

# Dynamic Programming: Activity Selection

- Select the maximum number of non-overlapping activities from a set of  $n$  activities  $A = \{a_1, \dots, a_n\}$  (sorted by finish times).
- Identify "easier" subproblems to solve.

$$A_1 = \{a_1\}$$

$$A_2 = \{a_1, a_2\}$$

$$A_3 = \{a_1, a_2, a_3\}, \dots,$$

$$A_n = A$$

- Subproblems: Select the max number of non-overlapping activities from  $A_i$

# Dynamic Programming: Activity Selection

- Solving for  $A_n$  solves the original problem.
- Solving for  $A_1$  is easy.
- If you have optimal solutions  $S_1, \dots, S_{i-1}$  for subproblems on  $A_1, \dots, A_{i-1}$ , how to compute  $S_i$ ?
- The optimal solution for  $A_i$  either
  - Case 1: does not include  $a_i$  or
  - Case 2: includes  $a_i$
- Case 1:
  - $S_i = S_{i-1}$
- Case 2:
  - $S_i = S_k \cup \{a_i\}$ , for some  $k < i$ .
  - How to find such a  $k$ ? We know that  $a_k$  cannot overlap  $a_i$ .

# Dynamic Programming Features

- Identification of **subproblems**
- **Recurrence relation** for solution of subproblems
- **Overlapping** subproblems (sometimes)
- Identification of a **hierarchy/ordering** of subproblems
- Use of table to store solutions of subproblems (**MEMOIZATION**)
- Optimal Substructure

# Longest Common Subsequence

$S_1 = \text{CORIANDER}$

$\text{CORIANDER}$

$S_2 = \text{CREDITORS}$

$\text{CREDITORS}$

Longest Common Subsequence( $S_1[1..9]$ ,  $S_2[1..9]$ ) = CRIR

Subproblems:

- $\text{LCS}[S_1[a..b], S_2[c..d]]$ , for all  $a, b, c$ , and  $d$
- $\text{LCS}[S_1[1..i], S_2[1..j]]$ , for all  $i$  and  $j$  [BETTER]

• Recurrence Relation:

-  $\text{LCS}[i,j] = \text{LCS}[i-1, j-1] + 1$ , if  $S_1[i] = S_2[j]$

$\text{LCS}[i,j] = \max \{ \text{LCS}[i-1, j], \text{LCS}[i, j-1] \}$ , otherwise

- Table ( $m \times n$  table)
- Hierarchy of Solutions?

# LCS Problem

LCS\_Length (X, Y )

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{Length}[Y]$
3. for  $i = 1$  to  $m$
4. do  $c[i, 0] \leftarrow 0$
5. for  $j = 1$  to  $n$
6. do  $c[0, j] \leftarrow 0$
7. for  $i = 1$  to  $m$
8.     do for  $j = 1$  to  $n$
9.         do if (  $x_i = y_j$  )
10.             then  $c[i, j] \leftarrow c[i-1, j-1] + 1$
11.              $b[i, j] \leftarrow \text{"\}"}$
12.             else if  $c[i-1, j] < c[i, j-1]$
13.                 then  $c[i, j] \leftarrow c[i-1, j]$
14.                  $b[i, j] \leftarrow \text{"\^"}$
15.             else
16.                  $c[i, j] \leftarrow c[i, j-1]$
17.                  $b[i, j] \leftarrow \text{"\leftarrow"}$
18. return

# LCS Example

		H	A	B	I	T	A	T
	0	0	0	0	0	0	0	0
A	0	0↑	1↖	1←	1←	1←	1↖	1←
L	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
P	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
H	0	1↖	1↑	1↑	1↑	1↑	1↑	1↑
A	0	1↑	2↖	2←	2←	2←	2↖	2←
B	0	1↑	2↑	3↖	3←	3←	3←	3←
E	0	1↑	2↑	3↑	3↑	3↑	3↑	3↑
T	0	1↑	2↑	3↑	3↑	4↖	4←	4↖

# Dynamic Programming vs. Divide-&-conquer

- Divide-&-conquer works best when all subproblems are **independent**. So, pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem.
- Dynamic programming is needed when subproblems are **dependent**; we don't know where to partition the problem.  
For example, let  $S_1 = \{\text{ALPHABET}\}$ , and  $S_2 = \{\text{HABITAT}\}$ .  
Consider the subproblem with  $S_1' = \{\text{ALPH}\}$ ,  $S_2' = \{\text{HABI}\}$ .  
Then,  $LCS(S_1', S_2') + LCS(S_1 - S_1', S_2 - S_2') \neq LCS(S_1, S_2)$
- Divide-&-conquer is best suited for the case when no "overlapping subproblems" are encountered.
- In dynamic programming algorithms, we typically solve each subproblem only once and store their solutions. But this is at the cost of space.

# Dynamic programming vs Greedy

1. Dynamic Programming solves the sub-problems bottom up. The problem can't be solved until we find all solutions of sub-problems. The solution comes up when the whole problem appears.

Greedy solves the sub-problems from top down. We first need to find the greedy choice for a problem, then reduce the problem to a smaller one. The solution is obtained when the whole problem disappears.

2. Dynamic Programming has to try every possibility before solving the problem. It is much more expensive than greedy. However, there are some problems that greedy can not solve while dynamic programming can. Therefore, we first try greedy algorithm. If it fails then try dynamic programming.



# Fractional Knapsack Problem

- Burglar's choices:

Items:  $x_1, x_2, \dots, x_n$

Value:  $v_1, v_2, \dots, v_n$

Max Quantity:  $q_1, q_2, \dots, q_n$

Weight per unit quantity:  $w_1, w_2, \dots, w_n$

Getaway Truck has a weight limit of  $B$ .

Burglar can take "fractional" amount of any item.

How can burglar maximize value of the loot?

- Greedy Algorithm works!

Pick the maximum possible quantity of highest value per weight item. Continue until weight limit of truck is reached.

# 0-1 Knapsack Problem

- Burglar's choices:

Items:  $x_1, x_2, \dots, x_n$

Value:  $v_1, v_2, \dots, v_n$

Weight:  $w_1, w_2, \dots, w_n$

Getaway Truck has a weight limit of  $B$ .

Burglar cannot take "fractional" amount of item.

How can burglar maximize value of the loot?

- Greedy Algorithm does not work! Why?
- Need dynamic programming!

# 0-1 Knapsack Problem

- Subproblems?
  - $V[j, L]$  = Optimal solution for knapsack problem assuming a truck of weight limit  $L$  and choice of items from set  $\{1, 2, \dots, j\}$ .
  - $V[n, B]$  = Optimal solution for original problem
  - $V[1, L]$  = easy to compute for all values of  $L$ .
- Table of solutions?
  - $V[1..n, 1..B]$
- Ordering of subproblems?
  - Row-wise
- Recurrence Relation? [Either  $x_j$  included or not]
  - $V[j, L] = \max \{ V[j-1, L], v_j + V[j-1, L-w_j] \}$