Design & Analysis of Algorithms -I

Dynamic Programming

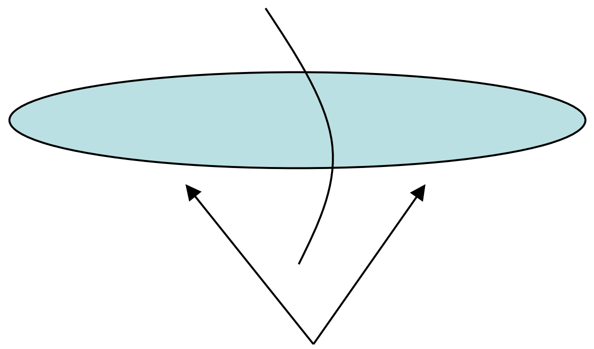
Dynamic Programming

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

DP - Two key ingredients

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

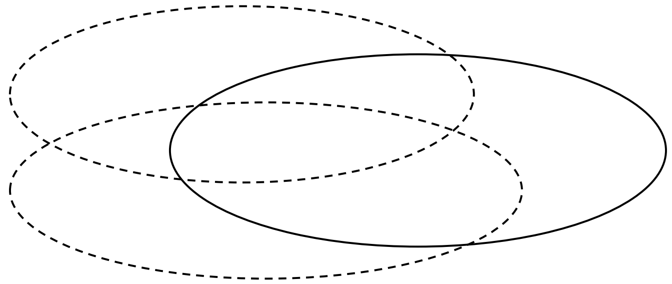
1. optimal substructures



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} ?

$$\begin{tikzcd}
 & & F_8 & \\
 & F_9 \arrow[ur, no head] \arrow[dr, no head] & & \\
 & & F_7 & \\
F_{10} \arrow[uur, no head] \arrow[ddr, no head] & & & \cdots\cdots \\
 & & F_7 & \\
 & F_8 \arrow[ur, no head] \arrow[dr, no head] & & \\
 & & F_6 &
\end{tikzcd}$$

Dynamic Programming

- Applicable when subproblems are not independent
  - Subproblems share sub-subproblems

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

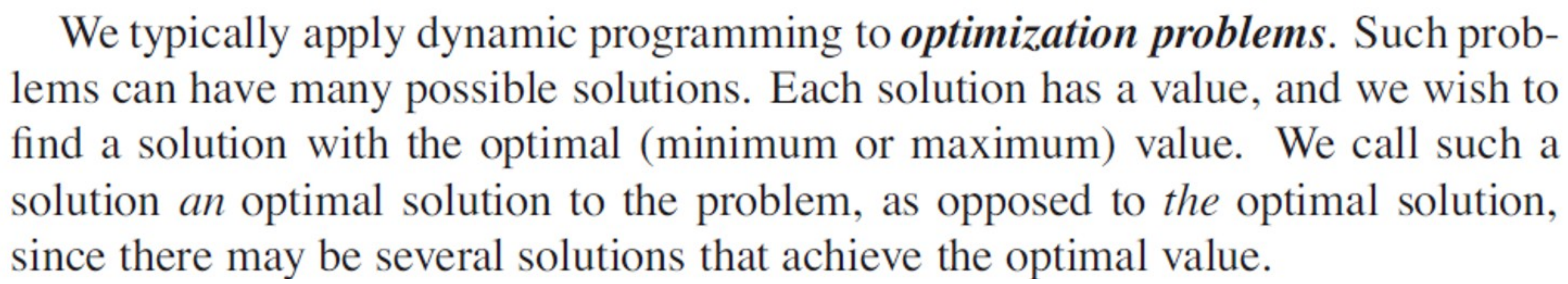
Tabular computation

- 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

The DP Methodology



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.

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

- Steps 1–3 form the basis of a dynamic-programming solution to a problem.
- For the value of an **optimal solution**, and not **the solution itself**, then we can omit step 4.
- When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Rod Cutting

- Serling Enterprises buys long steel rods and cuts them into shorter rods
  - which it then sells.
- Each cut is free. The management of Serling enterprises wants to know the best way to cut up the rods.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

We can cut up a rod of length n in 2^{n-1} different ways

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

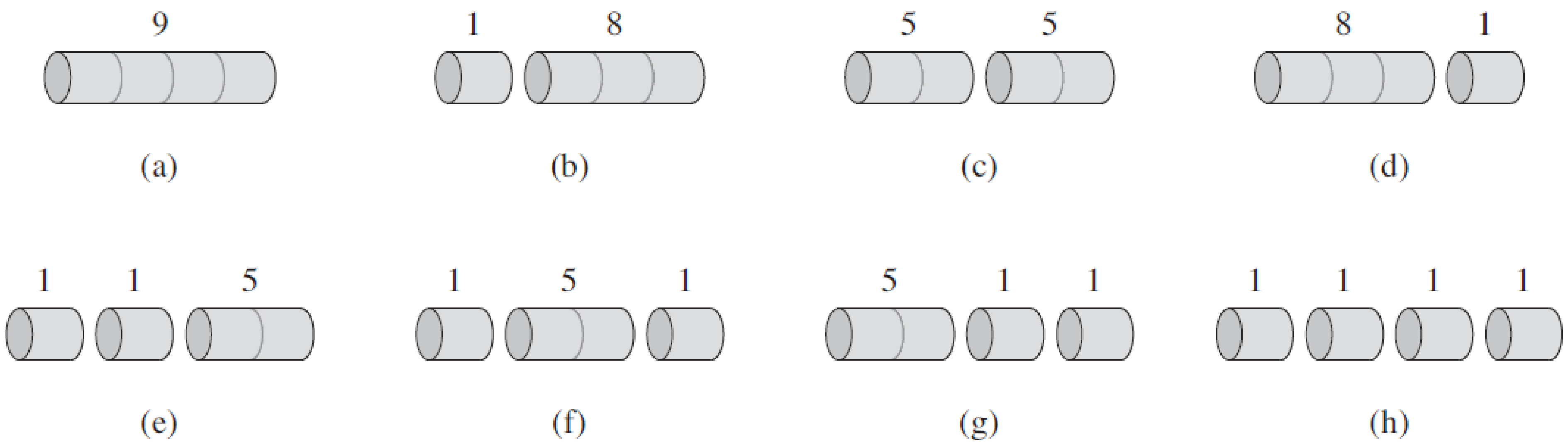


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

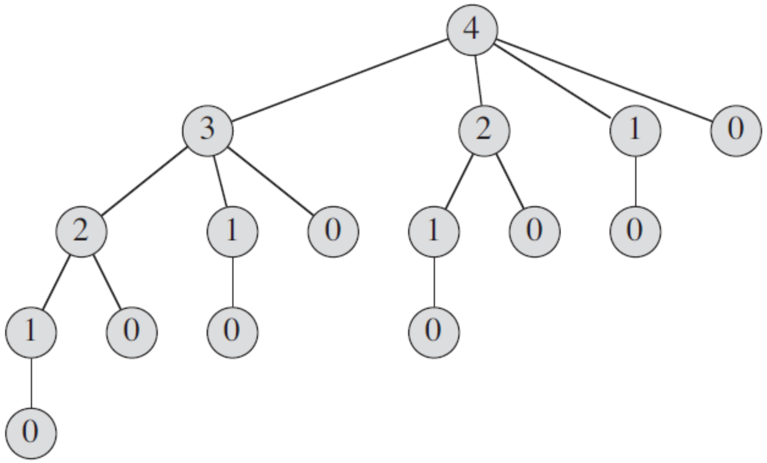
We can cut up a rod of length n in 2^{n-1} different ways.

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- For n = 40, the program takes at least several minutes, and most likely more than an hour.
- In fact, each time you increase n by 1, your program's running time would approximately double.
- this recursion tree has 2^n nodes and 2^{n-1} leaves.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = −∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

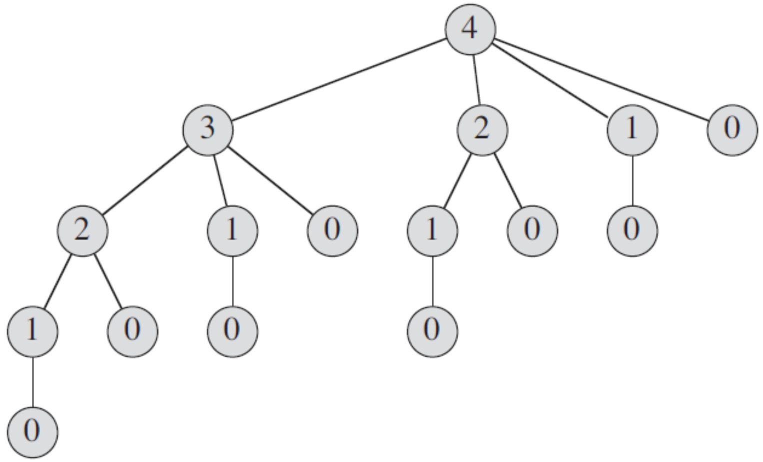


We can cut up a rod of length n in 2^{n-1} different ways.

Rod Cutting – using DP

```
CUT-ROD(p, n)
1  if $n == 0$
2      return 0
3  $q = -\infty$
4  for $i = 1$ to $n$
5      $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
6  return $q$
```

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

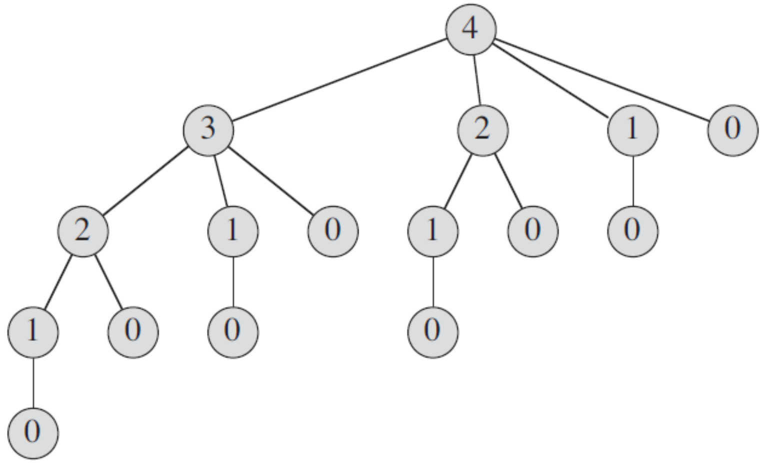


- The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly
- we arrange for each subproblem to be solved only *once*, saving its solution.
- If we need to refer to this subproblem's solution again later, we can just look it up, rather than compute it.

Rod Cutting – using DP

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

```
CUT-ROD(p, n)
1  if n == 0
2        return 0
3  q = −∞
4  for i = 1 to n
5        q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```



- Dynamic programming thus uses additional memory to save computation time; it serves an example of a **time-memory trade-off.**

- The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution.

A dynamic-programming approach runs in **polynomial time** when the **number of distinct subproblems involved is polynomial** in the input size and we can solve each such subproblem in polynomial time.

DP – Two variants

top-down with memoization

- we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).

- The procedure now first checks to see whether it has previously solved this subproblem.
    - If so, it returns the saved value, saving further computation at this level;
    - if not, the procedure computes the value in the usual manner.
- We say that the recursive procedure has been **memoized**;
- it "remembers" what results it has computed previously.

bottom-up method.

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "**smaller**" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.
- We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

DP – Two variants

- These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems.

- The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

top-down with memoization

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

- The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our previous procedure, CUT-ROD

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

top-down with memoization

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

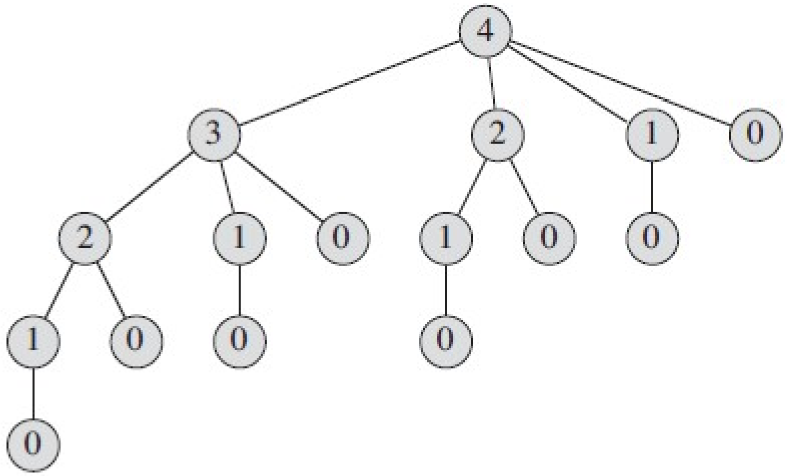
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

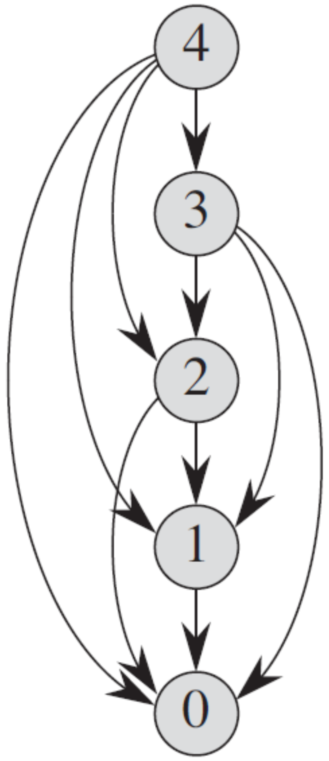
```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```





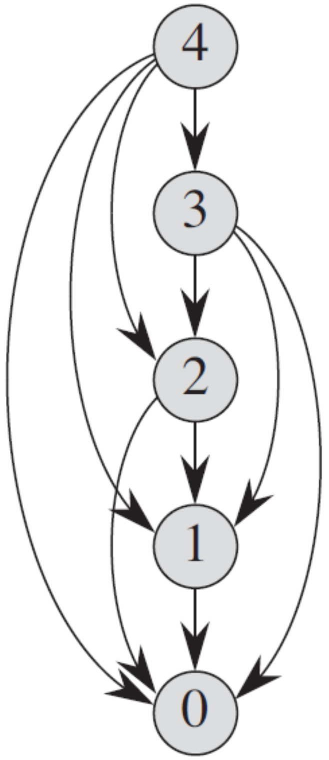
A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x.

This graph is a reduced version of the first tree, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

```
BOTTOM-UP-CUT-ROD(p, n)
1  let $r[0..n]$ be a new array
2  $r[0] = 0$
3  for $j = 1$ to $n$
4      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j-i])$
7      $r[j] = q$
8  return $r[n]$
```



- Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, **but they do not return an actual solution: a list of piece sizes.**

- We can extend the dynamic-programming approach to record not only the **optimal value computed** for each subproblem, but also a **choice** that led to the optimal value.

With this information, we can readily print an optimal solution.

Rod Cutting

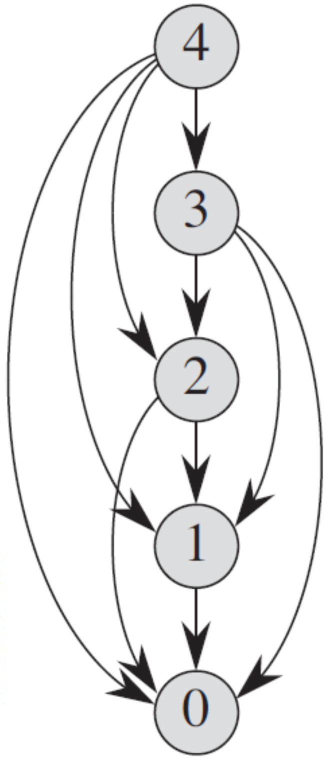
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n − s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10



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.

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 $L[0]=0$)

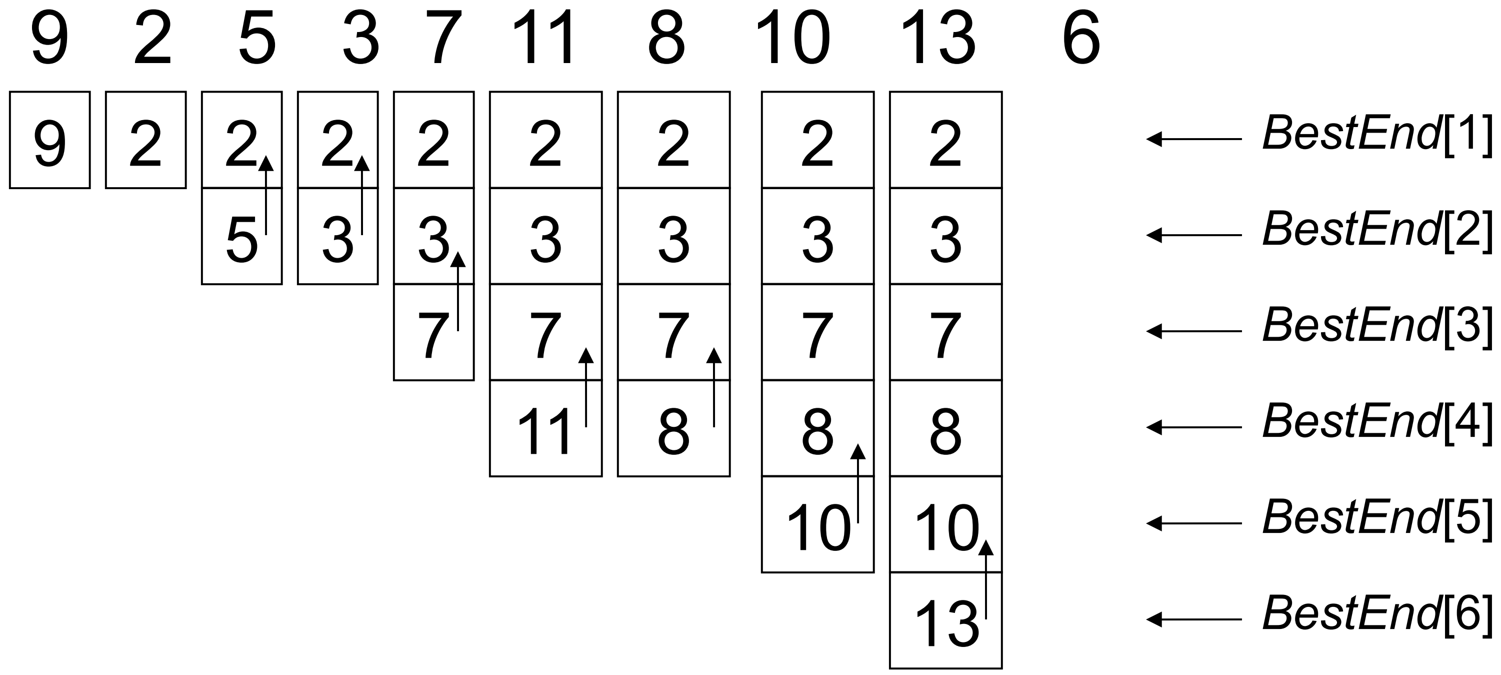
Index	0	1	2	3	4	5	6	7	8	9	10
Input	0	9	2	5	3	7	11	8	10	13	6
Length	0	1	1	2	2	3	4	4	5	6	3
Prev	-1	0	0	2	2	4	5	5	7	8	4
Path	1	1	1	1	1	2	2	2	2	2	2

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

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

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

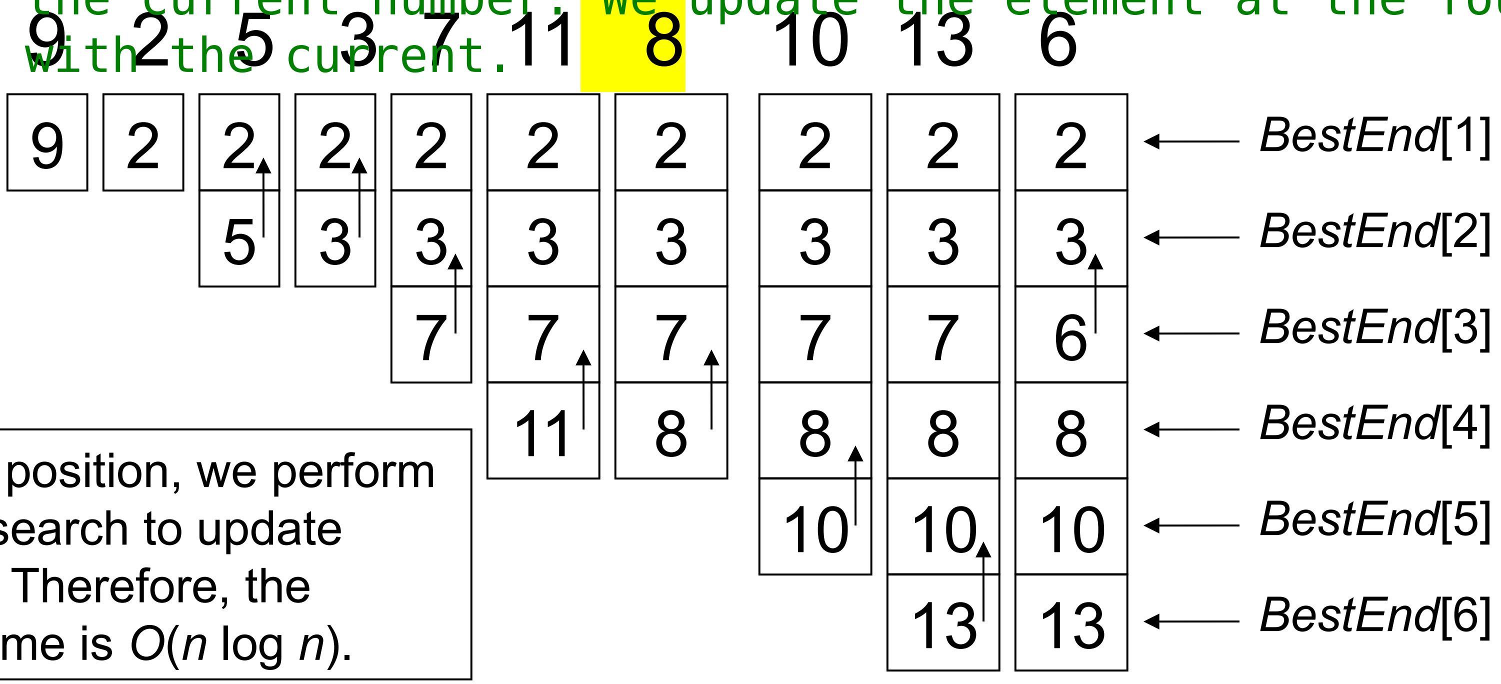
- Define *BestEnd*[*k*] to be the smallest number of an increasing subsequence of length *k*.



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

If the current number is greater than the last element of the answer list, it means we have found a longer increasing subsequence. Hence, we append the current number to the answer list.

If the current number is not greater than the last element of the answer list, we perform a binary search to find the smallest element in the answer list that is greater than or equal to the current number. We update the element at the found position with the current.



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

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

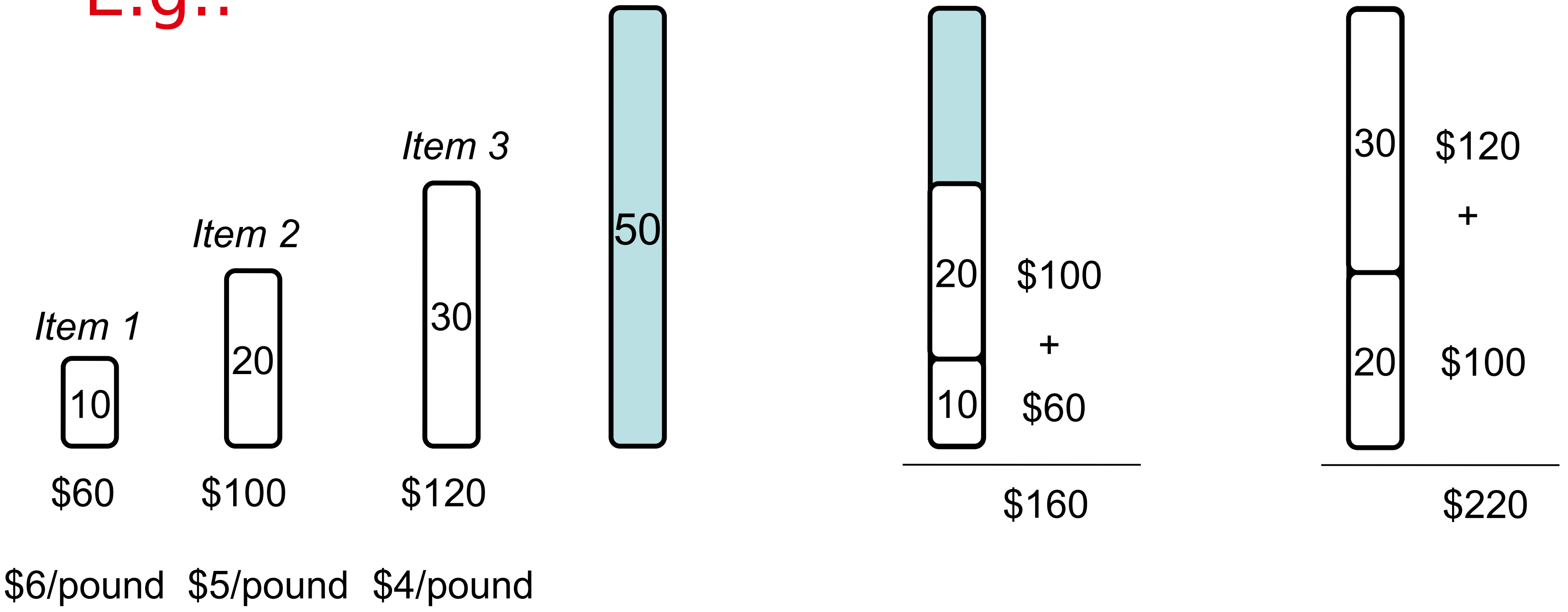
- Thief has a knapsack of capacity W
- There are n items: for i-th item value v_i and weight w_i
- Goal:
  - find x_i such that for all $x_i = \{0, 1\}$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ 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

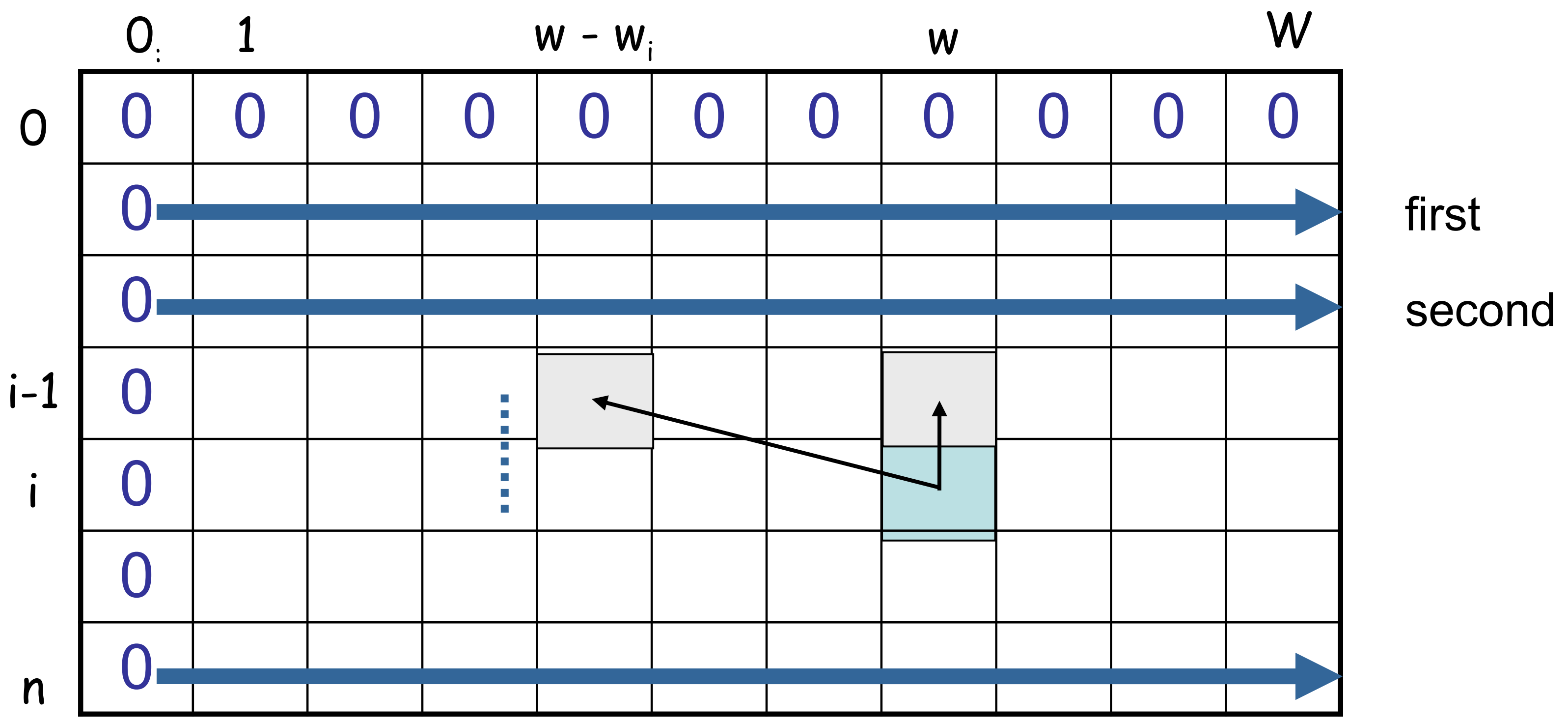
- $P(i, w)$ – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

  $P(i, w) = v_i + P(i - 1, w - w_i)$
- Case 2: thief does not take item i

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

0-1 Knapsack - Dynamic Programming

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



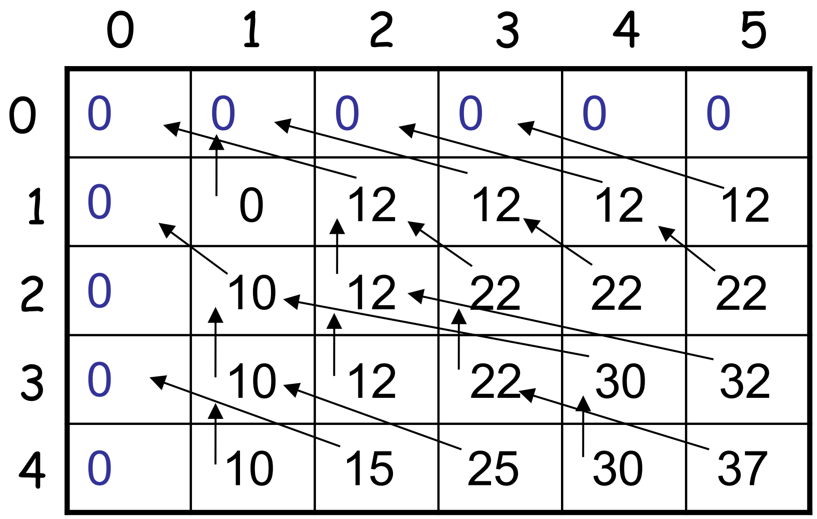
Example:

W = 5

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

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

$\longrightarrow$ wt



	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

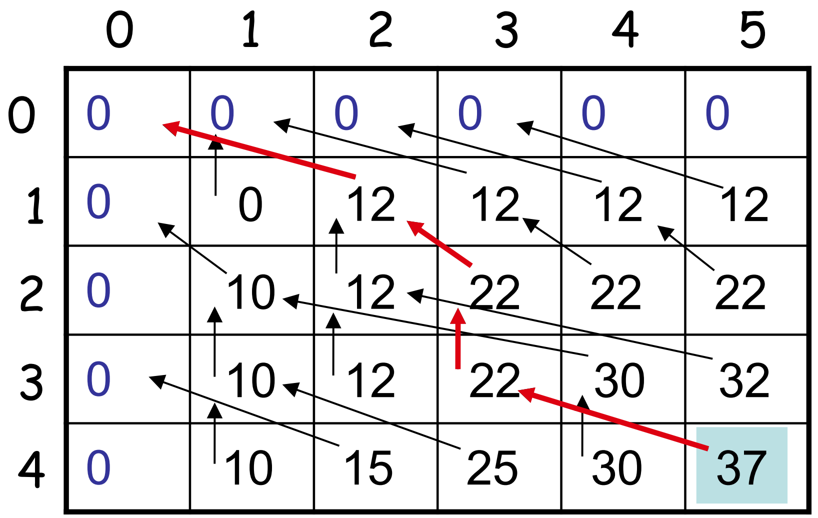
P(4, 2)= max{15+0, 12} = 15

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 ⇒ item i has been taken
- When you go straight up ⇒ 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 $P(i-1, w)$

Longest Common Subsequence (LCS)

- Application: comparison of two DNA strings
- Ex: X= {A B C B D A B }, Y= {B D C A B A}
- Longest Common Subsequence:
- X = A **B** **C** **B** D **A** B
- Y = **B** D **C** A **B** **A**
- Brute force algorithm would compare each subsequence of X with the symbols in Y

Longest Common Subsequence

- Given two sequences

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

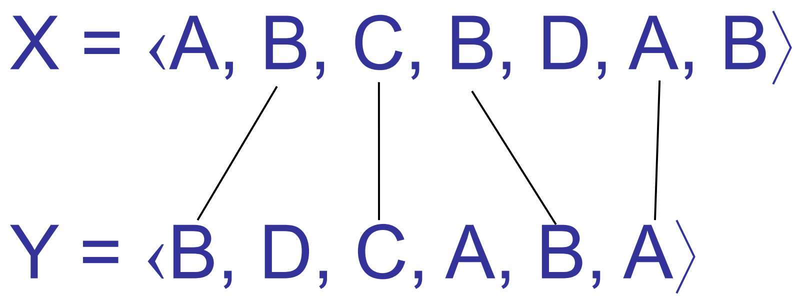
  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

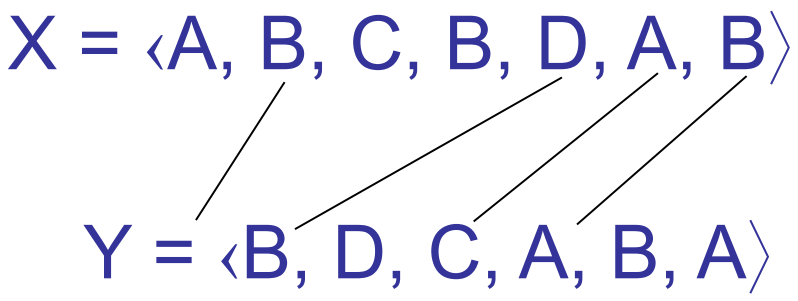
  find a maximum length common subsequence (LCS) of X and Y
- E.g.:

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

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

Example





- ‹B, C, B, A› and ‹B, D, A, B› are longest common subsequences of X and Y (length = 4)
- ‹B, C, A›, however is not a LCS of X and Y

Brute-Force Solution

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

LCS Algorithm

$$X = \langle x_1, x_2, \ldots, x_m \rangle \text{ and } Y = \langle y_1, y_2, \ldots, y_n \rangle.$$

sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith ***prefix*** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

- Define *c[i,j]* to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be *c[m,n]*

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

$X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n$

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

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

LCS recursive solution

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

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

LCS recursive solution

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

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

LCS recursive solution

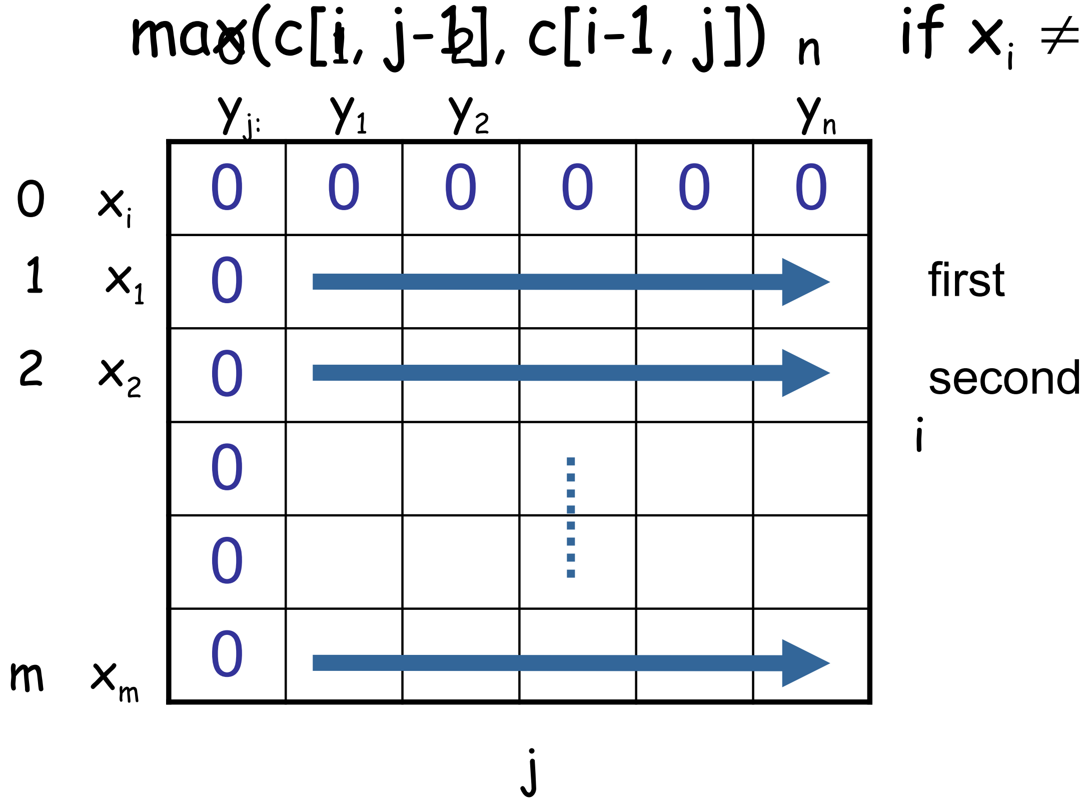
$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

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

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

3. Computing the Length of the LCS

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



Additional Information

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

b & c:

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

i

j

A matrix b[i, j]:

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

  b[i, j] = "↖"
- Else, if

  c[i - 1, j] ≥ c[i, j-1]

  b[i, j] = " ↑ "

  else

  b[i, j] = " ← "

LCS-LENGTH(X, Y, m, n)

```
1.  for i ← 1 to m
2.      do c[i, 0] ← 0
3.  for j ← 0 to n
4.      do c[0, j] ← 0
5.  for i ← 1 to m
6.      do for j ← 1 to n
7.          do if x_i = y_j
8.              then c[i, j] ← c[i - 1, j - 1] + 1
9.                  b[i, j ] ← "↖"
10.             else if c[i - 1, j] ≥ c[i, j - 1]
11.                 then c[i, j] ← c[i - 1, j]
12.                     b[i, j] ← "↑"
13.                 else c[i, j] ← c[i, j - 1]
14.                     b[i, j] ← "←"
15. return c and b
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

Example

X = ⟨A, B, C, B, D, A⟩
Y = ⟨B, D, C, A, B, A⟩

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

If $x_i = y_j$

$b[i, j] =$ " ↖ "

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

$b[i, j] =$ " ↑ "

else

$b[i, j] =$ " ← "

		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

4. Constructing a LCS

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

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B	0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

B

C A B

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

Running time: $\Theta(m + n)$

```
1.  if i = 0 or j = 0
2.     then return
3.  if b[i, j] = "↖"
4.           then PRINT-LCS(b, X, i - 1, j - 1)
5.                 print $x_i$
6.  elseif b[i, j] = "↑"
7.                 then PRINT-LCS(b, X, i - 1, j)
8.                 else PRINT-LCS(b, X, i, j - 1)
```

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

Improving the Code

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

LCS Algorithm Running Time

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

O(m*n)

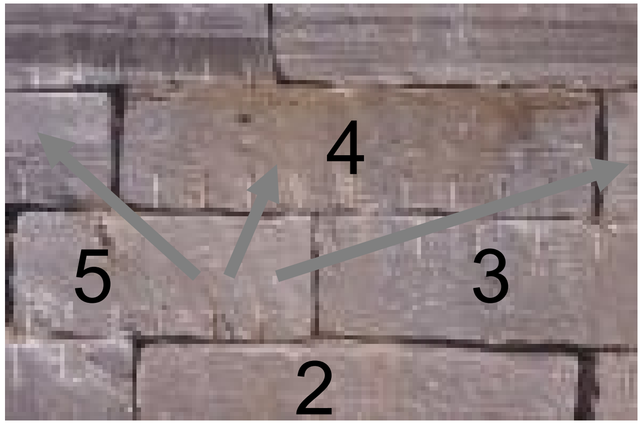
since each c[i,j] is calculated in constant time, and there are m*n elements in the array

Rock Climbing Problem

- A rock climber wants to get from the bottom of a rock to the top by the safest possible path.
- At every step, he reaches for handholds above him; some holds are safer than other.
- From every place, he can only reach a few nearest handholds.

Rock climbing (cont)

- Suppose we have a wall instead of the rock.



At every step our climber can reach exactly three handholds: above, above and to the right and above and to the left.

There is a table of "danger ratings" provided. The "Danger" of a path is the sum of danger ratings of all handholds on the path.

Rock Climbing (cont)

- We represent the wall as a table.

- Every cell of the table contains the danger rating of the corresponding block.

2	8	9	5	8
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

The obvious greedy algorithm does not give an optimal solution. The rating of this path is 13.

The rating of an optimal path is 12.

However, we can solve this problem by a dynamic programming strategy in polynomial time.

Idea: once we know the rating of a path to every handhold on a layer, we can easily compute the ratings of the paths to the holds on the next layer.

For the top layer, that gives us an answer to the problem itself.

For every handhold, there is only one "path" rating. Once we have reached a hold, we don't need to know how we got there to move to the next level.

This is called an "optimal substructure" property. Once we know optimal solutions to subproblems, we can compute an optimal solution to the problem itself.

Recursive solution:

To find the best way to get to stone j in row i, check the cost of getting to the stones

- (i-1,j-1),
- (i-1,j) and
- (i-1,j+1), and take the cheapest.

Problem: each ~~recursion level~~ makes three calls for itself, making a total of 3^n calls – too much!

Solution - memorization

We query the value of A(i,j) over and over again.

Instead of computing it each time, we can compute it once, and remember the value.

A simple recurrence allows us to compute A(i,j) from values below.

Dynamic programming

- Step 1: Describe an array of values you want to compute.
- Step 2: Give a recurrence for computing later values from earlier (bottom-up).
- Step 3: Give a high-level program.
- Step 4: Show how to use values in the array to compute an optimal solution.

Rock climbing: step 1.

- *Step 1: Describe an array of values you want to compute.*
- For $1 \le i \le n$ and $1 \le j \le m$, define $A(i,j)$ to be the cumulative rating of the least dangerous path from the bottom to the hold (i,j).
- The rating of the best path to the top will be the minimal value in the last row of the array.

Rock climbing: step 2.

- *Step 2: Give a recurrence for computing later values from earlier (bottom-up).*
- Let C(i,j) be the rating of the hold *(i,j)*. There are three cases for *A(i,j)*:
- Left *(j=1): C(i,j)+min{A(i-1,j),A(i-1,j+1)}*
- Right *(j=m): C(i,j)+min{A(i-1,j-1),A(i-1,j)}*
- Middle: *C(i,j)+min{A(i-1,j-1),A(i-1,j),A(i-1,j+1)}*
- For the first row *(i=1), A(i,j)=C(i,j).*

Rock climbing: simpler step 2

- Add initialization row: $A(0,j)=0$. No danger to stand on the ground.
- Add two initialization columns: $A(i,0)=A(i,m+1)=\infty$. It is infinitely dangerous to try to hold on to the air where the wall ends.
- Now the recurrence becomes, for every i,j:

$$A(i,j) = C(i,j)+min\{A(i-1,j-1),A(i-1,j),A(i-1,j+1)\}$$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞						∞
2	∞						∞
3	∞						∞
4	∞						∞

Initialization: $\boxed{A(i,0)=A(i,m+1)=\infty,\ A(0,j)=0}$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞						∞
3	∞						∞
4	∞						∞

The values in the first row are the same as C(i,j).

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7					∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\underline{\{\infty,3,2\}}=7.$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9				∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\{\underline{\infty,3,2}\}=7.$ $A\underline{(2,2)=7+\min}\{3,2,5\}=9$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7			∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\{\infty,3,2\}=7$. $A(2,2)=7+\min\{3,2,5\}=9$
$A(2,3)=5+\min\{2,5,4\}=7$.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	12	7	8	∞
4	∞						∞

The best cumulative rating on the second row is 5.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞						∞

The best cumulative rating on the third row is 7.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The best cumulative rating on the last row is 12.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The best cumulative rating on the last row is 12.

So the rating of the best path to the top is 12.

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The last hold was (4,4).

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	**2**	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The hold before the last was (3,4), since min{13,7,8} was 7.

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

The hold before that was (2,5), since min{7,10,5} was 5.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	**4**	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

Finally, the first hold was (1,4), since min{5,4,8} was 4.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	**4**	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

We are done!

Printing out the solution recursively

```
PrintBest(A,i,j) // Printing the best path ending at (i,j)
    if (i==0) OR (j=0) OR (j=m+1)
            return;
    if (A[i-1,j-1]<=A[i-1,j]) AND (A[i-1,j-1]<=A[i-1,j+1])
            PrintBest(A,i-1,j-1);
    elseif (A[i-1,j]<=A[i-1,j-1]) AND (A[i-1,j]<=A[i-1,j+1])
            PrintBest(A,i-1,j);
    elseif (A[i-1,j+1]<=A[i-1,j-1]) AND (A[i-1,j+1]<=A[i-1,j])
            PrintBest(A,i-1,j+1);
    printf(i,j)
```

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

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