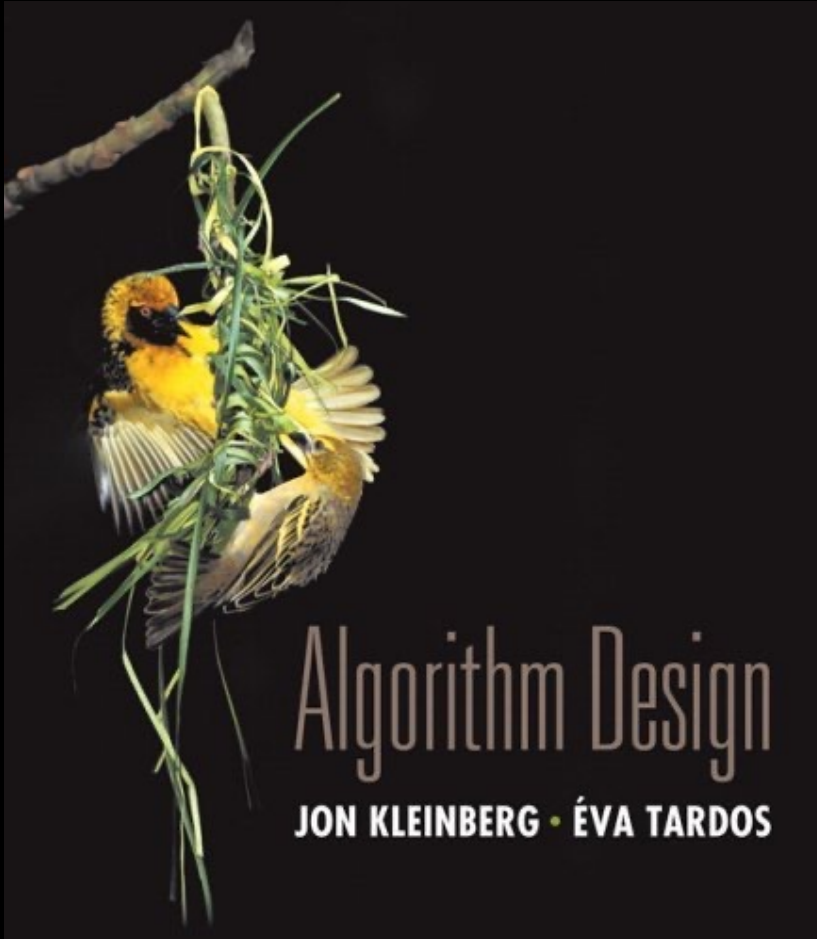


Chapter 6

Dynamic Programming



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

Annotated and slightly changed by Yang Xu (徐炆)
Contact: xuyang@sustech.edu.cn
Not for commercial use.

Algorithmic Paradigms

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

miopic:近视的; 目光短浅的

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*.

“Those who cannot remember the past are condemned to repeat it”
-- Dynamic Programming

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.

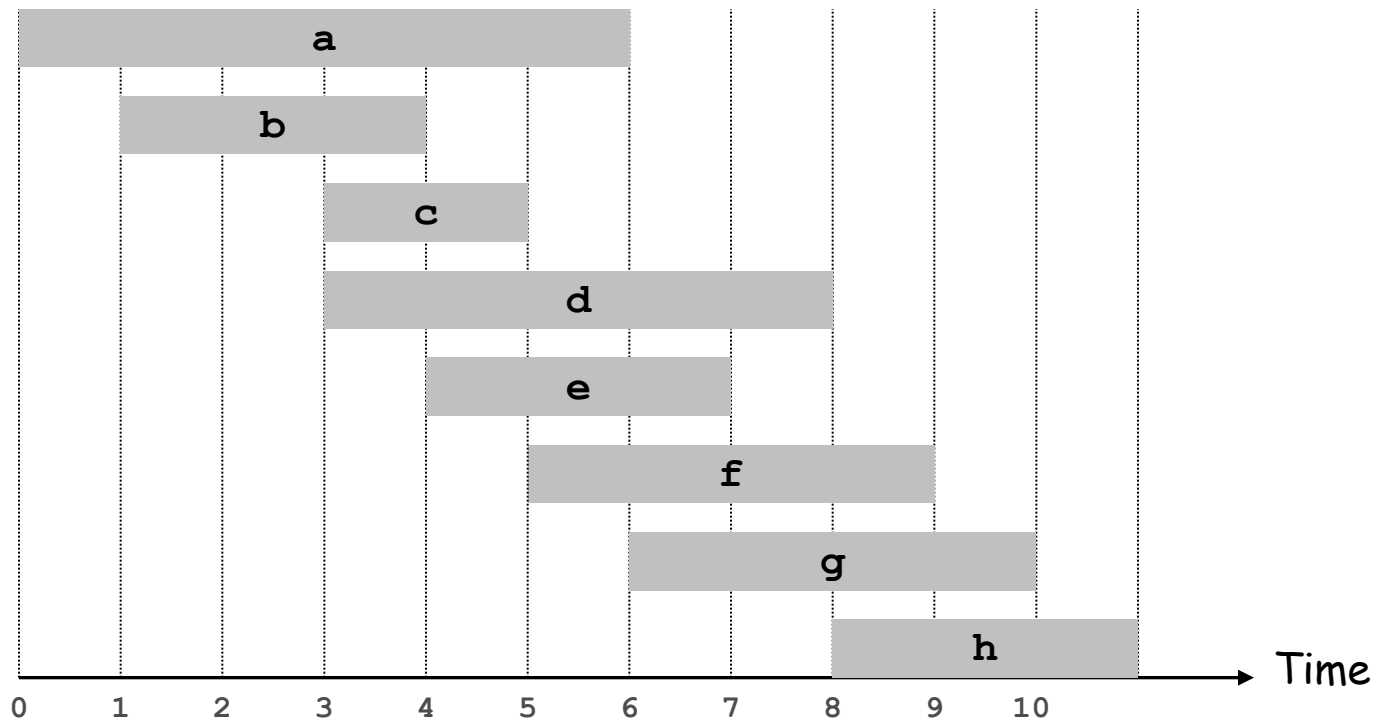
6.1-6.2 Weighted Interval Scheduling

(Sec 6.1 -- 6.2 on textbook)

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.

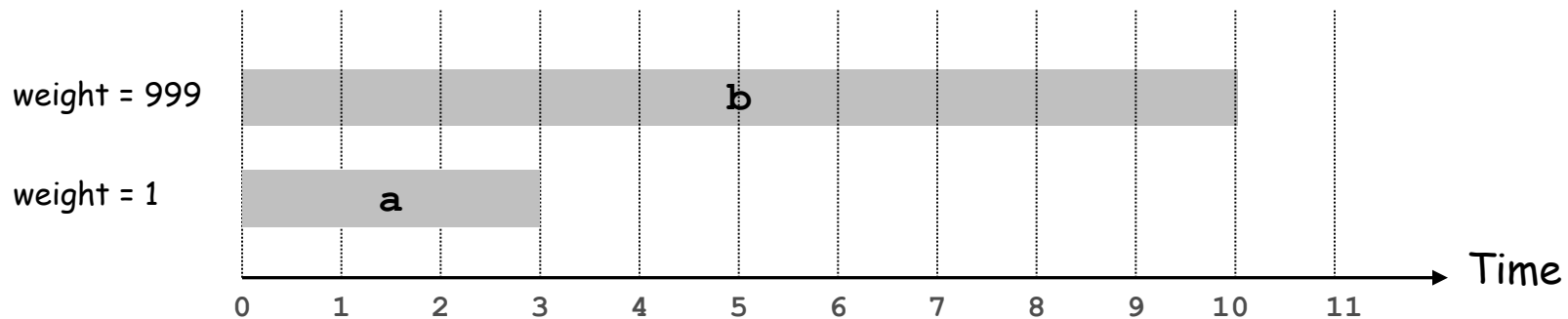


Unweighted Interval Scheduling Review

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.

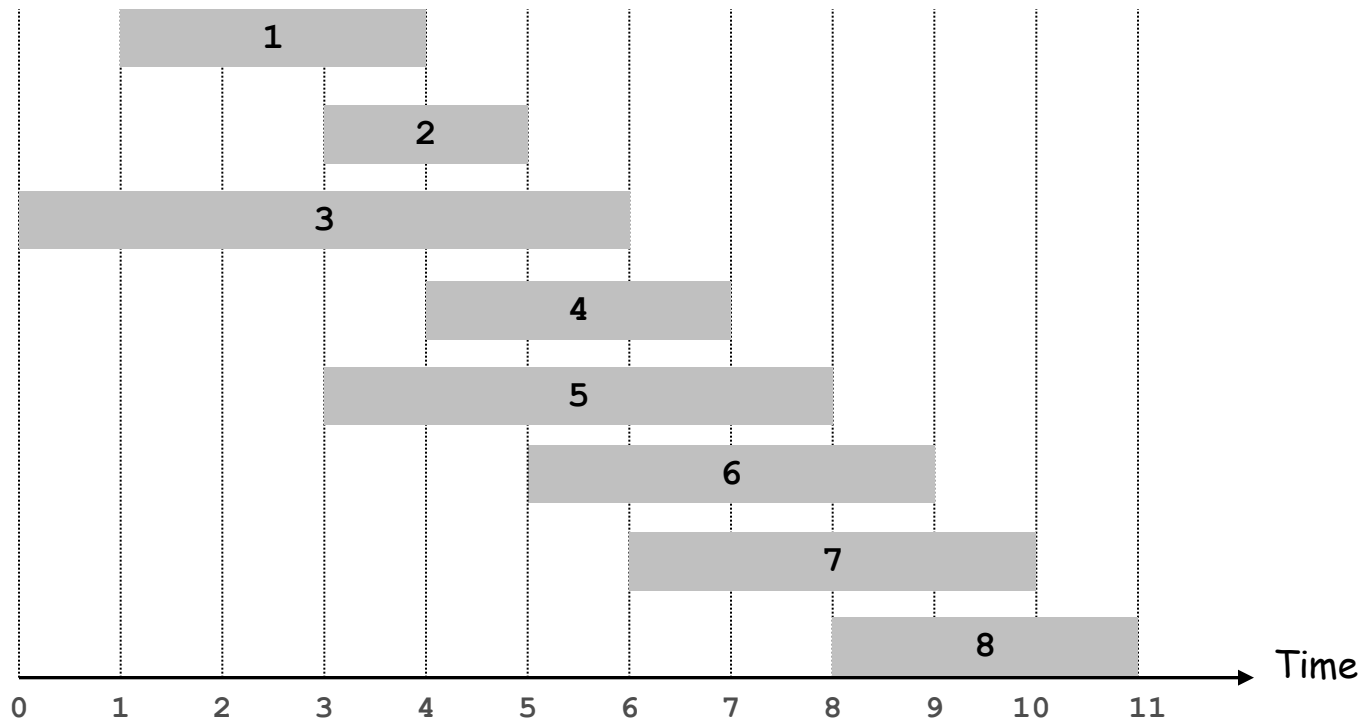


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \dots \leq f_n$.


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 $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
 - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $p(j)$
 - Case 2: OPT does not select job j.
 - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $j-1$
- 

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

Optimal Substructure

Optimal substructure: A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

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

Original problem

Subproblems, because $p(j) < j$ and $j - 1 < j$

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

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

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

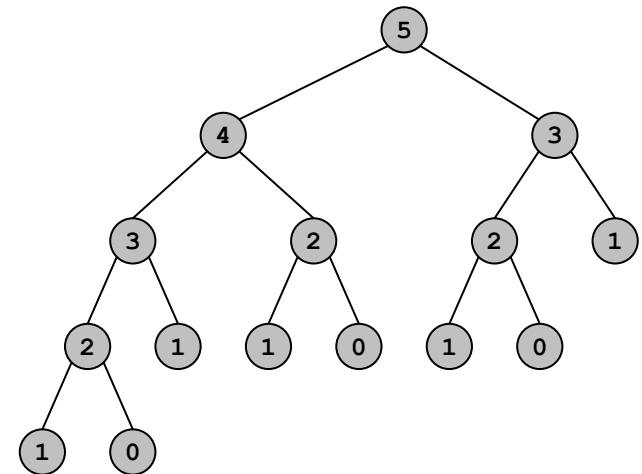
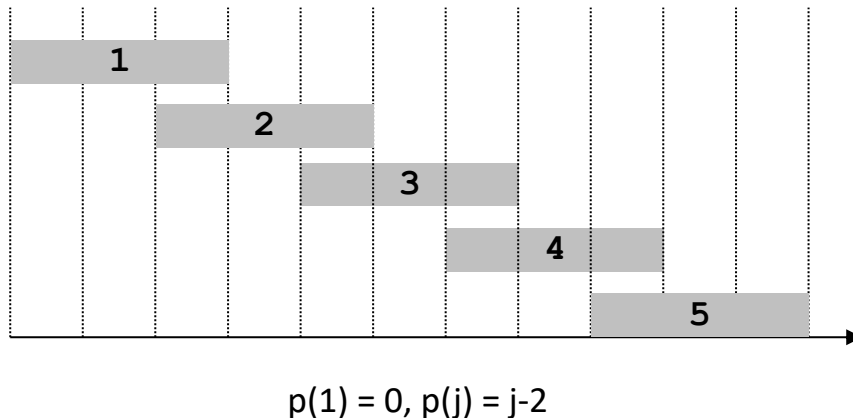
Compute $p(1), p(2), \dots, p(n)$

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

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.



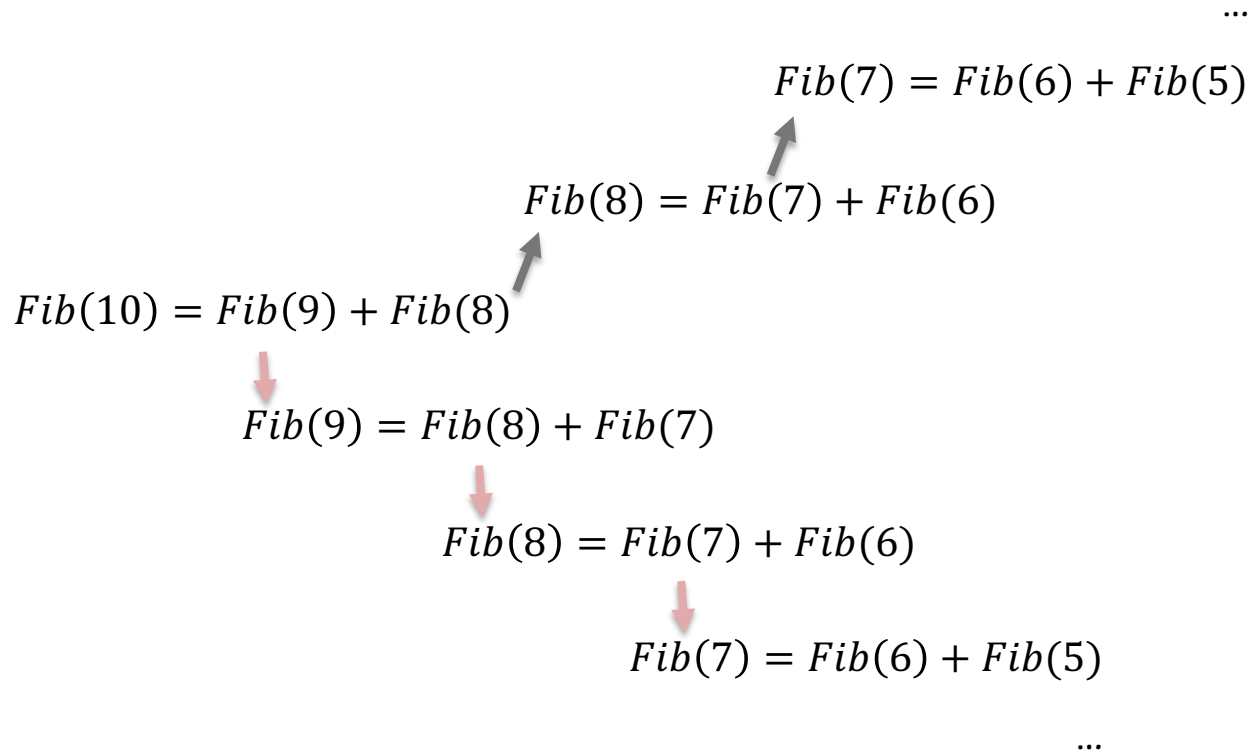
```
Compute-Opt(j) {  
  if (j = 0)  
    return 0  
  else  
    return max( $v_j + \text{Compute-Opt}(p(j))$ ,  
               $\text{Compute-Opt}(j-1)$ )  
}
```

$\text{Opt}(5) \Rightarrow \text{Opt}(4) + \text{Opt}(3)$

$\text{Opt}(j) \Rightarrow \text{Opt}(j-1) + \text{Opt}(j-2)$

Analogy to Fibonacci problem

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$



Subproblems are solved repeatedly

Weighted Interval Scheduling: Memoization

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

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

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

Compute $p(1), p(2), \dots, p(n)$

for $j = 1$ to n

$M[j] = \text{empty}$

$M[0] = 0$

 **global array**

M-Compute-Opt(j) {

if ($M[j]$ is empty)

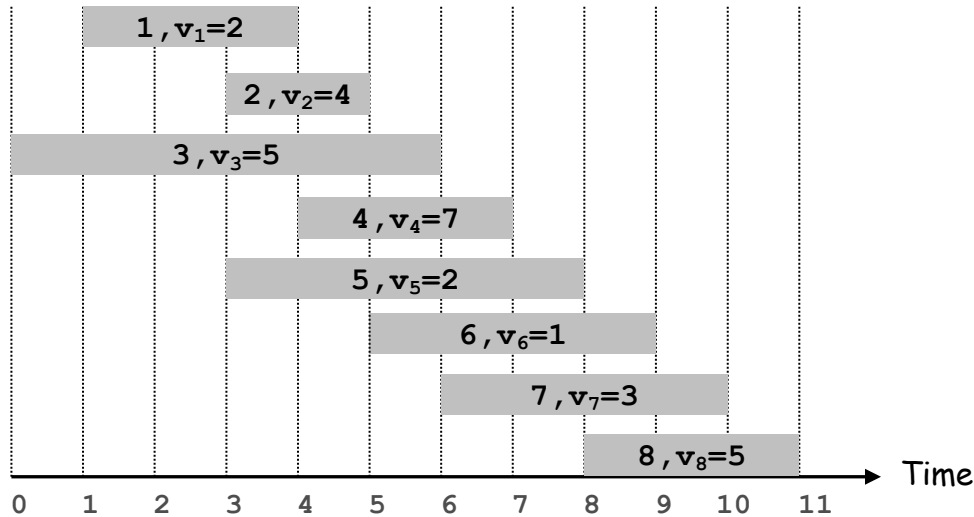
$M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$

return $M[j]$

}

Call **M-Compute-Opt**(n) to solve the problem

Memoization solution



Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j .

Compute $p(1), p(2), \dots, p(n)$

$p(1) = 0$
 $p(2) = 0$
 $p(3) = 0$
 $p(4) = 1$
 $p(5) = 0$
 $p(6) = 2$
 $p(7) = 3$
 $p(8) = 5$

Call **M-Compute-Opt**(n)

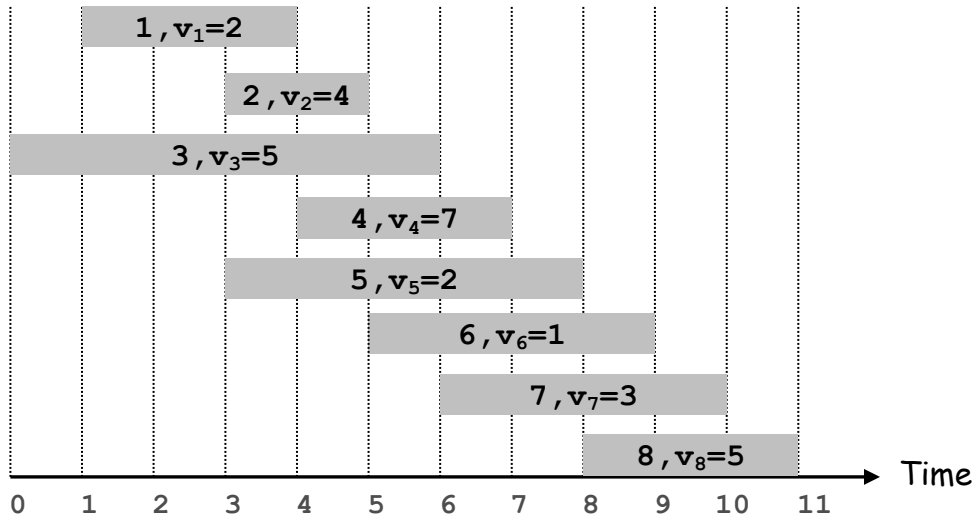
$M[]$	0								
index	0	1	2	3	4	5	6	7	8

```

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

$$\begin{aligned}
 \text{M-Compute-Opt}(8) &= \max \begin{cases} v_8 + \text{M-Compute-Opt}(p(8)) \\ \text{M-Compute-Opt}(8-1) \end{cases} = \begin{cases} v_8 + \text{M-Compute-Opt}(5) \\ \text{M-Compute-Opt}(7) \end{cases}
 \end{aligned}$$

Memoization solution



$p(1) = 0, p(2) = 0$
 $p(3) = 0, p(4) = 1$
 $p(5) = 0, p(6) = 2$
 $p(7) = 3, p(8) = 5$

```

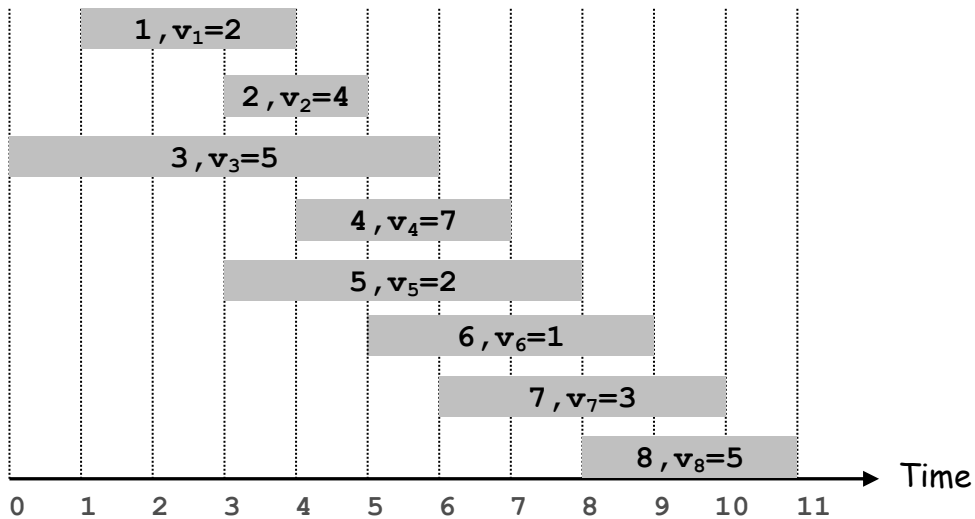
M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max(vj + M-Compute-Opt(p(j)),
                    M-Compute-Opt(j-1))
    return M[j]
}
    
```

M[]	0								
index	0	1	2	3	4	5	6	7	8

return M[0]

$$= \begin{cases} v_8 + \text{M-Compute-Opt}(5) \\ \text{M-Compute-Opt}(7) \end{cases} = \max \begin{cases} v_5 + \text{M-Compute-Opt}(p(5)) \\ \text{M-Compute-Opt}(5-1) \end{cases} = \begin{cases} v_5 + \text{M-Compute-Opt}(0) \\ \text{M-Compute-Opt}(4) \end{cases}$$

Memoization solution



$p(1) = 0, p(2) = 0$
 $p(3) = 0, p(4) = 1$
 $p(5) = 0, p(6) = 2$
 $p(7) = 3, p(8) = 5$

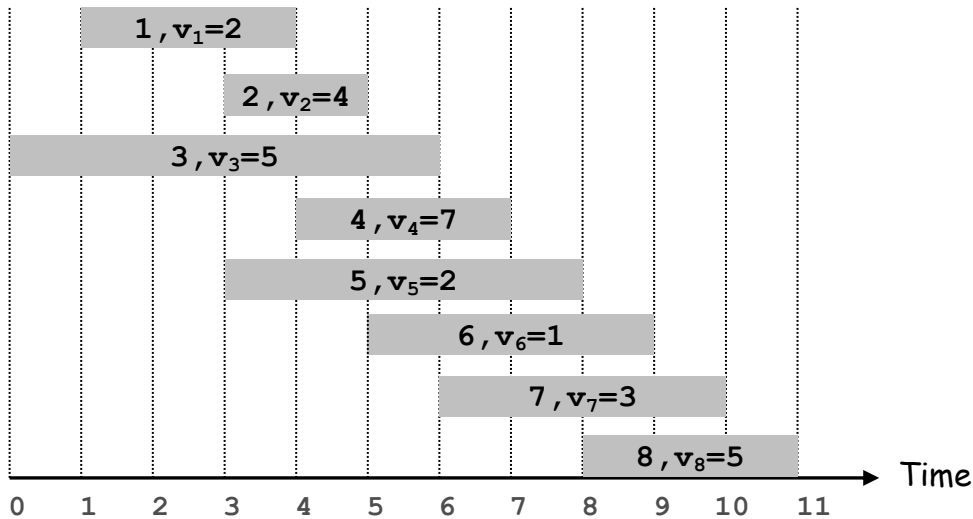
```

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]
}
    
```

M[]	0								
index	0	1	2	3	4	5	6	7	8

$$\begin{aligned}
 \max \left\{ \begin{array}{l} v_5 + 0 = 5 \\ \text{M-Compute-Opt}(4) \end{array} \right\} &= \max \left\{ \begin{array}{l} v_4 + \text{M-Compute-Opt}(p(4)) \\ \text{M-Compute-Opt}(4-1) \end{array} \right\} \\
 &= \max \left\{ \begin{array}{l} v_4 + \text{M-Compute-Opt}(1) \\ \text{M-Compute-Opt}(3) \end{array} \right\}
 \end{aligned}$$

Memoization solution



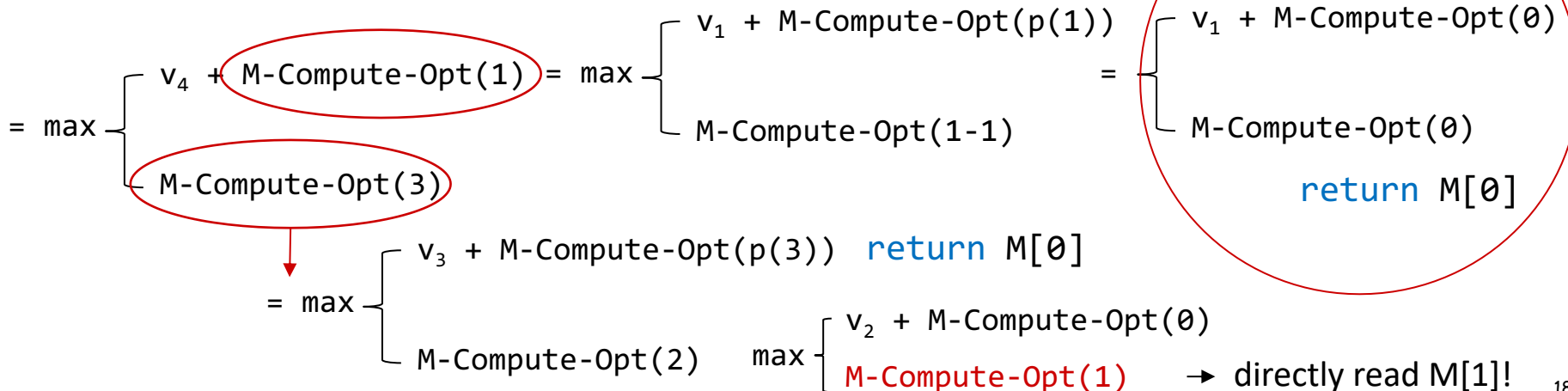
$p(1) = 0, p(2) = 0$
 $p(3) = 0, p(4) = 1$
 $p(5) = 0, p(6) = 2$
 $p(7) = 3, p(8) = 5$

```

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]
}
    
```

M[]	0	2	4	5	9				
index	0	1	2	3	4	5	6	7	8

M-Compute-Opt(1) is solved, fill in M[1]



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 $\Phi = \# \text{ nonempty entries of } \mathbf{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 ( $v_j + M[p(j)] > M[j-1]$ )
        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: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

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

Compute $p(1), p(2), \dots, p(n)$

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



$p(j) < j, j-1 < j$
previously computed values

Compare two approaches: Top-down and bottom-up

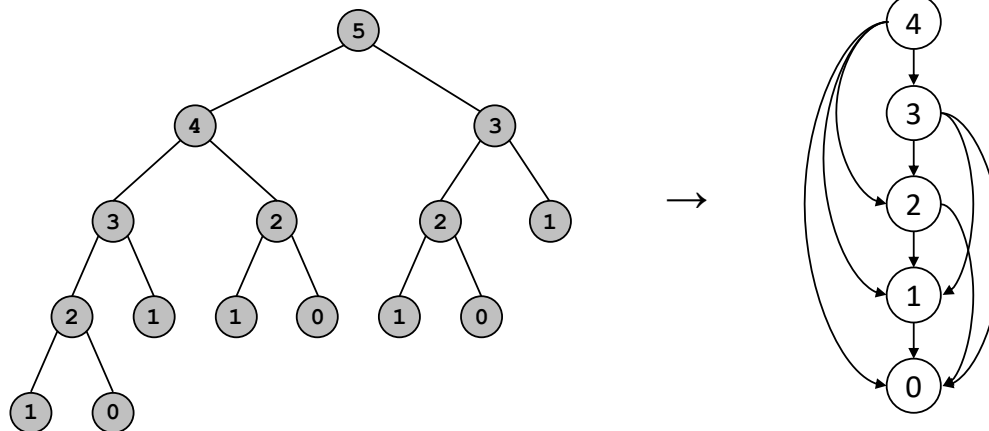
Top-down: Memoization

```
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]  
}
```

Bottom-up: Unwind recursion

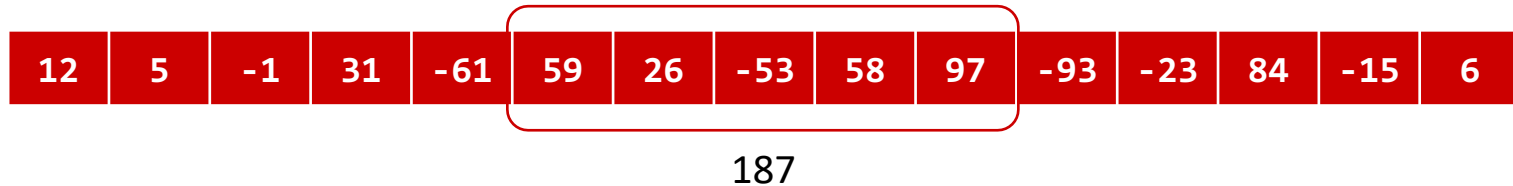
```
Iterative-Compute-Opt {  
  M[0] = 0  
  for j = 1 to n  
    M[j] = max( $v_j$  + M[p(j)],  
              M[j-1])  
}
```

Same outcome: smaller subproblems are solved first, without repetition



Bonus problem: Maximum subarray

Goal: Given an array a of n integer (positive or negative), find a contiguous subarray whose sum is maximum.



Brute force:

- For each i and j : compute $a[i] + a[i+1] + \dots + a[j]$
- Takes $O(n^2)$

Divide and conquer:

- Divide a into left and right halves: $a[1 \dots mid]$ and $a[mid \dots n]$
- $\text{max subarray} = \max(a[1 \dots mid], a[mid \dots n], \text{max-cross-mid})$
- $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = n \log n$

Dynamic programming?

Maximum subarray: Kadane's algorithm

Def. $OPT(i)$ = max sum of any subarray of a whose rightmost index is i .

Goal: $\max_i OPT(i)$

Optimal substructure:

$$OPT(i) = \begin{cases} a_i, & i = 1 \\ \max(a_i, a_i + OPT(i-1)), & i > 1 \end{cases}$$

only include element a_i

include element a_i
together with the
optimal subarray
ending at index $i-1$

$$OPT(1) = 12$$

$$OPT(2) = \max(5, OPT(1))$$

...

$$OPT(14) = \max(-15, OPT(13))$$

$$OPT(15) = \max(6, OPT(14))$$

12	5	-1	31	-61	59	26	-53	58	97	-93	-23	84	-15	6
----	---	----	----	-----	----	----	-----	----	----	-----	-----	----	-----	---

Maximum subarray: Kadane's algorithm

Use $M[i]$ to store $OPT(i)$

12	5	-1	31	-61	59	26	-53	58	97	-93	-23	84	-15	6
----	---	----	----	-----	----	----	-----	----	----	-----	-----	----	-----	---

[illegible]

max: 0 12 17 47 59 85 90 187

```
M[0] = 0, max = 0
for i from 1 to n:
    M[i] = max(a[i], a[i] + M[i-1])
    if M[i] > max:
        max = M[i]
```

Subproblems solved in a bottom-up way

Optimal substructure:

$$OPT(i) = \begin{cases} a_i, & i = 1 \\ \max(a_i, a_i + OPT(i-1)), & i > 1 \end{cases}$$

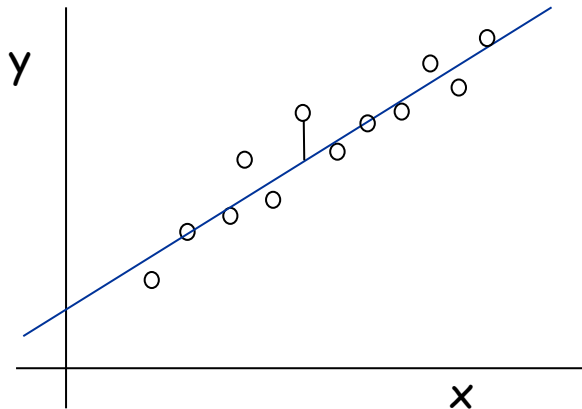
6.3 Segmented Least Squares

(Sec 6.3 on textbook)

Segmented Least Squares

Least squares.

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



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

Solution. Calculus \Rightarrow min error is achieved when

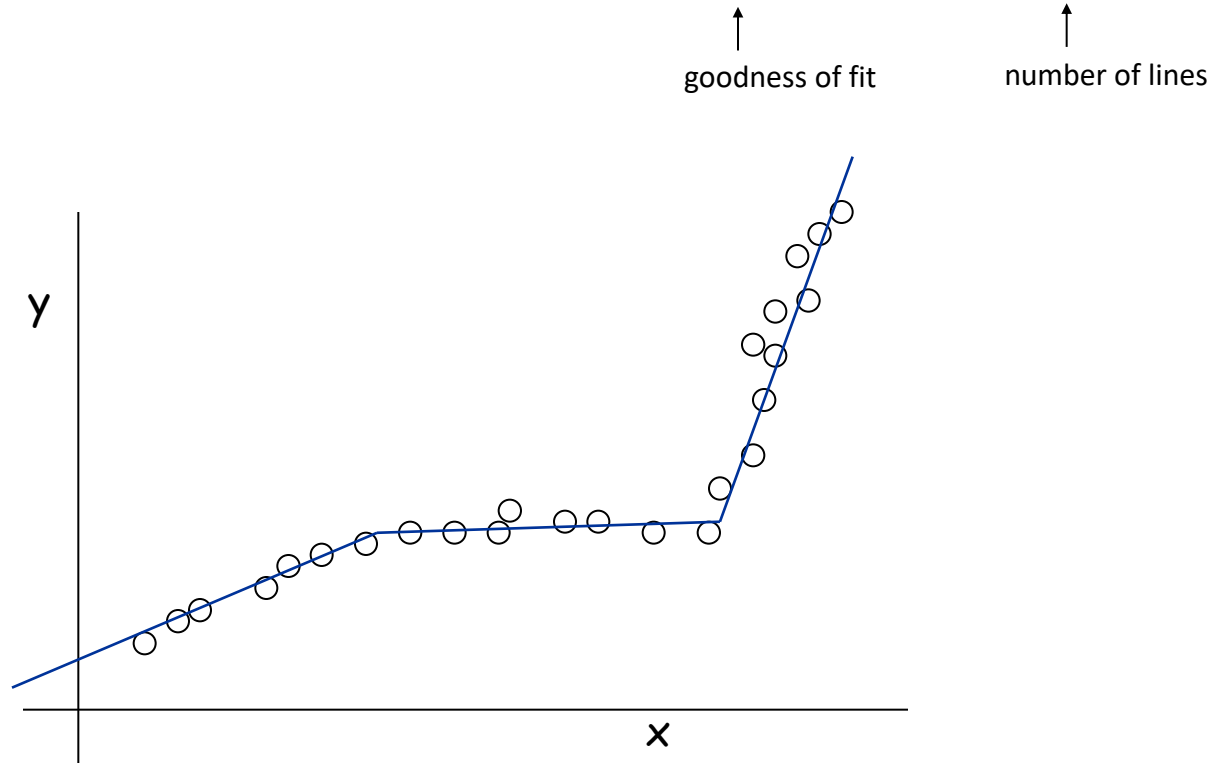
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i) (\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

Segmented least squares.

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

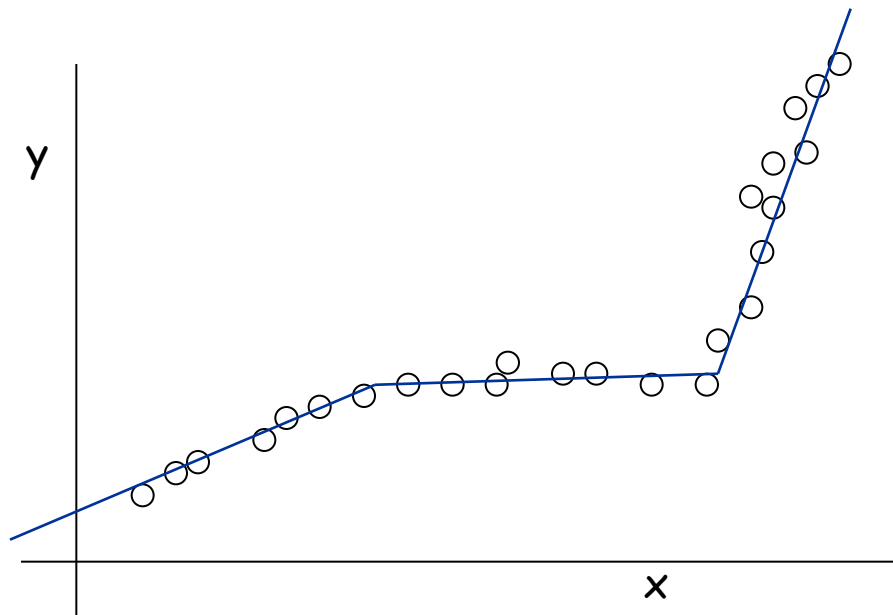
Q. What's a reasonable choice for $f(x)$ to balance accuracy and parsimony?



Segmented Least Squares: Formulation

Segmented least squares.

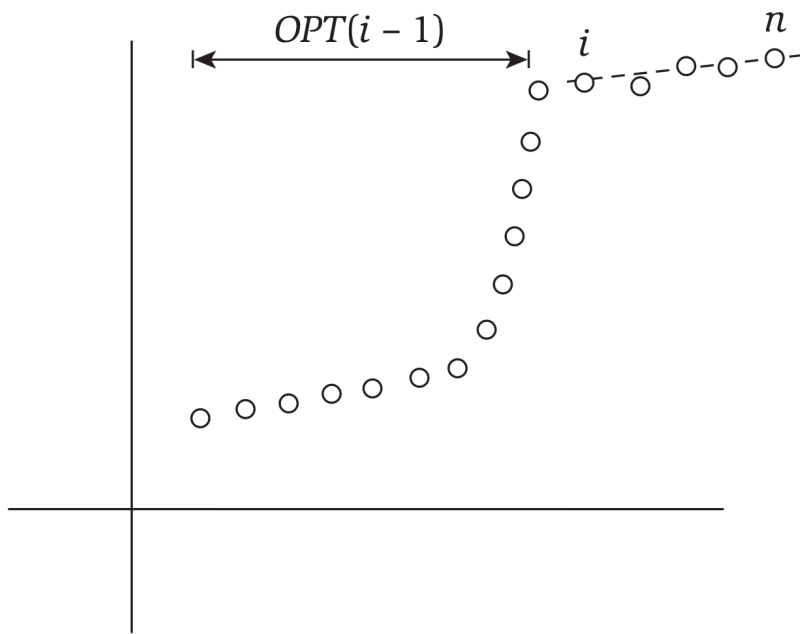
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with
- $x_1 < x_2 < \dots < x_n$, find a sequence of lines that minimizes:
 - the sum of the sums of the squared errors E in each segment
 - the number of lines L
- Tradeoff function: $E + C L$, for some constant $C > 0$.



Segmented Least Squares: Formulation

Observations

- The last point p_n belongs to a single segment beginning from p_i
- Remove $p_i \dots p_n$ and recursively solve the subproblem: $p_1 \dots p_{i-1}$



Assuming the last segment of the optimal partition is $p_i \dots p_n$, then the optimal solution can be expressed:

$$OPT(n) = e_{i,n} + C + OPT(i-1)$$

Dynamic Programming: Multiway Choice

Notation.

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

To compute $OPT(j)$:

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

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

Segmented Least Squares: Algorithm

```
INPUT:  $n, p_1, \dots, p_N, c$ 

Segmented-Least-Squares() {
   $M[0] = 0$ 
  for  $j = 1$  to  $n$ 
    for  $i = 1$  to  $j$ 
      compute the least square error  $e_{ij}$  for
      the segment  $p_i, \dots, p_j$ 

  for  $j = 1$  to  $n$ 
     $M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1])$ 

  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 Knapsack Problem

knapsack: n. 背包

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

$$W = 11$$

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i .

Ex: { 5, 2, 1 } achieves only value = 35 \Rightarrow greedy not optimal.

Dynamic Programming: False Start

Def. $\text{OPT}(i)$ = max profit 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 profit 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

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

Knapsack Problem: Bottom-Up

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

Input: $n, W, w_1, \dots, w_N, v_1, \dots, v_N$

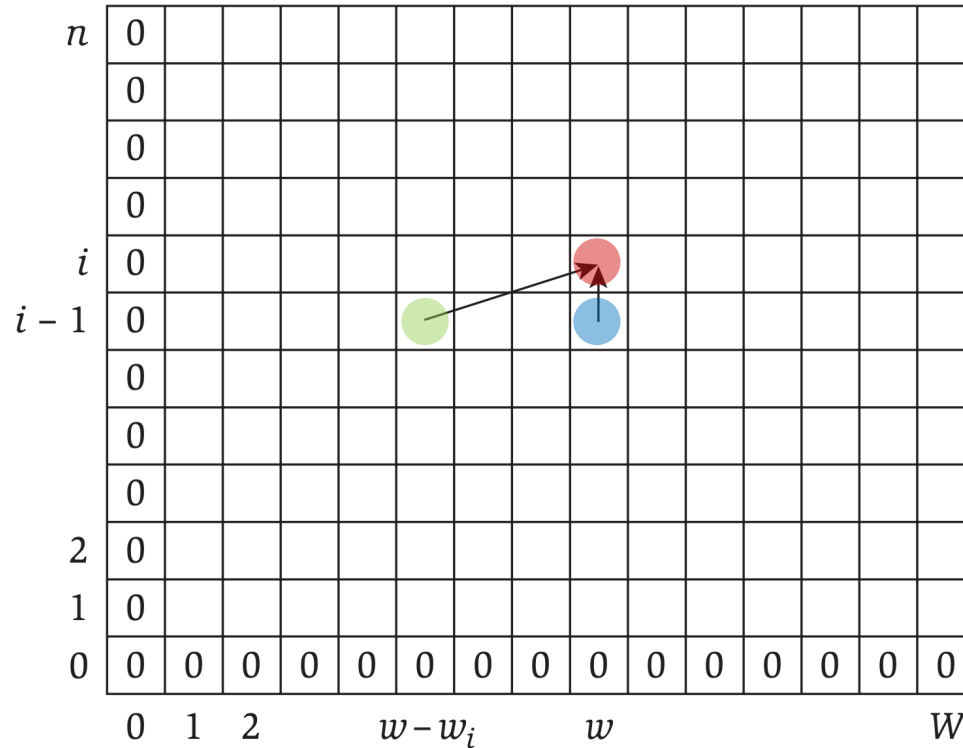
```
for  $w = 0$  to  $W$   
   $M[0, w] = 0$ 
```

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

```
return  $M[n, W]$ 
```

Knapsack Problem: Table

Use a **two-dimensional table** to store all subproblems



The entry $OPT(i, w)$ ● is computed from two other entries:
 $OPT(i-1, w)$ ● and $OPT(i-1, w-w_i)$ ●

Knapsack Problem: Example

Ex. $W=6$ and $n=3$ items, $w_1 = w_2 = 2$, and $w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
①	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Fill in values for $i = 1$

3							
②	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Fill in values for $i = 2$

③	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Fill in values for $i = 3$

Knapsack Algorithm

		$W + 1$											
		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
	$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
	$\{1, 2\}$	0	1	6	7	7	7	7	7	7	7	7	7
	$\{1, 2, 3\}$	0	1	6	7	7	18	19	24	25	25	25	25
	$\{1, 2, 3, 4\}$	0	1	6	7	7	18	22	24	28	29	29	40
	$\{1, 2, 3, 4, 5\}$	0	1	6	7	7	18	22	28	29	34	34	40

OPT: $\{4, 3\}$
value = $22 + 18 = 40$

$$W = 11$$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

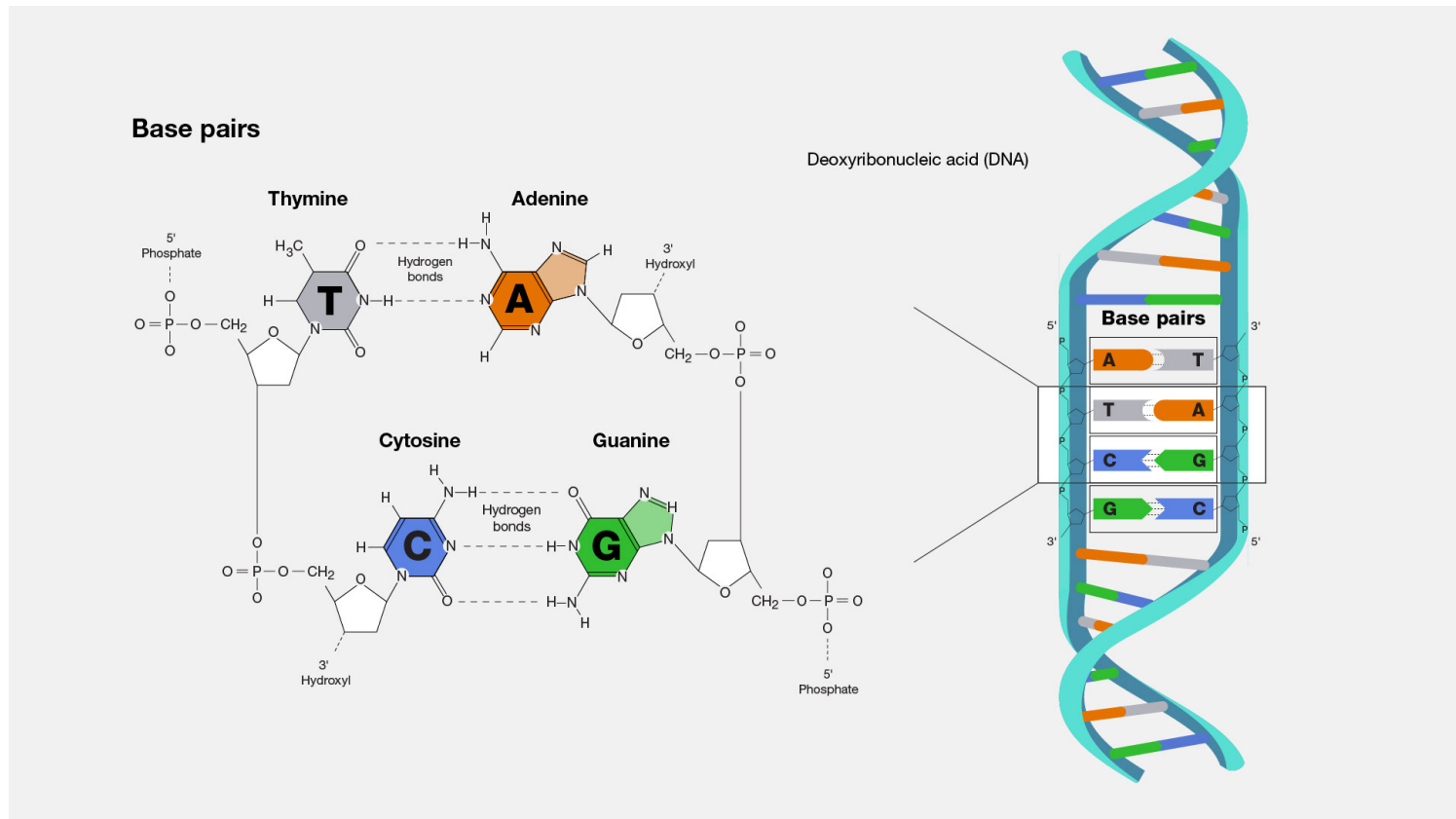
Knapsack Problem: Running Time

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

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

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

6.5 RNA Secondary Structure

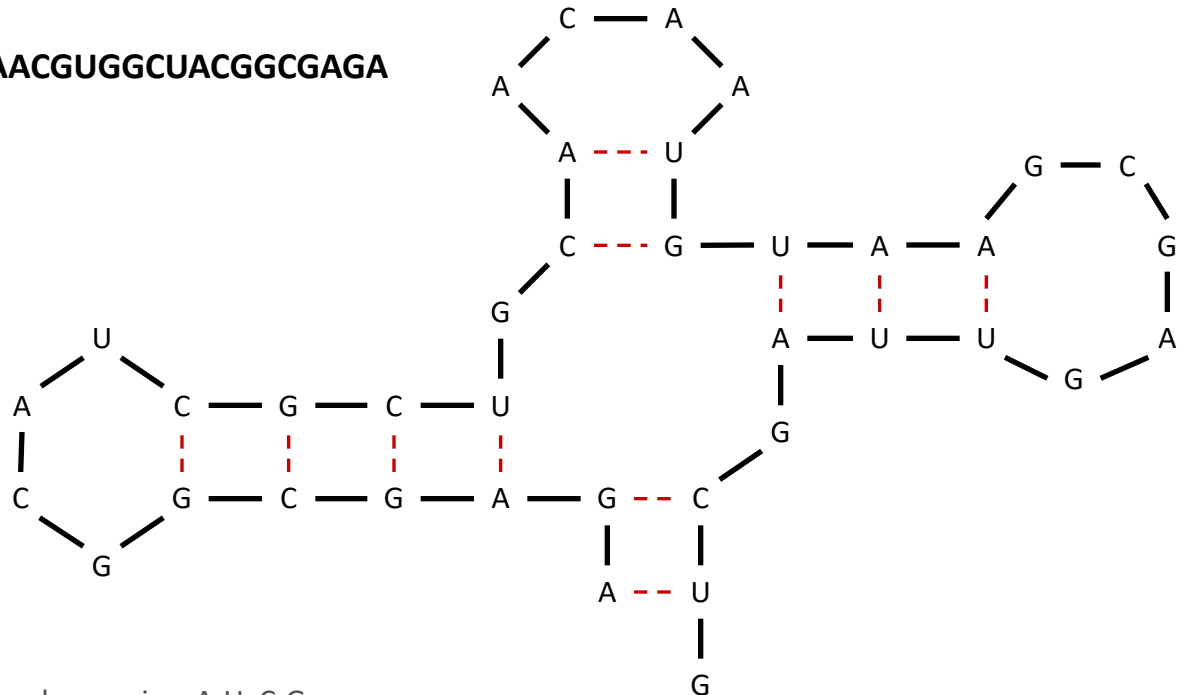


RNA Secondary Structure

RNA. String $B = b_1b_2\dots b_n$ over alphabet $\{A, C, G, U\}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



complementary base pairs: A-U, C-G

RNA Secondary Structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S , then we cannot have $i < k < j < l$.

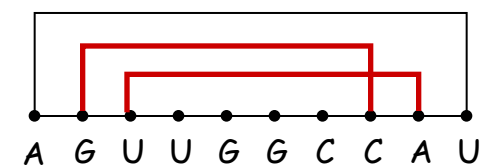
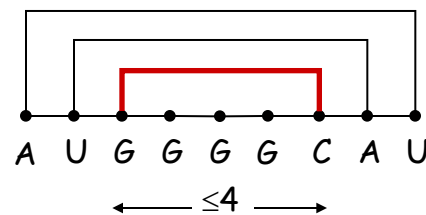
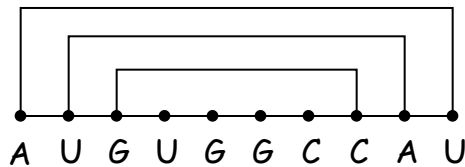
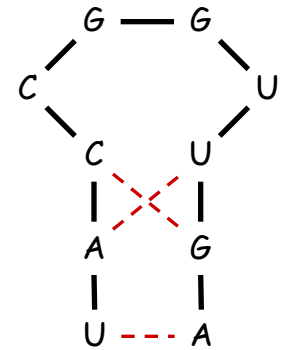
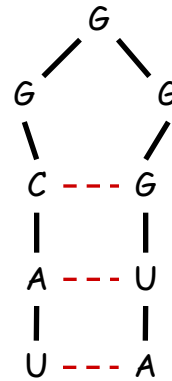
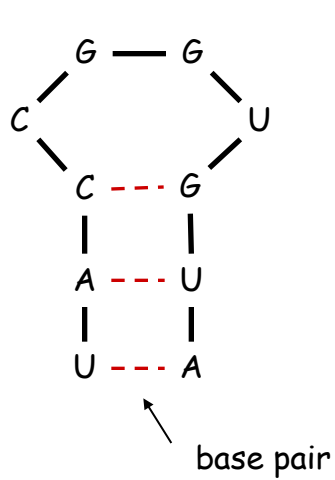
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

 approximate by number of base pairs

Goal. Given an RNA molecule $B = b_1b_2\dots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

Examples.



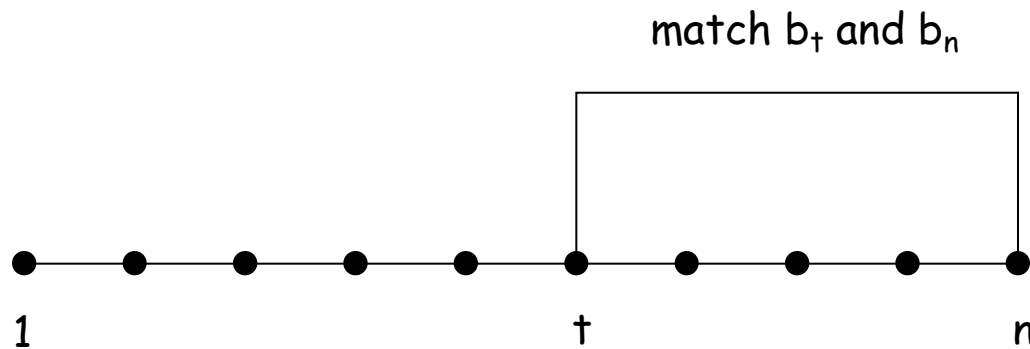
ok

sharp turn

crossing

RNA Secondary Structure: Subproblems

First attempt. $\text{OPT}(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1b_2\dots b_j$.




Difficulty. Results in two sub-problems.

- Finding secondary structure in: $b_1b_2\dots b_{t-1}$. ← $\text{OPT}(t-1)$
- Finding secondary structure in: $b_{t+1}b_{t+2}\dots b_{n-1}$. ← need more sub-problems

Dynamic Programming: Optimal Substructure

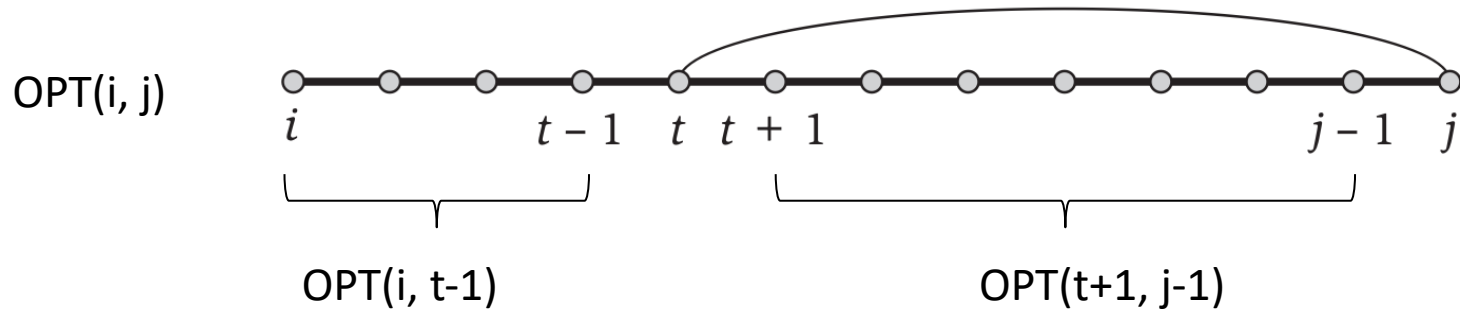
Notation. $\text{OPT}(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.

- Case 1. If $i \geq j - 4$.
 - $\text{OPT}(i, j) = 0$ by no-sharp turns condition.
- Case 2. Base b_j is not involved in a pair.
 - $\text{OPT}(i, j) = \text{OPT}(i, j-1)$
- Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.
 - non-crossing constraint decouples resulting sub-problems
 - $\text{OPT}(i, j) = 1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \}$


take max over t such that $i \leq t < j-4$ and
 b_t and b_j are Watson-Crick complements

Remark. Same core idea in CKY algorithm to parse context-free grammars.

Dynamic Programming: Optimal Substructure



$$OPT(i, j) = \max(\underbrace{OPT(i, j-1)}_{\text{If } b_j \text{ cannot pair with any base}}, \underbrace{\max(1 + OPT(i, t-1) + OPT(t+1, j-1))}_{\text{If } b_j \text{ can pair with some } b_t \text{ for } i \leq t < j-1}))$$

If b_j cannot pair with any base

If b_j can pair with some b_t for $i \leq t < j-1$

Bottom Up Dynamic Programming Over Intervals

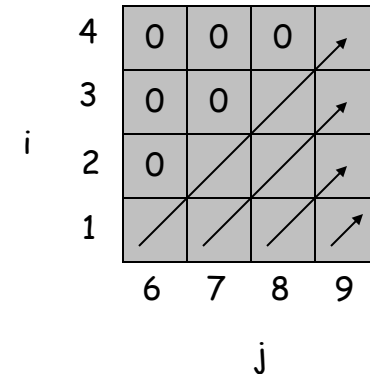
Q. What order to solve the sub-problems?

A. Do shortest intervals first.

Initialize $OPT(i, j) = 0$ whenever $i \geq j - 4$

```
RNA( $b_1, \dots, b_n$ ) {  
  for  $k = 5, 6, \dots, n-1$   
    for  $i = 1, 2, \dots, n-k$   
       $j = i + k$   
      Compute  $M[i, j]$   
  
  return  $M[1, n]$   
}
```

using recurrence



Running time. $O(n^3)$.

$$M[i, j] = 1 + \max_t \{ M[i, t-1] + M[t+1, j-1] \}$$

take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

RNA - Example

RNA sequence: b_1 A C C G G U A G U b_9

4	0	0	0	
3	0	0		
2	0			
$i = 1$				
	$j = 6$	7	8	9

Initial values

Initialize $OPT(i, j) = 0$ whenever $i \geq j - 4$

```

RNA( $b_1, \dots, b_n$ ) {
  for  $k = 5, 6, \dots, n-1$ 
    for  $i = 1, 2, \dots, n-k$ 
       $j = i + k$ 
      Compute  $M[i, j]$ 

  return  $M[1, n]$ 
}
```

Fill in the values for $k = 5$

$i = 1, 2, 3, 4$
 $j = 6, 7, 8, 9$

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			
	$j = 6$	7	8	9

$$OPT(1, 6) = \max(OPT(1, 5), \max_t(1 + OPT(1, t-1) + OPT(t+1, 5)))$$

because $1 = i \leq t < j-4 = 2, t = 1$

$b_1 = A$, which forms a pair with $b_6 = U$

Thus, $\max_t(1 + OPT(1, t-1) + OPT(t+1, 5)) = 1$

RNA - Example (cont.)

Fill in the values for $k = 5$

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			
	$j = 6$	7	8	9

$$\text{OPT}(2, 7) = \max(\text{OPT}(2, 6), \max_t(1 + \text{OPT}(2, t-1) + \text{OPT}(t+1, 7)))$$

$$2 = i \leq t < j-4 = 3, t = 2$$

$b_2 = C$, which **does NOT** form a pair with $b_7 = A$

$$\text{Thus, } \max_t(1 + \text{OPT}(2, t-1) + \text{OPT}(t+1, 7)) = 0$$

$b_1 b_2 \quad b_7 \quad b_9$
A C C G G U A G U

Initialize $\text{OPT}(i, j) = 0$ whenever $i \geq j-4$

```

RNA( $b_1, \dots, b_n$ ) {
  for  $k = 5, 6, \dots, n-1$ 
    for  $i = 1, 2, \dots, n-k$ 
       $j = i + k$ 
      Compute  $M[i, j]$ 
  return  $M[1, n]$ 
}
```

$k = 6$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		
	$j = 6$	7	8	9

$k = 7$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	
	$j = 6$	7	8	9

$k = 8$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2
	$j = 6$	7	8	9

RNA - Example (cont.)

$k = 8$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2
	$j = 6$	7	8	9

b_1 b_2 b_7 b_9
 A C C G G U A G U

$$\text{OPT}(1, 9) = \max(\text{OPT}(1, 8), \max_t(1 + \text{OPT}(2, t-1) + \text{OPT}(t+1, 8)))$$

$$1 = i \leq t < 9 - 4 = 5, t = \{1, 2, 3, 4\}$$

$t = 1, b_1 = A$ pairs with $b_9 = U$, $\text{OPT}(2, 8) = 1$

$t = 2, b_2 = C$ does not pair with $b_9 = U$

$t = 3, b_3 = C$ does not pair with $b_9 = U$

$t = 4, b_4 = G$ does not pair with $b_9 = U$

Dynamic Programming Summary

Recipe.

- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

← Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy

← CKY parsing algorithm for context-free grammar has similar structure

Top-down vs. bottom-up: different people have different intuitions.

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- occurrence
- occurrence

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

hyphen (-) indicates a *gap*

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Edit Distance

Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ ; mismatch penalty α_{pq} .
- Cost = sum of gap and mismatch penalties.

C	T	G	A	C	C	T	A	C	C	T
---	---	---	---	---	---	---	---	---	---	---

-	C	T	G	A	C	C	T	A	C	C	T
---	---	---	---	---	---	---	---	---	---	---	---

C	C	T	G	A	C	T	A	C	A	T
---	---	---	---	---	---	---	---	---	---	---

C	C	T	G	A	C	-	T	A	C	A	T
---	---	---	---	---	---	---	---	---	---	---	---

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

$$2\delta + \alpha_{CA}$$

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ find alignment of minimum cost.

Def. An **alignment** M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i - y_j$ and $x_{i'} - y_{j'}$ **cross** if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG vs. TACATG.

Sol: An alignment $M = \{x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6\}$.

$$\text{cost}(M) = \alpha_{x_5 y_4} + \delta_{x_1} + \delta_{y_5}$$

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G

	y_1	y_2	y_3	y_4	y_5	y_6
-	T	A	C	A	T	G

Sequence Alignment: Problem Structure

Def. $OPT(i, j)$ = **min cost** of aligning strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

- Case 1: OPT matches x_i - y_j .
 - pay mismatch for x_i - y_j + min cost of aligning two strings $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
 - pay gap for x_i and min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$
- Case 2b: OPT leaves y_j unmatched.
 - pay gap for y_j and min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

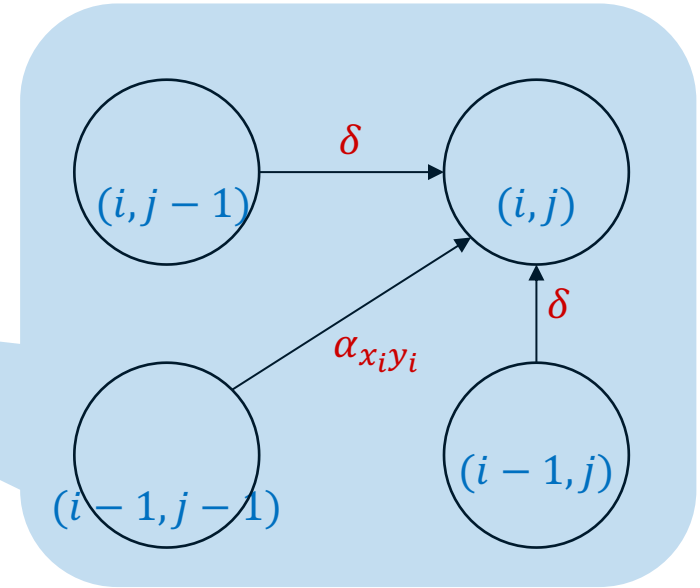
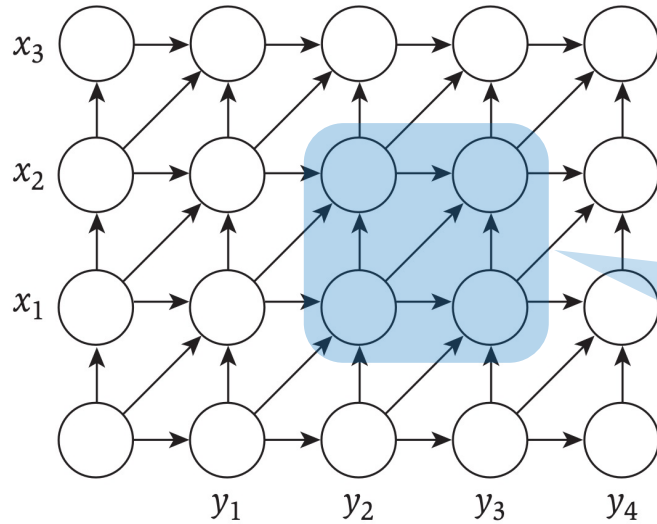
```
Sequence-Alignment( $m, n, x_1x_2\dots x_m, y_1y_2\dots y_n, \delta, \alpha$ ) {  
  for  $i = 0$  to  $m$   
     $M[i, 0] = i\delta$   
  for  $j = 0$  to  $n$   
     $M[0, j] = j\delta$   
  
  for  $i = 1$  to  $m$   
    for  $j = 1$  to  $n$   
       $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1],$   
                     $\delta + M[i-1, j],$   
                     $\delta + M[i, j-1])$   
  
  return  $M[m, n]$   
}
```

Analysis. $\Theta(mn)$ **time** and **space**.

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 100,000$. 10 billions ops OK, but 10GB array?

Sequence Alignment: Algorithm Analysis



Graph G_{XY} with a $m \times n$ grid

Finding the minimum-cost from $(0,0)$ to (i,j)

$$OPT(i,j) = \begin{cases} \alpha_{x_i y_i} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases}$$