南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Algorithm Design and Analysis (H)

CS216

Prof. Shiqi Yu (于仕琪)

yusq@sustech.edu.cn

http://faculty.sustech.edu.cn/yusq/

Assignment 1

A better assignment/project

The power of $O(K^n)$

Understand the resource demanded in fight against COVID-19.

Dynamic Programming

Algorithmic Paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

- Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming History

- Bellman. [1950s] Pioneered the systematic study of dynamic programming.

- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.

> "it's impossible to use dynamic in a pejorative sense"
> "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

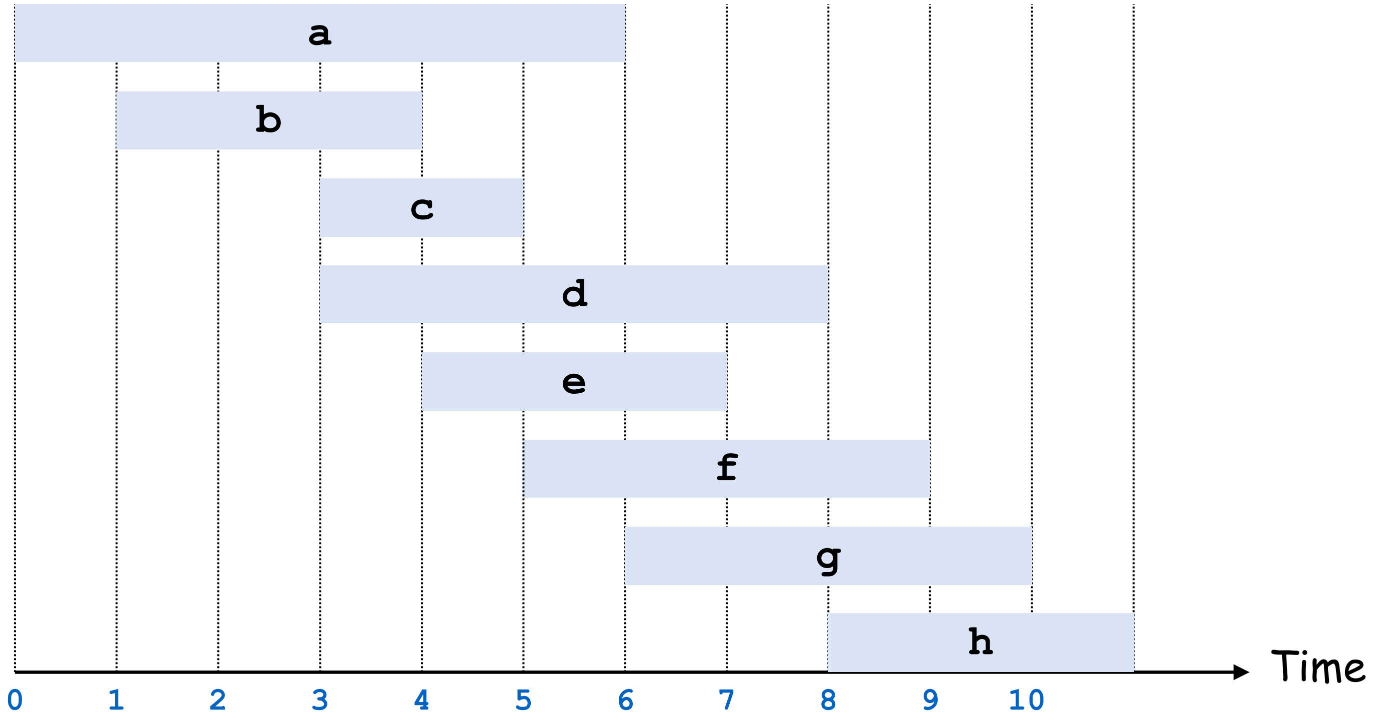
- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems, ….

- Some famous dynamic programming algorithms.
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models.
  - Smith-Waterman for genetic sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

1. Weighted Interval Scheduling

Weighted Interval Scheduling

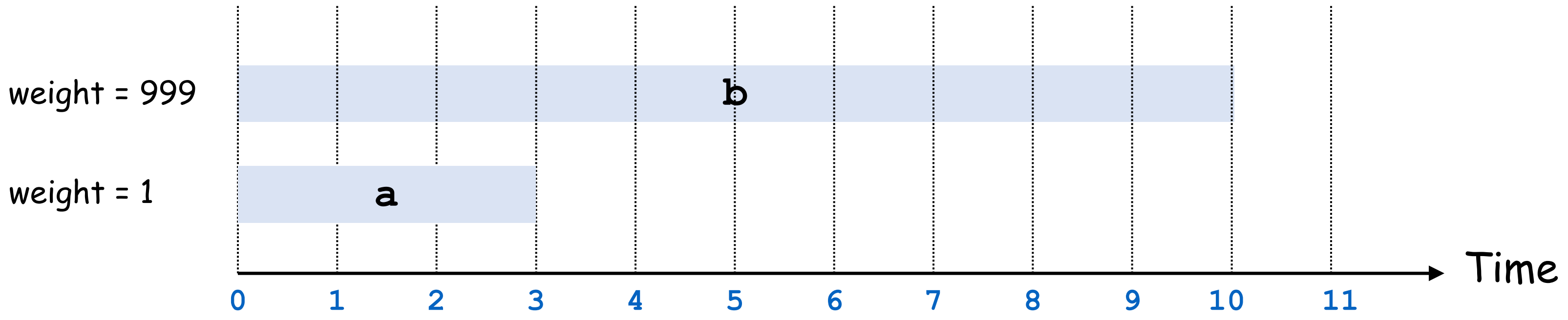
- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



Unweighted Interval Scheduling Review

- Recall. Greedy algorithm works if all weights are 1.
  - Consider jobs in ascending order of finish time.
  - Add job to subset if it is compatible with previously chosen jobs.

- Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

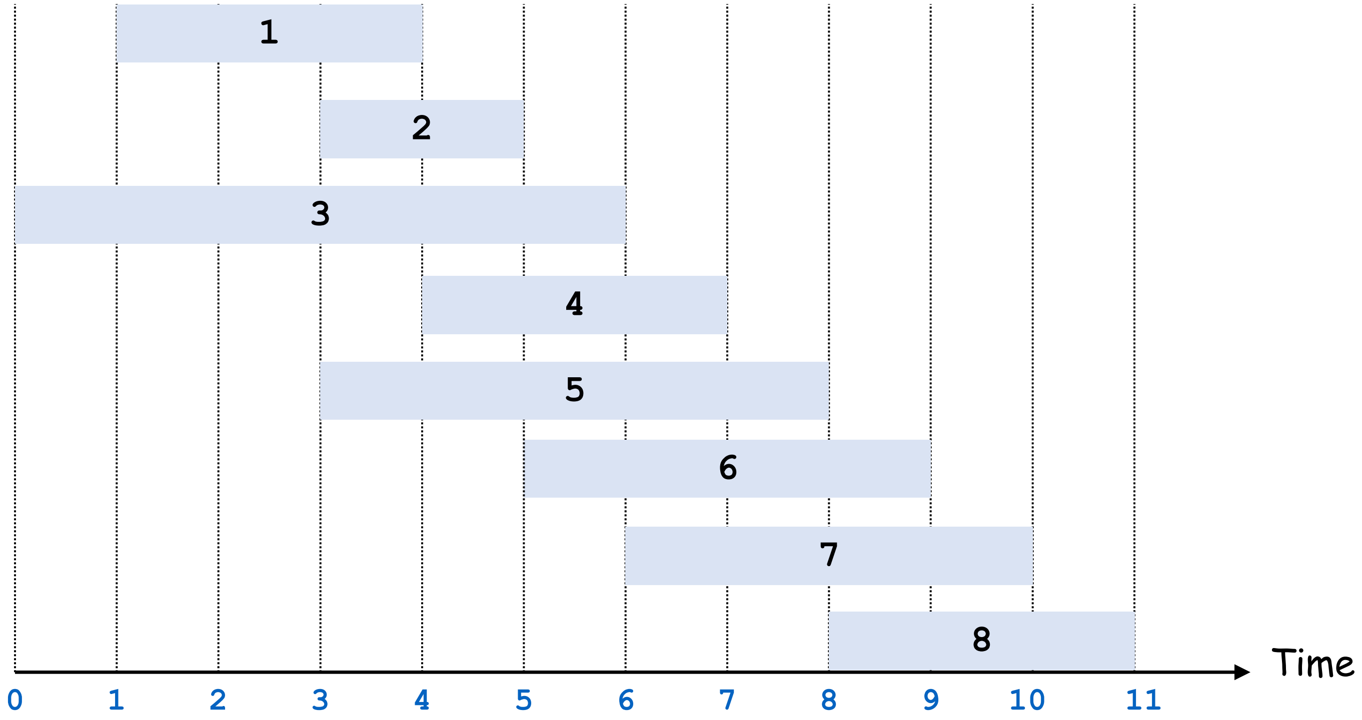


Weighted Interval Scheduling

Notation. 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.



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - collect profit v_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

  - $\mathrm{OPT}(j) = \max(v_j + OPT\big(p(j)\big), OPT(j-1))$

Weighted Interval Scheduling: Brute Force

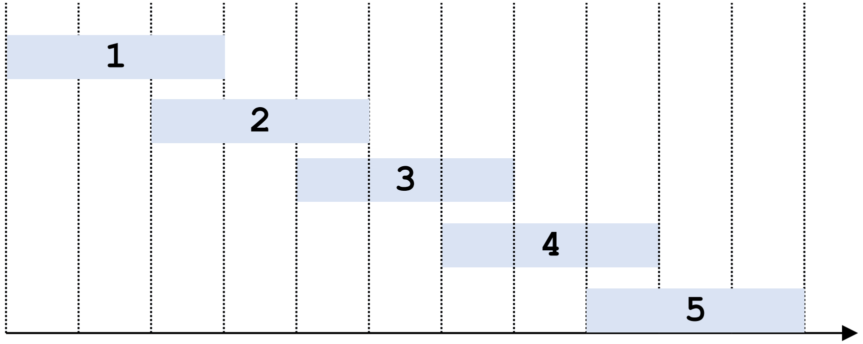
- Brute force algorithm.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

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

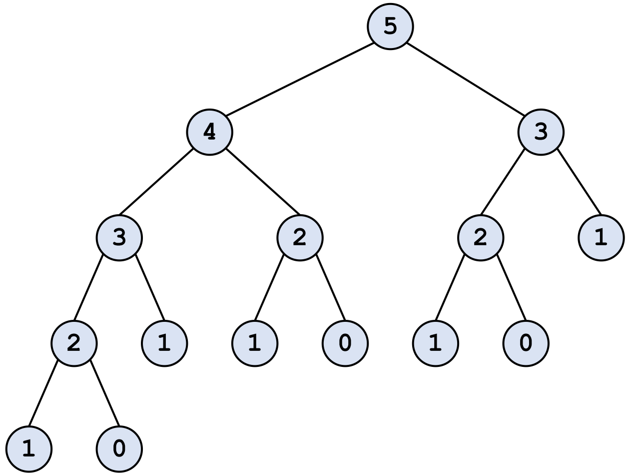
Weighted Interval Scheduling: Brute Force

- Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

- Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

• Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

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

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

Weighted Interval Scheduling: Running Time

- **Claim.** Memorized version of algorithm takes O(n log n) time.
  - Sort by finish time: O(n log n).
  - Computing $p(\cdot)$: O(n log n) via sorting by start time.

  - `M-Compute-Opt(j)`: each invocation takes O(1) time and either
    - ✓ (i) returns an existing value `M[j]`
    - ✓ (ii) fills in one new entry `M[j]` and makes two recursive calls

  - Progress measure Φ = # nonempty entries of `M[]`.
    - ✓ initially $\Phi = 0$, throughout $\Phi \le n$.
    - ✓ (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.

  - Overall running time of `M-Compute-Opt(n)` is O(n). ▪

- **Remark.** O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

- Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
- A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

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

➢ # of recursive calls $\leq$ n $\Rightarrow$ O(n).

Weighted Interval Scheduling: Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

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

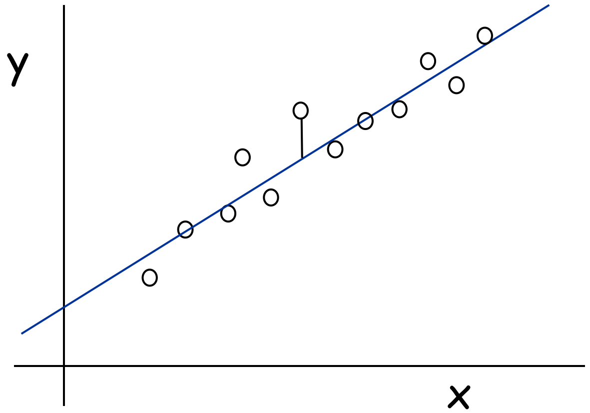
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

2. Segmented Least Squares

Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane P: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line L: $y = ax + b$ that minimizes the sum of the squared error:
- $\text{Error}(\text{L}, \text{P}) = \sum_{i=1}^{n} (y_i - a x_i - b)^2$



Solution. Nice closed-form solution $\Rightarrow$ min error is achieved when

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

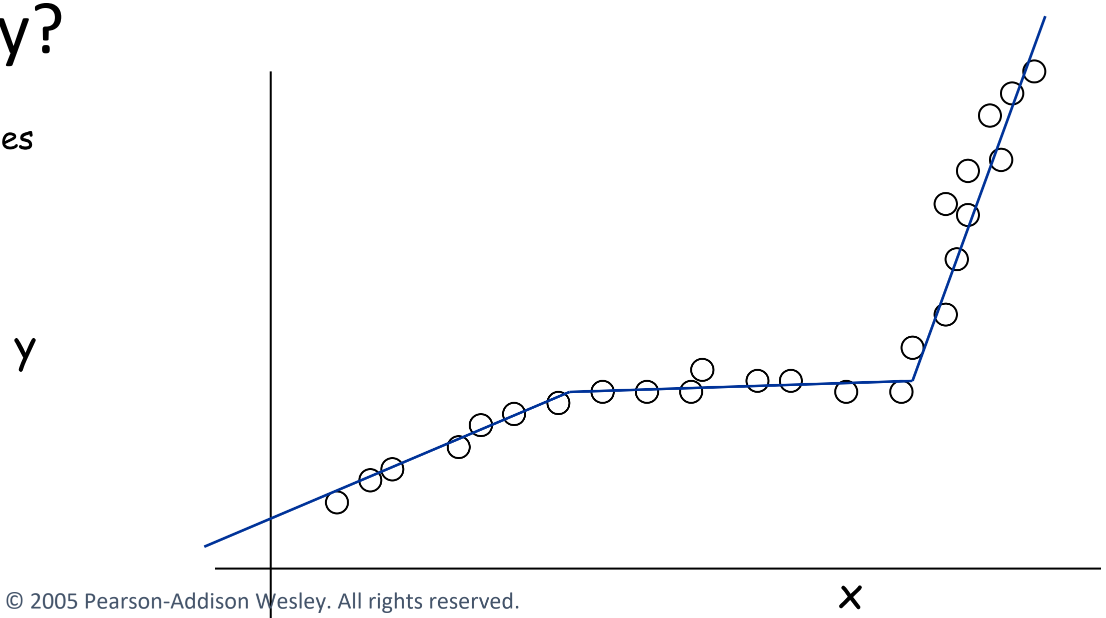
Segmented Least Squares

- Segmented least squares.
  - Points lie roughly on a sequence of several line segments.
  - Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
  - $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

goodness of fit

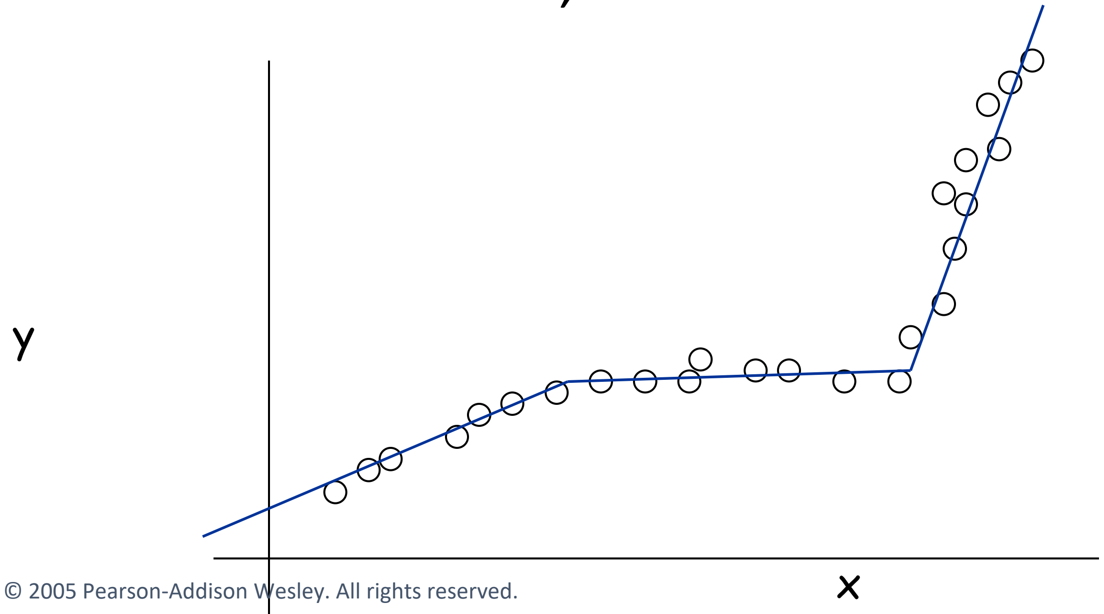
- Q. What's a reasonable choice for f(x) to balance accuracy and parsimony?

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 $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
  - $x_1 < x_2 < \ldots < x_n$, 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 $c > 0$.



Dynamic Programming: Multiway Choice

- Notation.
  - $OPT(j)$ = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
  - $e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

- To compute $OPT(j)$:
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
  - Cost = $e(i, j) + c + OPT(i-1)$.

Segmented Least Squares: Algorithm

```
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]
}
```

can be improved to $O(n^2)$ by pre-computing various statistics

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