CSE 421: Algorithms

Winter 2014

Lecture 15: RNA secondary structure, sequence alignment

Reading:
Sections 6.3-6.7

segmented least squares

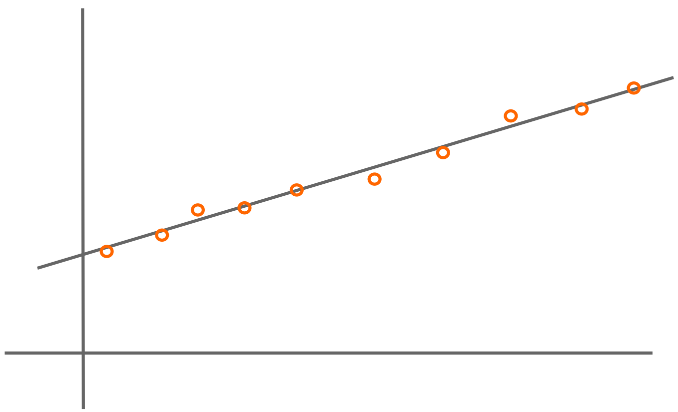
Least Squares

- Given a set **P** of **n** points in the plane $p_1=(x_1,y_1),\ldots,p_n=(x_n,y_n)$ with $x_1<\ldots<x_n$ determine a line **L** given by $y=ax+b$ that optimizes the totaled 'squared error'

  $\text{Error}(L,P)=\Sigma_i(y_i-ax_i-b)^2$
- A classic problem in statistics
- Optimal solution is known (see text for closed form)

  Call this line(**P**) and its error error(**P**)

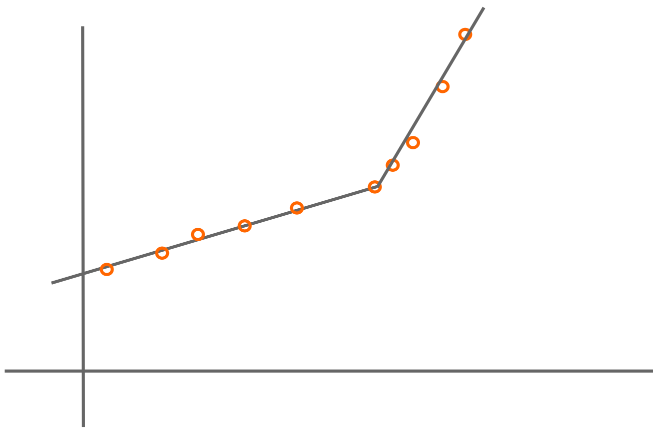
review: least squares



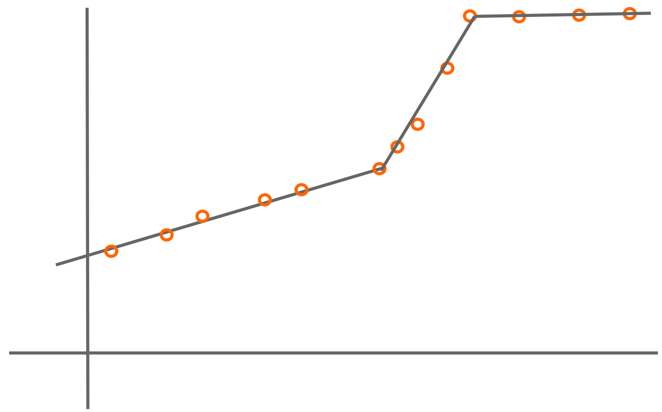
segmented least squares

What if data seems to follow a piece-wise linear model?

segmented least squares



segmented least squares



segmented least squares

- **What if data seems to follow a piece-wise linear model?**
- **Number of pieces to choose is not obvious**
- **If we chose n-1 pieces we could fit with 0 error**
  - Not fair
- **Add a penalty of C times the number of pieces to the error to get a total penalty**
- **How do we compute a solution with the smallest possible total penalty?**

segmented least squares

Recursive idea

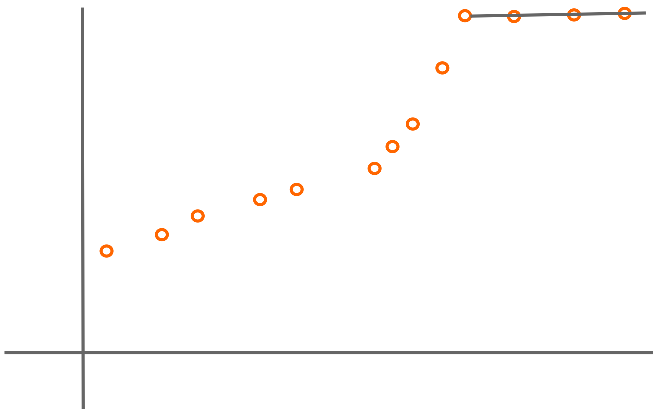
- **If we knew the point p_j where the last line segment began then we could solve the problem optimally for points $p_1,\ldots,p_j$ and combine that with the last segment to get a global optimal solution**

  Let OPT(**i**) be the optimal penalty for points $\{p_1,\ldots,p_i\}$

  Total penalty for this solution would be

  $\text{Error}(\{p_j,\ldots,p_n\}) + C + \text{OPT}(j-1)$

segmented least squares



segmented least squares

Recursive idea

- We don't know which point is p_j
  - But we do know that $1 \leq j \leq n$
  - The optimal choice will simply be the best among these possibilities
- Therefore:

segmented least squares

Recursive idea

- We don't know which point is p_j
  - But we do know that $1 \leq j \leq n$
  - The optimal choice will simply be the best among these possibilities
- Therefore:

$$\text{OPT}(\boldsymbol{n}) = \min_{1 \leq j \leq n} \{\text{Error}(\{\boldsymbol{p}_j, \dots, \boldsymbol{p}_n\}) + \boldsymbol{C} + \text{OPT}(\boldsymbol{j} - \mathbf{1})\}$$

dynamic programming solution

```
SegmentedLeastSquares(n)
   array OPT[0,...,n], Begin[1,...,n]
   OPT[0]←0
   for i=1 to n
    OPT[i]←Error{(p1,...,pi)}+C
    Begin[i]←1
    for j=2 to i-1
        e←Error{(pj,...,pi)}+C+OPT[j-1]
        if e <OPT[i] then
            OPT[i] ←e
            Begin[i]←j
        endif
    endfor
   endfor
   return(OPT[n])
```

knapsack (subset-sum) problem

- Given:
  - integer **W** (knapsack size)
  - **n** object sizes $x_1, x_2, \ldots, x_n$
- Find:
  - Subset **S** of $\{1,\ldots, n\}$ such that $\sum_{i \in S} x_i \le W$ but $\sum_{i \in S} x_i$ is as large as possible

recursive algorithm

- Let $K(n,W)$ denote the problem to solve for W and $x_1, x_2, \ldots, x_n$

recursive algorithm

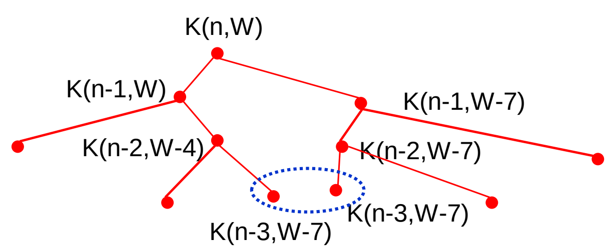
- Let $K(n,W)$ denote the problem to solve for W and $x_1, x_2, \ldots, x_n$
- For $n>0$,
  - The optimal solution for $K(n,W)$ is the better of the optimal solution for either

    $K(n-1,W)$ or $x_n+K(n-1,W-x_n)$
  - For $n=0$

    $K(0,W)$ has a trivial solution of an empty set **S** with weight **0**

recursive calls

Recursive calls on list ...,3, 4, 7



common sub-problems

- Only sub-problems are K(i,w) for
  - i = 0,1,..., n
  - w = 0,1,..., W
- Dynamic programming solution
  - Table entry for each K(i,w)

    OPT - value of optimal soln for first i objects and weight w

    belong flag - is x_i a part of this solution?
  - Initialize OPT[0,w] for w=0,...,W
  - Compute all OPT[i,*] from OPT[i-1,*] for i>0

dynamic knapsack algorithm

```
for w=0 to W;  OPT[0,w] ← 0;   end for
for i=1 to n do
    for w=0 to W do
        OPT[i,w]←OPT[i-1,w]
        belong[i,w]←0
        if  w ≥ x_i then
            val ←x_i+OPT[i-1,w-x_i]
            if val>OPT[i,w] then
                OPT[i,w]←val
                belong[i,w]←1
    end for
end for
return(OPT[n,W])
```

Time O(nW)

sample execution on 2, 3, 4, 7 with W=15

saving space

- To compute the value OPT of the solution only need to keep the last two rows of OPT at each step
- What about determining the set S?
  - Follow the belong flags O(n) time
  - What about space?

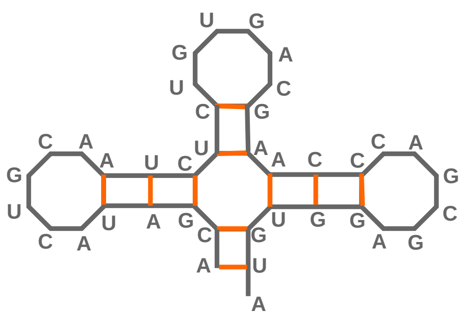
three steps to dynamic programming

- Formulate the answer as a recurrence relation or recursive algorithm
- Show that the number of different values of parameters in the recursive algorithm is "small"
  - e.g., bounded by a low-degree polynomial
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

RNA secondary structure

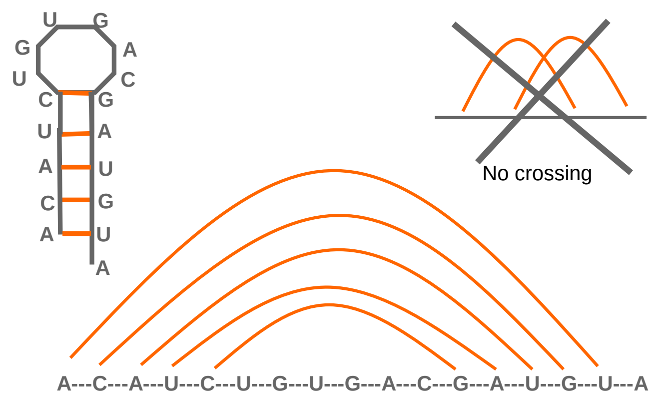
- RNA: sequence of bases
  - String over alphabet {A, C, G, U}
    U-G-U-A-C-C-G-G-U-A-G-U-A-C-A
- RNA folds and sticks to itself like a zipper
  - A bonds to U
  - C bonds to G
  - Bends can't be sharp
  - No twisting or criss-crossing
- How the bonds line up is called the RNA secondary structure

RNA secondary structure



ACGAUACUGCAAUCUCUGUGACGAACCCAGCGAGGUGUA

another view



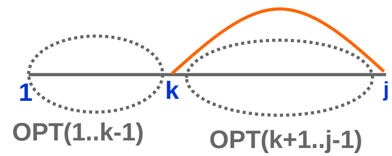
RNA secondary structure

- **Input:** String $x_1 \ldots x_n \in \{A,C,G,U\}^*$
- **Output:** Maximum size set **S** of pairs **(i,j)** such that
  - $\{x_i,x_j\}=\{A,U\}$ or $\{x_i,x_j\} =\{C,G\}$
  - The pairs in **S** form a matching
  - $i<j-4$ (no sharp bends)
  - No crossing pairs

    If (i,j) and (k,l) are in S then it is not the case that they cross as in $i<k<j<l$

recursive solution

Try all possible matches for the last base



$$OPT(1..j)=MAX(OPT(1..j-1),1+MAX_{k=1..j-5}\ (OPT(1..k-1)+OPT(k+1..j-1))$$

x_k matches x_j — Doesn't start at 1

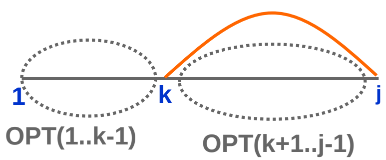
General form:

$$OPT(i..j)=MAX(OPT(i..j-1),\ 1+MAX_{k=i..j-5}\ (OPT(i..k-1)+OPT(k+1..j-1)))$$

x_k matches x_j

recursive solution

Try all possible matches for the last base



RNA secondary structure

- 2D Array **OPT(i,j)** for $i \le j$ represents optimal # of matches entirely for segment **i..j**
- For $j-i \le 4$ set $OPT(i,j)=0$ (no sharp bends)
- Then compute **OPT(i,j)** values when $j-i=5,6,\ldots,n-1$ in turn using recurrence.
- Return **OPT(1,n)**
- Total of $O(n^3)$ time
- Can also record matches along the way to produce **S**
  - Algorithm is similar to the polynomial-time algorithm for Context-Free Languages based on Chomsky Normal Form from 322
  - Both use dynamic programming over intervals

sequence alignment: edit distance

- **Given:**
  - Two strings of characters **$A=a_1\ a_2 \ldots a_n$** and **$B=b_1\ b_2 \ldots b_m$**
- **Find:**
  - The minimum number of edit steps needed to transform **A** into **B** where an edit can be:
  - insert a single character
  - delete a single character
  - substitute one character by another

sequence alignment vs editdDistance

- Sequence Alignment
  - Insert corresponds to aligning with a "**–**" in the first string

    Cost δ (in our case **1**)
  - Delete corresponds to aligning with a "**–**" in the second string

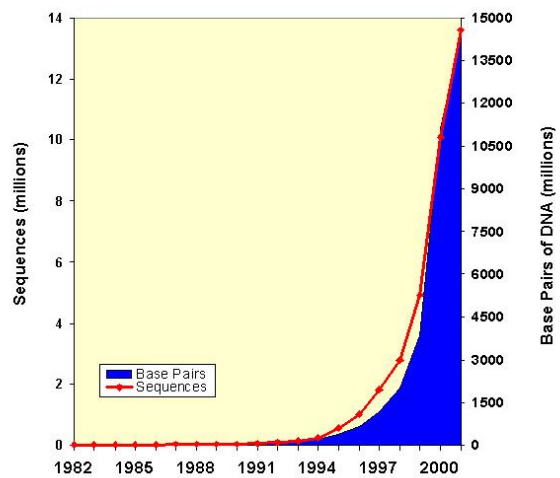
    Cost δ (in our case **1**)
  - Replacement of an **a** by a **b** corresponds to a mismatch

    Cost α_{ab} (in our case **1** if **$a \neq b$** and **0** if **$a=b$**)
- In Computational Biology this alignment algorithm is attributed to Smith & Waterman

applications

- "diff" utility – where do two files differ
- Version control & patch distribution – save/send only changes
- Molecular biology
  - Similar sequences often have similar origin and function
  - Similarity often recognizable despite millions or billions of years of evolutionary divergence

Growth of GenBank



recursive solutions

- Sub-problems: Edit distance problems for **all prefixes** of A and B that don't include all of both A and B
- Let D(i,j) be the number of edits required to transform $a_1\ a_2 \ldots a_i$ into $b_1\ b_2 \ldots b_j$
- Clearly D(0,0)=0

computing D(n,m)

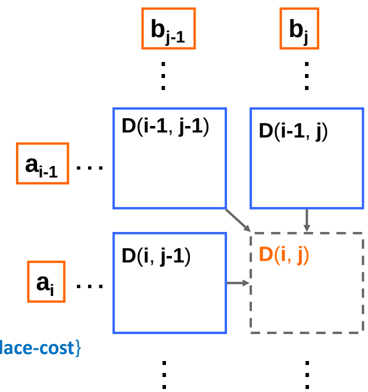
- Imagine how best sequence handles the last characters a_n and b_m
- If best sequence of operations
  - deletes a_n then D(n,m)=D(n-1,m)+1
  - inserts b_m then D(n,m)=D(n,m-1)+1
  - replaces a_n by b_m then D(n,m)=D(n-1,m-1)+1
  - matches a_n and b_m then D(n,m)=D(n-1,m-1)

recursive algorithm D(n,m)

```
if n=0 then
    return (m)
else if m=0 then
    return(n)
else
    if a_n=b_m then
        replace-cost ← 0
    else                          } cost of substitution of a_n by b_m (if used)
        replace-cost ← 1
    endif
    return(min{ D(n-1, m) + 1,
                D(n, m-1) + 1,
                D(n-1, m-1) + replace-cost } )
```

dynamic programming

```
for j = 0 to m; D(0,j) ← j; endfor
for i = 1 to n; D(i,0) ← i; endfor
for i = 1 to n
    for j = 1 to m
        if a_i=b_j then
            replace-cost ← 0
        else
            replace-cost ← 1
        endif
        D(i,j) ← min { D(i-1, j) + 1,
                       D(i, j-1) + 1,
                       D(i-1, j-1) + replace-cost}
    endfor
endfor
```



example run with AGACATTG and GAGTTA

	0	A 1	G 2	A 3	C 4	A 5	T 6	T 7	G 8
0									
G 1									
A 2									
G 3									
T 4									
T 5									
A 6									

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2								
G	3								
T	4								
T	5								
A	6								

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1					
G	3								
T	4								
T	5								
A	6								

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4								
T	5								
A	6								

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	**1**	2	3	4	5	6	7	8
G	1	1	**1**	2	3	4	5	6	7
A	2	1	2	**1**	2	3	4	5	6
G	3	2	1	2	**2**	**3**	4	5	5
T	4	3	2	2	3	3	**3**	4	5
T	5	4	3	3	3	4	3	**3**	4
A	6	5	4	3	4	3	4	4	**4**