Dynamic Programming

Chapter 6 of Dasgupta *et al.*

UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

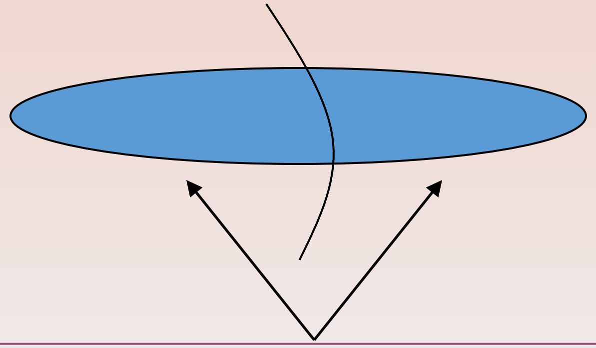
Outline

- Intro
- Stagecoach Problem
- Counting combinations
- 0-1 Knapsack (section 6.4)
- Longest common subsequence
- Later
  - Bellman-Ford (single source shortest path)
  - Floyd-Warshall (all pairs shortest path) (section 8.2)

Two key ingredients

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

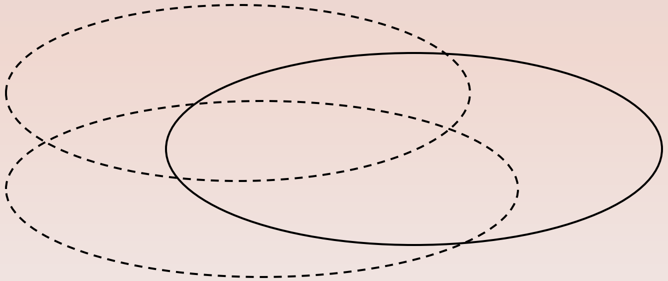
1. optimal substructures



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
  - a recurrence relation (for defining the value/cost of an optimal solution)
  - a tabular computation (for computing the value of an optimal solution)
  - a trace-back procedure (for delivering an optimal solution)

Dynamic Programming

Stagecoach Problem

Stagecoach Problem

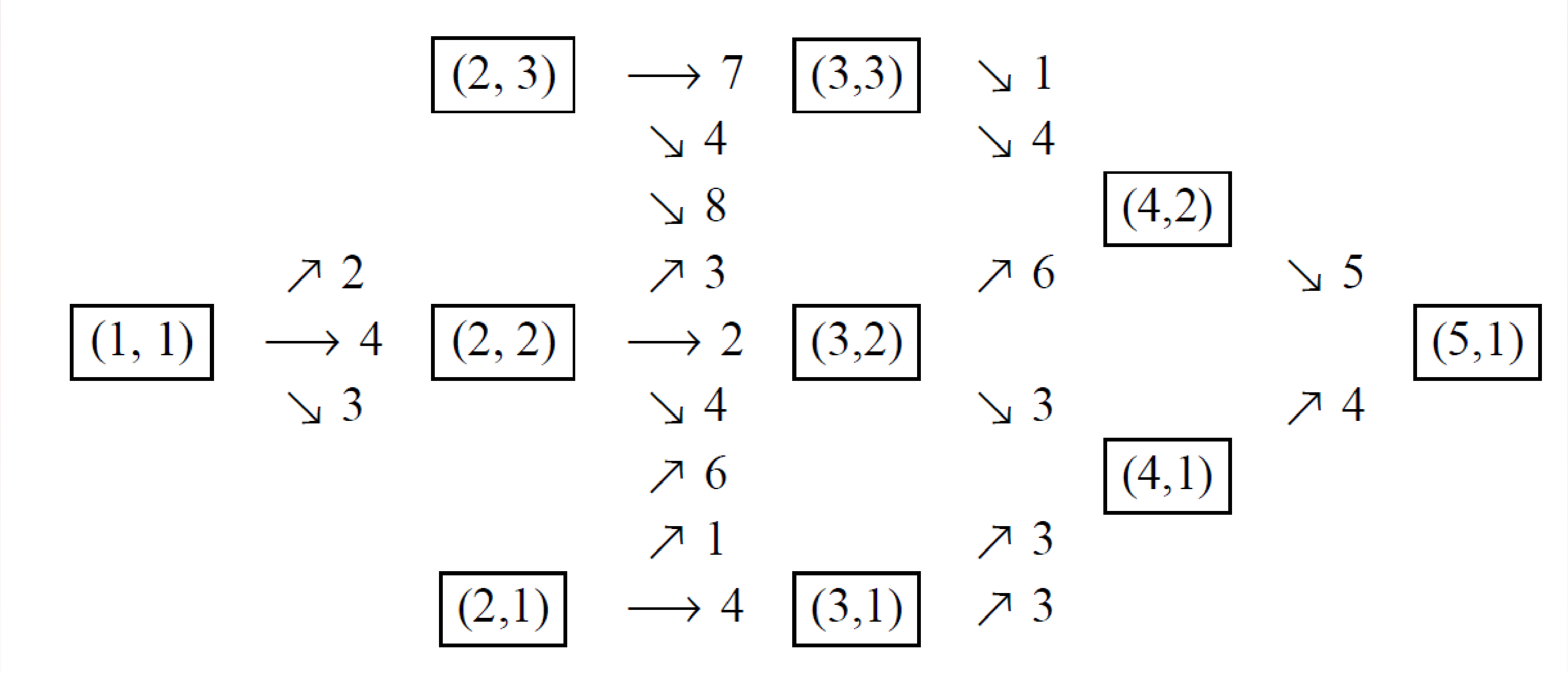
A 19th-century stagecoach company transports passengers from California to New York. Although the starting point and the destination are fixed, the company can choose the intermediate states.

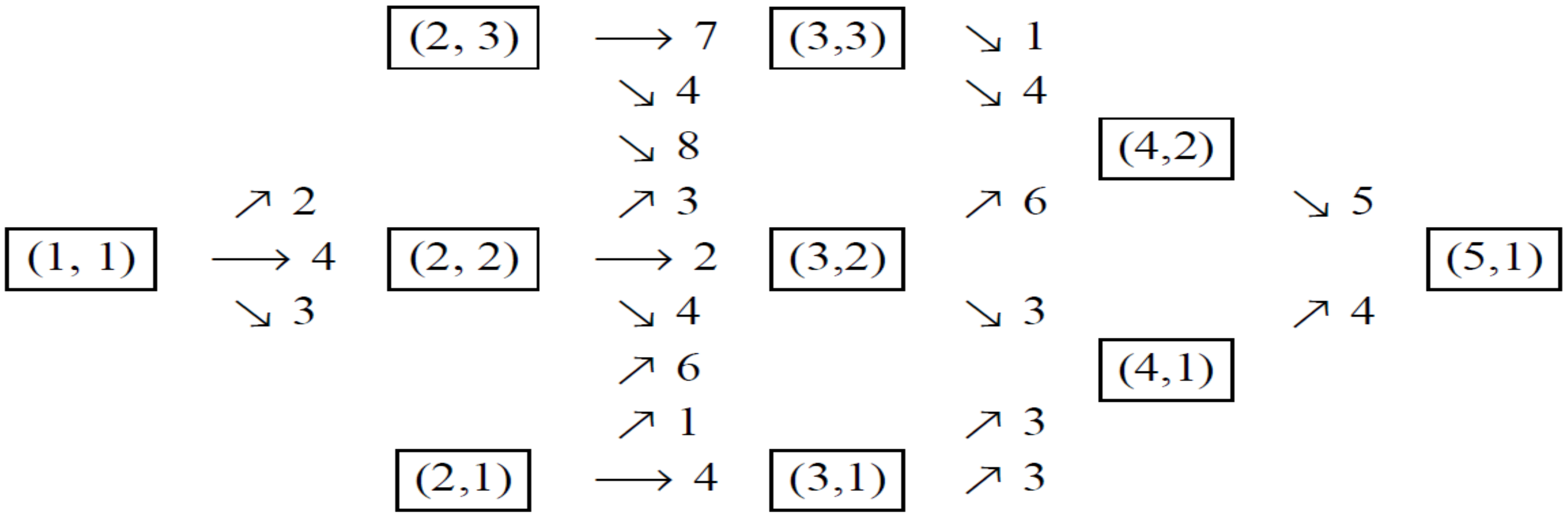
We assume stage 1 starts in California, stage 2 starts in one of three states in the Mountain Time Zone (say, Arizona, Utah or Montana), stage 3 starts in one of three states in the Central Time Zone (say, Oklahoma, Missouri or Iowa) and stage 4 starts in one of two states in the Eastern Time Zone (North Carolina or Ohio). Stage 5, the destination, is New York.

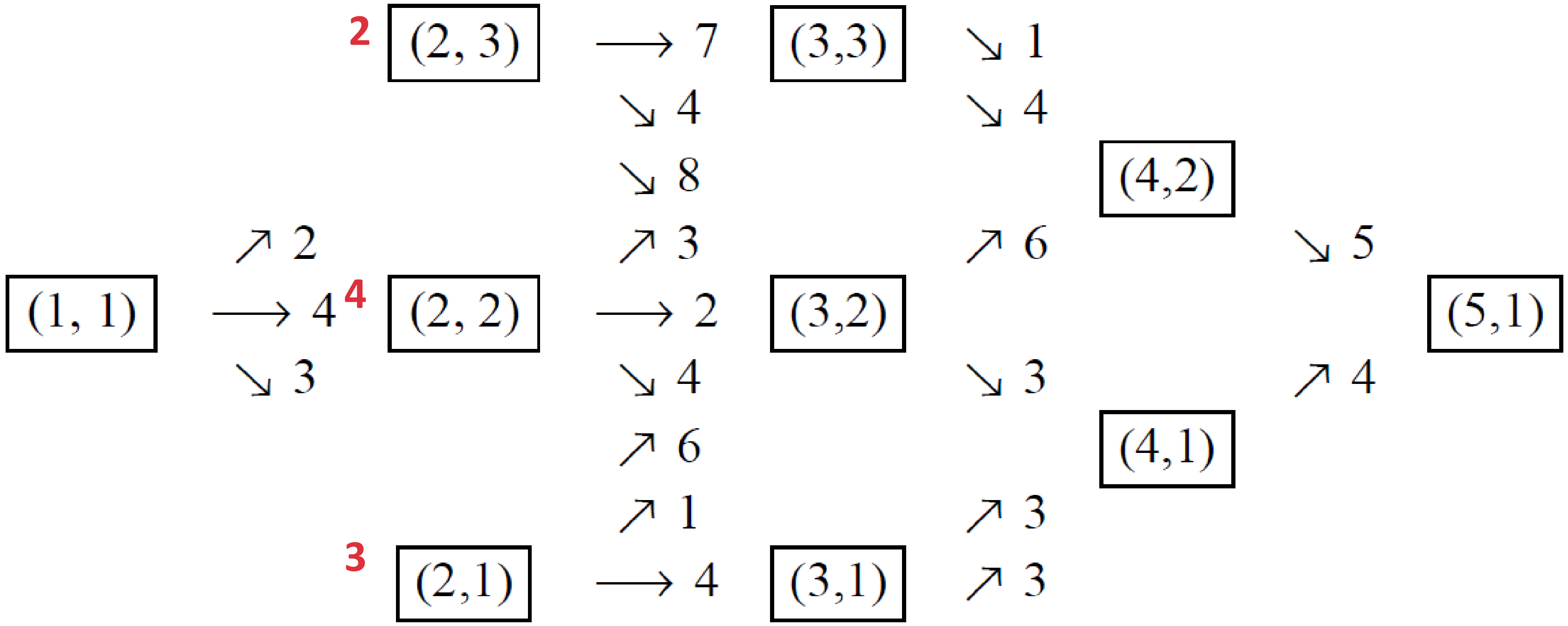
Since in those days travel by stagecoach was rather dangerous because of attacks by roaming criminals, life insurance was offered to the traveling passengers. The cost of the insurance **policy** was higher on more dangerous portions of the trip.

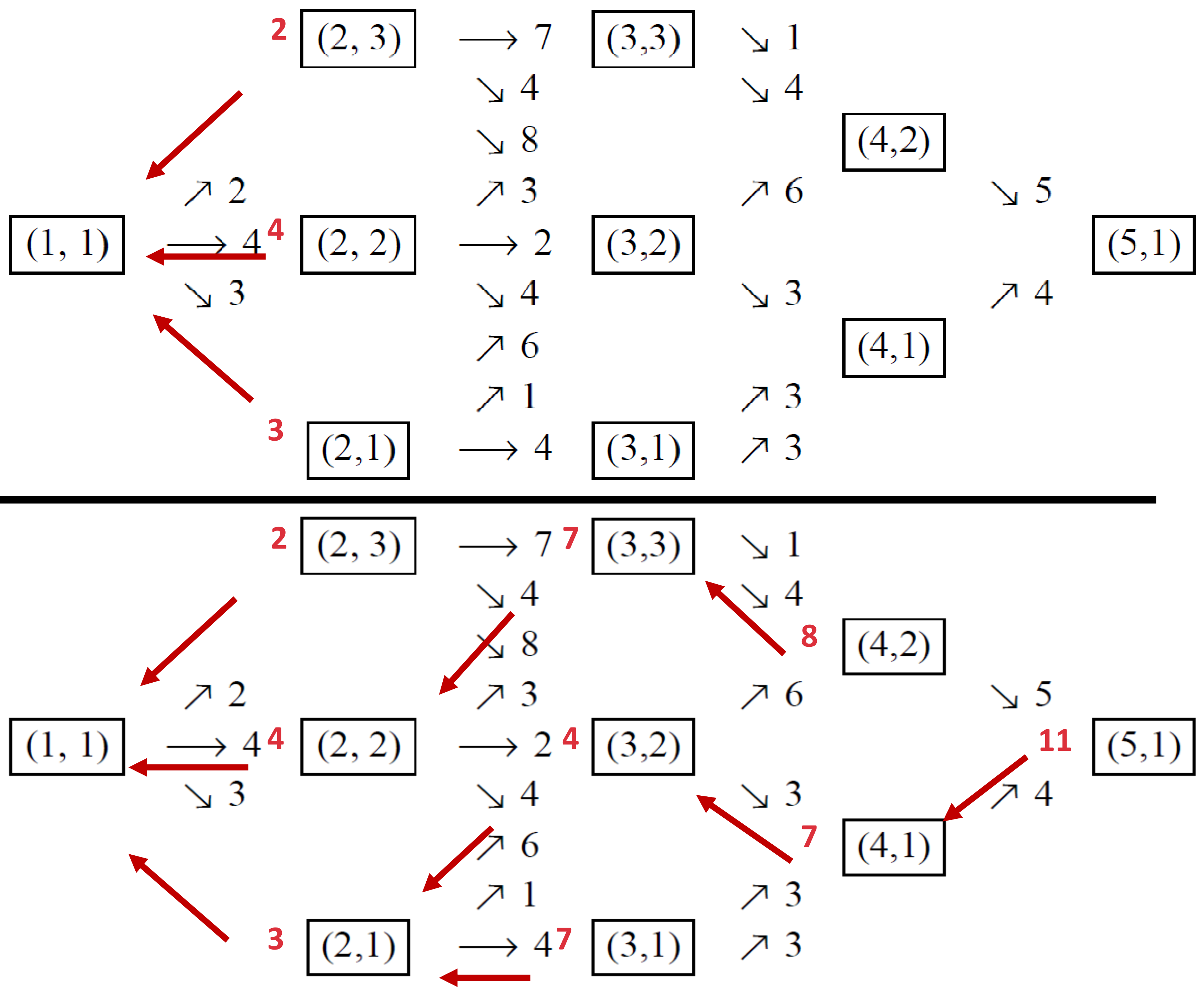
The stagecoach company thus faced the problem of choosing a route that would be cheapest and thus safest for its passengers.

Stagecoach Problem




(1, 1)
(2, 3)
(2, 2)
(2,1)
(3,3)
(3,2)
(3,1)
(4,2)
(4,1)
(5,1)


2
4
3
(1, 1)
(2, 3)
(2, 2)
(2,1)
(3,3)
(3,2)
(3,1)
(4,2)
(4,1)
(5,1)


(1, 1)
(2, 3)
(2, 2)
(2,1)
(3,3)
(3,2)
(3,1)
(4,2)
(4,1)
(5,1)
2
4
3
7
8
4
7
11

Stagecoach Problem

The objective function, V(i, j) is the minimum cost from state (i, j) to the final state (5, 1).

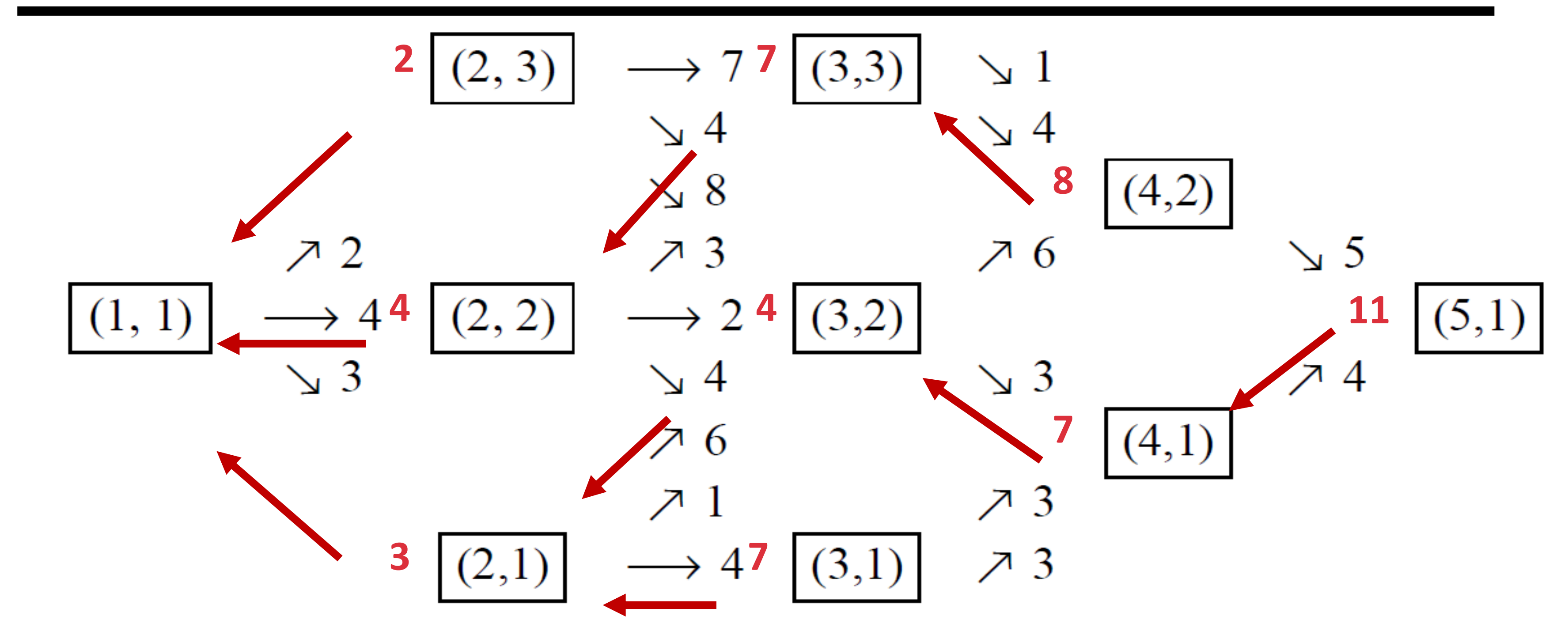
$$V(i, j) = \min_k \{c_{ij}(k) + V(i - 1, k)\}$$

i – stage; j – choice of the state in state i;

The objective function, V(i, j) is the minimum cost from state (i, j) to the final state (5, 1).

$$V(i, j) = \min_k \{c_{ij}(k) + V(i - 1, k)\}$$

i – stage; j – choice of the state in state i;



Counting combinations

Counting combinations

To choose r things out of n, either

- Choose the first item. Then we must choose the remaining $r-1$ items from the other $n-1$ items. Or
- Don't choose the first item. Then we must choose the r items from the other $n-1$ items.

Therefore,

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Counting combinations: D&C

- A simple divide & conquer algorithm for finding the number of combinations of *n* things chosen *r* at a time

```
def choose(n,r):
   if r == 0 or n == r:
      return 1
   else:
      return choose(n-1,r-1)+choose(n-1,r)
```

Counting combinations: D&C

Correctness Proof: A simple induction on n.

Analysis: Let $T(n)$ be the worst case running time of choose(n, r) over all possible values of r.

Then,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n-1) + d & \text{otherwise} \end{cases}$$

for some constants c, d.

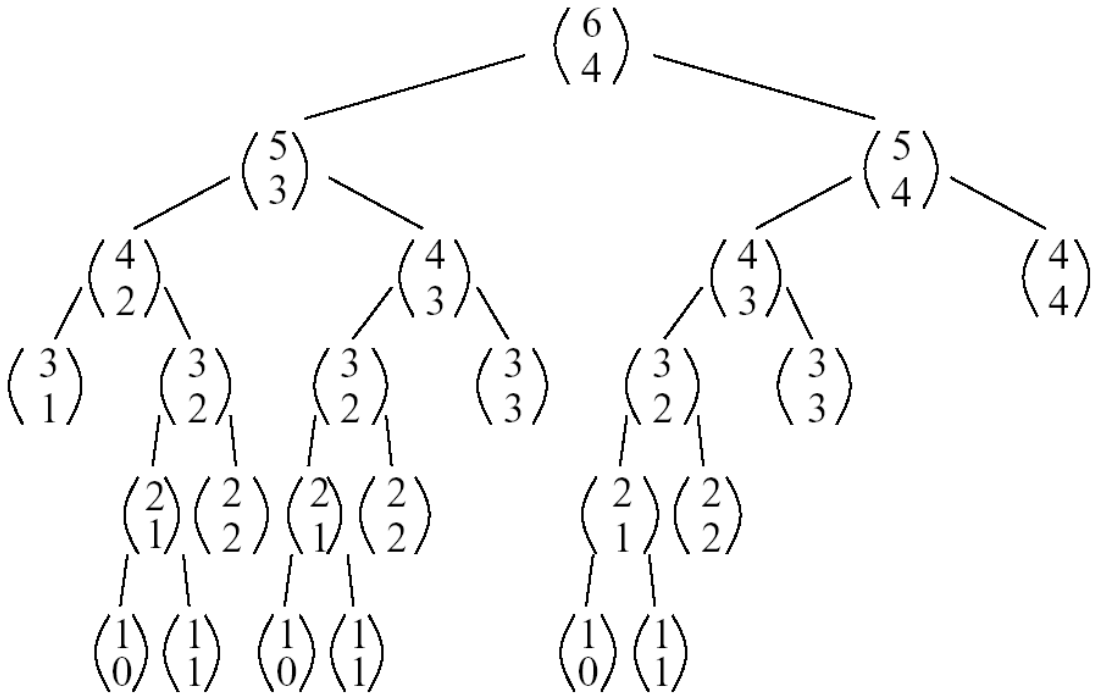
Counting combinations: D&C

$$
\begin{aligned}
T(n) &= 2T(n-1) + d \\
&= 2(2T(n-2) + d) + d \\
&= 4T(n-2) + 2d + d \\
&= 4(2T(n-3) + d) + 2 + d \\
&= 8T(n-3) + 4d + 2d + d \\
&= 2^i T(n-i) + d \sum_{j=0}^{i-1} 2^j \\
&= 2^{n-1} T(1) + d \sum_{j=0}^{n-2} 2^j \\
&= (c+d)2^{n-1} - d
\end{aligned}
$$

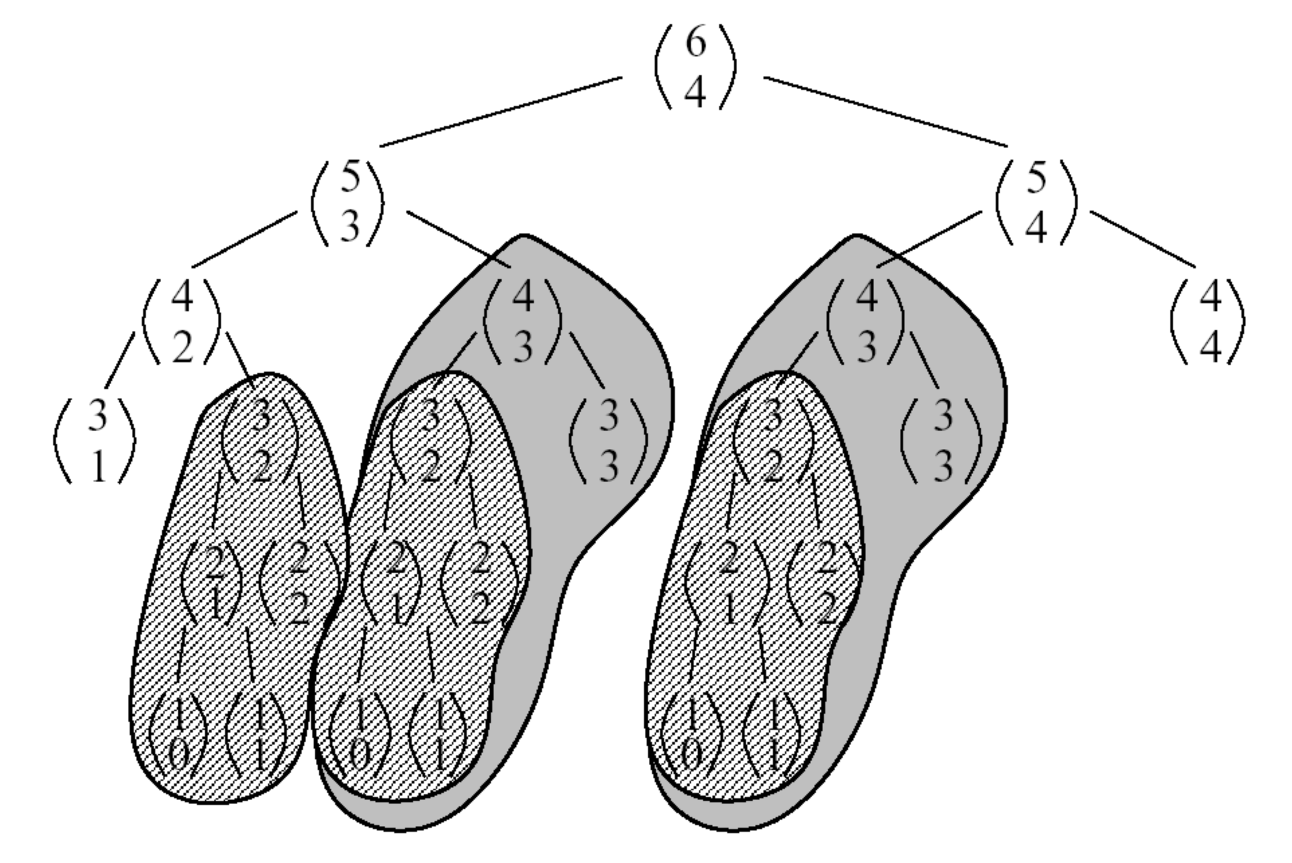
Hence, $T(n) = \Theta(2^n)$.

Counting combinations: Example

The problem is, the algorithm solves the same subproblems over and over again!



Counting combinations: Example



Counting Combinations

- Generate the Pascal’s triangle T[0..n, 0..r] where T[i,j] holds (*i* choose *j*)

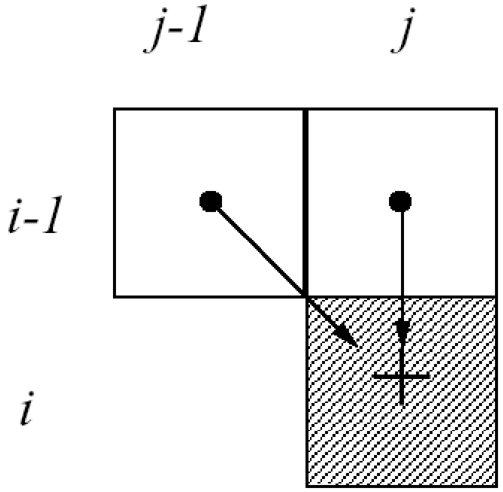
```
def choose(n,r):
    T = {}
    for i in range(n-r+1):
        T[i,0] = 1
    for i in range(r+1):
        T[i,i] = 1
    for j in range (1,r+1):
        for i in range(j+1,n-r+j+1):
            T[i,j] = T[i-1,j-1] + T[i-1,j]
    return T[n,r]
```

Initialization:

n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**		**1**	**x**	**x**
3	**1**			**1**	**x**
4	**1**				**1**
5	**1**				
6	**1**				
7	**1**				
8	**1**				

General Rule

To fill in T[i, j], we need T[$i-1, j-1$] and T[$i-1, j$] to be already filled in.



Filling the table:

n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**		**1**	**x**	**x**
3	**1**			**1**	**x**
4	**1**				**1**
5	**1**				
6	**1**				
7	**1**				
8	**1**				

n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**	**2**	**1**	**x**	**x**
3	**1**			**1**	**x**
4	**1**				**1**
5	**1**				
6	**1**				
7	**1**				
8	**1**				

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Filling the table:

n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**	**2**	**1**	**x**	**x**
3	**1**			**1**	**x**
4	**1**				**1**
5	**1**				
6	**1**				
7	**1**				
8	**1**				

n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**	**2**	**1**	**x**	**x**
3	**1**	**3**		**1**	**x**
4	**1**				**1**
5	**1**				
6	**1**				
7	**1**				
8	**1**				

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Filling the table:

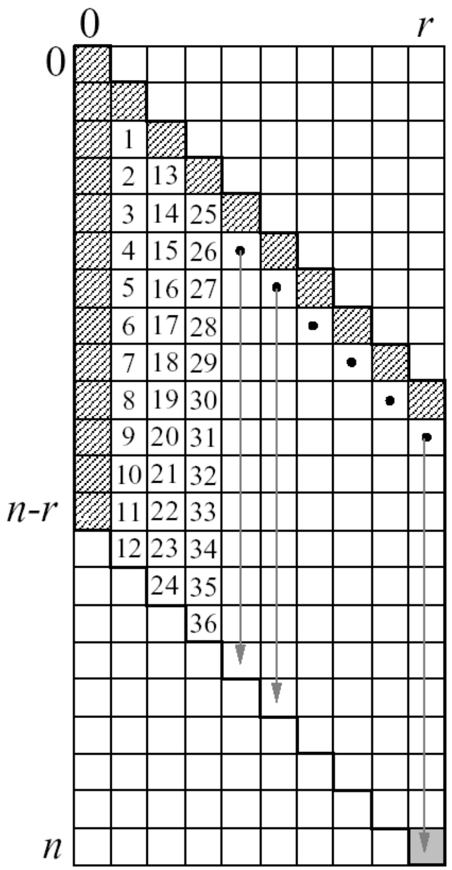
n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**	**2**	**1**	**x**	**x**
3	**1**	**3**	**3**	**1**	**x**
4	**1**	**4**	**6**	**4**	**1**
5	**1**	**5**	**10**	**10**	**5**
6	**1**	**6**	**15**	**20**	**15**
7	**1**	**7**	**21**	**35**	**35**
8	**1**	**8**	**28**	**56**	**70**

n \ r	0	1	2	3	4
0	**1**	**x**	**x**	**x**	**x**
1	**1**	**1**	**x**	**x**	**x**
2	**1**	**2**	**1**	**x**	**x**
3	**1**	**3**	**3**	**1**	**x**
4	**1**	**4**	**6**	**4**	**1**
5	**1**	**5**	**10**	**10**	**5**
6	**1**	**X**	**15**	**20**	**15**
7	**1**	**X**	**X**	**35**	**35**
8	**1**	**X**	**X**	**X**	**70**

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Filling in the Table

Fill in the columns from left to right. Fill each of the columns from top to bottom.



Numbers show the order in which the entries are filled in

Analysis

How many table entries are filled in?

$$(n-r+1)(r+1) = nr+n-r^2+1 \leq n(r+1)+1$$

Each entry takes time $O(1)$, so total time required is $O(n^2)$.

This is much better than $O(2^n)$.

Space: naive, $O(nr)$. Smart, $O(r)$.

Dynamic Programming

When divide and conquer generates a large number of identical subproblems, recursion is too expensive.

Instead, store solutions to subproblems in a table.

This technique is called <u>dynamic programming</u>.

Dynamic Programming

Identification:

- devise divide-and-conquer algorithm
- analyze — running time is exponential
- same subproblems solved many times

Dynamic Programming Construction

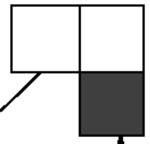
- take part of divide-and-conquer algorithm that does the "conquer" part and replace recursive calls with table lookups
- instead of returning a value, record it in a table entry
- use base of divide-and-conquer to fill in start of table
- devise "look-up template"
- devise for-loops that fill the table using "look-up template"

Divide and Conquer

```
function choose(n,r)
  if r=0 or r=n then return(1) else
    return(choose(n-1,r-1)+choose(n-1,r))
```

Dynamic Programming

```
function choose(n,r)
  for i:=0 to n-r do T[i,0]:=1
  for i:=0 to r do T[i,i]:=1
  for j:=1 to r do
    for i:=j+1 to n-r+j do
      T[i,j]:=T[i-1,j-1]+T[i-1,j]
  return(T[n,r])
```



0-1 knapsack

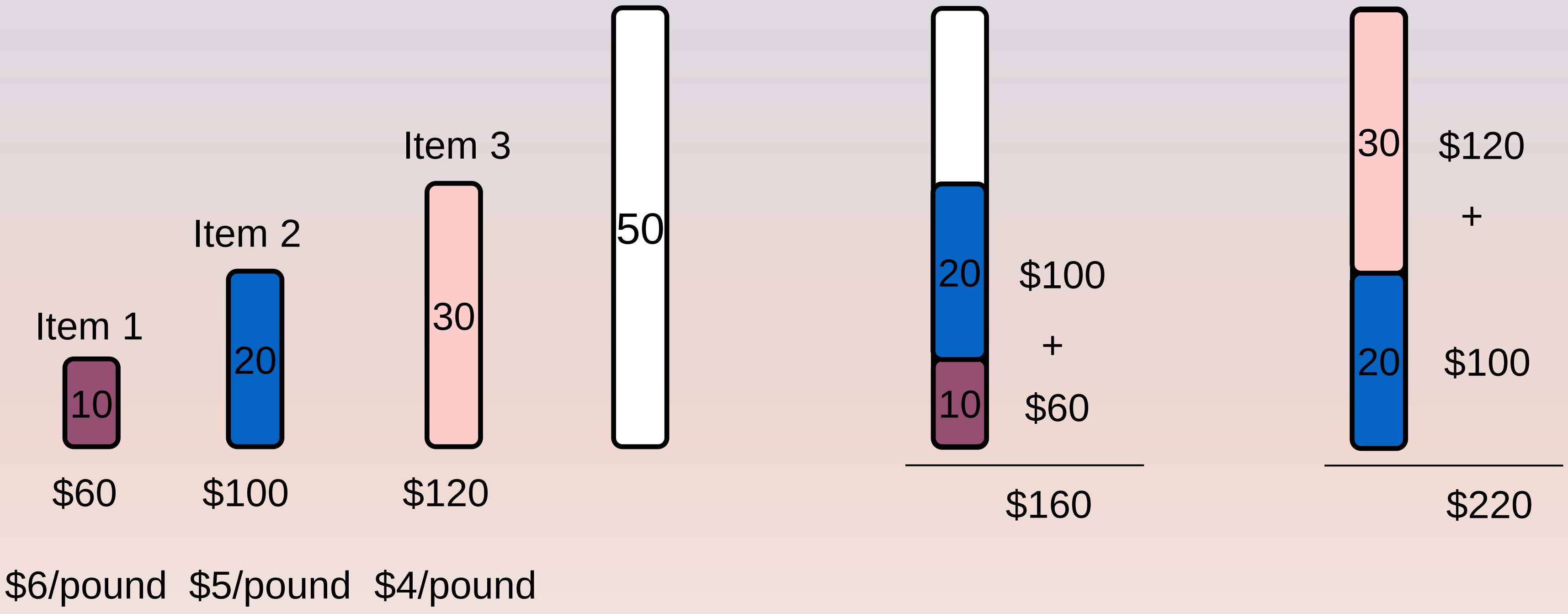
The Knapsack Problem

- A thief robbing a store finds n items
- The i^{th} item is worth b_i and weighs w_i pounds
- Thief's knapsack can carry at most W pounds
- Variables b_i, w_i and W are <u>integers</u>
- <u>Problem</u>: What items to select to maximize profit?

The 0-1 Knapsack Problem

- Each item must be either taken or left behind (a binary choice of 0 or 1)
- Exhibits *optimal substructure* property (for the same reason as for the fractional)
- 0-1 knapsack problem however *cannot* be solved by a greedy strategy
- Can be solved (less) efficiently by *dynamic programming*

0-1 Knapsack - Greedy Strategy



- The greedy choice property does not hold

0-1 Knapsack Problem

- Let x_i=1 denote item *i* is in the knapsack and x_i=0 denote it is not in the knapsack
- Problem stated formally as follows

$$\text{maximize} \quad \sum_{i=1}^{n} b_i x_i \qquad \text{(total profit)}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W \qquad \text{(weight constraint)}$$

Define the problem recursively ...

- Consider the first item *i*=1

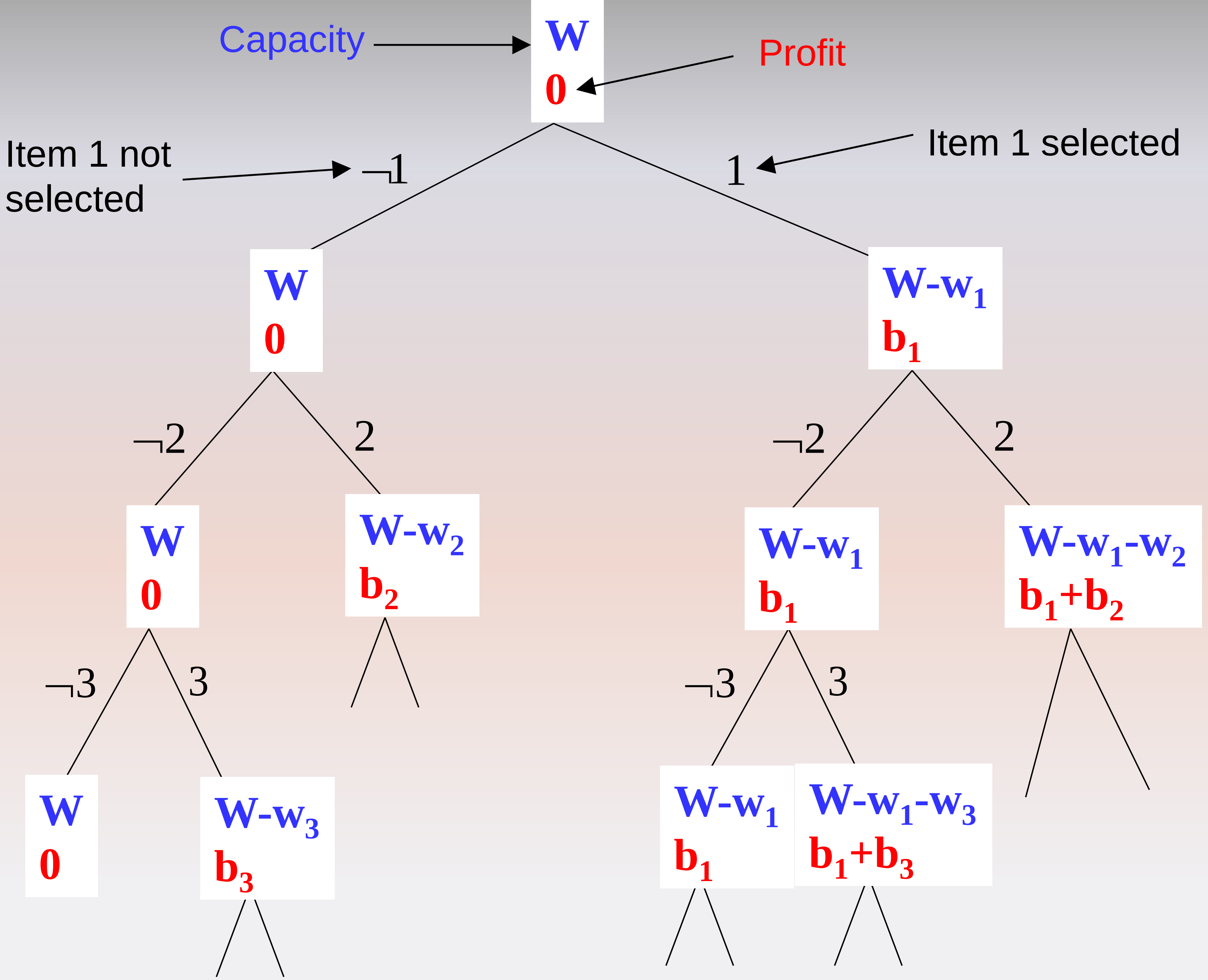
1. If it is selected (put in the knapsack)

maximize $$\sum_{i=2}^{n} b_i x_i$$ subject to $$\sum_{i=2}^{n} w_i x_i \le W - w_1$$

2. If it is not selected

maximize $$\sum_{i=2}^{n} b_i x_i$$ subject to $$\sum_{i=2}^{n} w_i x_i \le W$$

- Compute both cases, select the better one

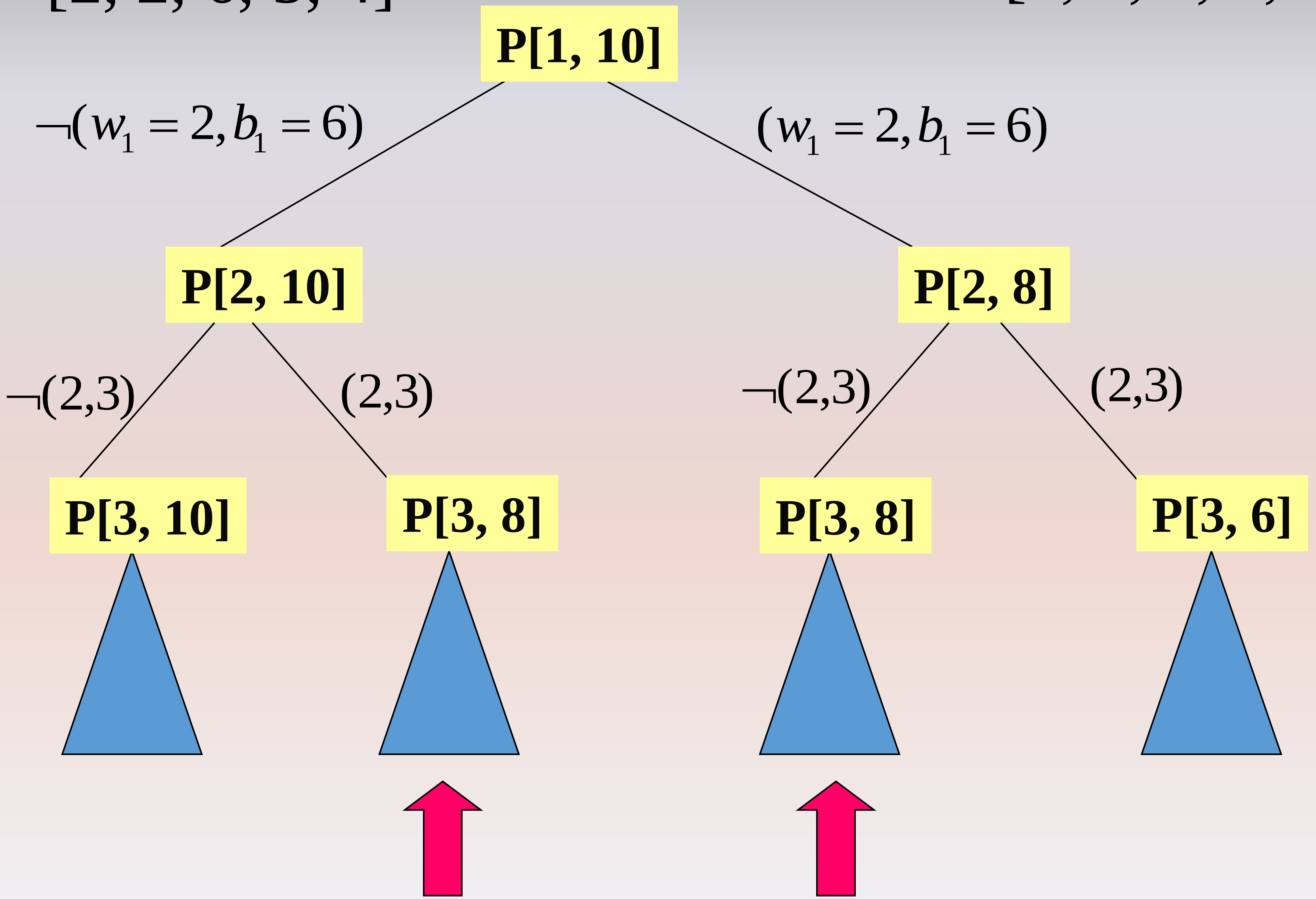

Capacity
Profit
W
0
Item 1 not selected
¬1
Item 1 selected
1
W
0
W-w_1
b_1
¬2
2
¬2
2
W
0
W-w_2
b_2
W-w_1
b_1
W-w_1-w_2
b_1+b_2
¬3
3
¬3
3
W
0
W-w_3
b_3
W-w_1
b_1
W-w_1-w_3
b_1+b_3

Recursive Solution

- We can write an algorithm for the recursive solution based on the four cases
- Recursive algorithm will take $O(2^n)$ time
- Inefficient because $P[i,k]$ for the same i and k will be computed many times
- Example
  - n=5, W=10, w=[2, 2, 6, 5, 4], b=[6, 3, 5, 4, 6]

w = [2, 2, 6, 5, 4]

b = [6, 3, 5, 4, 6]



Same subproblem

Dynamic Programming Solution

- Let us define *P*[*i*,*k*] as the maximum profit possible using items {1, 2, … *i*} and residual (knapsack) capacity *k*

$$
\mathrm{P}(\mathrm{i},\mathrm{k}) = \begin{cases} \mathrm{P}(0,\ \mathrm{k}) = 0 \\ \mathrm{P}(1,\ \mathrm{k}) = b_1 & \text{if } w_1 \le \mathrm{k} \\ \mathrm{P}(\mathrm{i}-1,\mathrm{k}) & \text{if } w_i > \mathrm{k} \\ max_{0\le i\le n}\{\mathrm{P}((\mathrm{i}-1,\ \mathrm{k}-w_i)+bi),\ \mathrm{P}(\mathrm{i}-1,\mathrm{k})\} & \text{if } w_i \le \mathrm{k} \end{cases}
$$

OR … Dynamic Programming Solution

- Let us define *P*[*i*,*k*] as the maximum profit possible using items {*i*, *i*+1,…, *n*} and residual (knapsack) capacity *k*
- We can define *P*[*i*,*k*] recursively as follows

$$P[i,k] = \begin{cases} 0 & i = n \ \& \ w_n > k \\ b_n & i = n \ \& \ w_n \leq k \\ P[i+1,k] & i < n \ \& \ w_i > k \\ \max\{P[i+1,k],\ b_i + P[i+1,k-w_i]\} & i < n \ \& \ w_i \leq k \end{cases}$$

0-1 knapsack (recursive) in Python

```
def knapsack(items,i,k):
  n = len(items)
  if i == n:
    return b(items[n-1]) if w(items[n-1])<=k else 0
```

Remark: i < n

```
  if w(items[i-1])>k:
     return knapsack(items,i+1,k)
  else:
    return max(knapsack(items,i+1,k),
      b(items[i-1])+knapsack(items,i+1,k-w(items[i-1])))
```

Dynamic Programming Solution

- The inefficiency can be eliminated by computing each *P*[*i*,*k*] once and storing the result in a table for future use
- The table is filled for *i=n,n-1, …,2,1* in that order for $1 \le k \le W$
- First row (initialization)

k	0	1	…	w_n-1	w_n	w_n+1	…	W
$P[n,k]$	*0*	*0*	…	*0*	b_n	b_n	…	b_n

Example

n=5, W=10, w = [2, 2, 6, 5, **4**], b = [2, 3, 5, 4, **6**]

$i \backslash k$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*6*	*6*
4											
3											
2											
1											

Example

n=5, *W*=10, *w* = [2, 2, 6, **5**, 4], *b* = [2, 3, 5, **4**, 6]

*i**k*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	*0* +4	*0*	*0*	*0*	*6* +4	*6*	*6*	*6*	*6*	*6*	*6*
4	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*10*	*10*
3											
2											
1											

$P[i,k] = \max\{P[i+1,k],\ b_i+P[i+1,k-w_i]\}$

Example

n=5, W=10, w = [2, 2, 6, 5, 4], b = [2, 3, 5, 4, 6]

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	+5 0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2											
1											

$P[i,k] = \max\{P[i+1,k],\ b_i+P[i+1,k-w_i]\}$

Example

n=5, W=10, w = [2, **2**, 6, 5, 4], b = [2, **3**, 5, 4, 6]

$i \backslash k$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*6*	*6*
4	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*10*	*10*
3	+3 *0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*10*	*11*
2	*0*	*0*	*3*	*3*	*6*	*6*	*9*	*9*	*9*	*10*	*11*
1											

$P[i,k] = \max\{P[i+1,k],\ b_i+P[i+1,k-w_i]\}$

Example

n=5, *W*=10, *w* = [**2**, 2, 6, 5, 4], *b* = [**2**, 3, 5, 4, 6]

*i**k*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*6*	*6*
4	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*10*	*10*
3	*0*	*0*	*0*	*0*	*6*	*6*	*6*	*6*	*6*	*10*	*11*
2	*0*	*0*	*3*	*3*	*6*	*6*	**+2** *9*	*9*	**+2** *9*	*10*	*11*
1	*0*	*0*	*3*	*3*	*6*	*6*	*9*	*9*	*11*	*11*	*11*

$P[i,k] = \max\{P[i+1,k],\ b_i+P[i+1,k-w_i]\}$

Example

n=5, W=10, w = [2, 2, 6, 5, 4], b = [2, 3, 5, 4, 6]

$i\backslash k$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1	0	0	3	3	6	6	9	9	11	11	11

x = [0,0,1,0,1]

x = [1,1,0,0,1]

0-1 knapsack in Python

```
def knapsack(items,w):
  P, n = {}, len(items)
  for j in range(w+1):
    P[n,j] = b(items[n-1]) if w(items[n-1])<=j else 0
  for i in range(len(items)-1,0,-1):
    for j in range(w+1):
      if w(items[i-1])>j:
        P[i,j] = P[i+1,j]
      else:
        P[i,j] = max(P[i+1,j],
                     b(items[i-1])+P[i+1,j-w(items[i-1])])
  return P
```

Time complexity

- Running time: $O(nW)$

- Technically, this is not a poly-time algorithm

- This class of algorithms is called *pseudo-polynomial*

Longest increasing subsequence (LIS)

- The longest increasing subsequence problem is to find a longest increasing subsequence of a given sequence of 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 standard DP 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 = maximum, and $L[0]=0$)

	9	2	5	3	7	11	8	10	13	6
$L[i]$	1	1	2	2	3	4	?			

A standard DP approach for LIS

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

	9	$\boxed{2}$	5	$\boxed{3}$	$\boxed{7}$	11	$\boxed{8}$	$\boxed{10}$	$\boxed{13}$	6
$L[i]$	1	1	2	2	3	4	4	5	6	3

The maximum length

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

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

Example: Trip-Hotel Problem

You are going on a long trip.

1. You start on the road at mile post 0.
2. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point.
3. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at.
4. You must stop at the final hotel (at distance a_n), which is your destination.

Example: Trip-Hotel Problem

5. You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$.
6. You want to plan your trip so as to minimize the total penalty - that is, the sum, over all travel days, of the daily penalties.

Question: Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Example: Trip-Hotel Problem

Subproblem definition: Let D(i) for 0 ≤ i ≤ n to be the minimum total penalty to get to hotel i.

Algorithm and Recursion: To get D(i), we consider all possible hotels j we can stay at the night before reaching hotel i.

For each of these possibilities, the minimum penalty to reach i is the sum of:

- the minimum penalty D(j) to reach j,
- and the cost $(200-(a_j-a_i))^2$ of a one-day trip from j to i.

Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all the j:

$$D(i) = min_{0\le j<i}\{D(j) + (200-(a_j-a_i))^2\}$$

And the base case is D(i)=0.

Example: Trip-Hotel Problem

Running Time: The running time is $O(n^2)$, as we have n subproblems and each takes time O(n) to solve, as we need to compute the minimum of O(n) values. Moreover, backtracking only takes time O(n).

Longest common subsequence

Longest Common Subsequence

A sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a *subsequence* of a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ if Z can be generated by striking out some (or none) elements from X

For example, $\langle b, c, d, b \rangle$ is a subsequence of $\langle a, b, c, a, d, c, a, b \rangle$

Longest Common Subsequence

The **longest common subsequence problem** is the problem of finding, for given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, a maximum-length common subsequence of X and Y.

Longest Common Subsequence

- For example, given
  X = B D C A B A
  Y = A B C B D A B
- $Z=LCS(X,Y)$ = BCBA
- X = B D C A B A
  Y = A B C B D A B

Longest Common Subsequence

Brute-force search for LCS requires exponentially many steps because if $m < n$, there are $\sum_{i=1}^{m} \binom{n}{i}$ candidate subsequences

Solve this problem by dynamic programming

The optimal-substructure of LCS

For a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ and $i, 1 \leq i \leq k$, let Z_i denote the length i prefix of Z, namely, $Z_i = \langle z_1, z_2, \ldots, z_i \rangle$.

Optimal Substructure

Theorem. Let $X=<x_1, \ldots, x_m>$ and $Y=<y_1,\ldots, y_n>$ be two sequences, and let $Z = <z_1, \ldots , z_k>$ be any LCS of X and Y

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Proof: omitted

Recursive Formulation

- Define $c[i, j]$ = length of LCS of X_i and Y_j

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

- We want $c[m,n]$
- This gives a recursive algorithm and solves the problem
- But is it efficient?

Longest Common Subsequence

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

$X =$ B D C A B A
$Y =$ A B C B D A B

- $Z = LCS(X,Y)$ = BCBA
- $X =$ B D C A B A
  $Y =$ A B C B D A B

Example

$$c[a,b] = \begin{cases} 0 & \text{if } a \text{ empty or } b \text{ empty}, \\ c[\text{prefix } a, \text{prefix } b] + 1 & \text{if end}(a) = \text{end}(b), \\ \max(c[\text{prefix } a, b], c[a, \text{prefix } b]) & \text{if end}(a) \neq \text{end}(b). \end{cases}$$

c[springtime, printing]

c[springtim, printing] c[springtime, printin]

[springti, printing] [springtim, printin] ~~[springtim, printin]~~ [springtime, printi]

[springt, printing] [springti, printin] [springtim, printi] [springtime, print]

$$c[a,b] = \begin{cases} 0 & \text{if } a \text{ empty or } b \text{ empty}, \\ c[\text{prefix } a, \text{prefix } b] + 1 & \text{if end}(a) = \text{end}(b), \\ \max(c[\text{prefix } a, b], c[a, \text{prefix } b]) & \text{if end}(a) \neq \text{end}(b). \end{cases}$$

Keep track of c[α,β] in a table of nm entries

		p	r	i	n	t	i	n	g
s									
p									
r									
i									
n									
g									
t									
i									
m									
e									

LCS in Python

```
def LCS(X,Y):
  c = {}
  for i in range(len(X)+1):
    for j in range(len(Y)+1):
      if i == 0 or j == 0:
        c[i,j] = 0
      elif X[i-1] == Y[j-1]:
        c[i,j] = c[i-1,j-1] + 1
      else:
        c[i,j] = max(c[i-1,j],c[i,j-1])
  #...continues
```

Remark: c[i,j] contains the length of an LCS of X[:i] and Y[:j]

Time: O(mn)

Reporting the LCS in Python

```
#...continued
i,j = len(X),len(Y)
LCS = []
while c[i,j]:
  while c[i,j] == c[i-1,j]:
    i -= 1
  while c[i,j] == c[i,j-1]:
    j -= 1
  i -= 1
  j -= 1
  LCS.append(X[i])
LCS.reverse()
return LCS
```

Remark: append matches

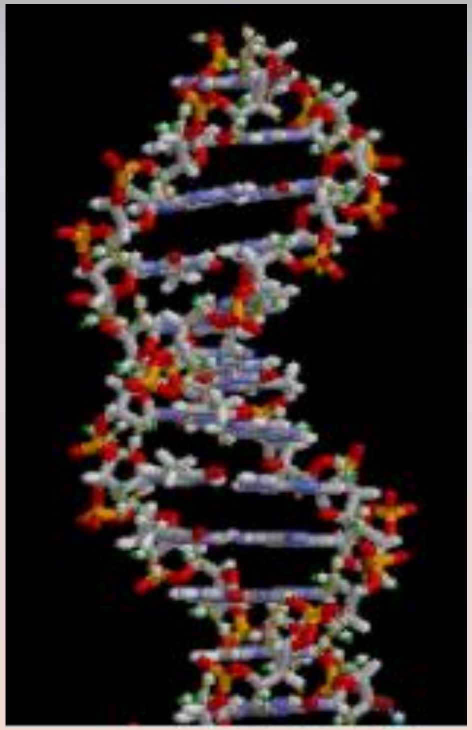
Time: O(m+n)

LCS algorithm

- Time complexity $O(nm)$
- Space complexity $O(nm)$
- Space can be reduced to linear by observing that we just need the previous row to compute the current row
- The length of the LCS can be computed easily in linear space

Sequence Alignment

- General Problem
  - Input
    - Strings S and T
  - Questions
    - How ***distant*** is S from T?
    - How ***similar*** is S to T?
- Solution Technique
  - Dynamic programming with cost/similarity/scoring matrix



Biological Motivation

In biomolecular sequences (DNA, RNA, amino acid sequences), high sequence similarity usually implies significant functional or structural similarity

AG-C	-AGC	AG-C	AGC-
AAAC	A-AAC	AAAC	AAAC

Alignment by Dynamic Programming

- Let w and y be two strings, $|w| = n, |y| = m$
- Define $V(i, j)$ as the value of the alignment of the strings $w_{[1 \dots i]}$ with $y_{[1 \dots j]}$
- The idea is to compute $V(i, j)$ for all values of $0 \leq i \leq n$ and $0 \leq j \leq m$
- In order to do that, we establish a recurrence relation between $V(i, j)$ and $V(i-1, j), V(i, j-1), V(i-1, j-1)$

Alignment by dynamic Programming

$$V(i,j) = max \begin{Bmatrix} V(i-1, j-1) + p(w_{[i]}, y_{[j]}) \\ V(i-1, j) + p(w_{[i]}, "-") \\ V(i, j-1) + p("-", y_{[j]}) \end{Bmatrix}$$

$$V(0,0) = 0$$
$$V(i,0) = V(i-1,0) + p\left(w_{[i]}, "-"\right)$$
$$V(0,j) = V(0, j-1) + p("-", y_{[j]})$$

Example

$p(a, a) = +1$ [match]
$p(a, b) = -1, if\ a \neq b$ [substitution]
$p(a, "-") = -2$ [deletion]
$p("-", a) = -2$ [insertion]

$$V(i,j) = max \begin{Bmatrix} V(i-1, j-1) + p(w_{[i]}, y_{[j]}) \\ V(i-1, j) + p(w_{[i]}, "-") \\ V(i, j-1) + p("-", y_{[j]}) \end{Bmatrix}$$

$V(0,0) = 0$
$V(i, 0) = V(i-1, 0) + p\left(w_{[i]}, "-"\right)$
$V(0, j) = V(0, j-1) + p("-", y_{[j]})$

		A	G	C
	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
A	-6	-3	-2	-1
C	-8	-5	-4	-1

Example

AG-C
AAC

		A	G	C
	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
A	-6	-3	-2	-1
C	-8	-5	-4	-1

Example

A-GC
AAAC

		A	G	C
	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
A	-6	-3	-2	-1
C	-8	-5	-4	-1

Example

-AGC
AAAC

		A	G	C
	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
A	-6	-3	-2	-1
C	-8	-5	-4	-1

Reading Assignment

- Counting combinations
- 0-1 Knapsack (6.4)
- Longest common subsequence
- Sequence Alignment

- Later in the course
  - Bellman-Ford (single source shortest path)
  - Floyd-Warshall (all pairs shortest path)