

# Dynamic Programming



# Algorithmic Paradigms

**Divide-and-conquer.** Break up a problem into two 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.** 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*.

# Dynamic Programming Applications

## Areas.

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

## Some famous dynamic programming algorithms.

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

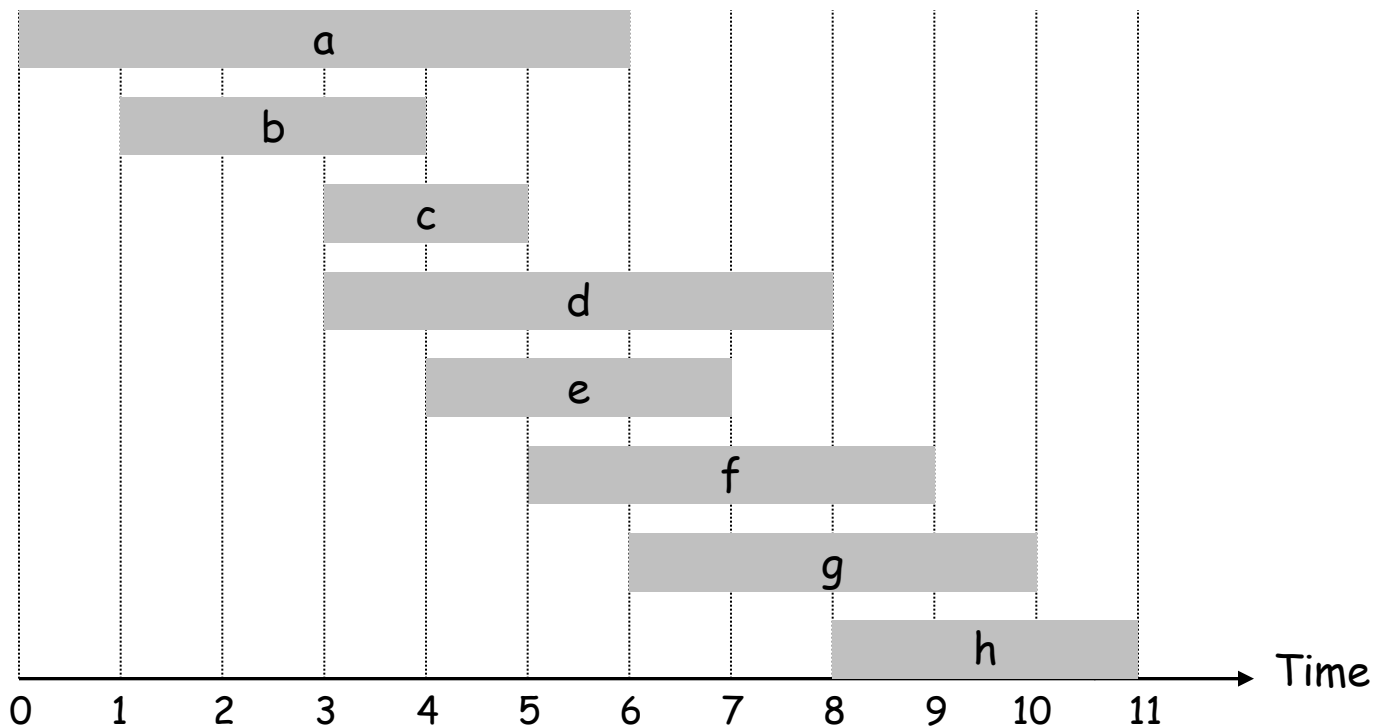
# 6.1 Weighted Interval Scheduling

---

# 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.

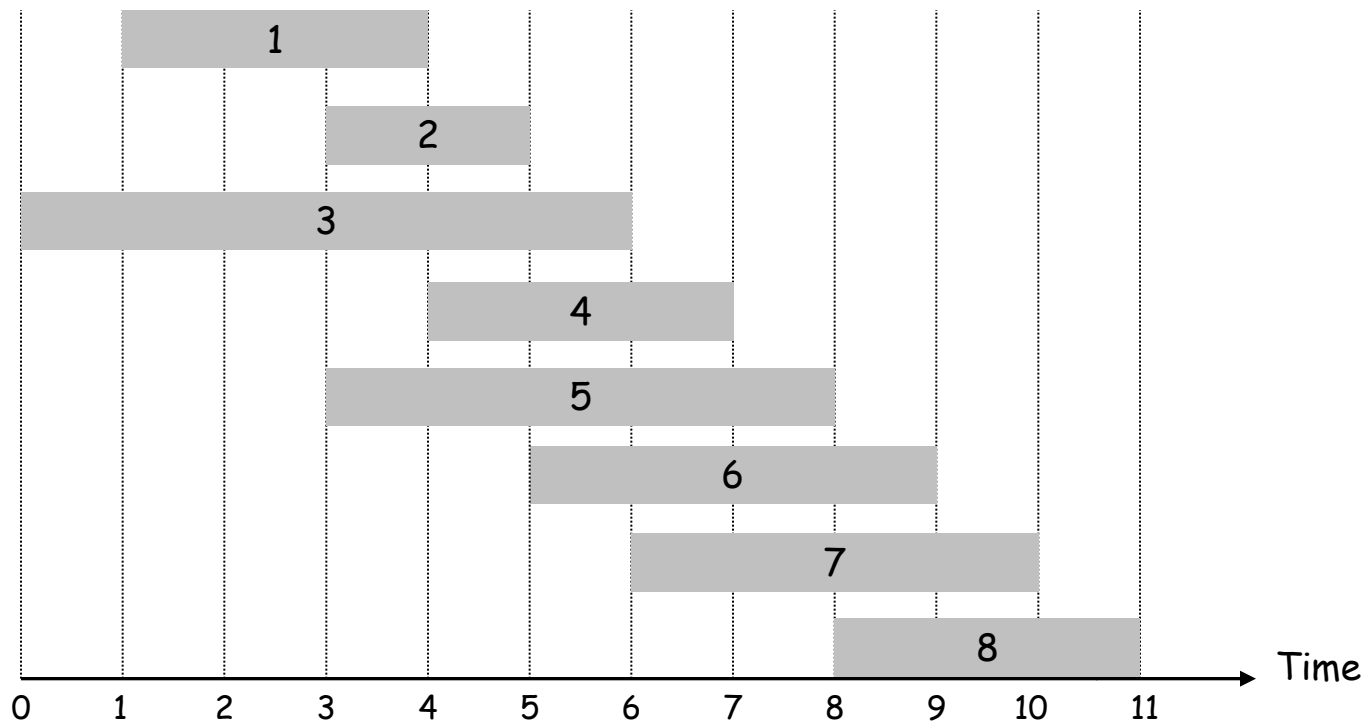


# Weighted Interval Scheduling

**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.
  - 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

↙ optimal substructure  
↘

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



# 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  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

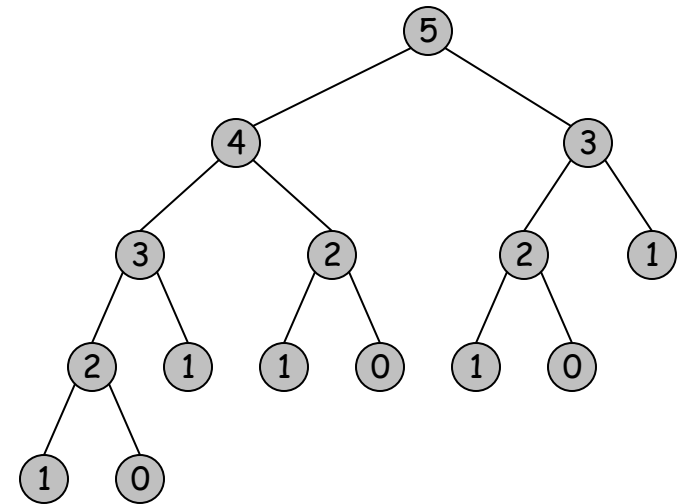
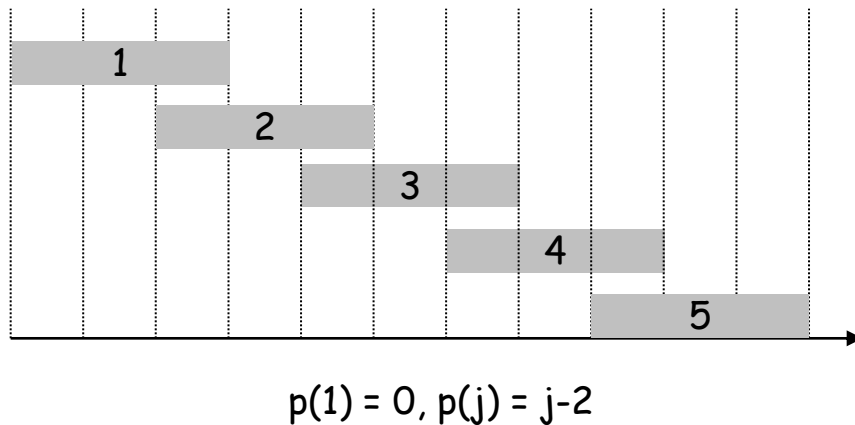
```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
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.



# 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  $n$ 
```

```
     $M[j] = \text{empty}$   $\leftarrow$  global array
```

```
 $M[j] = 0$ 
```

```
M-Compute-Opt( $j$ ) {
```

```
    if ( $M[j]$  is empty)
```

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

```
    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  $p(\cdot)$ :  $O(n)$  after sorting by start time.
- $M\text{-Compute-Opt}(j)$ : 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  $\Phi = \#$  nonempty entries of  $M[\ ]$ .
  - initially  $\Phi = 0$ , throughout  $\Phi \leq n$ .
  - (ii) increases  $\Phi$  by 1  $\Rightarrow$  at most  $2n$  recursive calls.
- Overall running time of  $M\text{-Compute-Opt}(n)$  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  $n$   
         $M[j] = \max(v_j + M[p(j)], M[j-1])$   
}
```

## 6.3 Segmented Least Squares

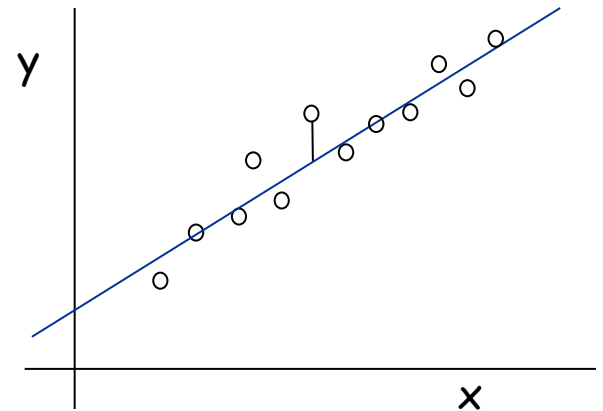
---

# Segmented Least Squares

## Least squares.

- Foundational problem in statistic and numerical analysis.
- Given  $n$  points in the plane:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ .
- 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

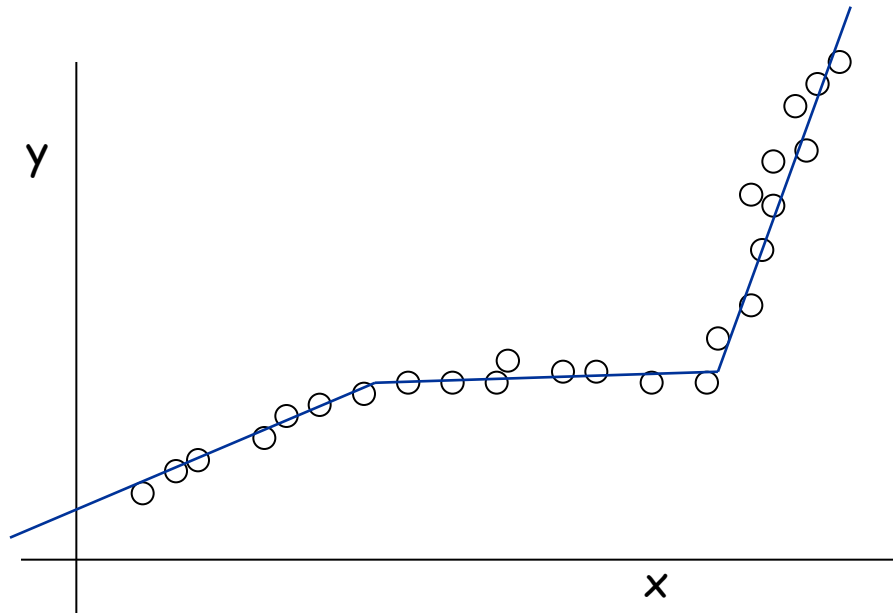
## Segmented least squares.

- 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?

↑  
number of lines

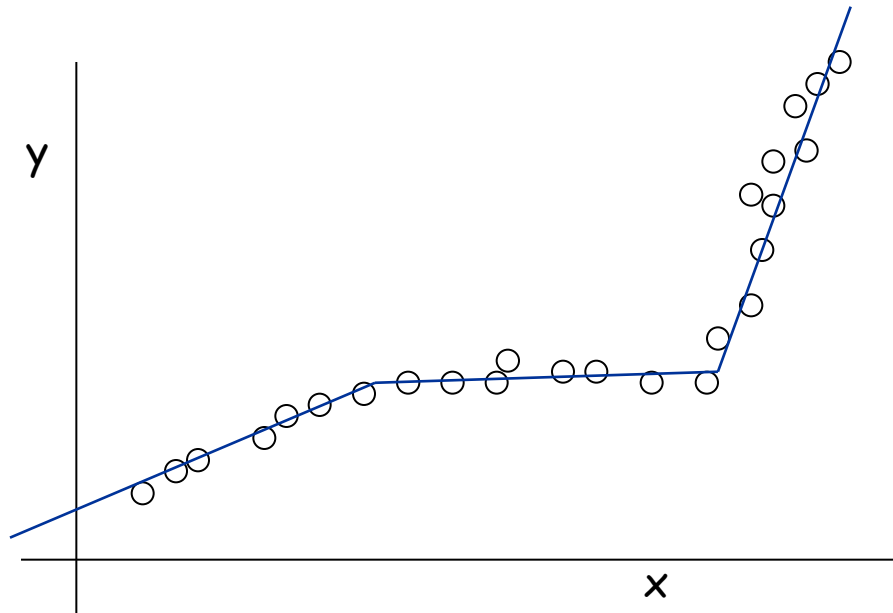
↑  
goodness of fit



# 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:
  - 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  $p_1, p_{i+1}, \dots, p_j$ .
- $e(i, j)$  = 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.

## 6.5 RNA Secondary Structure

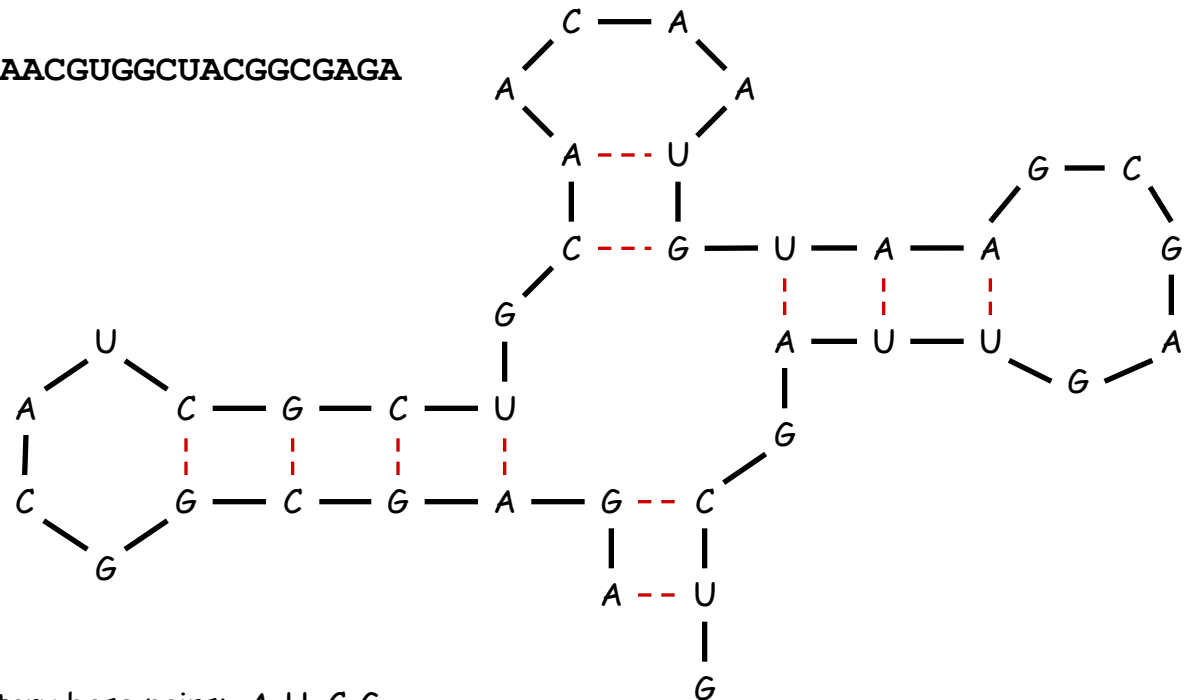
---

# 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 a 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  $i < k < j < l$ .

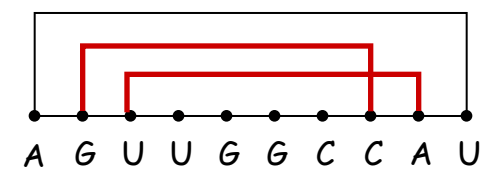
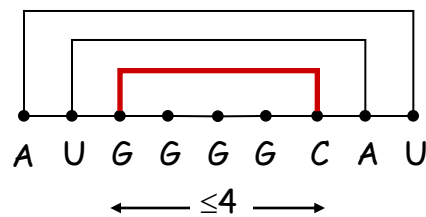
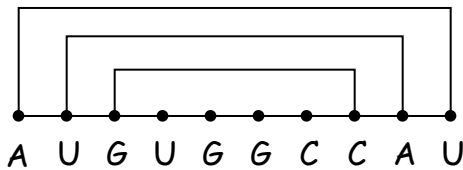
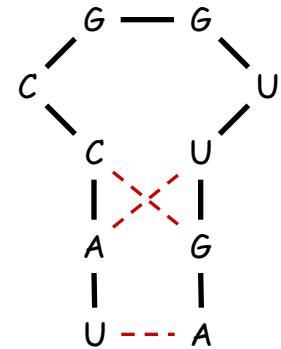
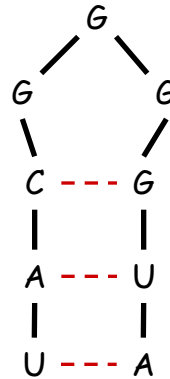
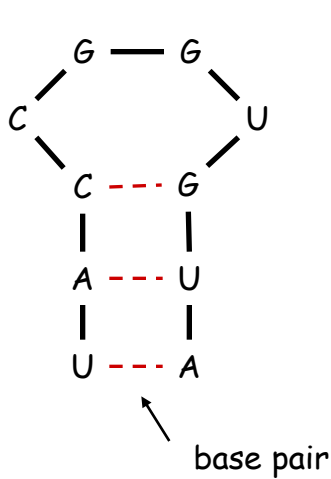
**Free energy.** Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

↑  
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

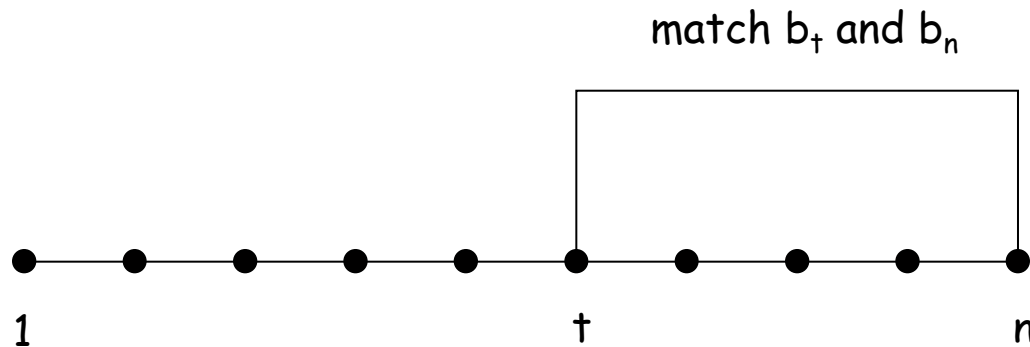
sharp turn

crossing



# RNA Secondary Structure: Subproblems

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

# Dynamic Programming Over Intervals

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

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

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

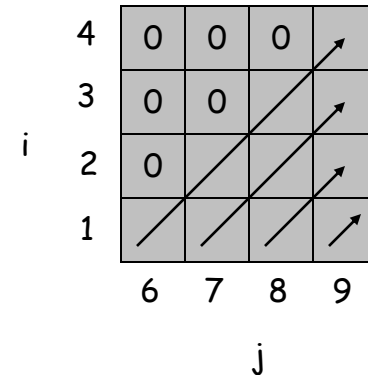
**Remark.** Same core idea in CKY algorithm to parse context-free grammars.

# Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

```
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.  $O(n^3)$ .

# 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. ← Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy
- Dynamic programming over intervals: RNA secondary structure.

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

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