

15.4 Longest common subsequence (LCS) problem

- Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a **subsequence** of X if there exists a **strictly increasing sequence** $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \dots, k$ we have $x_{i_j} = z_j$.

$\langle A, C, B, B, C, W \rangle$ is a subsequence of $\langle T, A, C, B, B, W, B, C, W, T, W \rangle$.

- Given two sequences X and Y , a sequence Z is called **common subsequence** of X and Y if Z is a subsequence of both X and Y .

$\langle A, C, B, B, C, W \rangle$ is a common subsequence of $\langle T, A, C, B, B, W, B, C, W, T, W \rangle$ and $\langle A, A, B, C, B, W, B, C, A, A, W, T \rangle$.

- The **longest-common-subsequence problem** is to find such a Z that has the maximum length.

15.4 Longest common subsequence problem (continued)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$.

Our task is to find a LCS of X and Y .

Key observation

- If $x_m = y_n$, we can assume that the LCS includes x_m and y_n . So, it consists of an LCS of $\langle x_1, x_2, \dots, x_{m-1} \rangle$ and $\langle y_1, y_2, \dots, y_{n-1} \rangle$.
- If $x_m \neq y_n$, the LCS cannot include both x_m and y_n . So, either it is an LCS of $\langle x_1, x_2, \dots, x_{m-1} \rangle$ and $\langle y_1, y_2, \dots, y_n \rangle$, or an LCS of $\langle x_1, x_2, \dots, x_m \rangle$ and $\langle y_1, y_2, \dots, y_{n-1} \rangle$ (or both).

15.4 Longest common subsequence problem (continued)

Let $c[i, j]$ be the length of an LCS of $\langle x_1, x_2, \dots, x_i \rangle$ and $Y = \langle y_1, y_2, \dots, y_j \rangle$.

Using the key observations above,

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

Compute using increasing i, j until $c[m, n]$ is obtained. Row by row or column by column orders are ok.

As usual, keeping track of which option provided the optimum at each step allows us to work backwards from the answer to find the actual subsequence with length $c[m, n]$.

15.4 Longest common subsequence problem (continued)

Example: $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B, A \rangle$.

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

Issues:

- This is a special case of longest path in an acyclic digraph.
- How much space is required?

15.4 Longest common subsequence problem (continued)

An array b can be used to record which case yielded the optimum at every step.

LCS_Length(X, Y)

```
1  for  $i \leftarrow 1$  to  $m$  do  $c[i, 0] \leftarrow 0$ 
2  for  $j \leftarrow 1$  to  $n$  do  $c[0, j] \leftarrow 0$ 
3  for  $i \leftarrow 1$  to  $m$ 
4  do for  $j \leftarrow 1$  to  $n$ 
5      do if  $x_i = y_j$ 
6          then  $c[i, j] \leftarrow c[i-1, j-1] + 1$ ;  $b[i, j] \leftarrow "\backslash"$ 
7          else if  $c[i-1, j] \geq c[i, j-1]$ 
8              then  $c[i, j] \leftarrow c[i-1, j]$ ;  $b[i, j] \leftarrow "\uparrow"$ 
9              else  $c[i, j] \leftarrow c[i, j-1]$ ;  $b[i, j] \leftarrow "\leftarrow"$ 
```

To find the LCS follow the path in b back from $b[m, n]$.

16.1. Incompatible task scheduling

Suppose there are n tasks T_1, T_2, \dots, T_n , where task T_i must start at time s_i and finish a time f_i ($s_i \leq t_i$).

No two tasks can be performed at the same time. That is, task T_i and T_j are incompatible if their time intervals (s_i, f_i) and (s_j, f_j) overlap.

The problem is to perform as many tasks as possible.

First attempt at greedy solution:

Repeatedly choose the earliest-starting task that is compatible with previously chosen tasks. **This doesn't work.**

Second attempt at greedy solution:

Repeatedly choose the earliest-finishing task that is compatible with previously chosen tasks. **This works!**

Chapter 16. Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.

A **greedy algorithm** always makes the choice that looks the best at the moment. That is, it makes a **locally optimal choice** in the **hope** that this choice will lead to a **globally optimal solution**. For some optimization problems, the greedy algorithm does not yield optimal solutions but for many problems, it does.

Examples of greedy algorithms that do not work:

- ▶ Shortest path through layered network: Construct a path by always adding an edge of shortest length.
- ▶ Matrix multiplication chain: Repeatedly do the cheapest of the available multiplications.

16.1. Incompatible task scheduling (continued)

Theorem. *This greedy solution is optimal:*

Repeatedly choose the earliest-finishing task that is compatible with previously chosen tasks.

Proof.

Let $S_1, S_2, S_3, \dots, S_k$ be any solution.

Let G_1, G_2, G_3, \dots be the greedy solution.

(In each case, a sequence of compatible tasks in the order of running.)

According to the greedy rule, G_1 finishes no later than S_1 . Therefore, S_2 is compatible with G_1 , so G_2 finishes no later than S_2 . And so on.

In general, for $1 \leq i < k$, G_i finishes no later than S_i and so S_{i+1} is compatible with G_1, \dots, G_i . Therefore the greedy solution can be continued for another task G_{i+1} which finishes no later than S_{i+1} .

16.1. Incompatible task scheduling (continued)

Therefore the greedy solution has at least as many tasks as any other solution. Therefore, it must have the maximum possible number of tasks. (We have also proved that the greedy solution finishes earliest of all the optimal solutions.)

To implement the algorithm, sort the tasks in order of finishing time. Then for each task in the list, execute it if its starting time has not already past, and wait until it is finished before continuing. Time: $O(n \lg n)$ for sorting, $O(n)$ for the rest.

Greedy_CPU_Scheduling(s, f)

```
1   $A \leftarrow \{T_1\}$ 
2   $j \leftarrow 1$ 
3  for  $i \leftarrow 2$  to  $n$ 
4      do if  $s_i \geq f_j$ 
5          then  $A \leftarrow A \cup \{T_i\}$ 
6               $j \leftarrow i$ 
7  return  $A$ 
```