 to n do

   for w ← W downto w_k do

      if B[w-w_k]+b_k > B[w] then

         B[w] ← B[w-w_k]+b_k
```

O(nW)