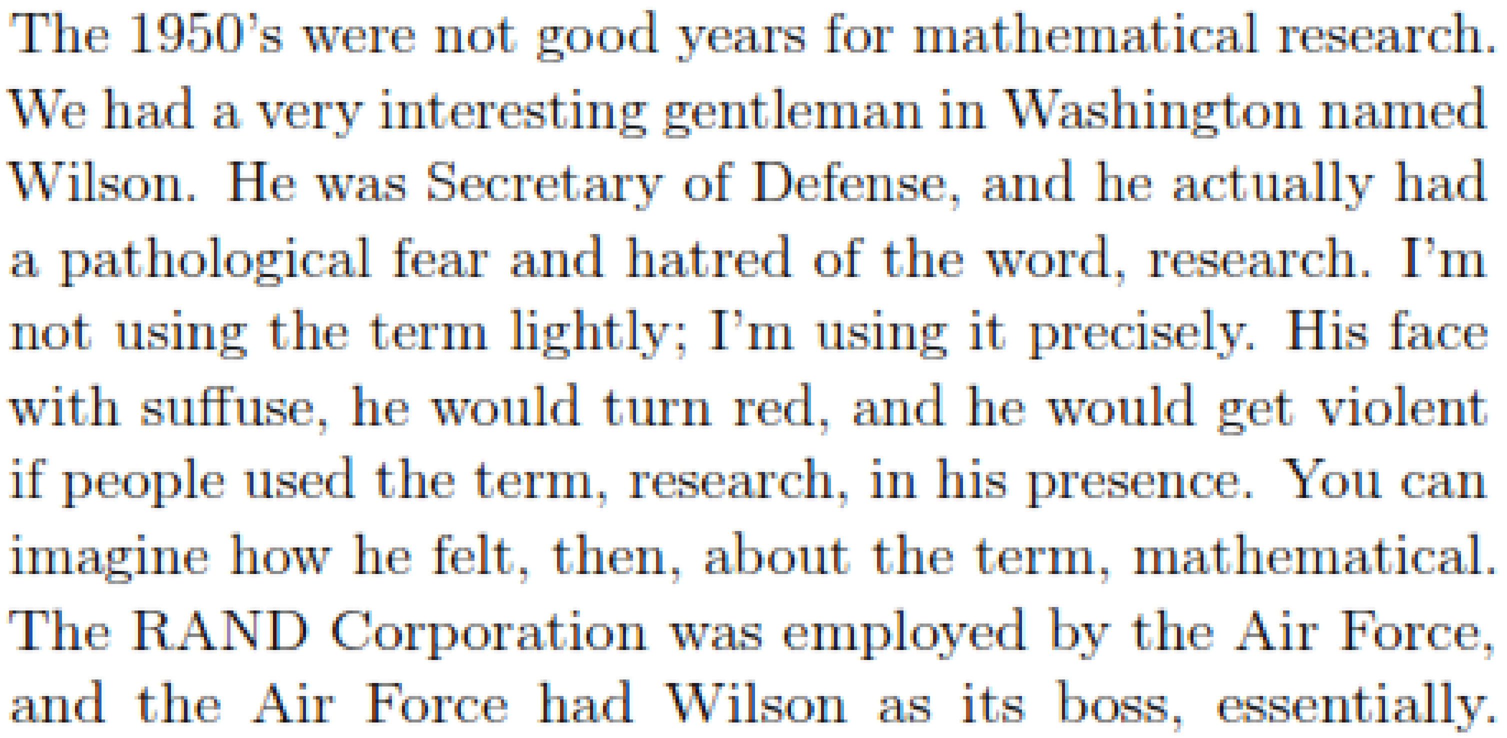
Dynamic Programming – Knapsack Problem

V. Balasubramanian

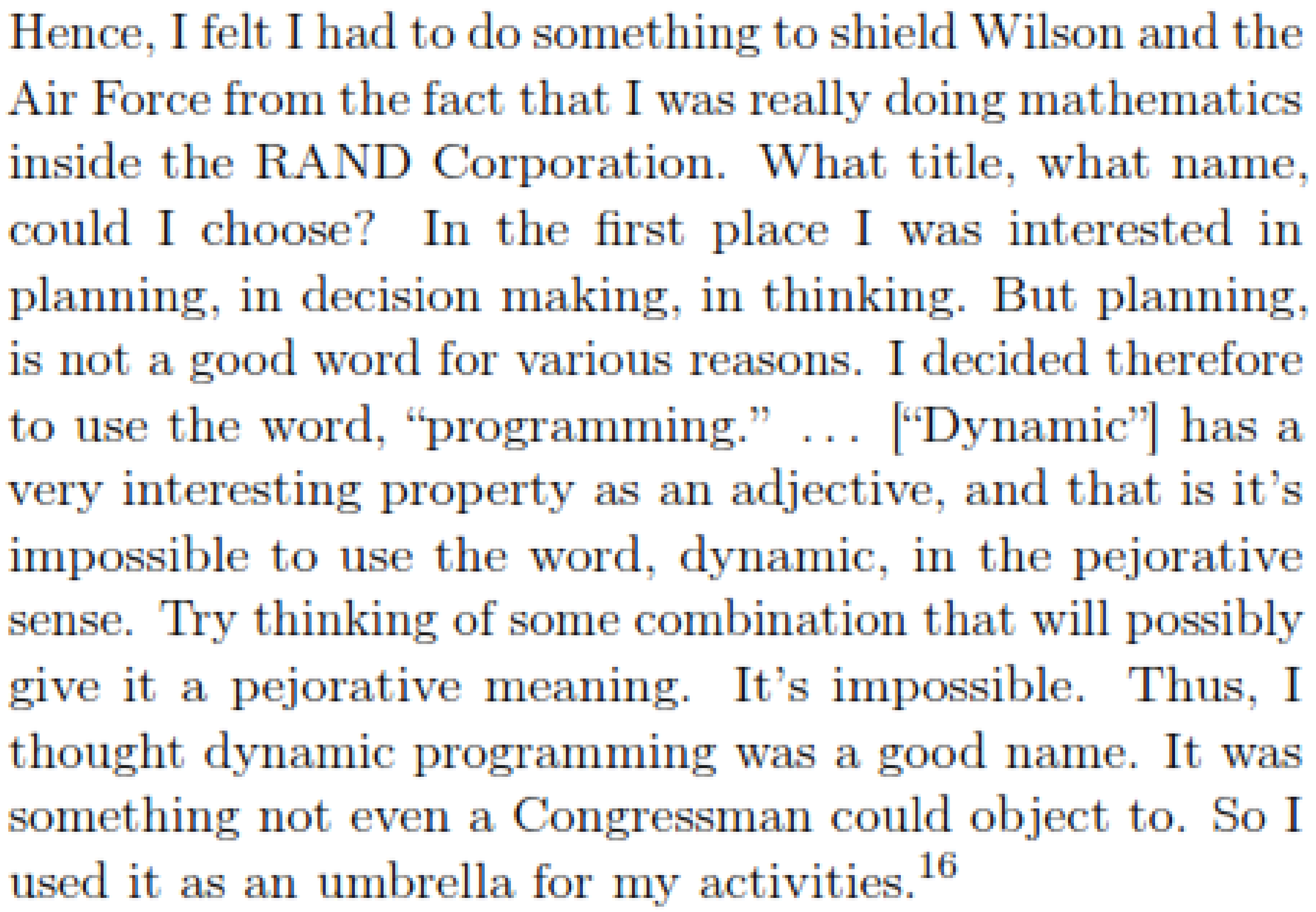
SSN College of Engineering

Richard Bellman



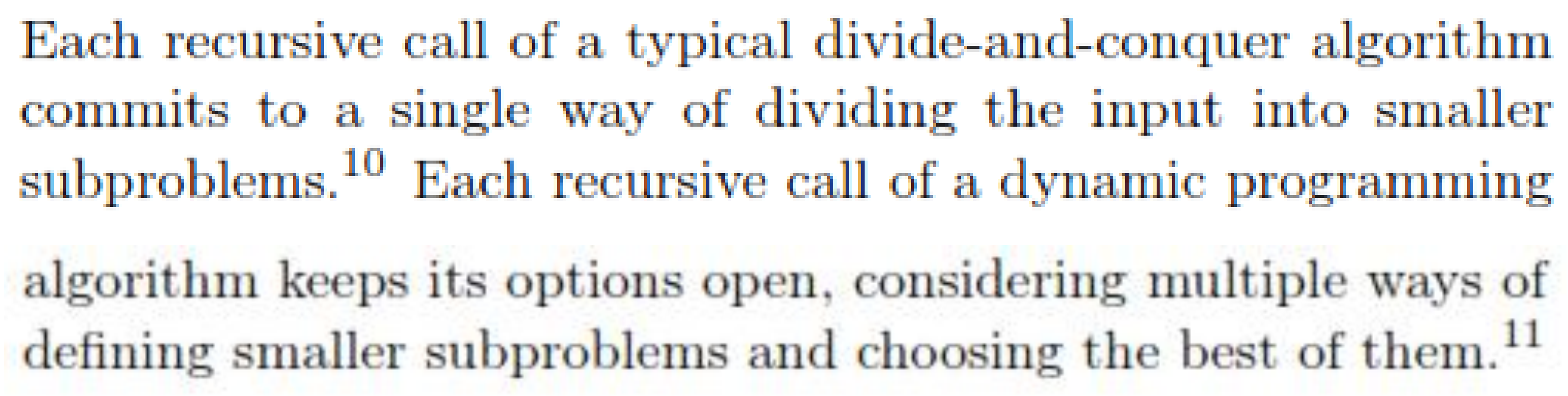
The 1950's were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face with suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially.

Bellaman



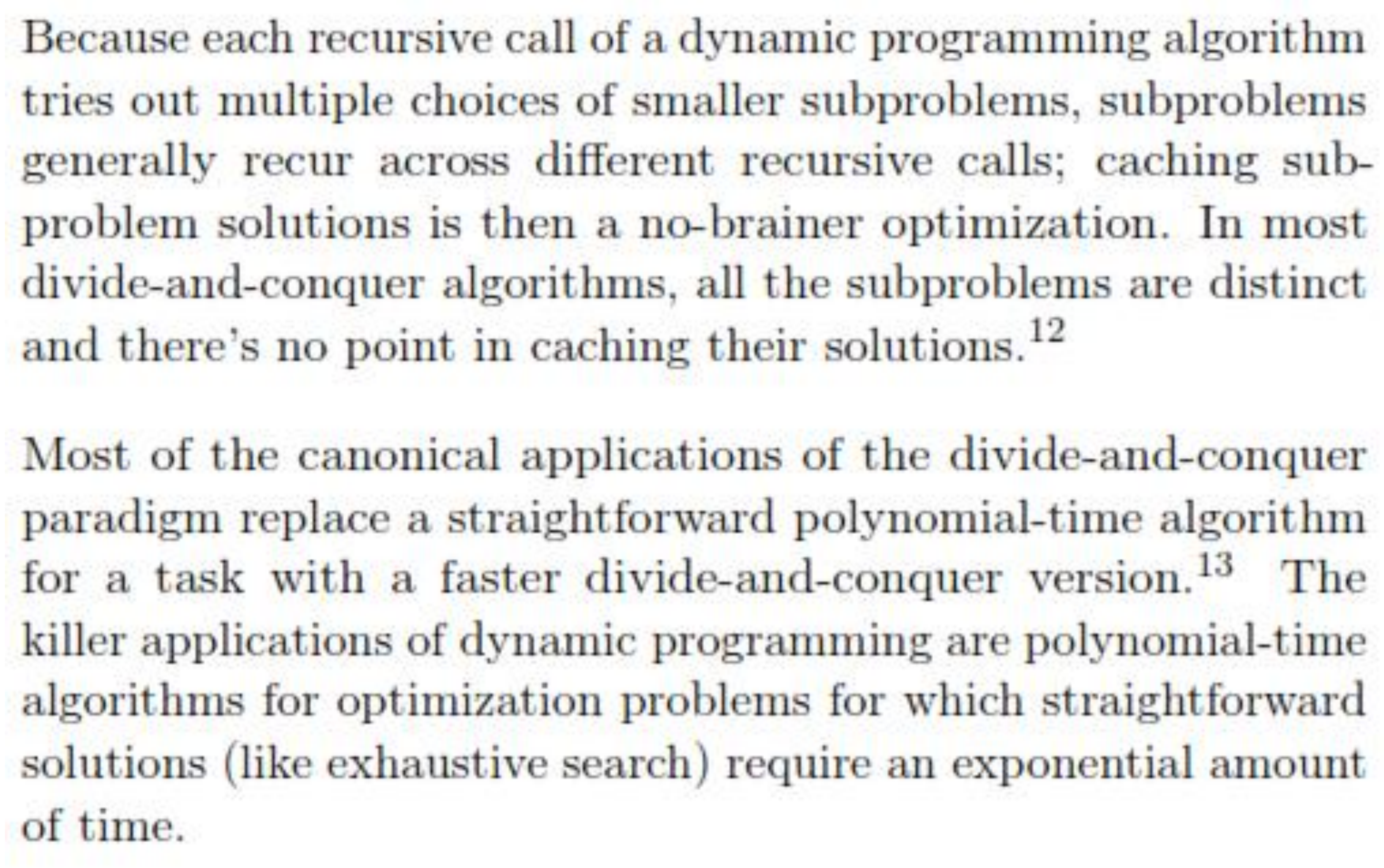
Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, “programming.” . . . [“Dynamic”] has a very interesting property as an adjective, and that is it’s impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.[16]

Difference with divide and conquer



Each recursive call of a typical divide-and-conquer algorithm commits to a single way of dividing the input into smaller subproblems.[10] Each recursive call of a dynamic programming algorithm keeps its options open, considering multiple ways of defining smaller subproblems and choosing the best of them.[11]

Difference



Because each recursive call of a dynamic programming algorithm tries out multiple choices of smaller subproblems, subproblems generally recur across different recursive calls; caching subproblem solutions is then a no-brainer optimization. In most divide-and-conquer algorithms, all the subproblems are distinct and there's no point in caching their solutions.[12]

Most of the canonical applications of the divide-and-conquer paradigm replace a straightforward polynomial-time algorithm for a task with a faster divide-and-conquer version.[13] The killer applications of dynamic programming are polynomial-time algorithms for optimization problems for which straightforward solutions (like exhaustive search) require an exponential amount of time.

Difference (Weighted Independent set)

In a divide-and-conquer algorithm, subproblems are chosen primarily to optimize the running time; correctness often takes care of itself.[14] In dynamic programming, subproblems are usually chosen with correctness in mind, come what may with the running time.[15]

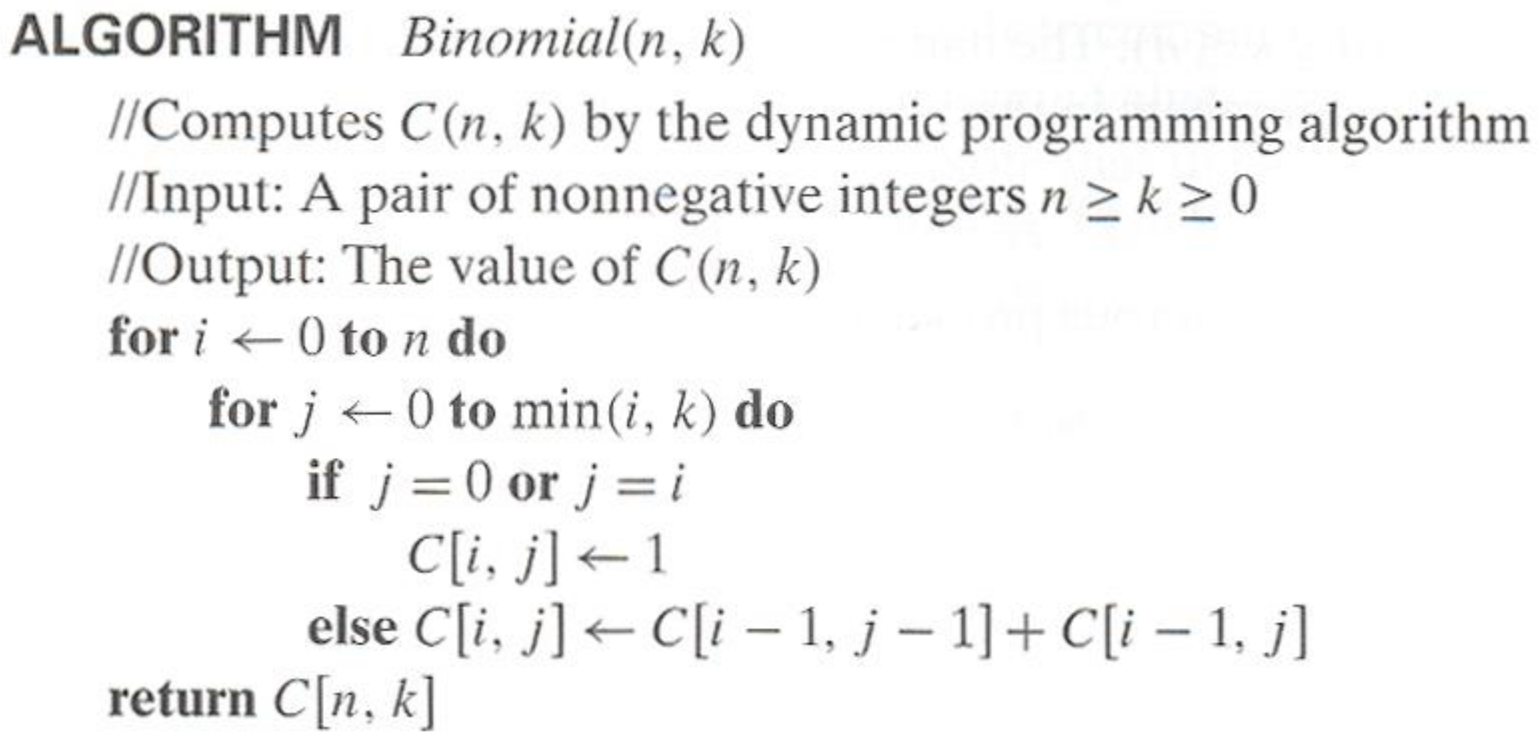
Relatedly, a divide-and-conquer algorithm generally recurses on subproblems with size at most a constant fraction (like 50%) of the input. Dynamic programming has no qualms about recursing on subproblems that are barely smaller than the input (like in the `WIS` algorithm), if necessary for correctness.

The divide-and-conquer paradigm can be viewed as a special case of dynamic programming, in which each recursive call chooses a fixed collection of subproblems to solve recursively. As the more sophisticated paradigm, dynamic programming applies to a wider range of problems than divide-and-conquer, but it is also more technically demanding to apply (at least until you've had sufficient practice).

Binomial Coefficient

- Binomial coefficients are coefficients of the binomial formula:
- $(a+b)^n = C(n,0)a^n b^0 + \cdots + C(n,k)a^{n-k}b^k + \ \ldots + C(n,n)a^0 b^n$
- Recurrence: C(n,k) = C(n-1,k) + C(n-1,k-1) for n > k > 0
- C(n,0) = 1, C(n,n) = 1 for n >= 0

Bionomial Coefficient

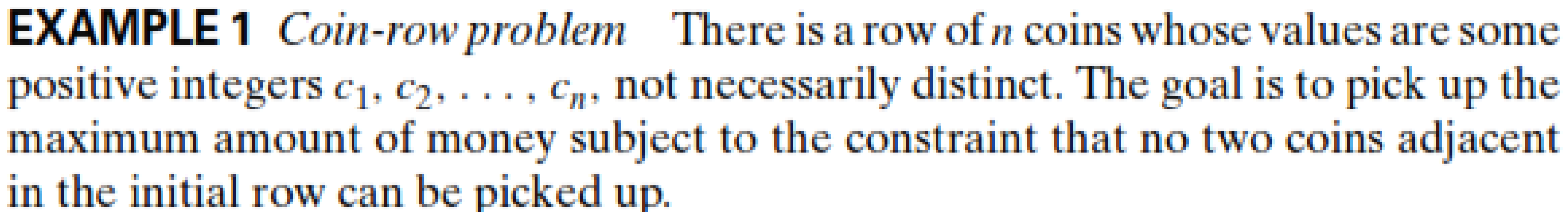


```
ALGORITHM Binomial(n, k)
    //Computes $C(n, k)$ by the dynamic programming algorithm
    //Input: A pair of nonnegative integers $n \geq k \geq 0$
    //Output: The value of $C(n, k)$
    for $i \leftarrow 0$ to $n$ do
        for $j \leftarrow 0$ to $\min(i, k)$ do
            if $j = 0$ or $j = i$
                $C[i, j] \leftarrow 1$
            else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
    return $C[n, k]$
```

	0	1	2	…	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
k	1					1
⋮						
$n-1$	1				$C(n-1, k-1)$	$C(n-1, k)$
n	1					$C(n, k)$

FIGURE 8.1 Table for computing the binomial coefficient $C(n, k)$ by the dynamic programming algorithm

Coin Row Problem



EXAMPLE 1 *Coin-row problem* There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Solution

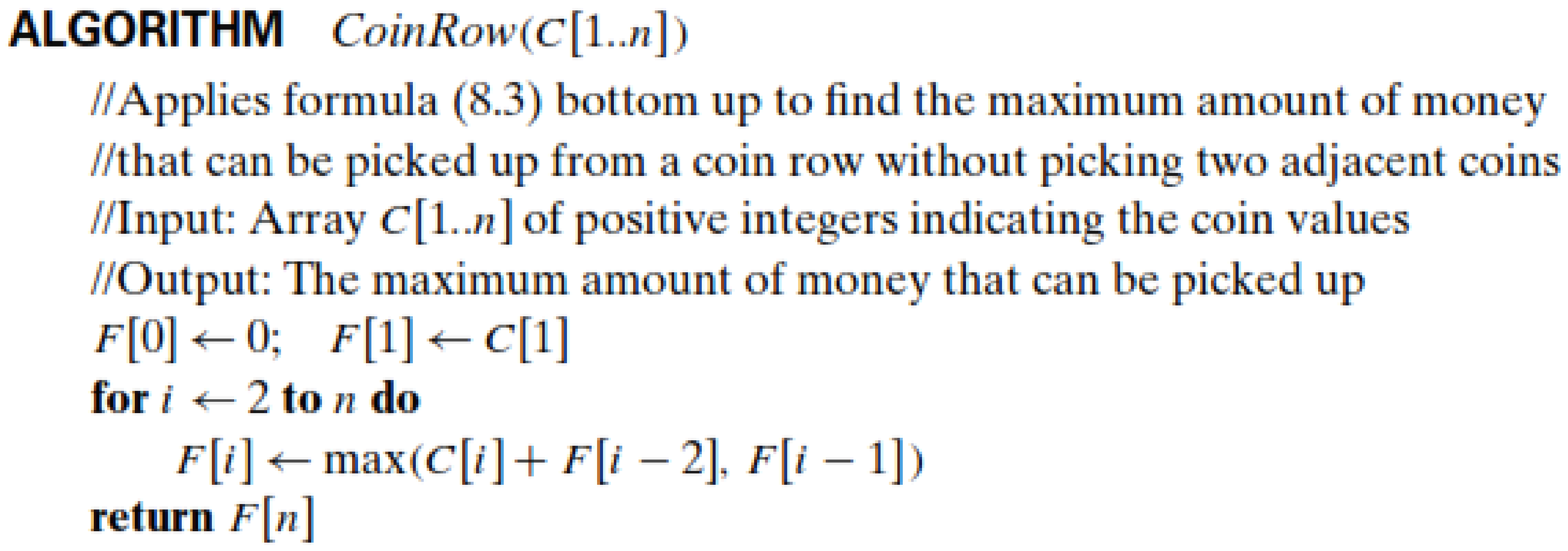
$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1,$$

$$F(0) = 0, \qquad F(1) = c_1.$$

Case 1: $n \notin S$. Because the optimal solution S excludes the last item, it can be regarded as a feasible solution (still with total value V and total size at most C) to the smaller problem consisting of only the first $n-1$ items (and knapsack capacity C). Moreover, S must be an optimal solution to the smaller subproblem: If there were a solution $S^* \subseteq \{1, 2, \ldots, n-1\}$ with total size at most C and total value greater than V, it would also constitute such a solution in the original instance. This would contradict the supposed optimality of S.

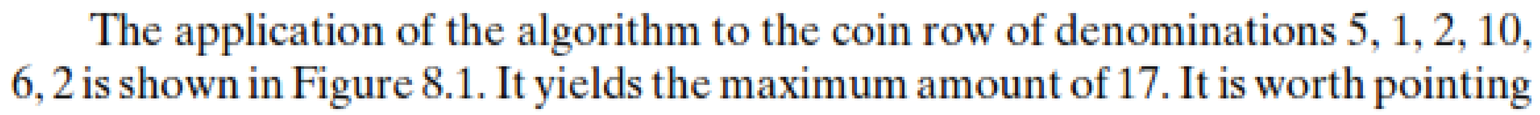
Case 2: $n \in S$. The trickier case is when the optimal solution S makes use of the last item n. This case can occur only when $s_n \leq C$ We can't regard S as a feasible solution to a smaller problem with only the first $n-1$ items, but we can after removing item n. Is $S - \{n\}$ an optimal solution to a smaller subproblem?

Algorithm



```
ALGORITHM CoinRow(C[1..n])
    //Applies formula (8.3) bottom up to find the maximum amount of money
    //that can be picked up from a coin row without picking two adjacent coins
    //Input: Array C[1..n] of positive integers indicating the coin values
    //Output: The maximum amount of money that can be picked up
    $F[0] \leftarrow 0$;   $F[1] \leftarrow C[1]$
    for $i \leftarrow 2$ to $n$ do
        $F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
    return $F[n]$
```

Example


The application of the algorithm to the coin row of denominations 5, 1, 2, 10, 6, 2 is shown in Figure 8.1. It yields the maximum amount of 17. It is worth pointing

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

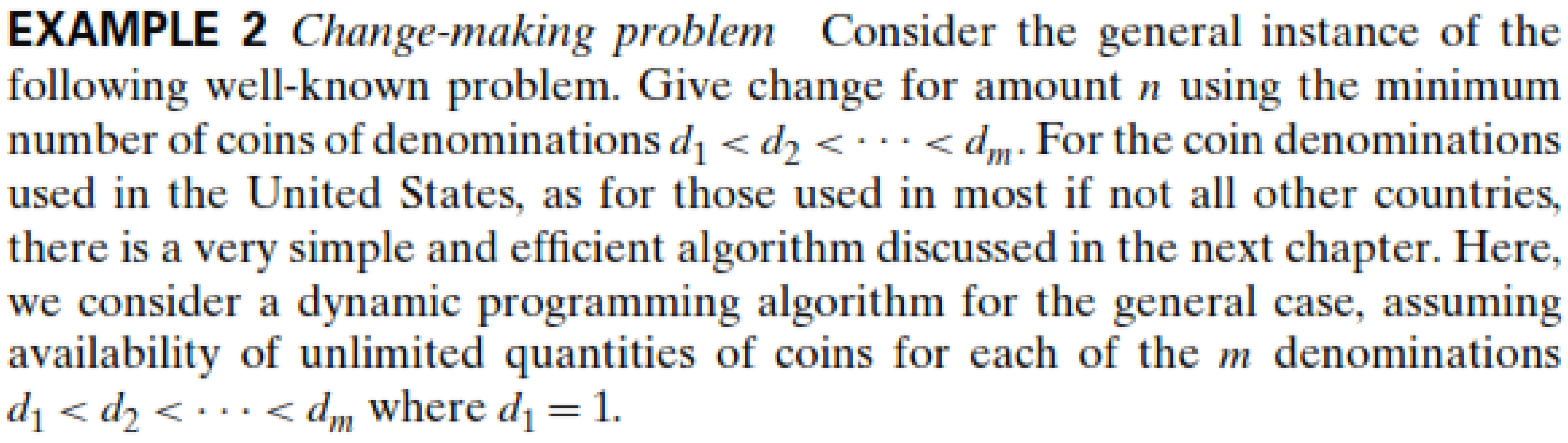
$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Example

Solve the instance 7, 2, 1, 12, 5 of the coin-row problem.

Change Making



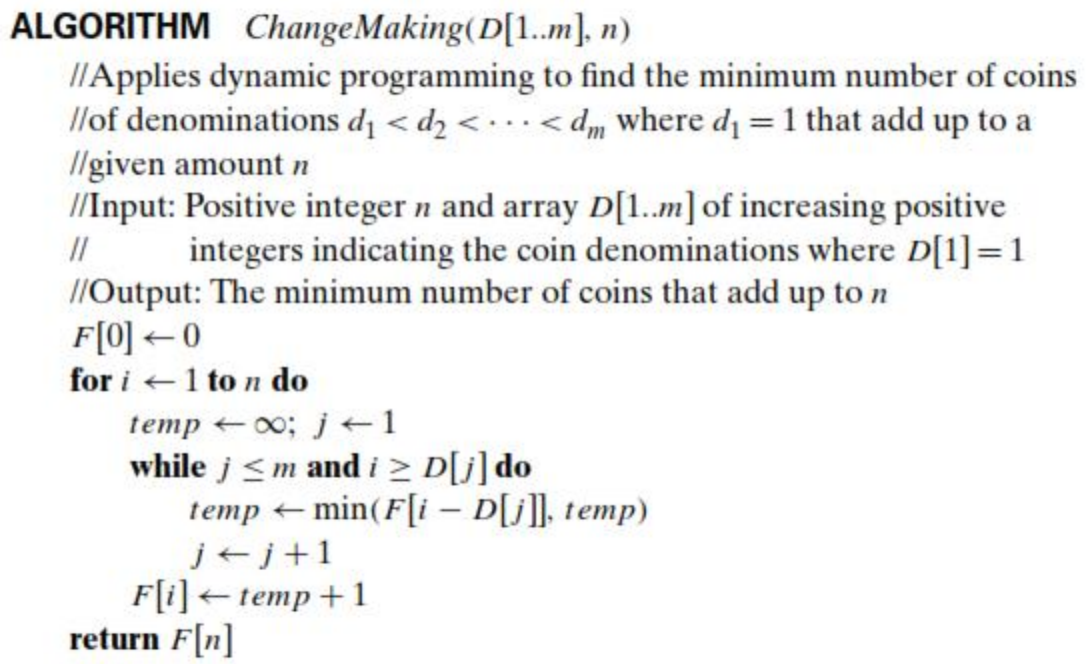
EXAMPLE 2 *Change-making problem* Consider the general instance of the following well-known problem. Give change for amount n using the minimum number of coins of denominations $d_1 < d_2 < \cdots < d_m$. For the coin denominations used in the United States, as for those used in most if not all other countries, there is a very simple and efficient algorithm discussed in the next chapter. Here, we consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the m denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$.

Recurrence Relation

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

Algorithm



```
ALGORITHM ChangeMaking($D[1..m], n$)
    //Applies dynamic programming to find the minimum number of coins
    //of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
    //given amount $n$
    //Input: Positive integer $n$ and array $D[1..m]$ of increasing positive
    //        integers indicating the coin denominations where $D[1] = 1$
    //Output: The minimum number of coins that add up to $n$
    $F[0] \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        $temp \leftarrow \infty$; $j \leftarrow 1$
        while $j \leq m$ and $i \geq D[j]$ do
            $temp \leftarrow \min(F[i - D[j]], temp)$
            $j \leftarrow j + 1$
        $F[i] \leftarrow temp + 1$
    return $F[n]$
```

Example

- Coins 1,2,4 change for n=6

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1 - 1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2 - 1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3 - 1], F[3 - 3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4 - 1], F[4 - 3], F[4 - 4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5 - 1], F[5 - 3], F[5 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6 - 1], F[6 - 3], F[6 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

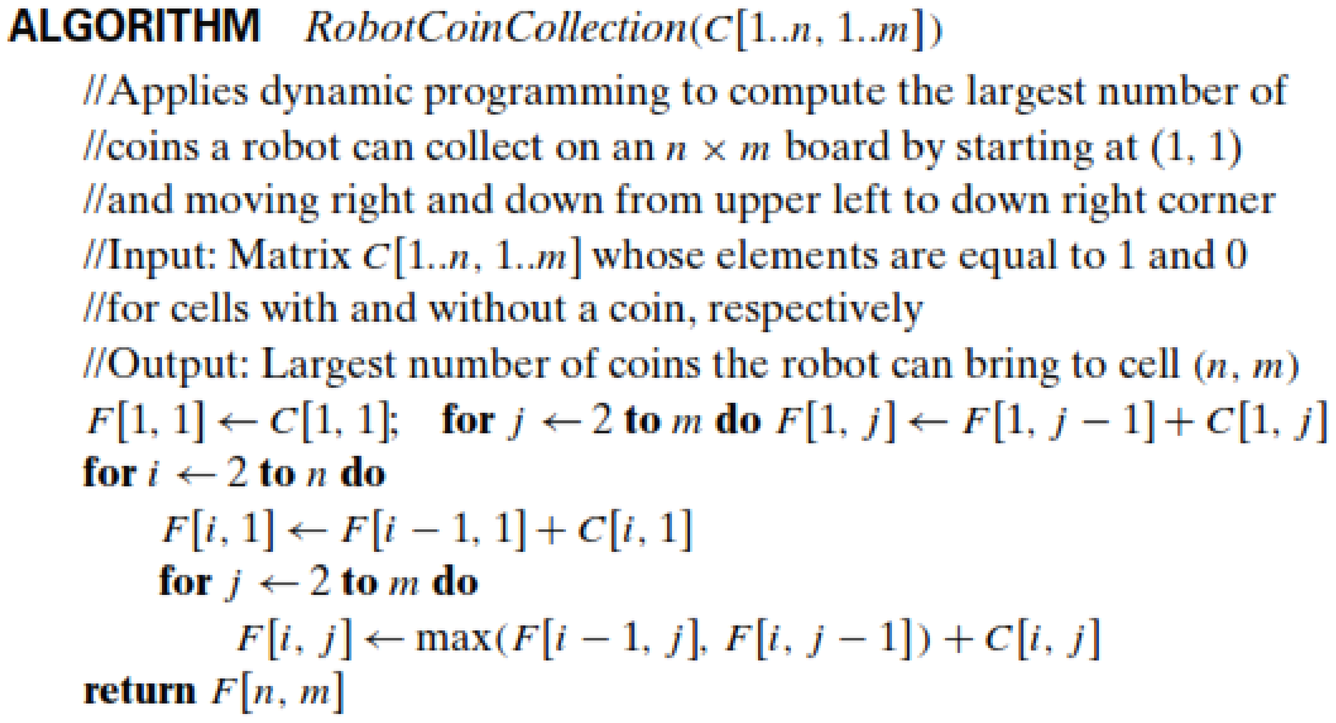
Exercise

Apply the dynamic programming algorithm to find all the solutions to the change-making problem for the denominations 1, 2, 5 and the amount $n = 8$.

Robot Coin Collection

EXAMPLE 3 *Coin-collecting problem* Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin. Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

Algorithm



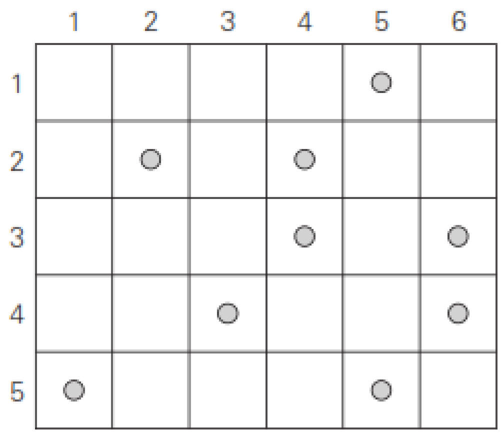
```
ALGORITHM  RobotCoinCollection($C[1..n, 1..m]$)
    //Applies dynamic programming to compute the largest number of
    //coins a robot can collect on an $n \times m$ board by starting at (1, 1)
    //and moving right and down from upper left to down right corner
    //Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
    //for cells with and without a coin, respectively
    //Output: Largest number of coins the robot can bring to cell $(n, m)$
    $F[1, 1] \leftarrow C[1, 1]$;   for $j \leftarrow 2$ to $m$ do $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
    for $i \leftarrow 2$ to $n$ do
        $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
        for $j \leftarrow 2$ to $m$ do
            $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
    return $F[n, m]$
```

Recurrence

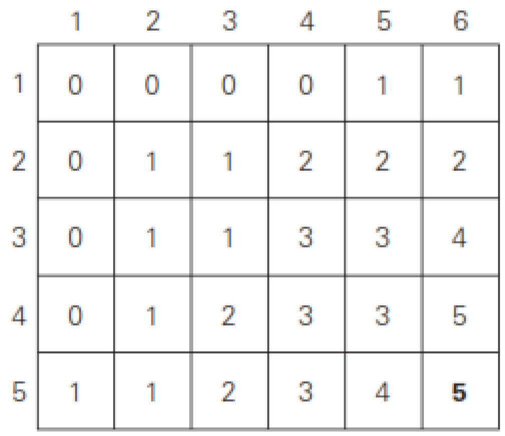
$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \ 1 \le j \le m$$

$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n,$$

Solution



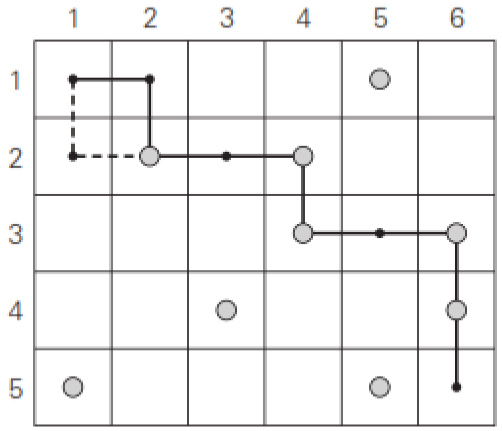
(a)



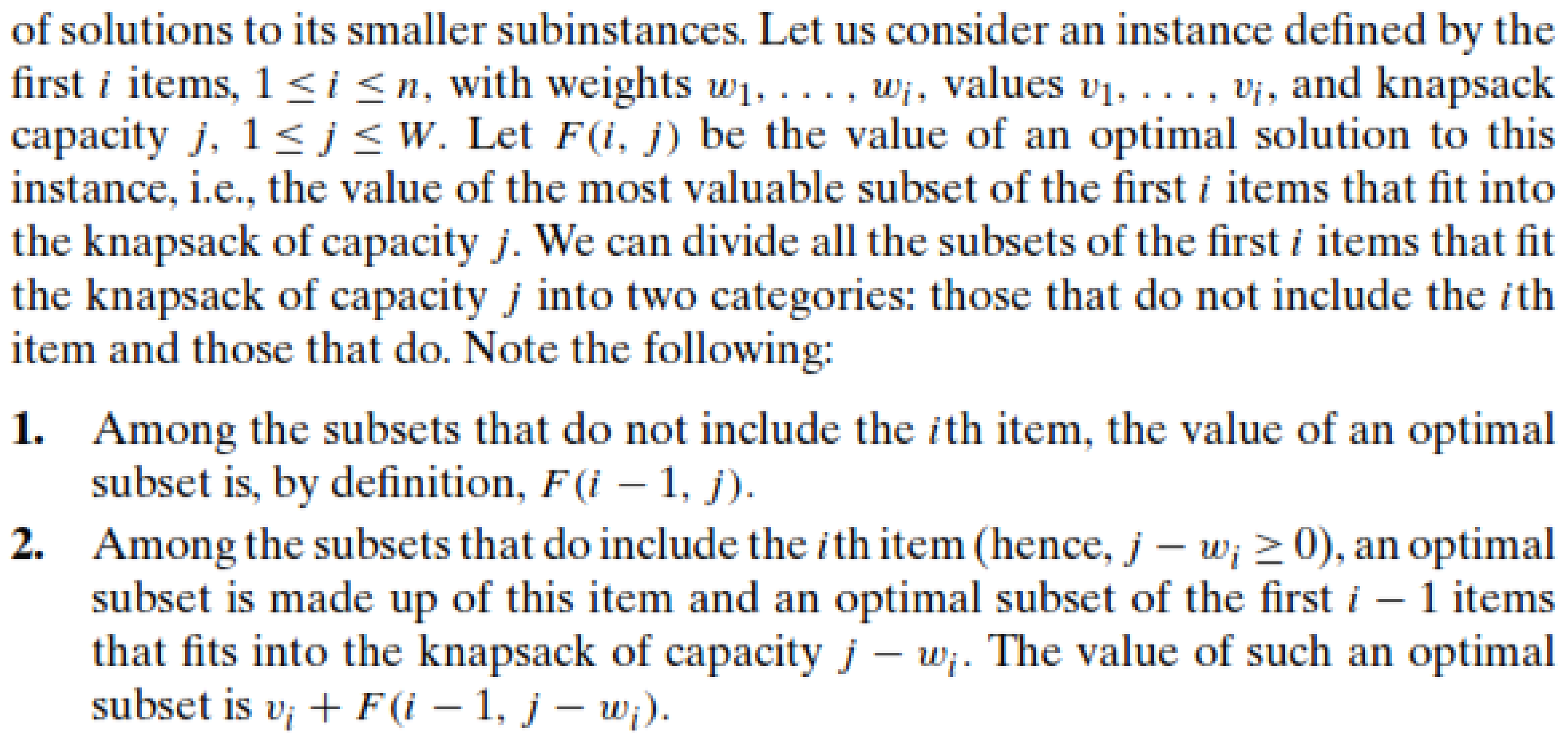
	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

(b)

solution



Knapsack Problem



of solutions to its smaller subinstances. Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let $F(i, j)$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do. Note the following:

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $F(i-1, j)$.
2. Among the subsets that do include the ith item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.

Recurrence

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases} \quad \textbf{(8.6)}$$

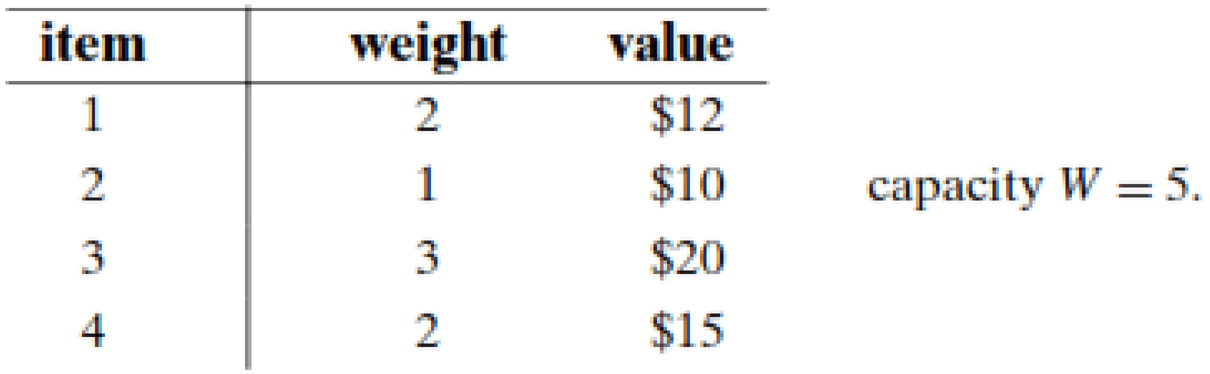
It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0. \quad \textbf{(8.7)}$$

Table

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

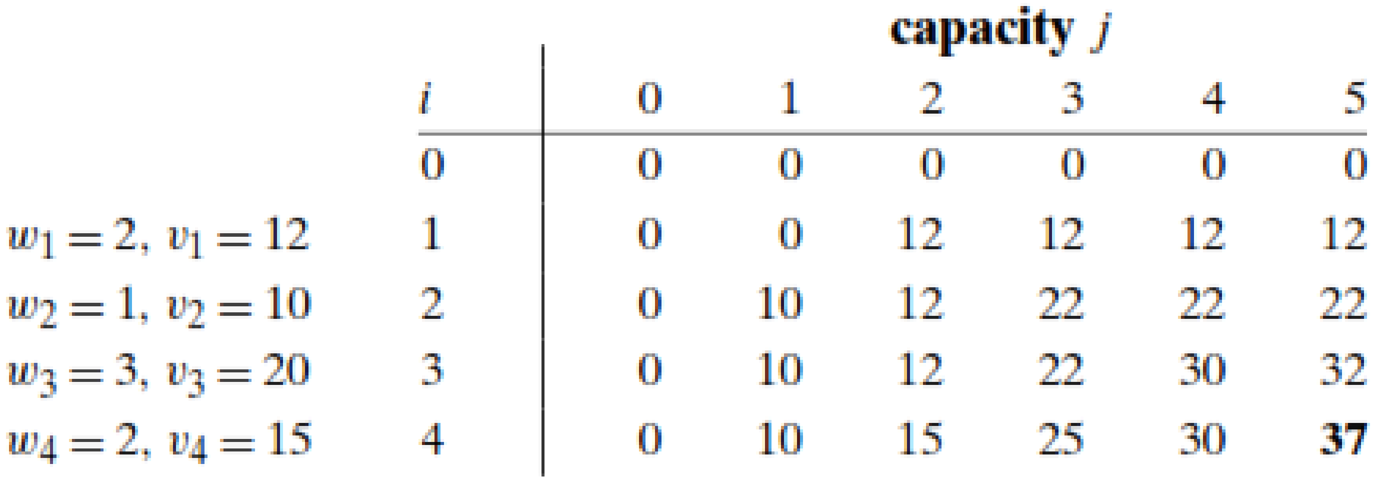
Example



item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

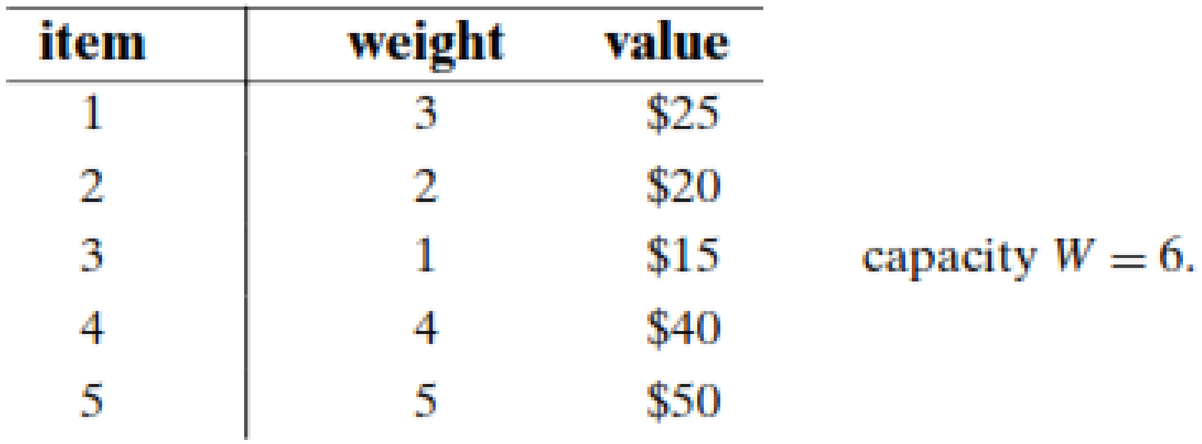
capacity $W = 5$.

Table



		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

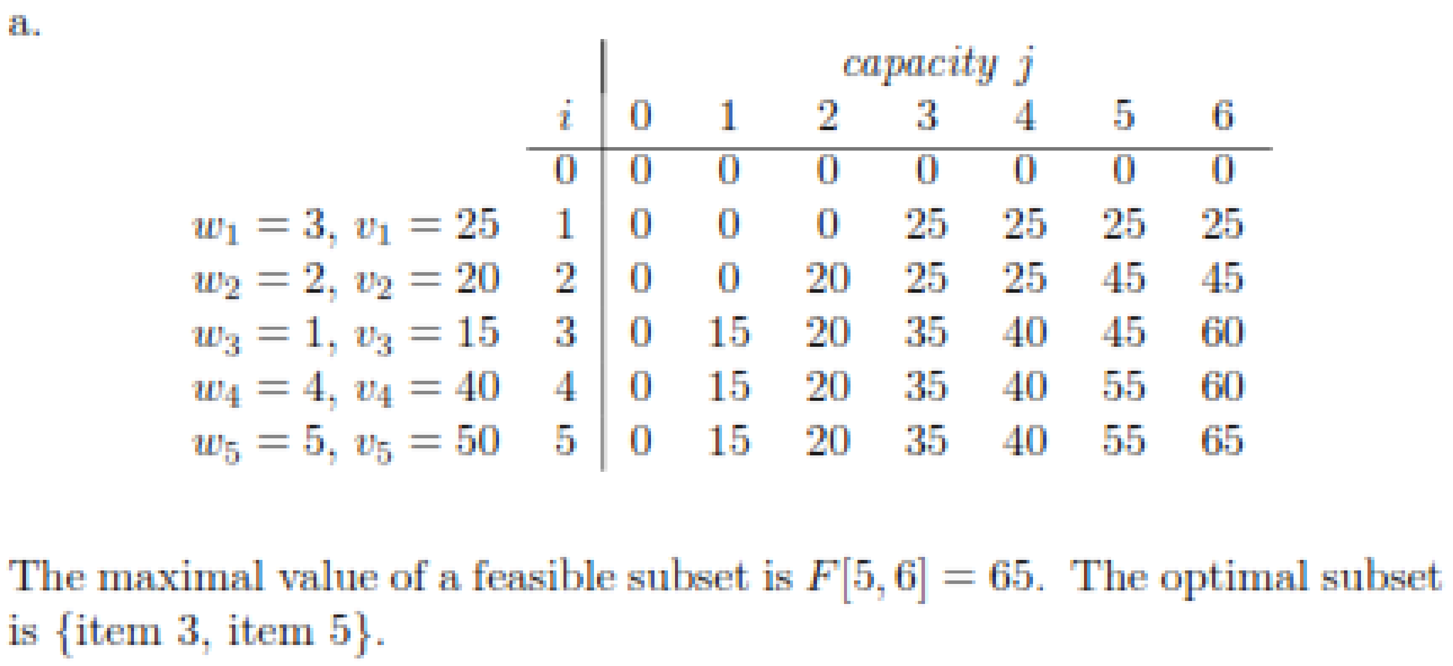
Example



item	weight	value
1	3	$25
2	2	$20
3	1	$15
4	4	$40
5	5	$50

capacity $W = 6$.

Solution



a.

		capacity j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3,\ v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2,\ v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1,\ v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4,\ v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5,\ v_5 = 50$	5	0	15	20	35	40	55	65

The maximal value of a feasible subset is $F[5,6] = 65$. The optimal subset is $\{$item 3, item 5$\}$.

Consider an instance of the knapsack problem with knapsack capacity $C = 6$ and four items:

Item	Value	Size
1	3	4
2	2	3
3	4	2
4	4	3

a. True or false: A sequence of values in a row of the dynamic programming table for the knapsack problem is always nondecreasing?

b. True or false: A sequence of values in a column of the dynamic programming table for the knapsack problem is always nondecreasing?

a. $F(i, j-1) \leq F(i, j)$ for $1 \leq j \leq W$ is true because it simply means that the maximal value of a subset that fits into a knapsack of capacity $j-1$ cannot exceed the maximal value of a subset that fits into a knapsack of capacity j.

b. $F(i-1, j) \leq F(i, j)$ for $1 \leq i \leq n$ is true because it simply means that the maximal value of a subset of the first $i-1$ items that fits into a knapsack of capacity j cannot exceed the maximal value of a subset of the first i items that fits into a knapsack of the same capacity j.

Item	Value	Size
1	1	1
2	2	3
3	3	2
4	4	5
5	5	4

ity $C = 9$. What are the fin