Part III: Dynamic Programming

Course: Design and Analysis of Algorithms
by Dr. Partha Basuchowdhuri

Department of Computer Science and Engineering, Heritage Institute of Technology, Kolkata, India
Phone:(+91)9163883328, E-mail: parthabasu.chowdhuri@heritageit.edu

March 1, 2017

Outline for Part I

Why do we need dynamic programming?

Why do we need dynamic programming?

Solving optimization problems, where finding solutions to the subproblems lead to the solution of the problem.

Why do we need dynamic programming?

Solving optimization problems, where finding solutions to the subproblems lead to the solution of the problem.

Commonly uses additional storage to record intermediate results to avoid repeated calculations.

Why do we need dynamic programming?

Solving optimization problems, where finding solutions to the subproblems lead to the solution of the problem.

Commonly uses additional storage to record intermediate results to avoid repeated calculations.

Primary steps of solving a problem with dynamic programming are -

- Characterize the structure of an optimal solution.

Why do we need dynamic programming?

Solving optimization problems, where finding solutions to the subproblems lead to the solution of the problem.

Commonly uses additional storage to record intermediate results to avoid repeated calculations.

Primary steps of solving a problem with dynamic programming are -

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.

Why do we need dynamic programming?

Solving optimization problems, where finding solutions to the subproblems lead to the solution of the problem.

Commonly uses additional storage to record intermediate results to avoid repeated calculations.

Primary steps of solving a problem with dynamic programming are -

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.

Why do we need dynamic programming?

Solving optimization problems, where finding solutions to the subproblems lead to the solution of the problem.

Commonly uses additional storage to record intermediate results to avoid repeated calculations.

Primary steps of solving a problem with dynamic programming are -

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from computed information.

Outline for Part II

Matrix Chain Multiplication

We are given *n* matrices in a sequence (say, A_1, A_2, . . . , A_n), which can be multiplied to generate the product.

Matrix Chain Multiplication

We are given *n* matrices in a sequence (say, A_1, A_2, ..., A_n), which can be multiplied to generate the product.

Due to associativity, irrespective of the order in which the matrices are multiplied, the product would be same, but ...

Matrix Chain Multiplication

We are given n matrices in a sequence (say, A_1, A_2, ..., A_n), which can be multiplied to generate the product.

Due to associativity, irrespective of the order in which the matrices are multiplied, the product would be same, but ... an interesting observation is that **the number scalar multiplications** involved in the process **can be minimized** by choosing the order of multiplication.

Matrix Chain Multiplication

We are given *n* matrices in a sequence (say, A_1, A_2, ..., A_n), which can be multiplied to generate the product.

Due to associativity, irrespective of the order in which the matrices are multiplied, the product would be same, but . . . an interesting observation is that **the number scalar multiplications** involved in the process **can be minimized** by choosing the order of multiplication.

For example, if we have a sequence of matrices $A_1A_2A_3$ with dimensions 10 $\times$ 100, 100 $\times$ 5 and 5 $\times$ 50, there can be two possible orders of generating the product - $((A_1A_2)A_3)$ and $(A_1(A_2A_3))$.

Matrix Chain Multiplication

We are given *n* matrices in a sequence (say, A_1, A_2, ..., A_n), which can be multiplied to generate the product.

Due to associativity, irrespective of the order in which the matrices are multiplied, the product would be same, but . . . an interesting observation is that **the number scalar multiplications** involved in the process **can be minimized** by choosing the order of multiplication.

For example, if we have a sequence of matrices $A_1A_2A_3$ with dimensions 10×100, 100×5 and 5×50, there can be two possible orders of generating the product - $((A_1A_2)A_3)$ and $(A_1(A_2A_3))$.

In $((A_1A_2)A_3)$, we perform $10 \times 100 \times 5 + 10 \times 5 \times 50 =$ 7,500 multiplications. (Preferred)
In $(A_1(A_2A_3))$, we perform $100 \times 5 \times 50 + 10 \times 100 \times 50 =$ 75,000 multiplications.

Step 1: Structure of an optimal parenthesization

Non-trivial to parenthesize $A_i A_{i+1} \ldots A_j$ when $i<j$.

Step 1: Structure of an optimal parenthesization

Non-trivial to parenthesize $A_i A_{i+1} \ldots A_j$ when $i<j$.

We choose a k, such that $i \leq k < j$, k divides the matrix chain in order to minimize scalar multiplications during generating the product.

Step 1: Structure of an optimal parenthesization

Non-trivial to parenthesize $A_i A_{i+1} \ldots A_j$ when $i<j$.

We choose a k, such that $i \leq k < j$, k divides the matrix chain in order to minimize scalar multiplications during generating the product.

The first split for optimal parenthesization for $A_i A_{i+1} \ldots A_j$ will be $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$.

Step 1: Structure of an optimal parenthesization

Non-trivial to parenthesize $A_i A_{i+1} \ldots A_j$ when $i<j$.

We choose a k, such that $i \leq k < j$, k divides the matrix chain in order to minimize scalar multiplications during generating the product.

The first split for optimal parenthesization for $A_i A_{i+1} \ldots A_j$ will be $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$.

If there is an k' that splits $A_i A_{i+1} \ldots A_j$ for minimum number of multiplications, then we can say $k = k'$.

Step 1: Structure of an optimal parenthesization

Non-trivial to parenthesize $A_i A_{i+1} \ldots A_j$ when $i<j$.

We choose a k, such that $i \leq k < j$, k divides the matrix chain in order to minimize scalar multiplications during generating the product.

The first split for optimal parenthesization for $A_i A_{i+1} \ldots A_j$ will be $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$.

If there is an k' that splits $A_i A_{i+1} \ldots A_j$ for minimum number of multiplications, then we can say $k = k'$.

We can further divide the two optimal substructures to split them into smaller-sized optimal substructures until the substructures can be trivially split into optimal substructures (i.e., $i = j$).

Step 2: A recursive solution

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i A_{i+1} \ldots A_j$.

Step 2: A recursive solution

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i A_{i+1} \ldots A_j$.

$m[i,j]$ can be expressed as,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

Step 2: A recursive solution

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i A_{i+1} \ldots A_j$.

$m[i,j]$ can be expressed as,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

Recursive definition for the minimum cost of parenthesizing the product can be expressed as,

Step 2: A recursive solution

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i A_{i+1} \dots A_j$.

$m[i,j]$ can be expressed as,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

Recursive definition for the minimum cost of parenthesizing the product can be expressed as,

$$m[i,j] = \begin{cases} 0, \text{if } i = j \\ min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, \text{if } i < j \end{cases}$$

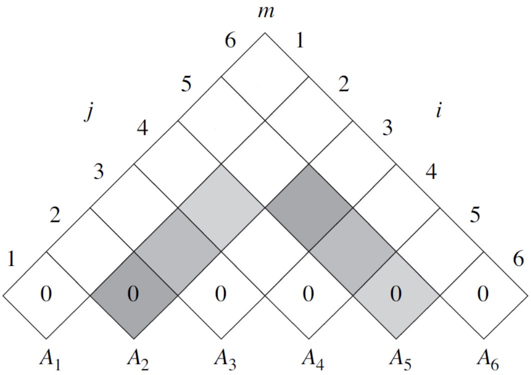
Step 3: Algorithm

Algorithm 1: MATRIX-CHAIN-ORDER(p))

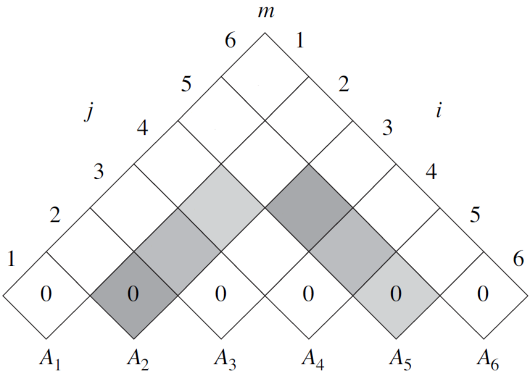
Input : Sequence of matrix dimensions *p*
Output: Matrices *m*, *s*

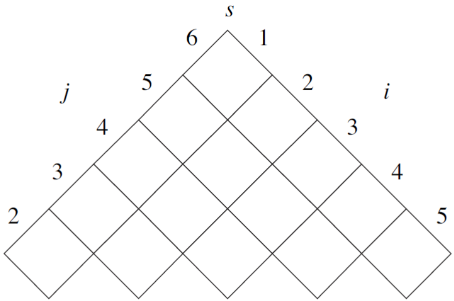
$n \leftarrow p.length$ - 1
Let $m[1 \dots n$, 1 $\dots n]$ and $s[1 \dots (n$-1), 2 $\dots n]$ be new tables
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
for $l = 2$ **to** n **do**
  for $i = 1$ **to** $n - l + 1$ **do**
    $j = i + l$-1
    $m[i,i] = \infty$
    for $k = i$ **to** j-1 **do**
      $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$
        $s[i,j] = k$

Find optimal parenthesization for $A_1A_2A_3A_4A_5A_6$, where $p = [30,35,15,5,10,20,25]$

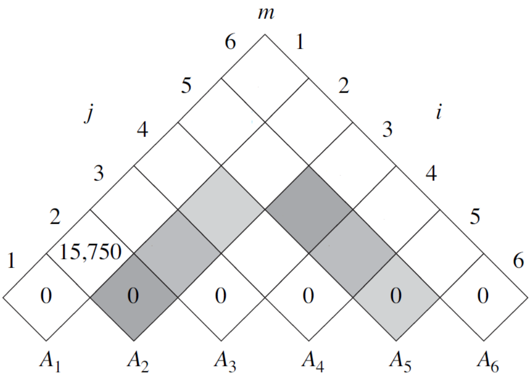


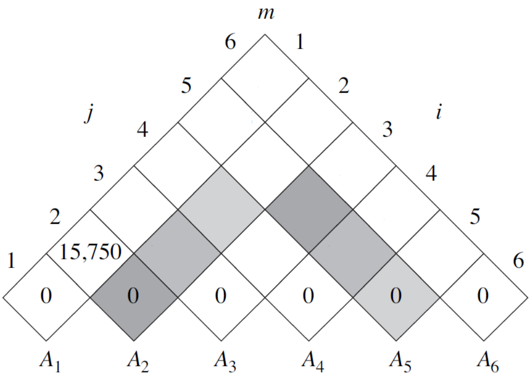
Find optimal parenthesization for $A_1A_2A_3A_4A_5A_6$, where $p = [30,35,15,5,10,20,25]$

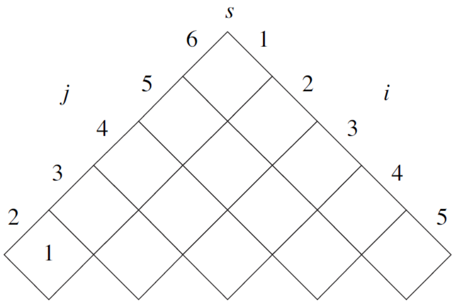




Initially, all the $m[i,i]$ values are set to zero.

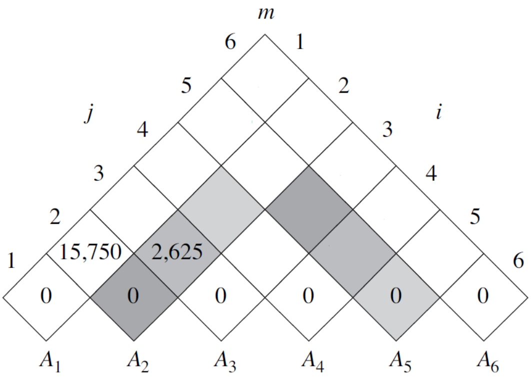

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
15,750
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

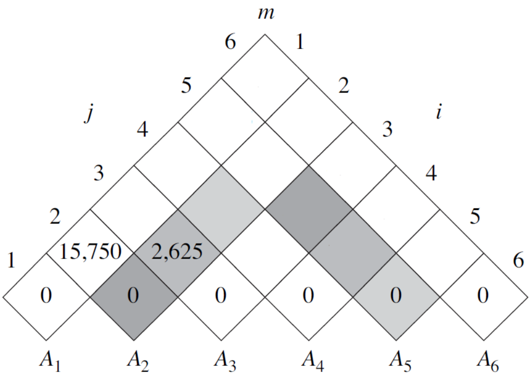


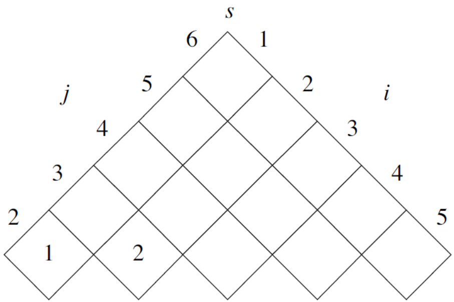


l=2, i=1, j=i+l-1=1+2-1=2

for k=1, m[1,2] = m[1,1] + m[2,2] + $p_0p_1p_2$ = 0 + 0 + 30 × 35 × 15 = 15,750

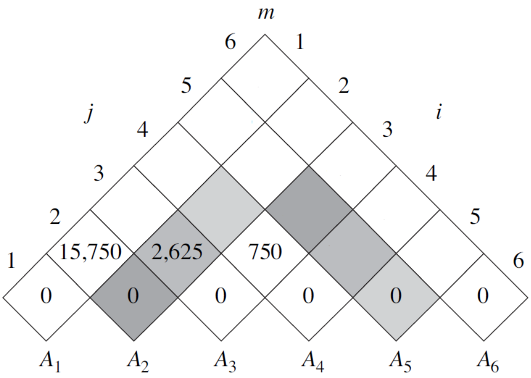

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
15,750
2,625
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

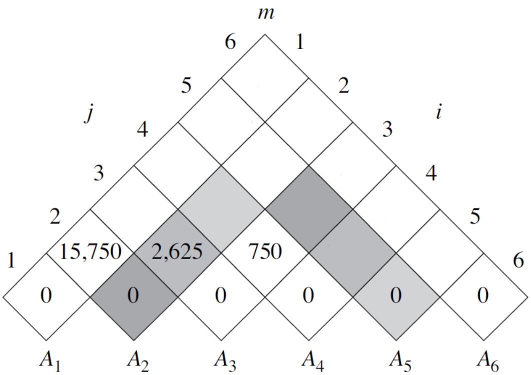


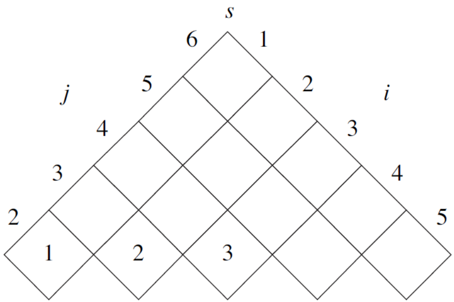


l=2, i=2, j=i+l-1=2+2-1=3

for k=2, m[2,3] = m[2,2] + m[3,3] + $p_1 p_2 p_3$ = 0 + 0 + 35 × 15 × 5 = 2,625

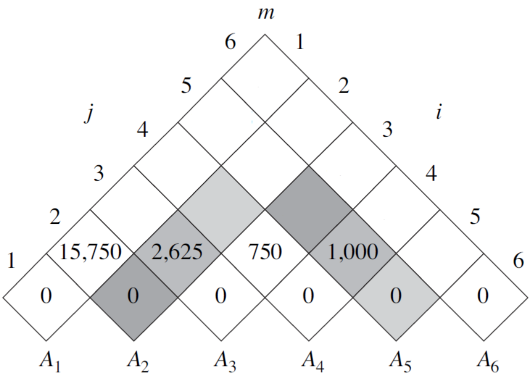

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
15,750
2,625
750
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

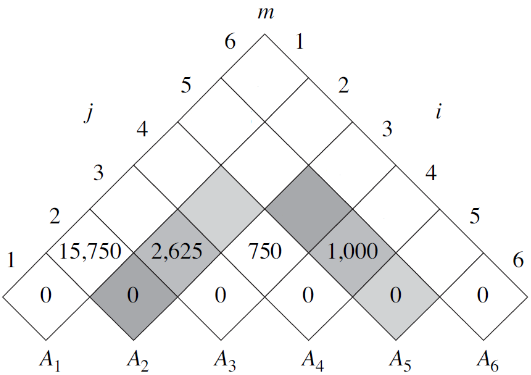


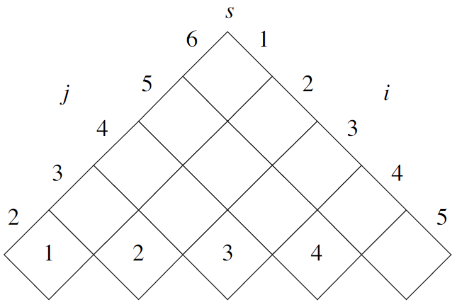


l=2, i=3, j=i+l-1=3+2-1=4

for k=3, m[3,4] = m[3,3] + m[4,4] + $p_2p_3p_4$ = 0 + 0 + 15 × 5 × 10 = 750

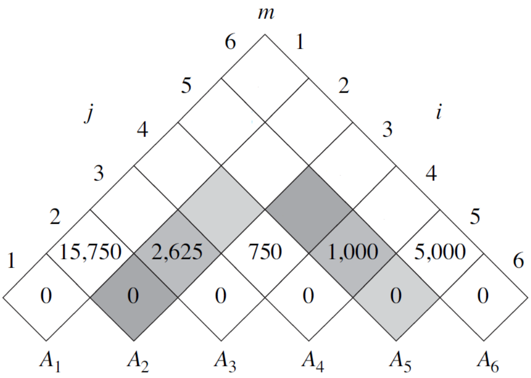

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
15,750
2,625
750
1,000
0
0
0
0
0
0
A1
A2
A3
A4
A5
A6

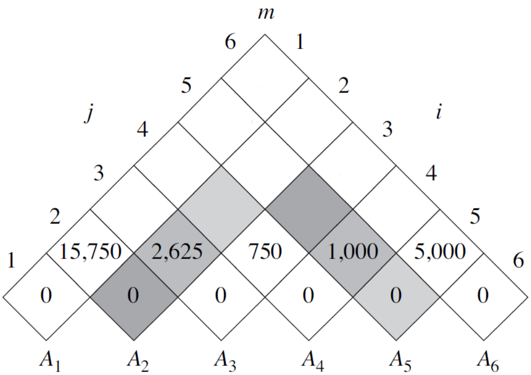


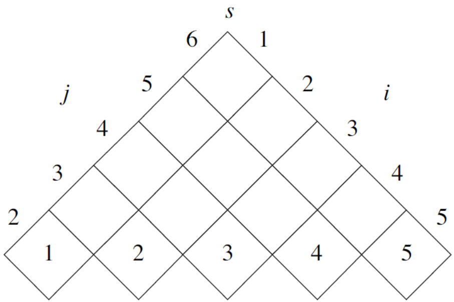


l=2, i=4, j=i+l-1=4+2-1=5

for k=4, m[4,5] = m[4,4] + m[5,5] + $p_3p_4p_5$ = 0 + 0 + 5 × 10 × 20 = 1,000

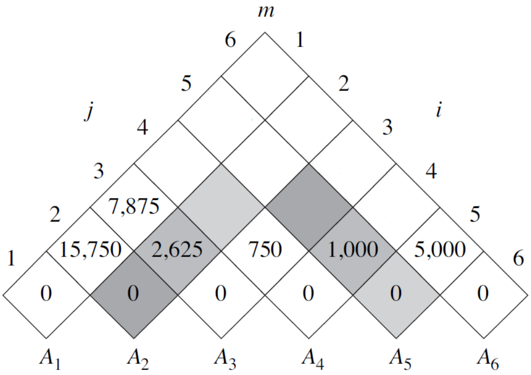

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

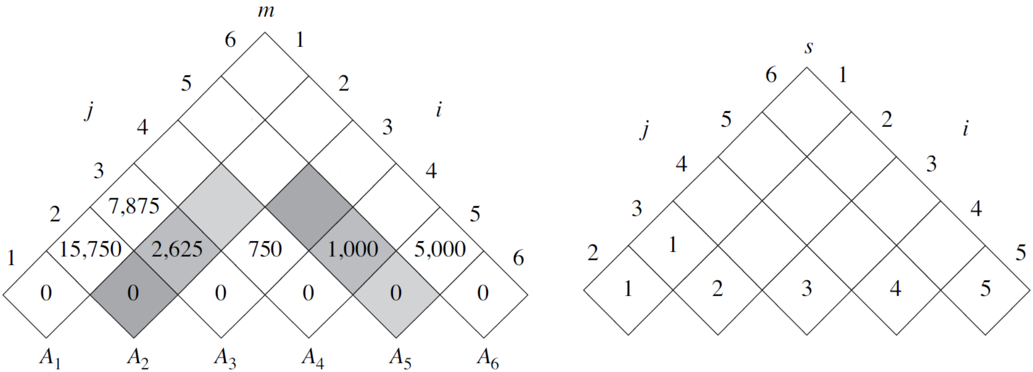




l=2, i=5, j=i+l-1=5+2-1=6

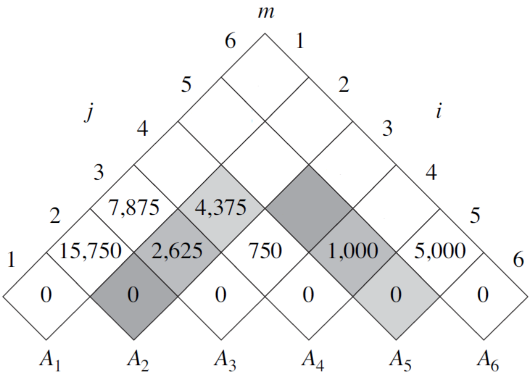
for k=5, m[5,6] = m[5,5] + m[6,6] + $p_4p_5p_6$ = 0 + 0 + 10 × 20 × 25 = 5,000

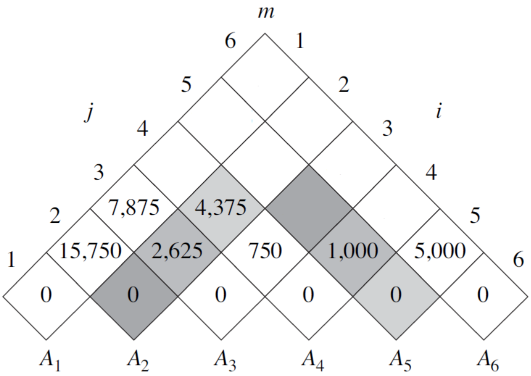

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
7,875
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A1
A2
A3
A4
A5
A6

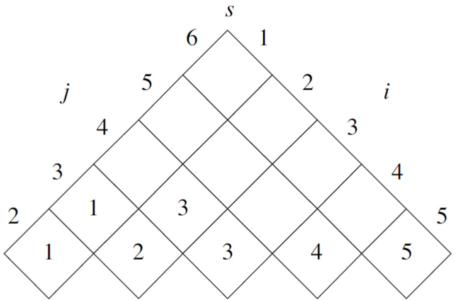


l=3, i=1, j=i+l-1=1+3-1=3

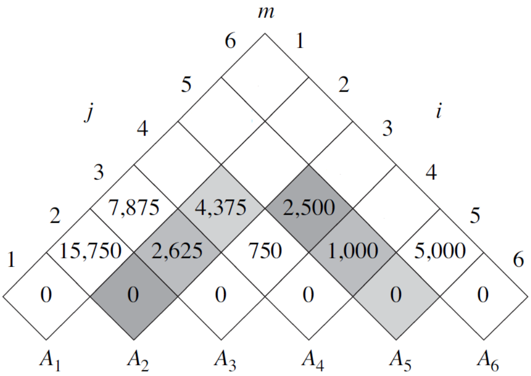
for k=1, m[1,3] = m[1,1] + m[2,3] + $p_0p_1p_3$ = 0 + 2625 + 30 × 35 × 5 = 7,875
for k=2, m[1,3] = m[1,2] + m[3,3] + $p_0p_2p_3$ = 15750 + 0 + 30 × 15 × 5 = 20,250

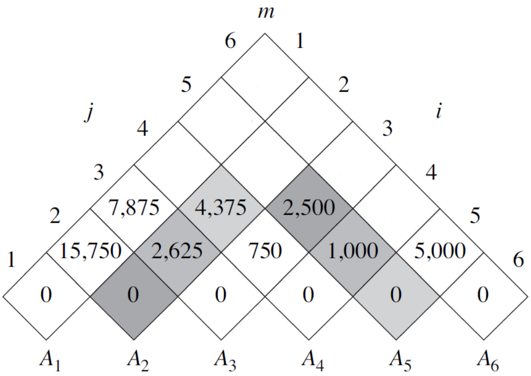

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
7,875
4,375
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

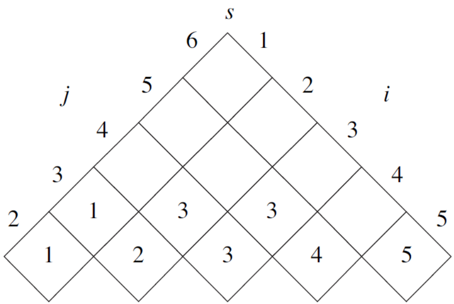




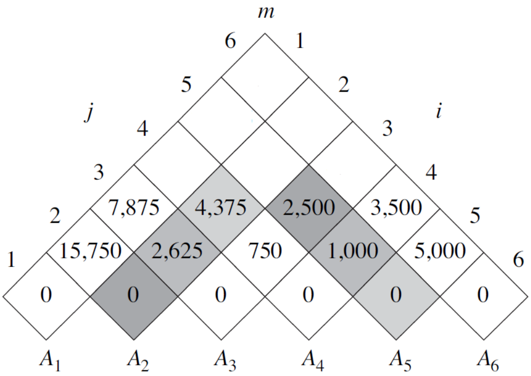
Do it yourself.

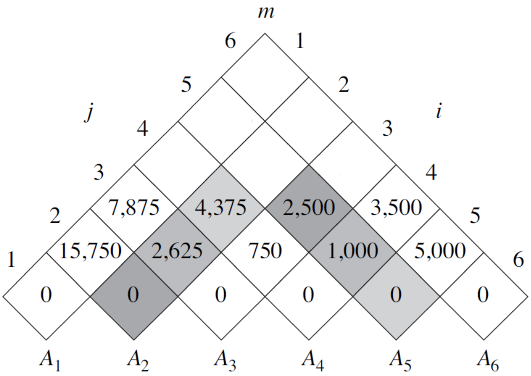

m
j
i
6
1
5
2
4
3
3
4
2
5
1
6
7,875
4,375
2,500
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

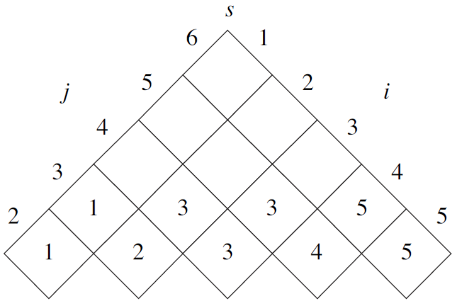




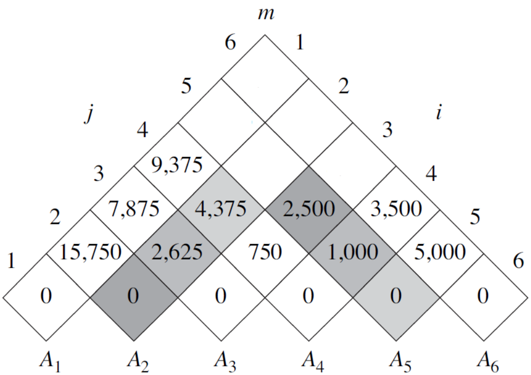
Do it yourself.

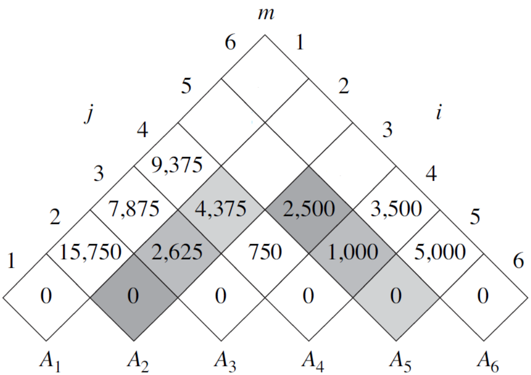

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
7,875
4,375
2,500
3,500
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

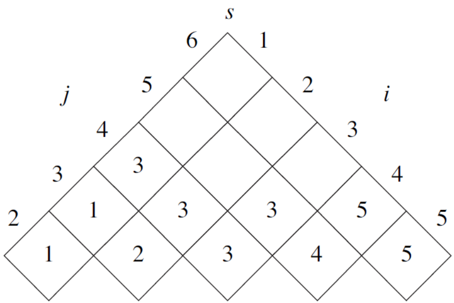




Do it yourself.


m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
9,375
7,875
4,375
2,500
3,500
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6



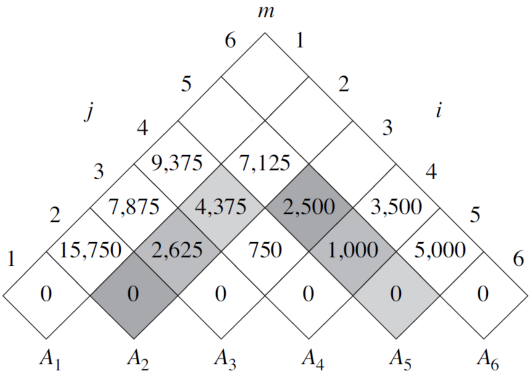


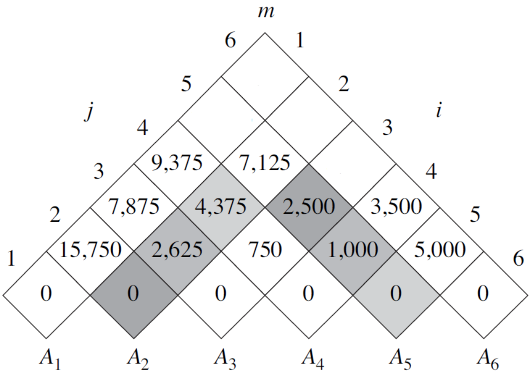
l=4, i=1, j=i+l-1=1+4-1=4

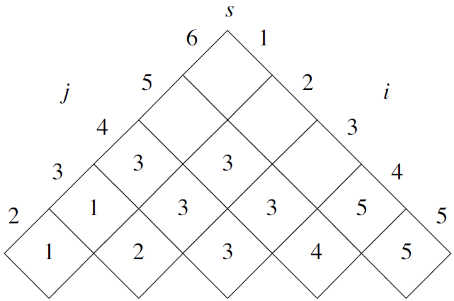
k=1, m[1,4] = m[1,1] + m[2,4] + $p_0p_1p_4$ = 0 + 4375 + 30 × 35 × 10 = 14,875

k=2, m[1,4] = m[1,2] + m[3,4] + $p_0p_2p_4$ = 15750 + 750 + 30 × 15 × 10 = 21,000

k=3, m[1,4] = m[1,3] + m[4,4] + $p_0p_3p_4$ = 7875 + 0 + 30 × 5 × 10 = 9,375


m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
9,375
7,125
7,875
4,375
2,500
3,500
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6



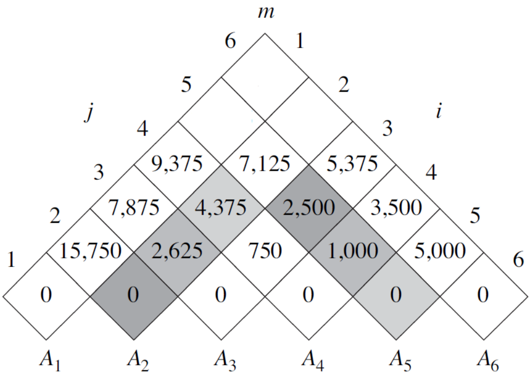


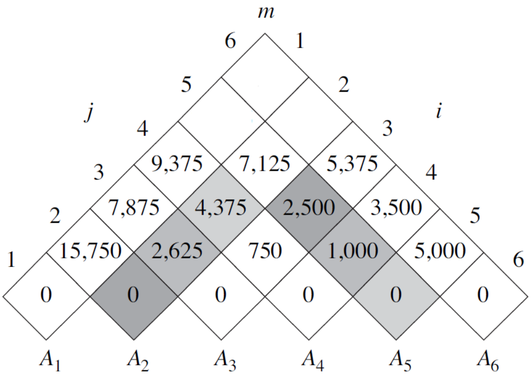
l=4, i=2, j=i+l-1=2+4-1=5

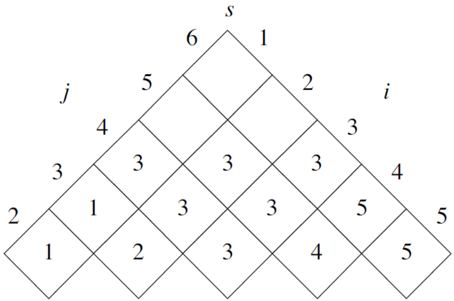
k=2, m[2,5] = m[2,2] + m[3,5] + $p_1p_2p_5$ = 0 + 2500 + 35 × 15 × 20 = 13,000

k=3, m[2,5] = m[2,3] + m[4,5] + $p_1p_3p_5$ = 2625 + 1000 + 35 × 5 × 20 = 7,125

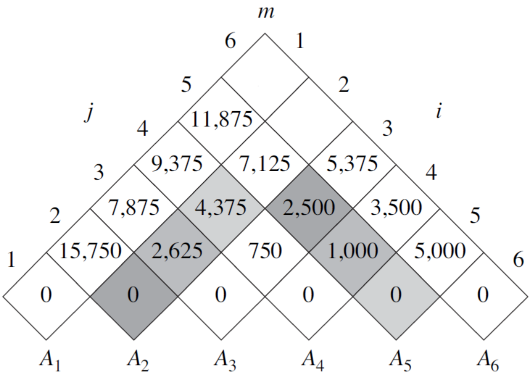
k=4, m[2,5] = m[2,4] + m[5,5] + $p_1p_4p_5$ = 4375 + 0 + 35 × 10 × 20 = 11,375

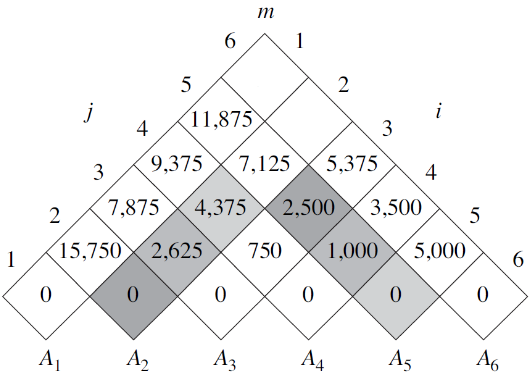

m
6
1
5
2
j
i
4
3
9,375
7,125
5,375
3
4
7,875
4,375
2,500
3,500
2
5
15,750
2,625
750
1,000
5,000
1
6
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

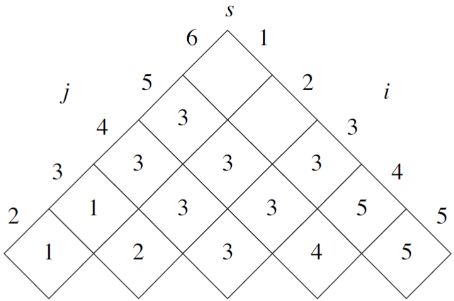




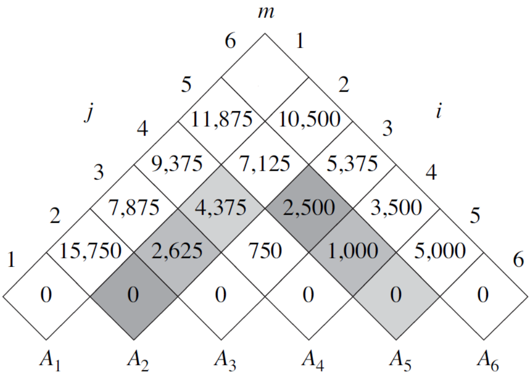
Do it yourself.

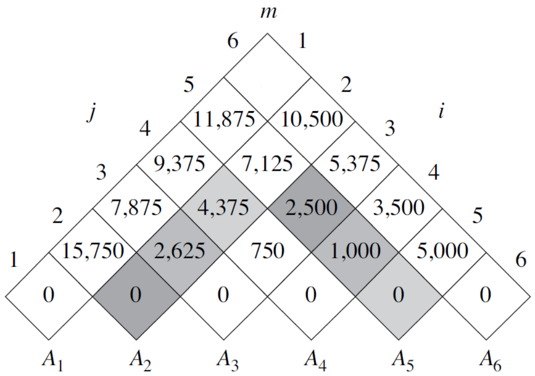

m
j
i
1
2
3
4
5
6
11,875
9,375
7,125
5,375
7,875
4,375
2,500
3,500
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A_1
A_2
A_3
A_4
A_5
A_6

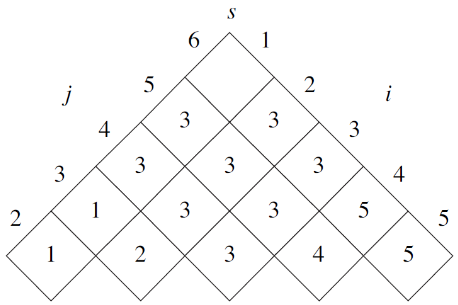




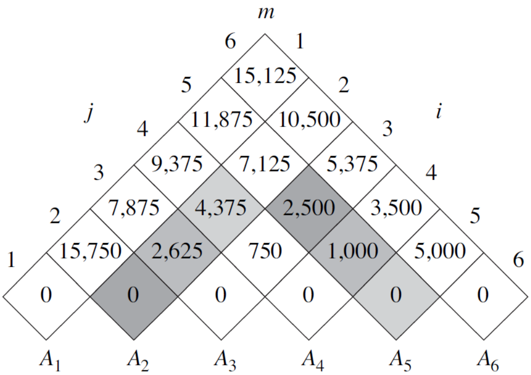
Do it yourself.

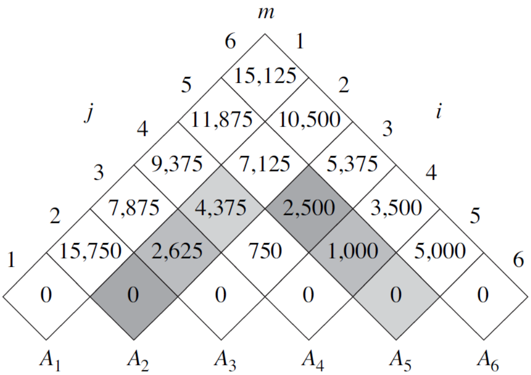

m
j
i
6
5
4
3
2
1
1
2
3
4
5
6
11,875
10,500
9,375
7,125
5,375
7,875
4,375
2,500
3,500
15,750
2,625
750
1,000
5,000
0
0
0
0
0
0
A1
A2
A3
A4
A5
A6

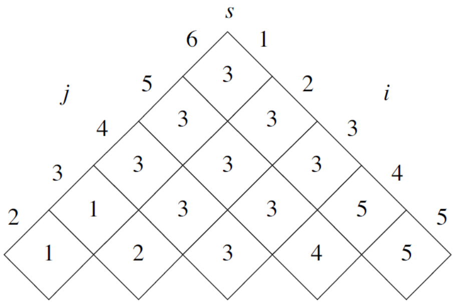




Do it yourself.


m
j
i
6 1
5 2
4 3
3 4
2 5
1 6
15,125
11,875 10,500
9,375 7,125 5,375
7,875 4,375 2,500 3,500
15,750 2,625 750 1,000 5,000
0 0 0 0 0 0
A1 A2 A3 A4 A5 A6





Do it yourself. $m[1,6] = 15{,}125$

Step 4: Constructing an optimal solution

Algorithm 2: PRINT-OPTIMAL-PARENS(s,i,j)

Input : Matrix *s*, Indices *i*,*j*
Output: Optimal parenthesization

```
begin
    if i == j then
        print "A"
    else
        print "("
        PRINT-OPTIMAL-PARENS(s,i,s[i,j])
        PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
        print ")"
```

The optimal parenthesization for the example can be expressed as,
$((A_1(A_2A_3))((A_4A_5)A_6))$

Outline for Part III

Knapsack Problem - Informal Representation

The Problem

Knapsack Problem - Informal Representation

The Problem

Warning: The problem is meant for a mathematically sound thief!

Knapsack Problem - Informal Representation

The Problem

Warning: The problem is meant for a mathematically sound thief!

- A thief enters a store with a bag of capacity W.
- There are n items in the store with weights $\{w_1, w_2, \ldots, w_i, \ldots, w_n\}$.
- Valuation of those n items can be represented by a set $\{v_1, v_2, \ldots, v_i, \ldots, v_n\}$.

Knapsack Problem - Informal Representation

The Problem

Warning: The problem is meant for a mathematically sound thief!

- A thief enters a store with a bag of capacity W.
- There are n items in the store with weights $\{w_1, w_2, \ldots, w_i, \ldots, w_n\}$.
- Valuation of those n items can be represented by a set $\{v_1, v_2, \ldots, v_i, \ldots, v_n\}$.

Two variants of the problem -

1. **0-1 Knapsack**: Items can be accrued only in whole. Either you take an item or you don't. Example: Where items are indivisible and partial items are valueless.
2. **Fractional Knapsack**: Items can be accrued in part as well. You can take as much of an item you want to fill your bag. Example: Consider containers consisting of valuable metals such as gold, silver, copper, etc.

0-1 Knapsack Problem - Informal Representation

Camera
Weight: 1 kg
Value: 1000$

Laptop
Weight: 3 kg
Value: 2000$

Necklace
Weight: 4 kg
Value: 4000$

Vase
Weight: 5 kg
Value: 4500$

Knapsack
Capacity: 7 kg
Max value: ???

0-1 Knapsack Problem - Informal Representation

Capacity of the bag is 7 kgs.

Item 1 Camera: $w_1 = 1$ kg, $v_1 = 60$K

Item 2 Laptop: $w_2 = 3$ kg, $v_2 = 150$K

Item 3 Jewellery: $w_3 = 4$ kg, $v_3 = 300$K

Item 4 Collectible: $w_4 = 5$ kg, $v_4 = 400$K

0-1 Knapsack Problem - Formal Representation

Problem Definition

Given a knapsack with maximum capacity W, and a set S consisting of n items, where each item i has some weight w_i and benefit value b_i (all w_i, b_i and W are integer values). The objective is to find a $T \subseteq S$, such that

$$\sum_{i \in T} b_i \text{ is maximized, subject to } \sum_{i \in T} w_i \leq W.$$

0-1 Knapsack Problem - Formal Representation

Problem Definition

Given a knapsack with maximum capacity W, and a set S consisting of n items, where each item i has some weight w_i and benefit value b_i (all w_i, b_i and W are integer values). The objective is to find a $T \subseteq S$, such that

$\sum_{i \in T} b_i$ is maximized, subject to $\sum_{i \in T} w_i \leq W$.

How to solve 0-1 Knapsack

- Brute-force algorithm to find optimal solution.

0-1 Knapsack Problem - Formal Representation

Problem Definition

Given a knapsack with maximum capacity W, and a set S consisting of n items, where each item i has some weight w_i and benefit value b_i (all w_i, b_i and W are integer values). The objective is to find a $T \subseteq S$, such that

$$\sum_{i \in T} b_i \text{ is maximized, subject to } \sum_{i \in T} w_i \leq W.$$

How to solve 0-1 Knapsack

- Brute-force algorithm to find optimal solution.
- We go through all combinations and find the one with maximum value and with total weight less or equal to W. O(2^n)

0-1 Knapsack Problem - Formal Representation

Problem Definition

Given a knapsack with maximum capacity W, and a set S consisting of n items, where each item i has some weight w_i and benefit value b_i (all w_i, b_i and W are integer values). The objective is to find a $T \subseteq S$, such that

$$\sum_{i \in T} b_i \text{ is maximized, subject to } \sum_{i \in T} w_i \leq W.$$

How to solve 0-1 Knapsack

- Brute-force algorithm to find optimal solution.
- We go through all combinations and find the one with maximum value and with total weight less or equal to W. O(2^n)
- Can we find an easier solution applying dynamic programming ?

Item	w_i	b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

- If items are labeled 1, . . ., n, then a subproblem would be to find an optimal solution for $S_k = \{$items labeled 1, 2, . . . $k\}$
- The question is: can we describe the final solution(S_n) in terms of subproblems(S_k)?

Item	w_i	b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

- If items are labeled 1, . . ., n, then a subproblem would be to find an optimal solution for $S_k = \{$items labeled 1, 2, . . . $k\}$
- The question is: can we describe the final solution(S_n) in terms of subproblems(S_k)?
- $S_4 = \{1, 2, 3, 4\}$, $S_5 = \{1, 3, 4, 5\}$

Item	w_i	b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

- If items are labeled 1, . . ., n, then a subproblem would be to find an optimal solution for $S_k = \{$items labeled 1, 2, . . . $k\}$
- The question is: can we describe the final solution(S_n) in terms of subproblems(S_k)?
- $S_4 = \{1, 2, 3, 4\}$, $S_5 = \{1, 3, 4, 5\}$
- Solution S_4 is not a part of solution S_5. Therefore, the framing of the dynamic programming solution is incorrect.

- Let us 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].

- Let us 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].
- The recursive formula may look like this,

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

- Let us 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].
- The recursive formula may look like this,

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

- It means, that the best subset of S_k that has total weight w is,
  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

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

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

- The best subset of S_k that has the total weight w, either contains item k or not.

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

- 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 cannot be part of the solution, since if it was, the total weight would be $>$ w, which is unacceptable.

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

- 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 cannot be part of the solution, since if it was, the total weight would be $>$ w, which is unacceptable.
- **Second case:** $w_k \leq w$. Then the item k can be in the solution, and we choose the case with greater value.

0-1 Knapsack Problem - Algorithm

Algorithm 3: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[i-1, w − w_i] > B[i-1, w] then
                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]
```

- Time complexity: O(nW)
- Let us solve the problem for n = 4, W = 5
- Elements (weight, benefit) : {(2, 3), (3, 4), (4, 5), (5, 6)}

0-1 Knapsack Problem - Example

Algorithm 4: 01-KNAPSACK(W, $\{x_1,..., x_n\}$)

Input : W, $\{w_1,..., w_n\}$, $\{b_1,..., b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Solve the problem for n = 4, W = 5
- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$

w	0	1	2	3	4	5
i_0						
i_1						
i_2						
i_3						
i_4						

0-1 Knapsack Problem - Example

Algorithm 5: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Solve the problem for n = 4, W = 5
- Elements (weight, benefit) : {(2, 3), (3, 4), (4, 5), (5, 6)}

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0					
i_2	0					
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 6: 01-KNAPSACK(W, $\{x_1,\ldots, x_n\}$)

Input : W, $\{w_1,\ldots, w_n\}$, $\{b_1,\ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 1$, $w_i = 2$, $b_i = 3$, $w = 1$, $w - w_i =$ -1

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0				
i_2	0					
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 7: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 1$, $w_i = 2$, $b_i = 3$, $w = 2$, $w - w_i = 0$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3			
i_2	0					
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 8: 01-KNAPSACK(W, $\{x_1,..., x_n\}$)

Input : W, $\{w_1,..., w_n\}$, $\{b_1,..., b_n\}$
Output: Maximum benefit (within W capacity)

for $w \leftarrow 0$ **to** W **do**
  B[0,w] = 0

for $i \leftarrow 1$ **to** n **do**
  B[i,0] = 0

for $i \leftarrow 1$ **to** n **do**
  for $w \leftarrow 0$ **to** W **do**
    if $w_i \leq w$ **then**
      if b_i+ B[i-1, $w - w_i$] > B[i-1, w] **then**
        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]

- Elements (weight, benefit) : {(2, 3), (3, 4), (4, 5), (5, 6)}
- $i = 1$, $w_i = 2$, $b_i = 3$, $w = 3$, $w - w_i = 1$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3		
i_2	0					
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 9: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 1$, $w_i = 2$, $b_i = 3$, $w = 4$, $w - w_i = 2$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	
i_2	0					
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 10: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 1$, $w_i = 2$, $b_i = 3$, $w = 5$, $w - w_i = 3$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0					
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 11: 01-KNAPSACK(W, $\{x_1,..., x_n\}$)

Input : W, $\{w_1,..., w_n\}$, $\{b_1,..., b_n\}$
Output: Maximum benefit (within W capacity)

for $w \leftarrow 0$ **to** W **do**
  B[0,w] = 0
for $i \leftarrow 1$ **to** n **do**
  B[i,0] = 0
for $i \leftarrow 1$ **to** n **do**
  for $w \leftarrow 0$ **to** W **do**
    if $w_i \leq w$ **then**
      if b_i+ B[i-1, $w - w_i$] > B[i-1, w] **then**
        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]

- Elements (weight, benefit) : {(2, 3), (3, 4), (4, 5), (5, 6)}
- $i = 2$, $w_i = 3$, $b_i = 4$, $w = 1$, $w - w_i =$ -2

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0				
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 12: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 2$, $w_i = 3$, $b_i = 4$, $w = 2$, $w - w_i =$ -1

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3			
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 13: 01-KNAPSACK(W, $\{x_1,\ldots,x_n\}$)

Input : W, $\{w_1,\ldots,w_n\}$, $\{b_1,\ldots,b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 2$, $w_i = 3$, $b_i = 4$, $w = 3$, $w - w_i = 0$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4		
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 14: 01-KNAPSACK(W, $\{x_1,\ldots, x_n\}$)

Input : W, $\{w_1,\ldots, w_n\}$, $\{b_1,\ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i+ B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 2$, $w_i = 3$, $b_i = 4$, $w = 4$, $w - w_i = 1$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 15: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 2$, $w_i = 3$, $b_i = 4$, $w = 5$, $w - w_i = 2$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0					
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 16: 01-KNAPSACK(W, $\{x_1,\ldots, x_n\}$)

Input : W, $\{w_1,\ldots, w_n\}$, $\{b_1,\ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 3$, $w_i = 4$, $b_i = 5$, $w = 1$, $w - w_i = -3$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0				
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 17: 01-KNAPSACK(W, $\{x_1,..., x_n\}$)

Input : W, $\{w_1,..., w_n\}$, $\{b_1,..., b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 3$, $w_i = 4$, $b_i = 5$, $w = 2$, $w - w_i =$ -2

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3			
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 18: 01-KNAPSACK(W, $\{x_1,..., x_n\}$)

Input : W, $\{w_1,..., w_n\}$, $\{b_1,..., b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 3$, $w_i = 4$, $b_i = 5$, $w = 3$, $w - w_i =$ -1

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4		
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 19: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 3$, $w_i = 4$, $b_i = 5$, $w = 4$, $w - w_i = 0$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 20: 01-KNAPSACK(W, $\{x_1,\ldots, x_n\}$)

Input : W, $\{w_1,\ldots, w_n\}$, $\{b_1,\ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 3$, $w_i = 4$, $b_i = 5$, $w = 5$, $w - w_i = 1$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0					

0-1 Knapsack Problem - Example

Algorithm 21: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 4$, $w_i = 5$, $b_i = 6$,
  $w = 1, 2, 3, 4,$
  $w - w_i =$ -4, -3, -2, -1

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	

0-1 Knapsack Problem - Example

Algorithm 22: 01-KNAPSACK(W, $\{x_1, \ldots, x_n\}$)

Input : W, $\{w_1, \ldots, w_n\}$, $\{b_1, \ldots, b_n\}$
Output: Maximum benefit (within W capacity)

```
for w ← 0 to W do
    B[0,w] = 0
for i ← 1 to n do
    B[i,0] = 0
for i ← 1 to n do
    for w ← 0 to W do
        if w_i ≤ w then
            if b_i + B[ i-1, w − w_i] > B[i-1, w] then
                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]
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 4$, $w_i = 5$, $b_i = 6$, $w = 5$, $w - w_i = 0$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	7

0-1 Knapsack Problem - Example (Finding the items)

Algorithm 23: 01-KSP-FIND-ITEMS(B)

Input : Table with benefit values (B)
Output: Items in the Knapsack

```
i = n, k = W
while i, k > 0 do
    if B[ i, k] ≠ B[i-1, k] then
        Mark i^th item as in Knapsack
        i = i - 1
        k = k - w_i
    else
        i = i - 1
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 4$, $k = 5$, $w_i = 5$, $b_i = 6$, $B[i,k] = 7$, $B[i-1,k] = 7$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	7

0-1 Knapsack Problem - Example (Finding the items)

Algorithm 24: 01-KSP-FIND-ITEMS(B)

Input : Table with benefit values (B)
Output: Items in the Knapsack

```
i = n, k = W
while i, k > 0 do
    if B[ i, k] ≠ B[i-1, k] then
        Mark i^th item as in Knapsack
        i = i - 1
        k = k - w_i
    else
        i = i - 1
```

- Elements (weight, benefit) : {(2, 3), (3, 4), (4, 5), (5, 6)}
- $i = 4$, $k = 5$, $w_i = 5$, $b_i = 6$, $B[i,k] = 7$, $B[i-1,k] = 7$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	7

0-1 Knapsack Problem - Example (Finding the items)

Algorithm 25: 01-KSP-FIND-ITEMS(B)

Input : Table with benefit values (B)
Output: Items in the Knapsack

```
i = n, k = W
while i, k > 0 do
    if B[ i, k] ≠ B[i-1, k] then
        Mark i^th item as in Knapsack
        i = i - 1
        k = k - w_i
    else
        i = i - 1
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 3$, $k = 5$, $w_i = 4$, $b_i = 5$, $B[i,k] = 7$, $B[i-1,k] = 7$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	7

0-1 Knapsack Problem - Example (Finding the items)

Algorithm 26: 01-KSP-FIND-ITEMS(B)

Input : Table with benefit values (B)
Output: Items in the Knapsack

```
i = n, k = W
while i, k > 0 do
    if B[ i, k] ≠ B[i-1, k] then
        Mark i^th item as in Knapsack
        i = i - 1
        k = k - w_i
    else
        i = i - 1
```

- Elements (weight, benefit) : $\{(2, 3), (3, 4), (4, 5), (5, 6)\}$
- $i = 2$, $k = 5$, $w_i = 3$, $b_i = 4$, $B[i,k] = 7$, $B[i-1,k] = 3$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	7

0-1 Knapsack Problem - Example (Finding the items)

Algorithm 27: 01-KSP-FIND-ITEMS(B)

Input : Table with benefit values (B)
Output: Items in the Knapsack

i = n, k = W
while $i, k > 0$ **do**
  if $B[\ i,\ k] \neq B[i\text{-}1,\ k]$ **then**
    Mark i^{th} item as in Knapsack
    $i = i$ - 1
    $k = k$ - w_i
  else
    $i = i$ - 1

- Elements (weight, benefit) : {(2, 3), (3, 4), (4, 5), (5, 6)}
- $i = 1$, $k = 2$, $w_i = 2$, $b_i = 3$, $B[i,k] = 3$, $B[i-1,k] = 0$

w	0	1	2	3	4	5
i_0	0	0	0	0	0	0
i_1	0	0	3	3	3	3
i_2	0	0	3	4	4	7
i_3	0	0	3	4	5	7
i_4	0	0	3	4	5	7

01-Knapsack Problem & Optimal Substructure

- Both solutions exhibit optimal substructure.
- To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds
  - If we remove item j from the load, what do we know about the remaining load?

01-Knapsack Problem & Optimal Substructure

- Both solutions exhibit optimal substructure.
- To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds
  - If we remove item j from the load, what do we know about the remaining load?
  - Answer: Remainder must be the most valuable load weighing at most W - w_j that thief could take, excluding item j.