Dynamic Programming

Dynamic Programming (DP)

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  – Partition the problem into independent subproblems
  – Solve the subproblems recursively
  – Combine the solutions to solve the original problem

DP vs. Divide and Conquire

- DP applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- A divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- A DP algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

More about DP

- We typically apply dynamic programming to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value

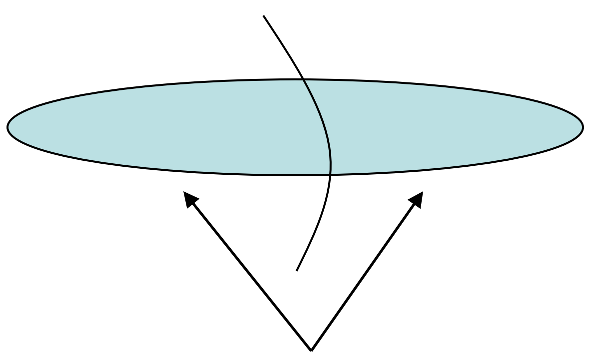
Steps taken in DP

- When developing a dynamic-programming algorithm, we follow a sequence of four steps:
  - 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.

DP - Two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic-programming solution:

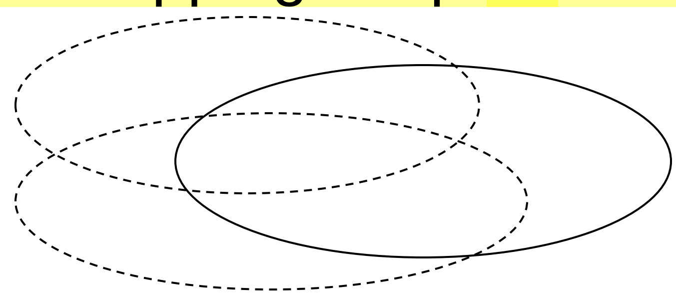
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Three basic components

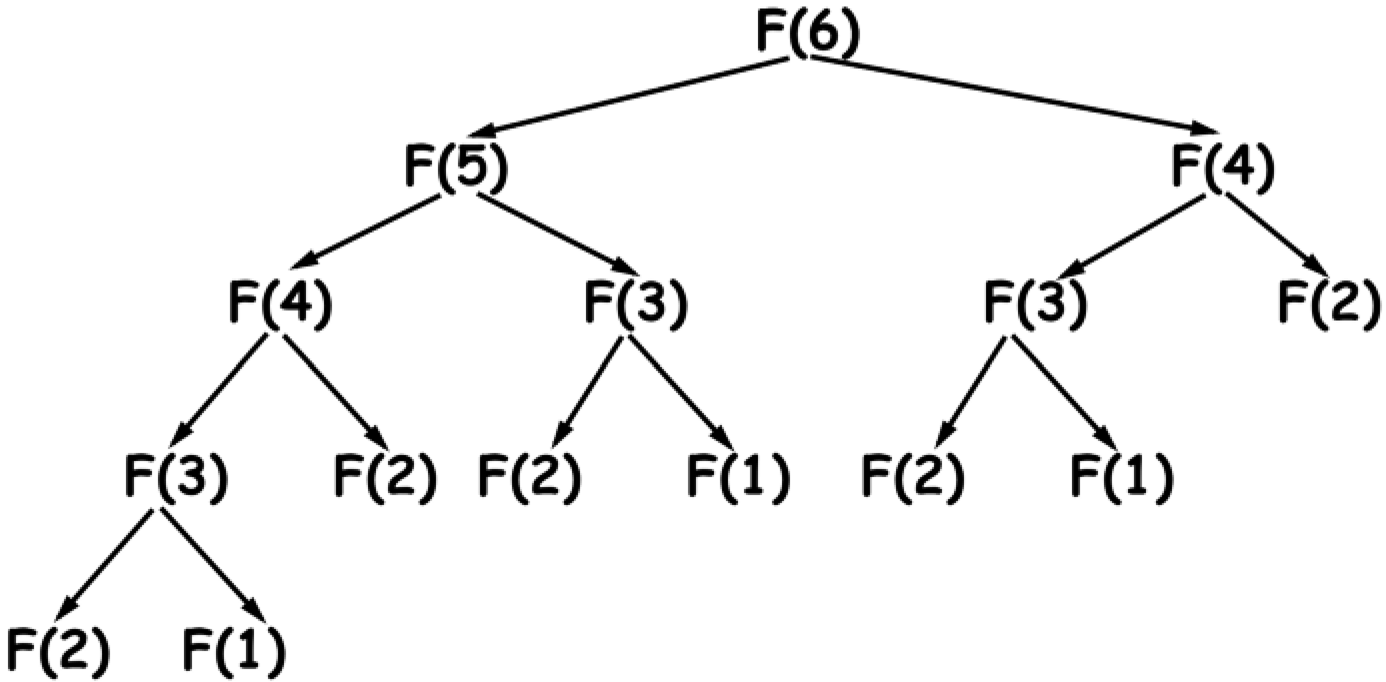
- The development of a dynamic-programming algorithm has three basic components:
  - The recurrence relation (for defining the value of an optimal solution);
  - The tabular computation (for computing the value of an optimal solution);
  - The traceback (for delivering an optimal solution).

Fibonacci numbers

The *Fibonacci numbers* are defined by the following recurrence:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i>1 .$$

How to compute F_{10}?



Dynamic Programming

- Applicable when subproblems are not independent
  - Subproblems share subsubproblems

E.g.: Fibonacci numbers:

  - Recurrence: $F(n) = F(n-1) + F(n-2)$
  - Boundary conditions: $F(1) = 0$, $F(2) = 1$
  - Compute: $F(5) = 3$, $F(3) = 1$, $F(4) = 2$

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Tabular computation

- The tabular computation can avoid recompuation.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Result

1. Solving Problems using DP

Sequences

Definition

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Definition

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if $1 \leq i_1 < \ldots < i_k \leq n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \ldots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \ldots \leq a_n$. Similarly **decreasing** and non-increasing.

Sequences

Example...

Example

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Subsequence: 5, 2, 1
- Increasing sequence: 3, 5, 9
- Increasing subsequence: 2, 7, 8

Longest increasing subsequence(LIS)

- The longest increasing subsequence is to find a longest increasing subsequence of a given sequence of distinct integers $a_1a_2\ldots a_n$.

e.g. 9 2 5 3 7 11 8 10 13 6

2 3 7

5 7 10 13

are increasing subsequences.

We want to find a longest one.

9 7 11

3 5 11 13

are not increasing subsequences.

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an *increasing subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
- Longest increasing subsequence: 3, 5, 7, 8

A naive approach for LIS

Assume $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ is contained in an array $\mathbf{A}$

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|

    Output max
```

A naive approach for LIS

Assume $a_1, a_2, \ldots, a_n$ is contained in an array A

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|

    Output max
```

Running time: $O(n2^n)$.
2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

A naive approach for LIS

- Let **L[i]** be the length of a longest increasing subsequence ending at position *i*.

$$L[i] = 1 + \max_{j = 0..i-1}\{L[j] \mid a_j < a_i\}$$

(use a dummy a_0 = minimum, and $L[0]=0$)

Index	0	1	2	3	4	5	6	7	8	9	10
Input	0	9	2	5	3	7	11	8	10	13	6
Length	0	1	1	2	2	3	4	4	5	6	3
Prev	-1	0	0	2	2	4	5	5	7	8	4
Path	1	1	1	1	1	2	2	2	2	2	2

The subsequence 2, 3, 7, 8, 10, 13 is a longest increasing subsequence.

This method runs in $O(n^2)$ time.

Algorithm

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]
          (* L[i] stores the value LIS-Ending(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i - 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Running time: $O(n^2)$

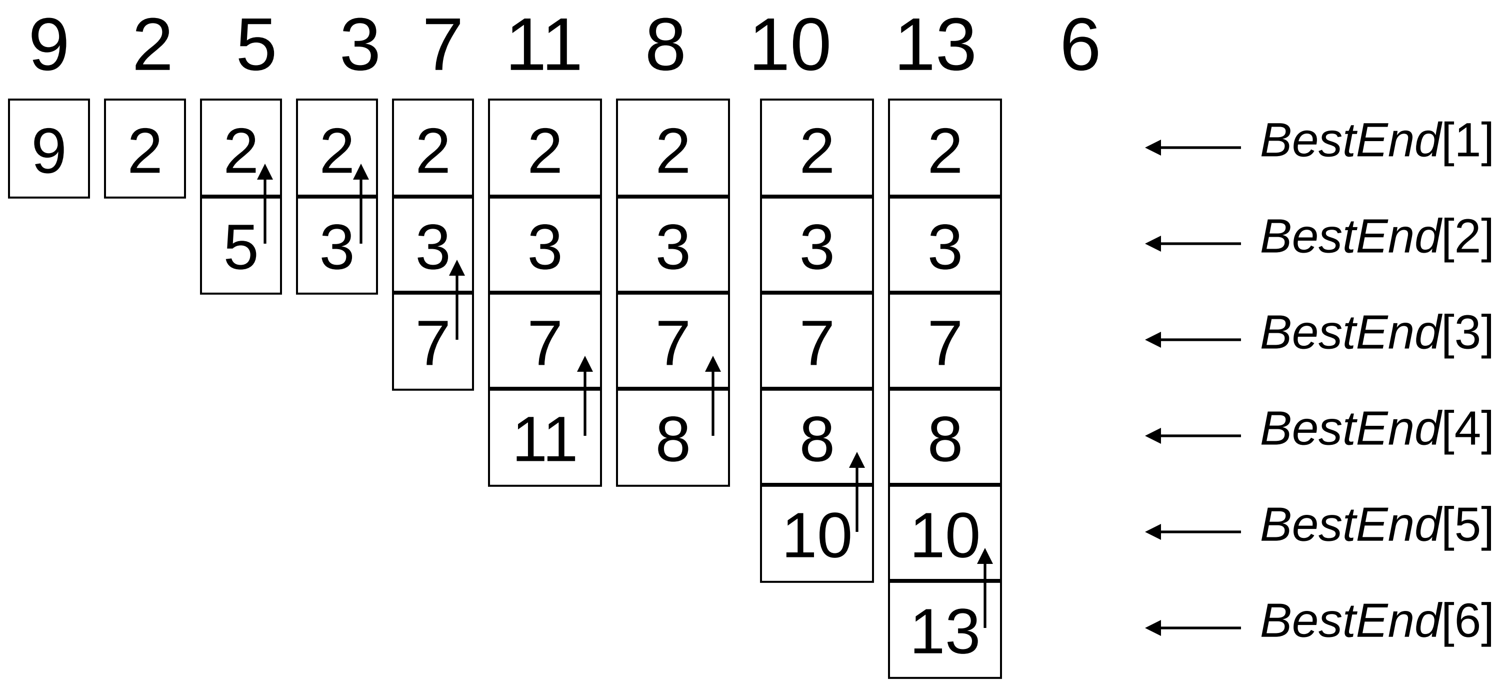
A better solution

One can compute LIS in $O(n \log n)$ time

TRY YOURSELF !!!!!!!

An $O(n \log n)$ method for LIS

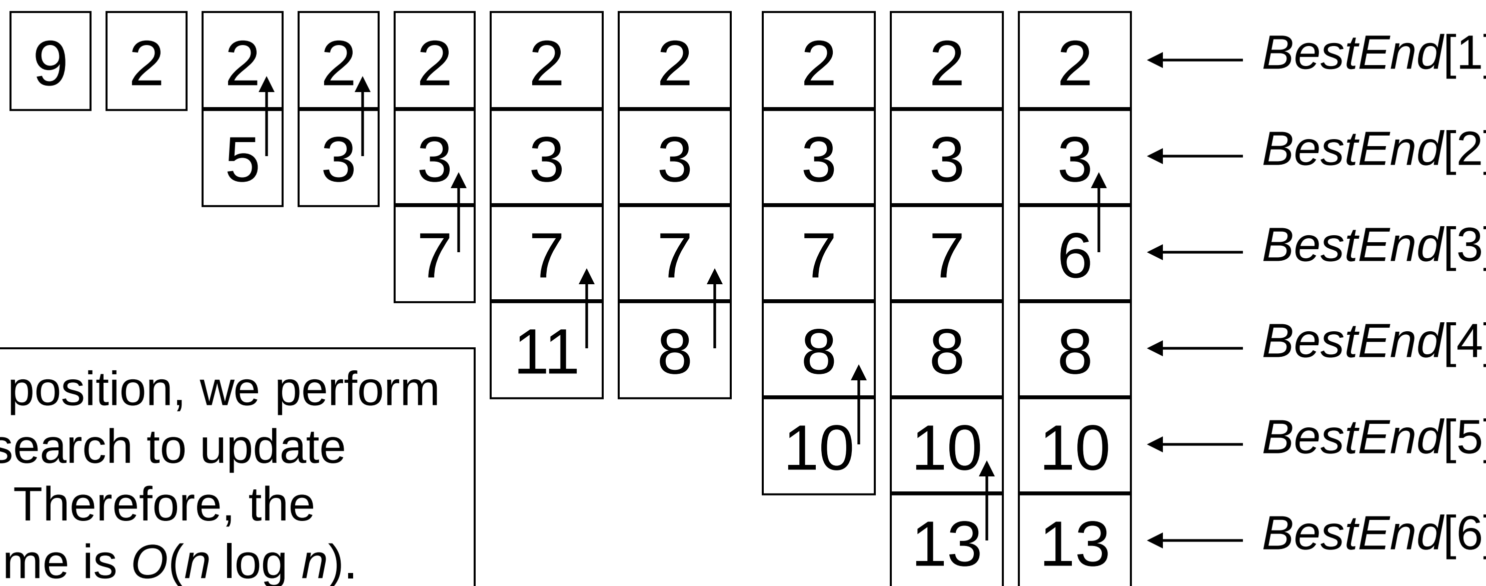
- Define *BestEnd*[*k*] to be the smallest number of an increasing subsequence of length *k*.



An $O(n \log n)$ method for LIS

- Define *BestEnd*[*k*] to be the smallest number of an increasing subsequence of length *k*.

9 2 5 3 7 11 8 10 13 6



For each position, we perform a binary search to update *BestEnd.* Therefore, the running time is $O(n \log n)$.

2. Sum of Subset Problem

- Problem:
  - Suppose you are given N positive integer numbers A[1…N] and it is required to produce another number K using a subset of A[1..N] numbers. How can it be done using Dynamic programming approach?
- Example:

  N = 6, A[1..N] = {2, 5, 8, 12, 6, 14}, K = 19

  Result: 2 + 5 + 12 = 19

Algorithm

- Use a two dimensional array

			w1	**w2**	**w3**	**…**	**ws**
Input[i]		0	1	2	3	…	s
	0						
2	**1**						
3	**2**						
7	**3**						
	.						
	.						
10	**n**						

j

i

Algorithm

- for (int i = 0; i <= n; i++)
  - s[0][i] = 1;

			w1	w2	w3	…	ws
Input[i]		0	1	2	3	…	s
	0	**1**					
2	**1**	**1**					
3	**2**	**1**					
7	**3**	**1**					
	.	**1**					
	.	**1**					
10	**n**	**1**					

j

i

Algorithm

- for (int j = 1; j <= s; j++)
  – s[j][0] = 0;

			w1	**w2**	**w3**	**…**	**ws**
Input[i]		0	1	2	3	…	s
	0	1	0	0	0	0	0
2	**1**	1					
3	**2**	1					
7	**3**	1					
	.	1					
	.	1					
10	**n**	1					

S

i

Algorithm

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

			w1	**w2**	**w3**	**…**	**ws**
Input[i]		0	1	2	3	…	s
	0	1	0	0	0	0	0
2	**1**	1					
3	**2**	1					
7	**3**	1					
	.	1					
	.	1					
10	**n**	1					

j

i

Example

$$
s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}
$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**												
2	**1**												
3	**2**												
7	**3**												
8	**4**												
10	**5**												

Input={2,3,7,8,10}
S=11

Example

$$
s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}
$$

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1											
3	**2**	1											
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

Example

$$s[i,j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0										
3	**2**	1											
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

Example

$$s[i,j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \mid\mid s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0										
3	**2**	1											
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

(1 || 0)=1

Example

$$s[i,j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \ || \ s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1									
3	**2**	1											
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

(1 || 0)=1

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1											
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0										
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1									
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1									
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

(1 || 0)=1

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1	1								
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

(1 || 0)=1

Example

$$s[i,j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1	1	0	1	0	0	0	0	0	0
7	**3**	1											
8	**4**	1											
10	**5**	1											

Input={2,3,7,8,10}
S=11

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1	1	0	1	0	0	0	0	0	0
7	**3**	1	0	1	1	0	1	0	1	0	1	1	0
8	**4**	1	0	1	1	0	1	0	1	1	1	1	1
10	**5**	1	0	1	1	0	1	0	1	1	1	1	1

Input={2,3,7,8,10}
S=11

Example

$$s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1	1	0	1	0	0	0	0	0	0
7	**3**	1	0	1	1	0	1	0	1	0	1	1	0
8	**4**	1	0	1	1	0	1	0	1	1	1	1	1
10	**5**	1	0	1	1	0	1	0	1	1	1	1	1

Input={2,3,7,8,10}
S=11

8,10

Example

$$
s[i, j] = \begin{cases} s[i-1, j] & \text{if } j < input[i] \\ s[i-1, j] \;||\; s[i-1, j-input[i]] & \text{otherwise} \end{cases}
$$

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
input	**0**	1	0	0	0	0	0	0	0	0	0	0	0
2	**1**	1	0	1	0	0	0	0	0	0	0	0	0
3	**2**	1	0	1	1	0	1	0	0	0	0	0	0
7	**3**	1	0	1	1	0	1	0	1	0	1	1	0
8	**4**	1	0	1	1	0	1	0	1	1	1	1	1
10	**5**	1	0	1	1	0	1	0	1	1	1	1	1

Input={2,3,7,8,10}
S=11

2,3, 8,10

Algorithm

```
for (int i = 0; i <= n; i++)
    s[0][i] = 1;
for (int j = 1; j <= s; j++)
    s[j][0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= s; j++)
      {
         if(j<input[i])
                  s[i][j]=s[i][j-1]
         else
                  s[i][j]=s[i][j-1] || s[i][j-input[i]]
      }
```

Coin Change Problem

- Suppose you are given ***n*** types of coin - C_1, C_2, … , C_n coin, and another number ***K.***
- Is it possible to make K using above types of coin?
  – Number of each coin is infinite
  – Number of each coin is finite
- Find minimum number of coin that is required to make ***K***?
  – Number of each coin is infinite
  – Number of each coin is finite

Maximum-sum interval

- Given a sequence of real numbers $a_1a_2\ldots a_n$, find a consecutive subsequence with the maximum sum.

9 −3 1 7 −15 2 3 −4 2 −7 6 −2 8 4 -9

For each position, we can compute the maximum-sum interval starting at that position in $O(n)$ time. Therefore, a naive algorithm runs in $O(n^2)$ time.

Try Yourself

The Knapsack Problem

- **The 0-1 knapsack problem**
  - A thief robbing a store finds n items: the i-th item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)
  - The thief can only carry W pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The fractional knapsack problem**
  - Similar to above
  - The thief can take fractions of items

The 0-1 Knapsack Problem

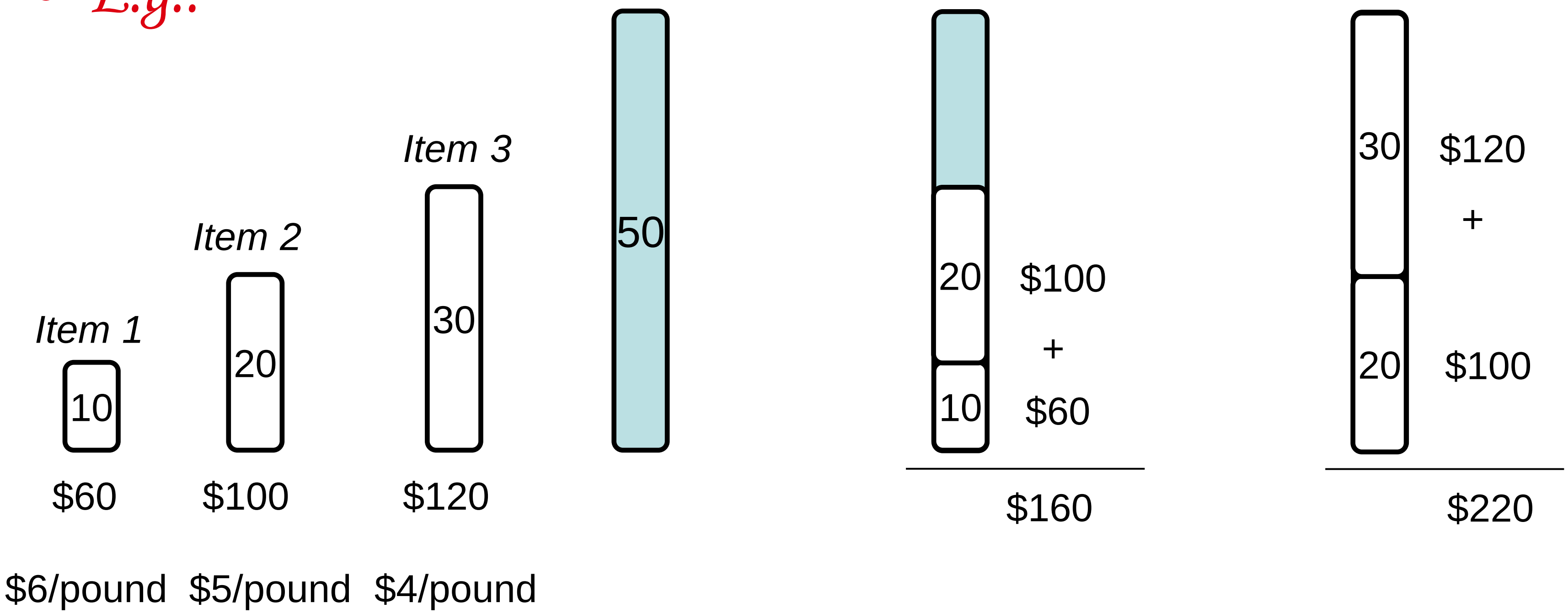
- Thief has a knapsack of capacity W
- There are n items: for i-th item value v_i and weight w_i
- Goal:
  - find x_i such that for all $x_i = \{0, 1\}$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

0-1 Knapsack - Greedy Strategy

- *E.g.:*



- None of the solutions involving the greedy choice (item 1) leads to an optimal solution
  - The greedy choice property does not hold

0-1 Knapsack - Dynamic Programming

- $P(i, w)$ – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

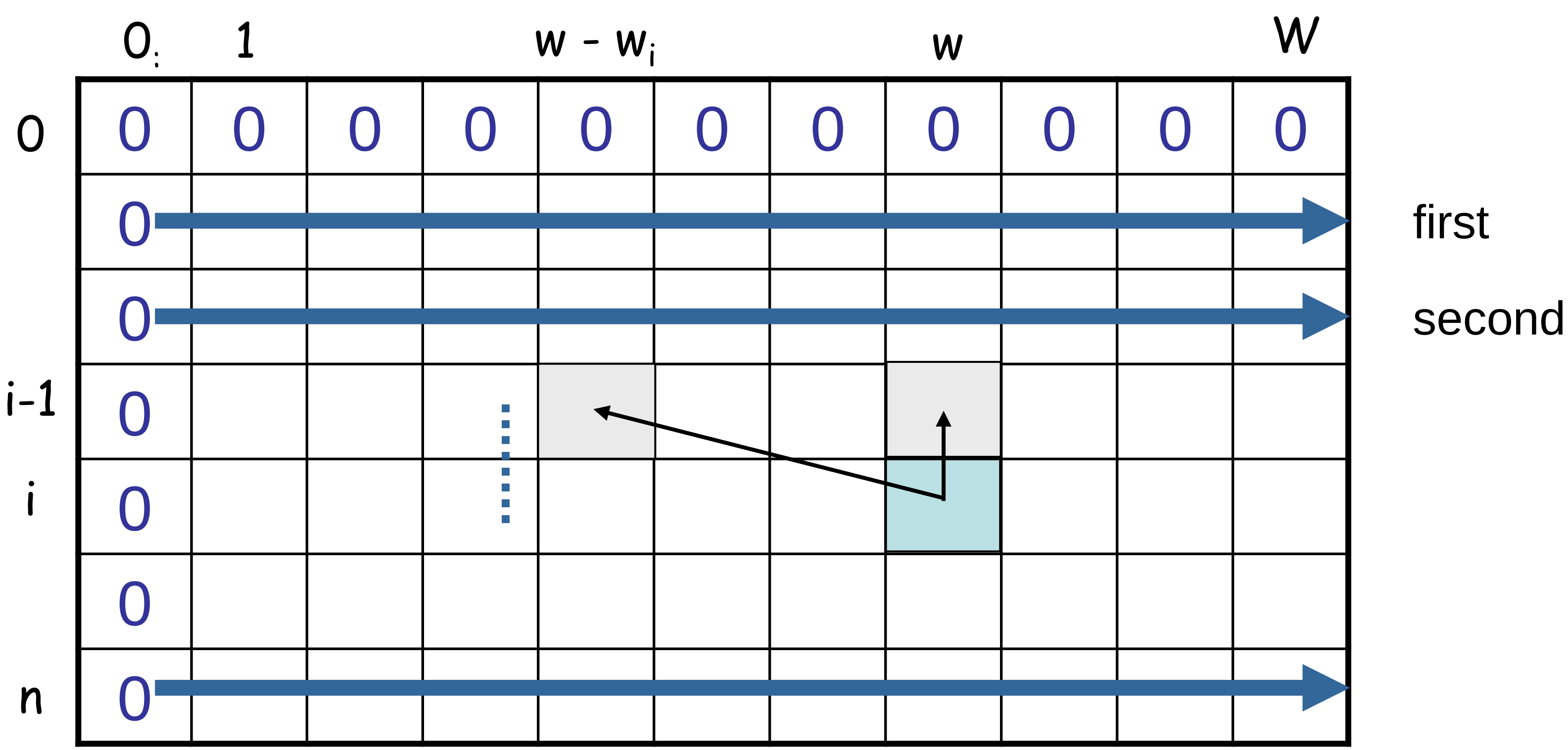
  $$P(i, w) = v_i + P(i - 1, w - w_i)$$

- Case 2: thief does not take item i

  $$P(i, w) = P(i - 1, w)$$

0-1 Knapsack - Dynamic Programming

$$P(i, w) = \max \{\overbrace{v_i + P(i - 1, w-w_i)}^{\text{Item } i \text{ was taken}}, \overbrace{P(i - 1, w)}^{\text{Item } i \text{ was not taken}} \}$$



Example:

W = 5

$$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(3, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at $P(n, W)$
- When you go left-up $\Rightarrow$ item i has been taken
- When you go straight up $\Rightarrow$ item i has not been taken

Overlapping Subproblems

$P(i, w) = \max \{v_i + P(i-1, w-w_i), P(i-1, w)\}$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

E.g.: all the subproblems shown in grey may depend on $P(i-1, w)$

Sudocode

```
main()
{
   int P[5]={0,1,2,5,6};
   int wt[5]={0,2,3,4,5};
   int m=8, n=4;
   int K[5][9];

  for(int i=0; i<=n; i++)
  {
    for(int w=0; w<=m; w++)
    {
        if(i==0 || w==0)
             K[i][w]=0;
        else if(wt[i] <= w)
             K[i][w]=max(P[i]+K[i-1][w-wt[i]],
                              K[i-1][w]);
        else K[i][w]=K[i-1][w];
    }
  }
  cout<<K[n][w];
```

Thanks All