Dynamic Programming

- Activity Problem Revisited: Given a set of \( n \) activities \( a_i = (s_i, f_i) \), we want to schedule the maximum number of non-overlapping activities.
- New Approach:
  - Observation: To solve the problem on activities \( A = \{a_1, ..., a_n\} \), we notice that either
    - optimal solution does not include \( a_n \)
      - then enough to solve subproblem on \( A_{n-1} = \{a_1, ..., a_{n-1}\} \)
    - optimal solution includes \( a_n \)
      - enough to solve subproblem on \( A_k = \{a_1, ..., a_k\} \), the set \( A \) without activities that overlap \( a_n \).

An efficient implementation

- Why not solve the subproblems on \( A_1, A_2, ..., A_{n-1}, A_n \) in that order?
- Is the problem on \( A_1 \) easy?
- Can the optimal solutions to the problems on \( A_1, ..., A_i \) help to solve the problem on \( A_{i+1} \)?
  - YES! Either:
    - optimal solution does not include \( a_{i+1} \)
      - problem on \( A_i \)
    - optimal solution includes \( a_{i+1} \)
      - problem on \( A_k = \{a_1, ..., a_k\} \) (equal to \( A_i \) without activities that overlap \( a_{i+1} \))
      - but this has already been solved according to our ordering.

Dynamic Programming: Activity Selection

- Select the maximum number of non-overlapping activities from a set of \( n \) activities \( A = \{a_1, ..., a_n\} \) (sorted by finish times).
- Identify "easier" subproblems to solve.
  \( A_1 = \{a_1\} \)
  \( A_2 = \{a_1, a_2\} \)
  \( A_3 = \{a_1, a_2, a_3\}, ..., A_n = A \)
- Subproblems: Select the max number of non-overlapping activities from \( A_i \)
Dynamic Programming: Activity Selection

- Solving for \( A_i \) solves the original problem.
- Solving for \( A_1 \) is easy.
- If you have optimal solutions \( S_1, \ldots, S_{i-1} \) for subproblems on \( A_1, \ldots, A_{i-1} \), 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:
  \[ S_i = S_k \cup \{ a_i \}, \text{ for some } k < i. \]
  - How to find such a \( k \)? We know that \( a_k \) cannot overlap \( a_i \).

Dynamic Programming: Activity Selection

```plaintext
DP-ACTIVITY-SELECTOR (s, f)
1. n = length[s]
2. N[1] = 1        // number of activities in \( S_1 \)
3. F[1] = 1        // last activity in \( S_1 \)
4. for i = 2 to n do
   5. let k be the last activity finished before \( s_i \)
   6. if \( N[i-1] > N[k] \) then // Case 1
      7. \( N[i] = N[i-1] \)
      8. \( F[i] = F[i-1] \)
   9. else // Case 2
      10. \( N[i] = N[k] + 1 \)
      11. \( F[i] = i \)

How to output \( S_n \)?
Backtrack!

Time Complexity?
\( O(n \lg n) \)
```

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

\[ S_1 = \text{CORIANDER} \]
\[ S_2 = \text{CREDITORS} \]

Longest Common Subsequence \((S_1[1..9], S_2[1..9]) = \text{CRIR}\)

Subproblems:
- \(\text{LCS}[S_1[1..i], S_2[1..j]]\), 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

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

LCS Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>I</th>
<th>T</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>I</th>
<th>T</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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, $\text{LCS}(S_1', S_2') + \text{LCS}(S_1-S_1', S_2-S_2') \neq \text{LCS}(S_1, S_2)$
- 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.