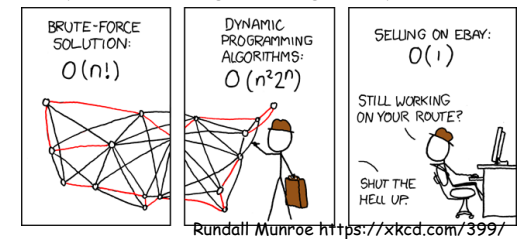


### Algorithmic Paradigms

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

**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.

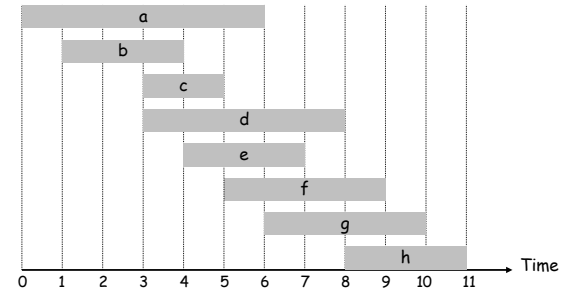
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

## 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.

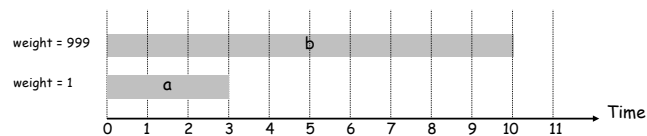


### Unweighted Interval Scheduling Review

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.

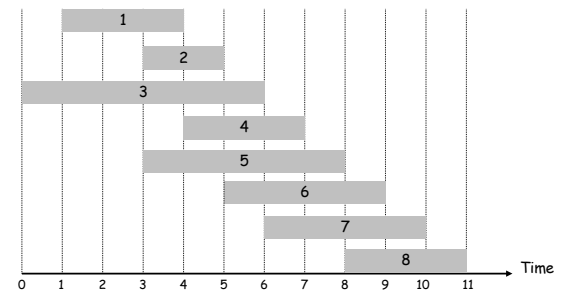


### Weighted Interval Scheduling

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.
  - 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

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

9

### Weighted Interval Scheduling: Brute Force

Brute force algorithm.

```

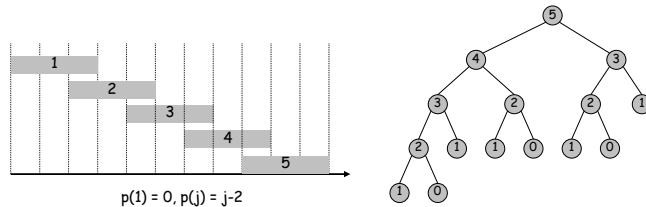
Input: n, s1, ..., sn, f1, ..., fn, v1, ..., vn
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), ..., p(n)
Compute-Opt(j) {
  if (j = 0)
    return 0
  else
    return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
    
```

10

### Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

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



11

### 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]
}
    
```

12

### 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-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  $M[\cdot]$ .
  - initially  $\Phi = 0$ , throughout  $\Phi \leq n$ .
  - (ii) increases  $\Phi$  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.

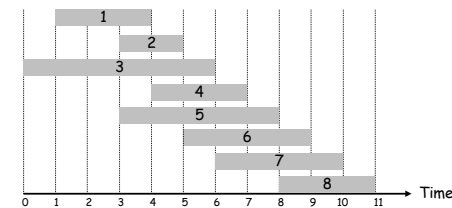
13

### Computing the Index $p(\cdot)$ in Linear Time

Interval	1	2	3	4	5	6	7	8
Finish time	4	5	6	7	8	9	10	11

Interval	3	1	2	5	4	6	7	8
Start time	0	1	3	3	4	5	6	8

Interval	1	2	3	4	5	6	7	8
$p(\cdot)$	0	0	0	1	0	2	3	5



14

### 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  $\leq n \Rightarrow O(n)$ .

15

### 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])
}
    
```

16

### Automated Memoization

Automated memoization. Many functional programming languages (e.g., Lisp) have built-in support for memoization.

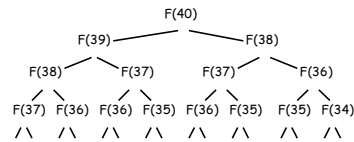
Q. Why not in imperative languages (e.g., Java)?

```
(defun F (n)
  (if (<= n 1)
      n
      (+ (F (- n 1)) (F (- n 2))))))
```

Lisp (efficient)

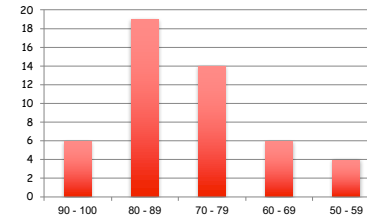
```
static int F(int n) {
  if (n <= 1) return n;
  else return F(n-1) + F(n-2);
}
```

Java (exponential)



17

### Midterm



Mean: 46.83 (78%)  
Median: 48 (80%)

18

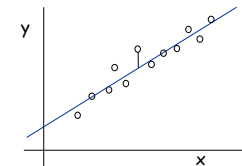
## 6.3 Segmented Least Squares

### 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

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

20

### Segmented Least Squares

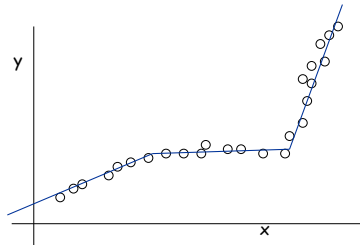
#### Segmented least squares.

- 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

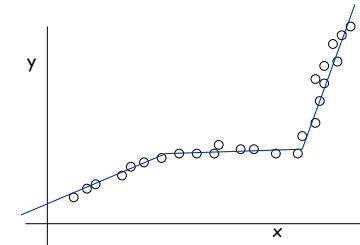


21

### 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:  $E + cL$ , for some constant  $c > 0$ .



22

### Dynamic Programming: Multiway Choice

#### Notation.

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

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

23

### 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 e_ij for
            the segment p_i, ..., p_j

        for j = 1 to n
            M[j] = min_{1 ≤ i ≤ 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.

24

## 6.4 Knapsack Problem

### 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.

$W = 11$

Item	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.

26

### Dynamic Programming: False Start

**Def.**  $OPT(i) = \text{max profit subset of items } 1, \dots, i.$

- 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!

27

### Dynamic Programming: Adding a New Variable

**Def.**  $OPT(i, w) = \text{max profit subset of items } 1, \dots, i \text{ with weight limit } w.$

- 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

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

28

### Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array

```

Input: n, w1, ..., wn, v1, ..., vn, W
for w = 0 to W
    M[0, w] = 0
for i = 1 to n
    for w = 1 to W
        if (wi > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
return M[n, W]
    
```

29

### 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

30

### Knapsack Problem: Running Time

Running time.  $\Theta(nW)$ .

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

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

31

## 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

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

33

### Edit Distance

Applications.

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

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

- Gap penalty  $\delta$ ; mismatch penalty  $\alpha_{pq}$ .
- 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  
 $\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$      $2\delta + \alpha_{CA}$

34

### 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:  $M = x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6$ .

x<sub>1</sub> x<sub>2</sub> x<sub>3</sub> x<sub>4</sub> x<sub>5</sub> x<sub>6</sub>  
C T A C C - G  
- T A C A T G  
y<sub>1</sub> y<sub>2</sub> y<sub>3</sub> y<sub>4</sub> y<sub>5</sub> y<sub>6</sub>

35

### 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  $y_1 y_2 \dots y_{j-1}$

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

36

### Sequence Alignment: Algorithm

```

Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
  for i = 0 to m
    M[i, 0] = iδ
  for j = 0 to n
    M[0, j] = jδ

  for i = 1 to m
    for j = 1 to n
      M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                    δ + M[i-1, j],
                    δ + 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?

37

### Sequence Alignment: Algorithm

Gap penalty = 1  
Mismatch penalty = 1

X<sub>1</sub> X<sub>2</sub> X<sub>3</sub> X<sub>4</sub> X<sub>5</sub> X<sub>6</sub>  
 C T A C C G  
 - T A C A T G  
 Y<sub>1</sub> Y<sub>2</sub> Y<sub>3</sub> Y<sub>4</sub> Y<sub>5</sub> Y<sub>6</sub>

38

## 6.5 RNA Secondary Structure

### RNA Secondary Structure

**RNA.** String  $B = b_1b_2...b_n$  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

40

### 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.

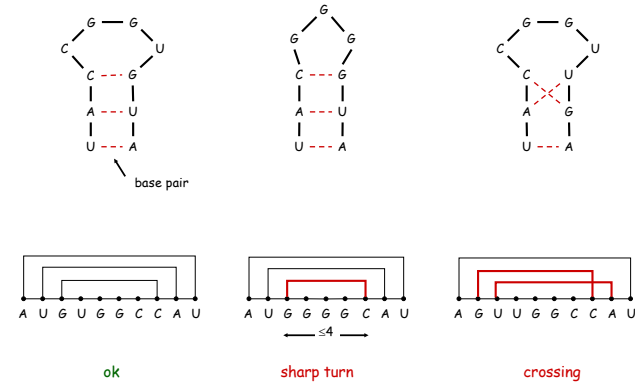
approximate by number of base pairs

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

41

### RNA Secondary Structure: Examples

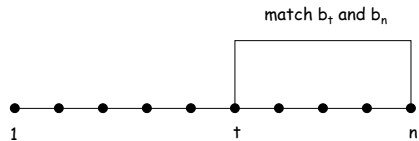
Examples.



42

### 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

43

### 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.

44

### Bottom Up Dynamic Programming Over Intervals

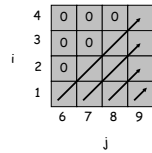
Q. What order to solve the sub-problems?

A. Do shortest intervals first.

```

RNA(b1, ..., bn) {
  for k = 5, 6, ..., n-1
    for i = 1, 2, ..., n-k
      j = i + k
      Compute M[i, j]
  return M[1, n]
}
    
```

using recurrence



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

45

### 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
- Adding a new variable: knapsack.
- 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.

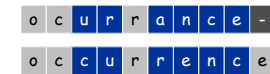
46

## 6.6 Sequence Alignment

### String Similarity

How similar are two strings?

- occurrence
- occurrence



5 mismatches, 1 gap



1 mismatch, 1 gap



0 mismatches, 3 gaps

48

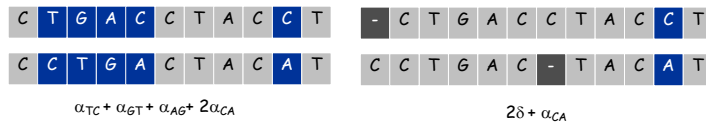
### Edit Distance

**Applications.**

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

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

- Gap penalty  $\delta$ ; mismatch penalty  $\alpha_{pq}$ .
- Cost = sum of gap and mismatch penalties.



49

### 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.

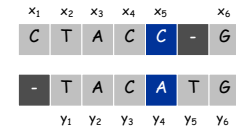
**Def.** An **alignment**  $M$  is a set of ordered pairs  $x_i-y_j$  such that each item occurs in at most one pair and no crossings.

**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:**  $M = x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6$ .



50

### 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  $y_1 y_2 \dots y_{j-1}$

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

51

### Sequence Alignment: Algorithm

```

Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, delta, alpha) {
  for i = 0 to m
    M[0, i] = i*delta
  for j = 0 to n
    M[j, 0] = j*delta

  for i = 1 to m
    for j = 1 to n
      M[i, j] = min(alpha[xi, yj] + 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?

52

## 6.7 Sequence Alignment in Linear Space

### Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in  $O(m + n)$  space and  $O(mn)$  time.

- Compute  $OPT(i, \cdot)$  from  $OPT(i-1, \cdot)$ .
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in  $O(m + n)$  space and  $O(mn)$  time.

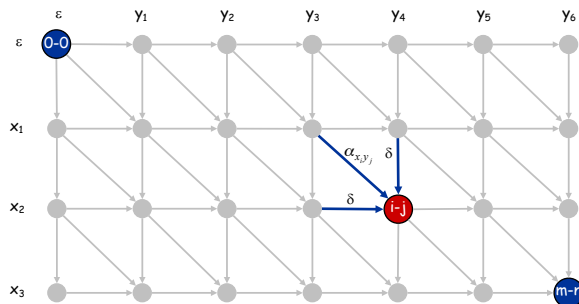
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

54

### Sequence Alignment: Linear Space

Edit distance graph.

- Let  $f(i, j)$  be shortest path from  $(0,0)$  to  $(i, j)$ .
- Observation:  $f(i, j) = OPT(i, j)$ .

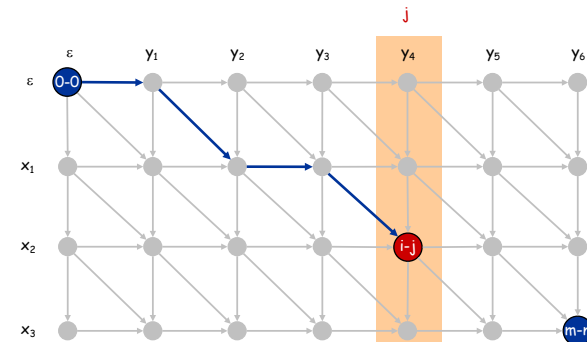


55

### Sequence Alignment: Linear Space

Edit distance graph.

- Let  $f(i, j)$  be shortest path from  $(0,0)$  to  $(i, j)$ .
- Can compute  $f(\cdot, j)$  for any  $j$  in  $O(mn)$  time and  $O(m + n)$  space.

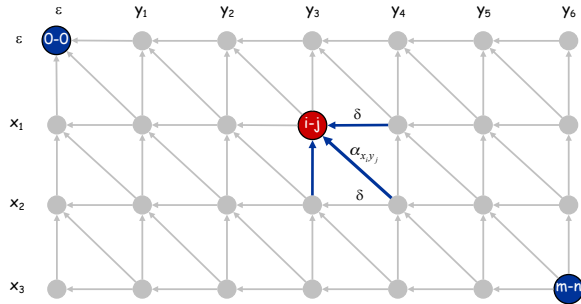


56

Sequence Alignment: Linear Space

Edit distance graph.

- Let  $g(i, j)$  be shortest path from  $(i, j)$  to  $(m, n)$ .
- Can compute by reversing the edge orientations and inverting the roles of  $(0, 0)$  and  $(m, n)$

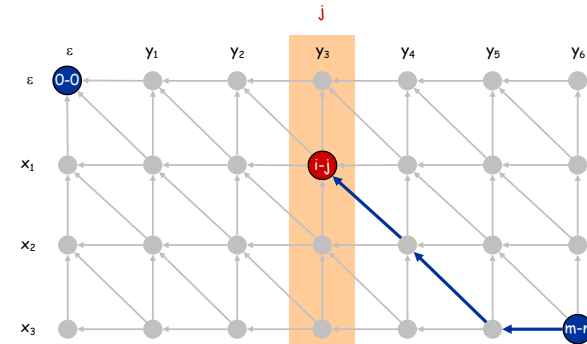


57

Sequence Alignment: Linear Space

Edit distance graph.

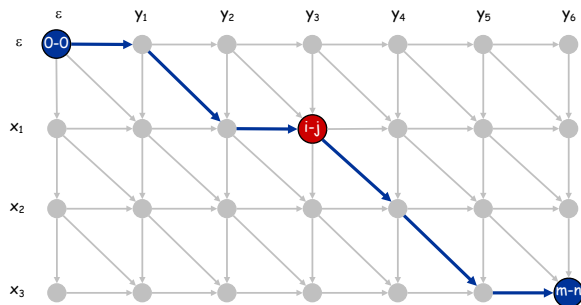
- Let  $g(i, j)$  be shortest path from  $(i, j)$  to  $(m, n)$ .
- Can compute  $g(\cdot, j)$  for any  $j$  in  $O(mn)$  time and  $O(m + n)$  space.



58

Sequence Alignment: Linear Space

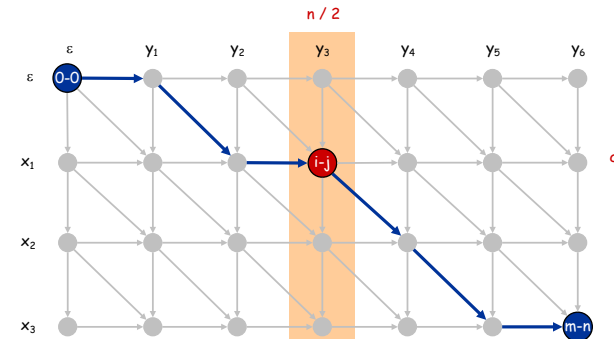
Observation 1. The cost of the shortest path that uses  $(i, j)$  is  $f(i, j) + g(i, j)$ .



59

Sequence Alignment: Linear Space

Observation 2. let  $q$  be an index that minimizes  $f(q, n/2) + g(q, n/2)$ . Then, the shortest path from  $(0, 0)$  to  $(m, n)$  uses  $(q, n/2)$ .



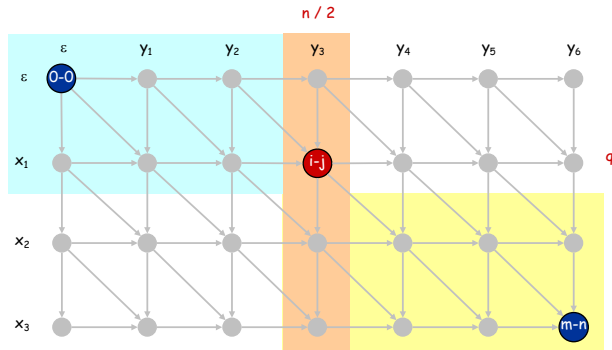
60

Sequence Alignment: Linear Space

Divide: find index  $q$  that minimizes  $f(q, n/2) + g(q, n/2)$  using DP.

- Align  $x_q$  and  $y_{n/2}$ .

Conquer: recursively compute optimal alignment in each piece.



61

Sequence Alignment: Running Time Analysis Warmup

Theorem. Let  $T(m, n)$  = max running time of algorithm on strings of length at most  $m$  and  $n$ .  $T(m, n) = O(mn \log n)$ .

$$T(m, n) \leq 2T(m, n/2) + O(mn) \Rightarrow T(m, n) = O(mn \log n)$$

Remark. Analysis is not tight because two sub-problems are of size  $(q, n/2)$  and  $(m - q, n/2)$ . In next slide, we save  $\log n$  factor.

62

Sequence Alignment: Running Time Analysis

Theorem. Let  $T(m, n)$  = max running time of algorithm on strings of length  $m$  and  $n$ .  $T(m, n) = O(mn)$ .

Pf. (by induction on  $n$ )

- $O(mn)$  time to compute  $f(\cdot, n/2)$  and  $g(\cdot, n/2)$  and find index  $q$ .
- $T(q, n/2) + T(m - q, n/2)$  time for two recursive calls.
- Choose constant  $c$  so that:

$$\begin{aligned} T(m, 2) &\leq cm \\ T(2, n) &\leq cn \\ T(m, n) &\leq cmn + T(q, n/2) + T(m - q, n/2) \end{aligned}$$

- Base cases:  $m = 2$  or  $n = 2$ .
- Inductive hypothesis:  $T(m, n) \leq 2cmn$ .

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cqn/2 + 2c(m - q)n/2 + cmn \\ &= cqn + cmn - cqn + cmn \\ &= 2cmn \end{aligned}$$

63