Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 18

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.:*

$$X = \langle A, B, C, B, D, A, B \rangle$$

- Subsequence of X:
  - A subset of elements in the sequence taken in order (but not necessarily consecutive)

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

Recursive Solution

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

b & c:

i		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					

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] = " ←"

Improving the Code

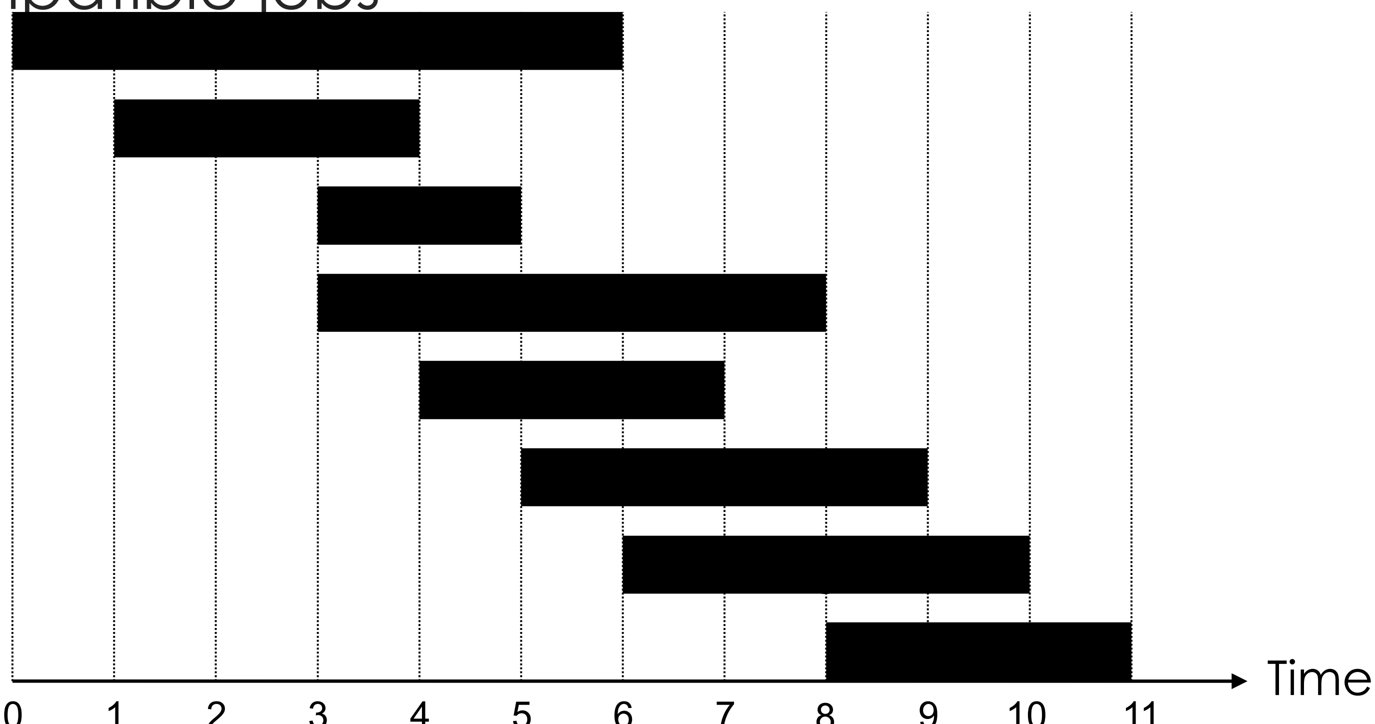
- What can we say about how each entry **c[i, j]** is computed?
  - It depends only on **c[i -1, j - 1]**, **c[i - 1, j]**, and **c[i, j - 1]**
  - Eliminate table **b** and compute in **O(1)** which of the three values was used to compute **c[i, j]**
  - We save **Θ(mn)** space from table **b**
  - However, we do not asymptotically decrease the auxiliary space requirements: still need table **c**

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

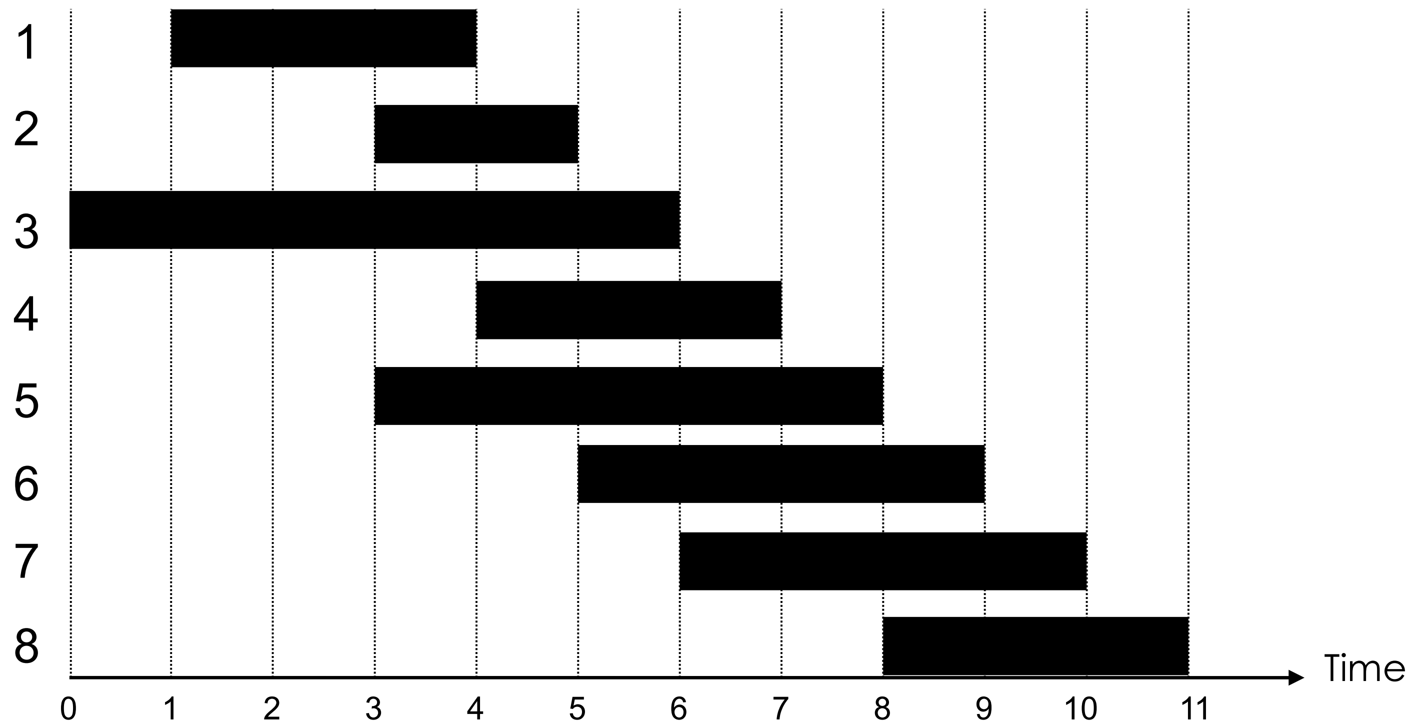
Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has weight or value v_j
- Two jobs are **compatible** if they don't overlap
- Goal: find maximum **weight** subset of mutually compatible jobs



Weighted Interval Scheduling

- Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- Def. p(j) = largest index i < j such that job i is compatible with j
- Ex: p(8) = 5, p(7) = 3, p(2) = 0



1. Making the Choice

- OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j
  - Case 1: OPT selects job j
    - Can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

  - Case 2: OPT does not select job j
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

2. A Recursive Solution

- OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j
  - Case 1: OPT selects job j
    - Can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)
    - OPT(j) = v_j + OPT(p(j))
  - Case 2: OPT does not select job j
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1
    - OPT(i) = OPT(j-1)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Top-Down Recursive Algorithm

WRONG!

Input: n, $s_1,\ldots,s_n$, $f_1,\ldots,f_n$, $v_1,\ldots,v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$

Compute p(1), p(2), …, p(n)

```
Compute-Opt(j)
{
  if (j = 0)
    return 0
  else
    return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

3. Compute the Optimal Value

- Compute values in increasing order of j

Input: n, $s_1,\ldots,s_n$, $f_1,\ldots,f_n$, $v_1,\ldots,v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$

Compute p(1), p(2), …, p(n)

```
Iterative-Compute-Opt
{
  M[0] = 0
  for j = 1 to n
    M[j] = max(v_j + M[p(j)], M[j-1])
}
```

Memoized Version

- Store results of each sub-problem; lookup as needed

Input: $n, s_1,\ldots,s_n, f_1,\ldots,f_n, v_1,\ldots,v_n$
Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.
Compute p(1), p(2), …, p(n)

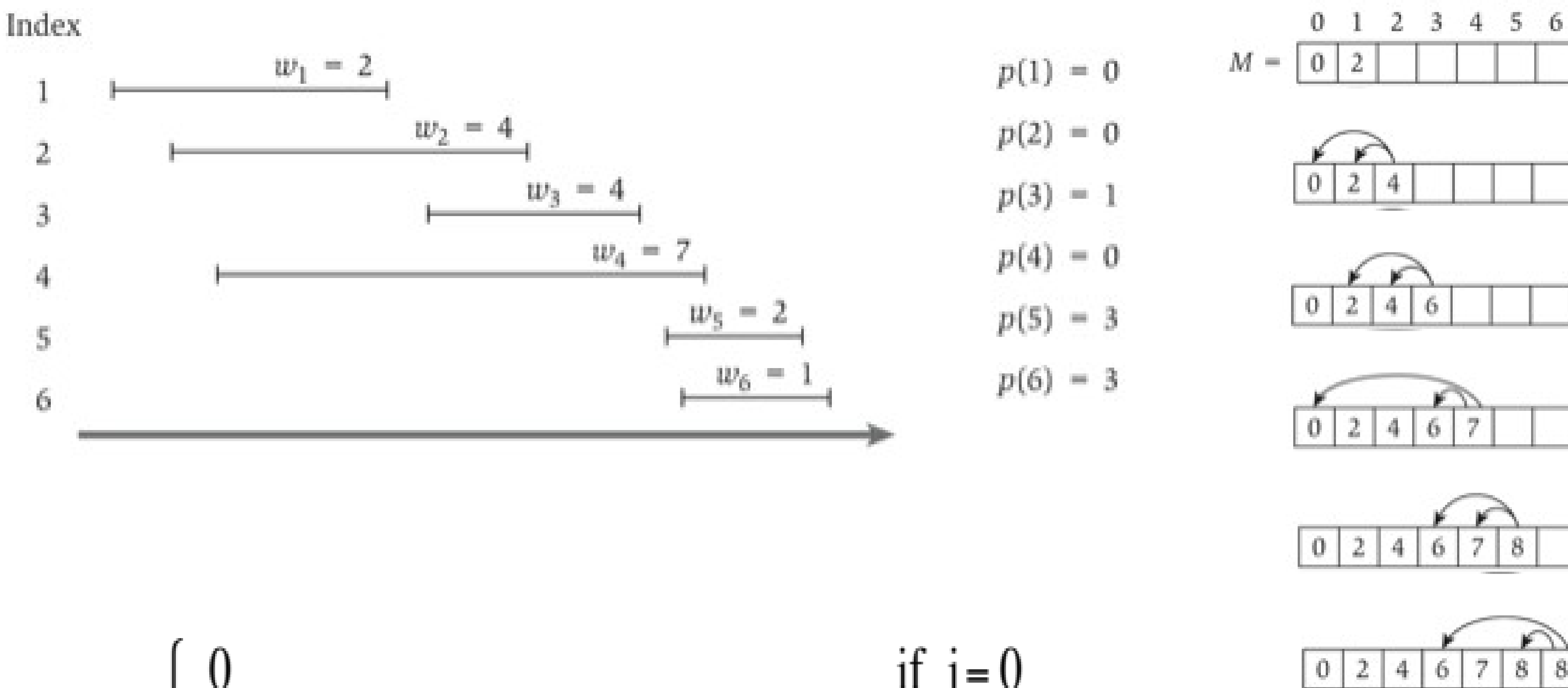
```
for j = 1 to n
   M[j] = empty          ← global array
M[j] = 0

M-Compute-Opt(j)
{
   if (M[j] is empty)
      M[j] = max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

4. Finding the Optimal Solution

- Two options
  1. Store additional information: at each time step store either j or p(j) – value that gave the optimal solution
  2. Recursively find the solution by iterating through array M

```
Find-Solution(j)
{
  if (j = 0)
    output nothing
  else if (v_j + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```

An Example

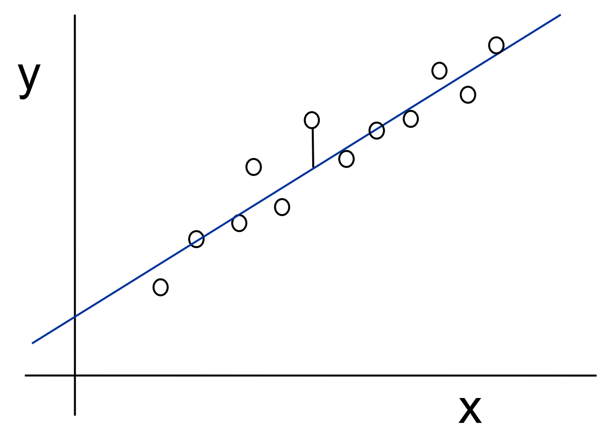


$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\, v_j + OPT(p(j)),\ OPT(j-1) \,\} & \text{otherwise} \end{cases}$$

Segmented Least Squares

- Least squares
  - Foundational problem in statistic and numerical analysis
  - Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
  - Find a line y = ax + b that minimizes the sum of the squared error:

$$Error = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



- Solution – closed form
  - Minimum error is achieved when

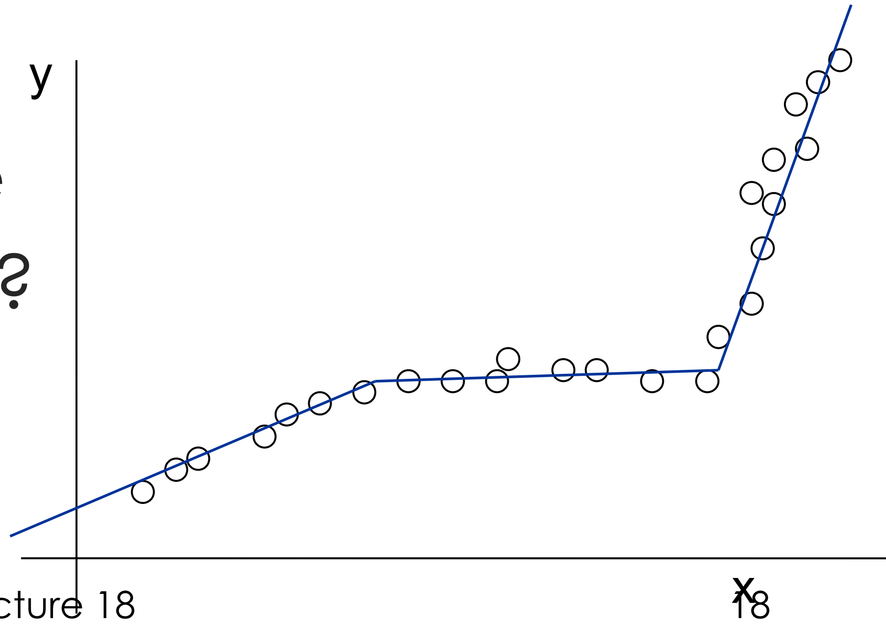
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Segmented least squares
  - Points lie roughly on a sequence of several line segments
  - Given n points in the plane (x_1, y_1), (x_2, y_2) , . . . , (x_n, y_n) with $x_1 < x_2 < ... < x_n$, find a sequence of lines that minimizes f(x)
- What is a reasonable choice for f(x) to balance accuracy and parsimony?

goodness of fit

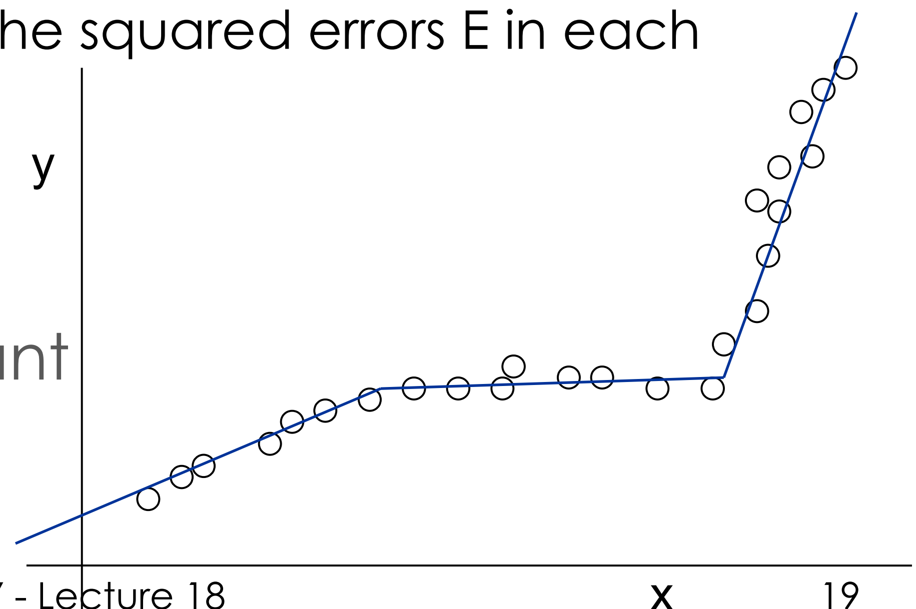
number of lines



Segmented Least Squares

- Segmented least squares
  - Points lie roughly on a sequence of several line segments
  - Given n points in the plane (x_1, y_1), (x_2, y_2) , . . . , (x_n, y_n) with $x_1 < x_2 < ... < x_n$, find a sequence of lines that minimizes:
    - the sum of the sums of the squared errors E in each segment
    - the number of lines L
- Tradeoff function:

  E + c L, for some constant c > 0



(1,2) Making the Choice and Recursive Solution

- Notation
  - OPT(j) = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$
  - e(i, j) = minimum sum of squares for points $p_i, p_{i+1}, .., p_j$
- To compute OPT(j)
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i
  - Cost = e(i, j) + c + OPT(i-1)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ \ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

3. Compute the Optimal Value

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

- Running time: $O(n^3)$
  - Bottleneck = computing e(i, j) for $O(n^2)$ pairs, $O(n)$ per pair using previous formula

Greedy Algorithms

- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems
- **Idea:** When we have a choice to make, make the one that looks best right now
  - Make a locally optimal choice in the hope of getting a globally optimal solution
- Greedy algorithms don't always yield an optimal solution
- When the **problem** has certain general characteristics (**greedy choice property**), greedy algorithms give optimal solutions

Activity Selection

- Problem
  - Schedule the largest possible set of non-overlapping activities for a given room

	Start	End	Activity
1	8:00am	9:15am	Numerical methods class
2	8:30am	10:30am	Movie presentation (refreshments served)
3	9:20am	11:00am	Data structures class
4	10:00am	noon	Programming club mtg. (Pizza provided)
5	11:30am	1:00pm	Computer graphics class
6	1:05pm	2:15pm	Analysis of algorithms class
7	2:30pm	3:00pm	Computer security class
8	noon	4:00pm	Computer games contest (refreshments served)
9	4:00pm	5:30pm	Operating systems class

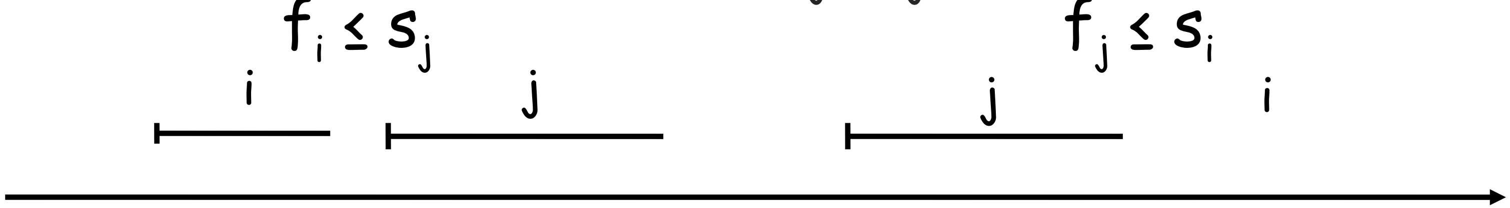
Activity Selection

- Schedule n **activities** that require exclusive use of a common resource

  $S = \{a_1, \ldots, a_n\}$ – set of activities

- a_i needs resource during period $[s_i, f_i)$
  - s_i = **start time** and f_i = **finish time** of activity a_i
  - $0 \le s_i < f_i < \infty$

- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap



Activity Selection Problem

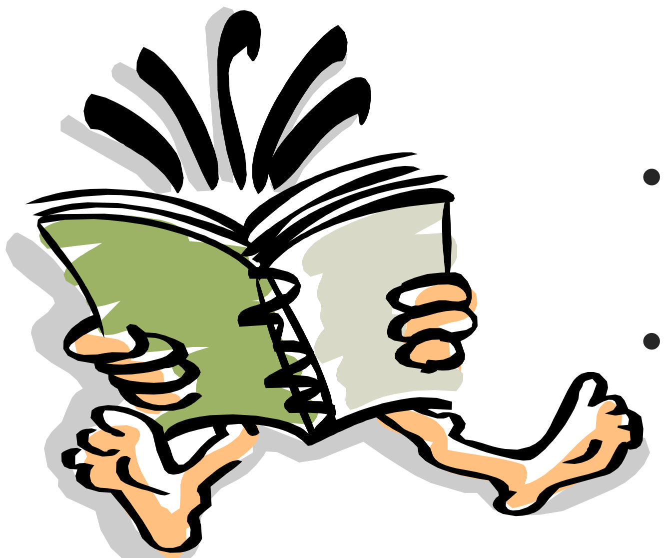
Select the largest possible set of non-overlapping (***compatible***) activities.

E.g.:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Activities are sorted in increasing order of finish times
- A subset of mutually compatible activities: $\{a_3, a_9, a_{11}\}$
- Maximal set of mutually compatible activities: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$

Readings



- For this lecture
  - Chapter 14
- Coming next
  - Chapter 14