CSE 417, Winter 2012

Dynamic Programming

Ben Birnbaum

Widad Machmouchi

Slides adapted from Larry Ruzzo, Steve Tanimoto, and Kevin Wayne

Dynamic Programming

Outline:

- General Principles
- Easy Examples – Fibonacci, Licking Stamps
- Meatier examples
  - Weighted interval scheduling
  - And others

Some Algorithm Design Techniques, I

- General overall idea
  - Reduce solving a problem to a smaller problem or problems of the same type
- Greedy algorithms
  - Used when one needs to build something a piece at a time
  - Repeatedly make the *greedy* choice - the one that looks the best right away
  - Usually fast if they work (but often don't)

Some Algorithm Design Techniques, II

Divide & Conquer

- Reduce problem to one or more sub-problems of the same type
- Typically, each sub-problem is at most a constant fraction of the size of the original problem
  - e.g. Mergesort, Binary Search, Strassen's Algorithm, Quicksort (kind of)

Some Algorithm Design Techniques, III

Dynamic Programming

- Give a solution of a problem using smaller sub-problems, e.g. a recursive solution
- Useful when the same sub-problems show up again and again in the solution

Dynamic Programming History

Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

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.*

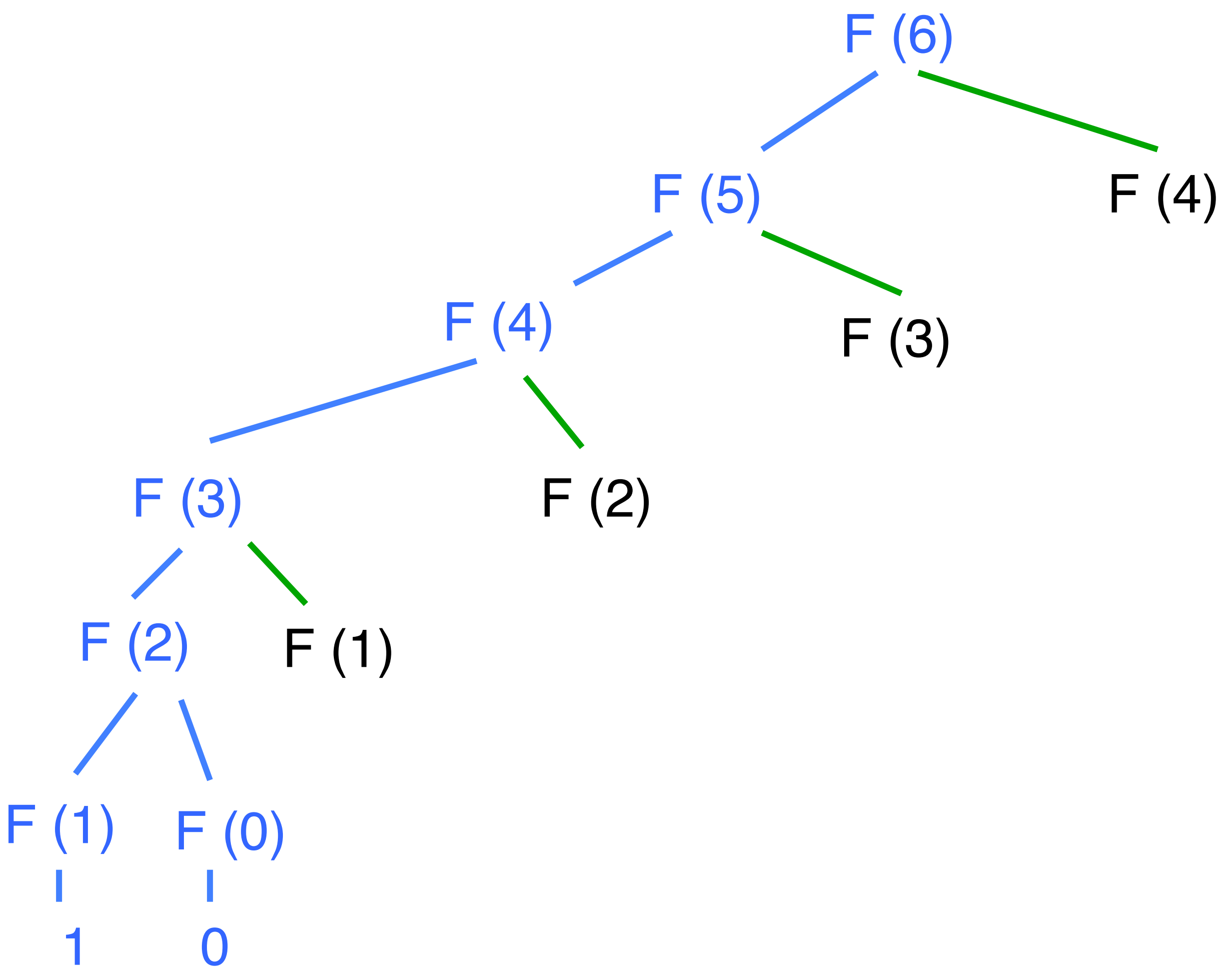
A very simple case: Computing Fibonacci Numbers

Recall $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0$, $F_1 = 1$

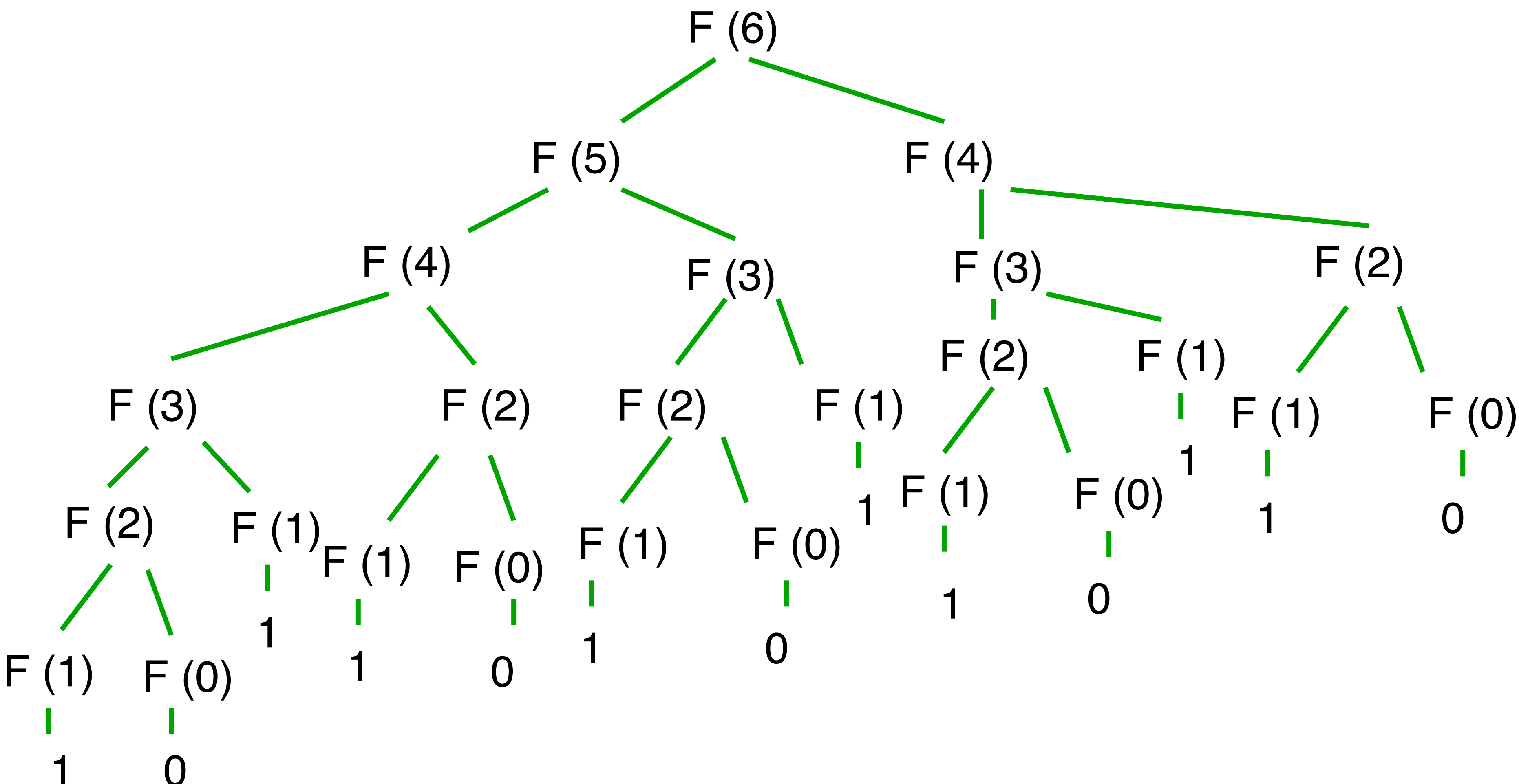
Recursive algorithm:

```
Fibo(n)
    if n=0 then return(0)
    else if n=1 then return(1)
    else return(Fibo(n-1)+Fibo(n-2))
```

Call tree - start



Full call tree



Memo-ization (Caching)

Save all answers from earlier recursive calls

Before recursive call, test to see if value has already been computed

Dynamic Programming

- *NOT* memoized; instead, convert memoized alg from a recursive one to an iterative one (top-down → bottom-up)

Fibonacci - Memoized Version

```
initialize: F[i] ← undefined for all i
F[0] ← 0
F[1] ← 1
FiboMemo(n):
    if(F[n] undefined) {
        F[n] ← FiboMemo(n-2)+FiboMemo(n-1)
    }
    return(F[n])
```

Fibonacci - Dynamic Programming Version

```
FiboDP(n):
    F[0] ← 0
    F[1] ← 1
    for i=2 to n do
        F[i]  ← F[i-1]+F[i-2]
    end
    return(F[n])
```

For this problem, keeping only last 2 entries instead of full array suffices, but about the same speed

Dynamic Programming

Useful when

- Same recursive sub-problems occur *repeatedly*
- Parameters of these recursive calls anticipated
- The solution to whole problem can be solved without knowing the *internal* details of how the sub-problems are solved
  - "principle of optimality"

Making change

Given:

- Large supply of 1¢, 5¢, 10¢, 25¢, 50¢ coins
- An amount N

Problem: choose fewest coins totaling N

Cashier's (greedy) algorithm works:

- Give as many as possible of the next biggest denomination

Licking Stamps

Given:

- Large supply of 5¢, 4¢, and 1¢ stamps
- An amount N

Problem: choose fewest stamps totaling N

How to Lick 27¢

# of 5¢ stamps	# of 4 ¢ stamps	# of 1¢ stamps	total number
5	0	2	7
4	1	3	8
3	3	0	6

Morals: Greed doesn't pay; success of "cashier's alg" depends on coin denominations

Better Idea

Theorem: If last stamp in an opt sol has value v, then previous stamps are opt sol for N-v.

Proof: if not, we could improve the solution for N by using opt for N-v.

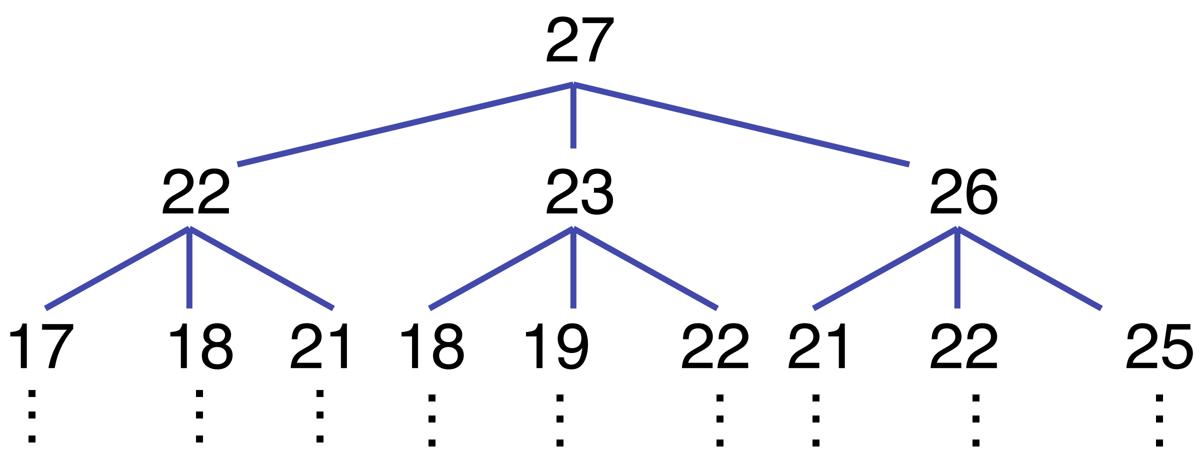
Alg: for i = 1 to n:

$$M(i) = \min \begin{Bmatrix} 0 & i=0 \\ 1+M(i-5) & i \geq 5 \\ 1+M(i-4) & i \geq 4 \\ 1+M(i-1) & i \geq 1 \end{Bmatrix}$$

where *M(i)* = min number of stamps totaling *i*¢

New Idea: Recursion

$$M(i) = \min \left\{ \begin{array}{ll} 0 & i=0 \\ 1+M(i-5) & i\geq 5 \\ 1+M(i-4) & i\geq 4 \\ 1+M(i-1) & i\geq 1 \end{array} \right\}$$



Time: $> 3^{N/5}$

Another New Idea: Avoid Recomputation

Tabulate values of solved subproblems

- Top-down: "memoization"
- Bottom up:

for i = 0, …, N do $M[i] = \min \begin{Bmatrix} 0 & i=0 \\ 1+M[i-5] & i \geq 5 \\ 1+M[i-4] & i \geq 4 \\ 1+M[i-1] & i \geq 1 \end{Bmatrix}$

Time: O(N)

Finding *How Many* Stamps

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M(i)	0	1	2	3	1	1	2	3	2						

1+Min(3,1,3) = 2

Finding *Which* Stamps: Trace-Back

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M(i)	0	1	2	3	1	1	2	3	2						

4¢

<u>1</u>+Min(3,**<u>1</u>**,3) = **<u>2</u>**

Trace-Back

Way 1: tabulate all

add data structure storing back-pointers indicating which predecessor gave the min. (more space, maybe less time)

Way 2: re-compute just what's needed

```
TraceBack(i):
  if i == 0 then return;
  for d in {1, 4, 5} do
    if M[i] == 1 + M[i - d]
      then break;
  print d;
  TraceBack(i - d);
```

$$M[i] = \min \begin{Bmatrix} 0 & i=0 \\ 1+M[i-5] & i\geq 5 \\ 1+M[i-4] & i\geq 4 \\ 1+M[i-1] & i\geq 1 \end{Bmatrix}$$

Elements of Dynamic Programming

What feature did we use?

What should we look for to use again?

"Optimal Substructure"

Optimal solution contains optimal subproblems

"Repeated Subproblems"

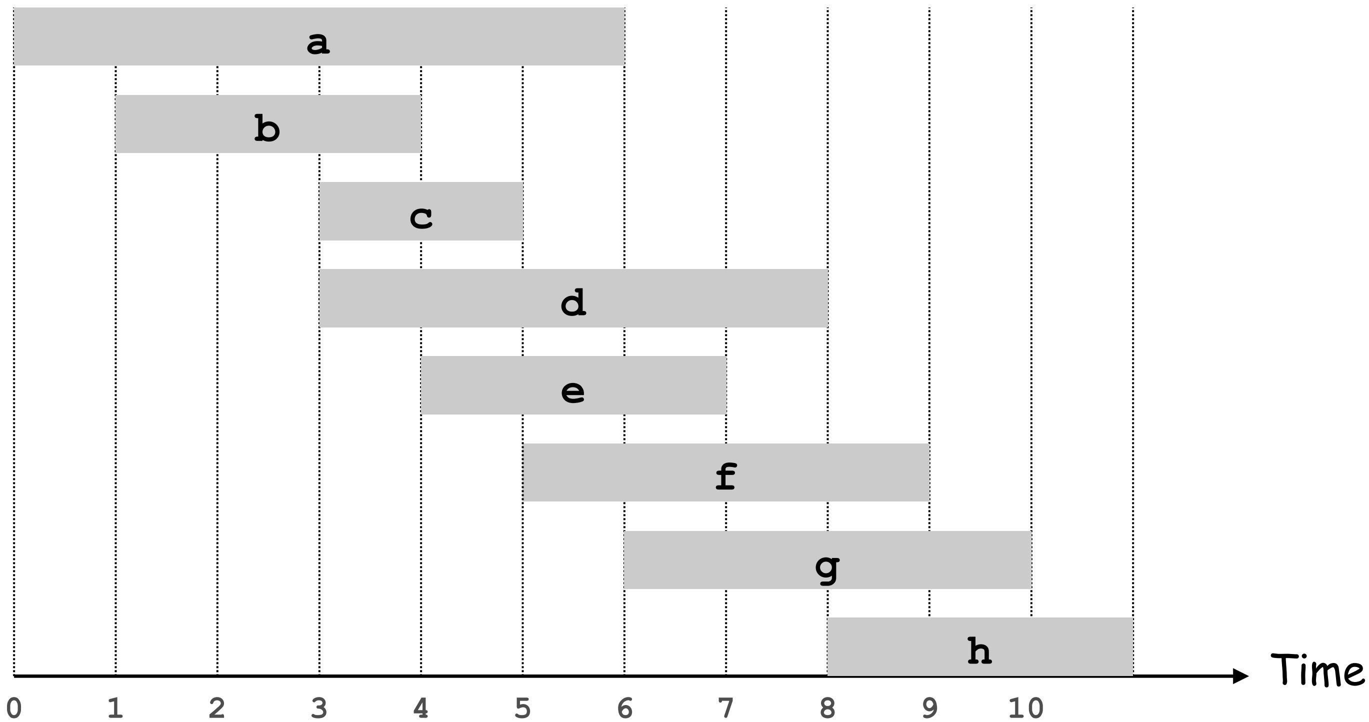
The same subproblems arise in various ways

6.1 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.

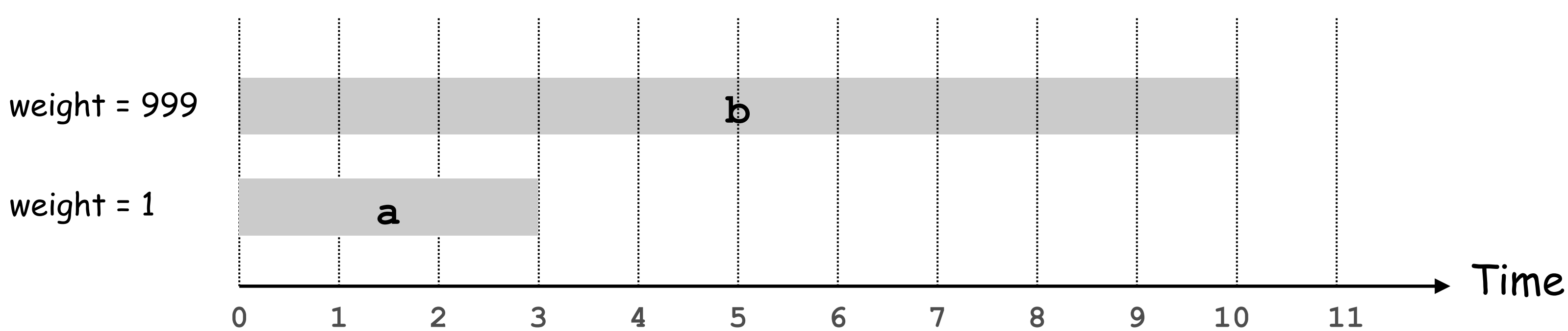


Unweighted Interval Scheduling Review

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.

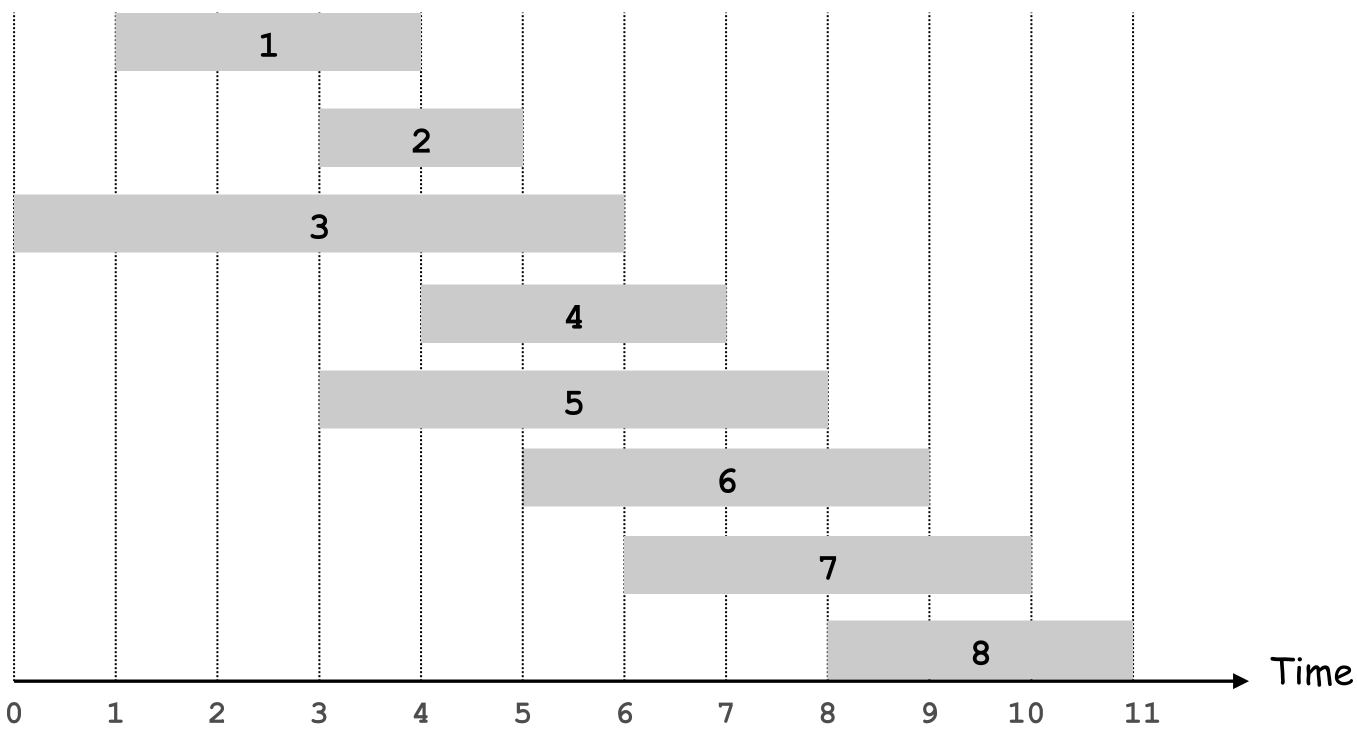
Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



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

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

Weighted Interval Scheduling: Brute Force

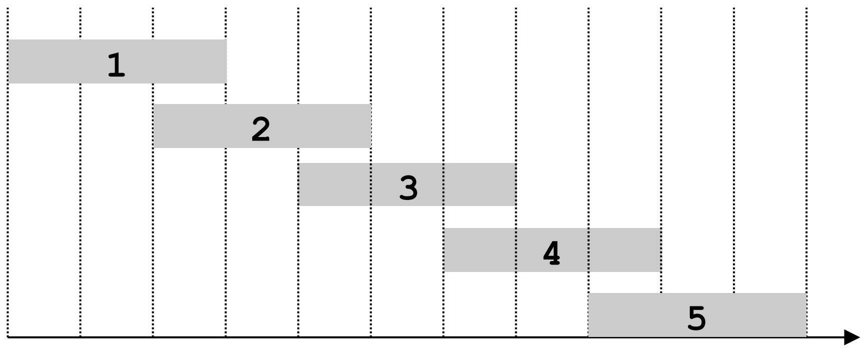
Brute force algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ 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))
}
```

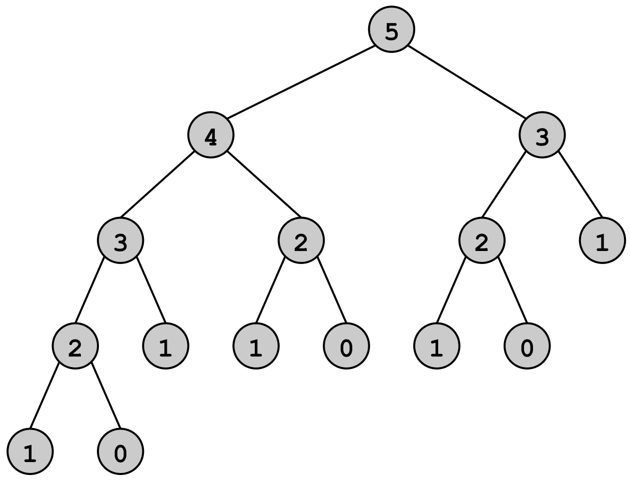
Weighted Interval Scheduling: Brute Force

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. Memoized 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 $\Phi \leq n$.
  - (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])
}
```

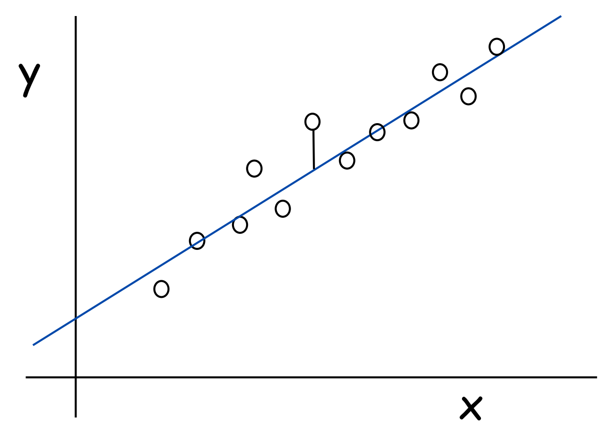
6.3 Segmented Least Squares

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:

$$SSE = \sum_{i=1}^{n} (y_i - a x_i - b)^2$$



Solution. Calculus $\Rightarrow$ min error is achieved when

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

Segmented Least Squares

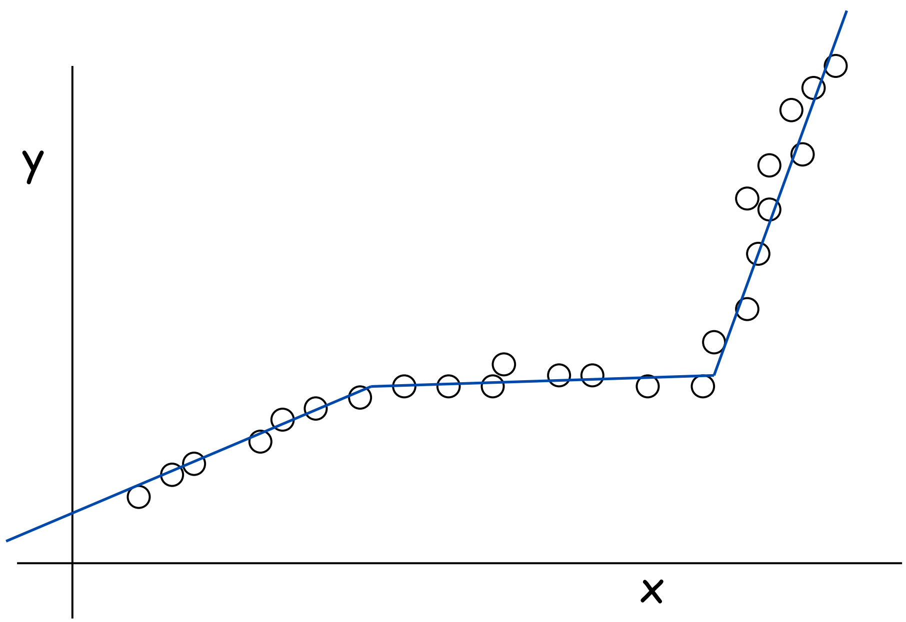
Segmented least squares.

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

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

goodness of fit

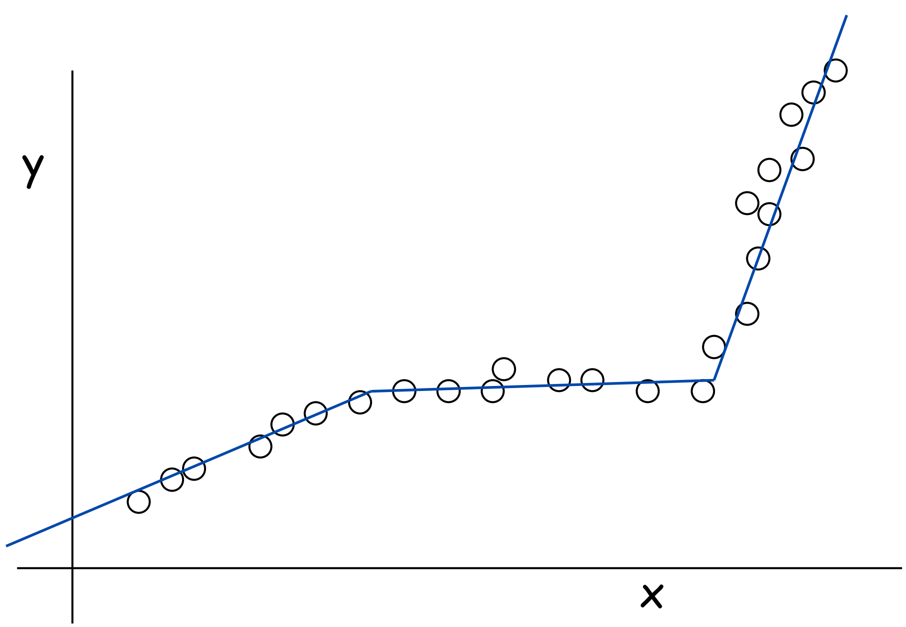
number of lines



Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ 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).

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

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]
}
```

Running time. $O(n^3)$. can be improved to $O(n^2)$ by pre-computing various statistics

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

6.4 Subset-Sum Problem

Subset-Sum Problem

Subset-Sum problem.

- Input: a set of items {1, …, n} with weights w_i and a capacity W
- Output: A subset S of items whose weights sum to ≤ W
- Goal: Maximize the sum of the weights of the items chosen

Dynamic Programming: False Start

Def. OPT(i) = max weight of a subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max weight of a subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\left\{ OPT(i-1, w),\quad w_i + OPT(i-1, w-w_i) \right\} & \text{otherwise} \end{cases}$$

Subset-Sum Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], w_i + M[i-1, w-w_i ]}

return M[n, W]
```

Subset-Sum Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Subset-Sum is NP-complete. [Chapter 8]