CSE 421: Algorithms

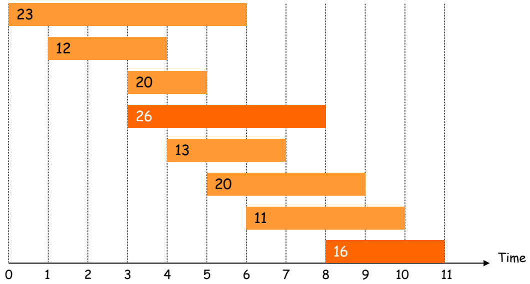
Winter 2014

Lecture 14: Dynamic programming II

Reading:
Sections 6.2-6.6

weighted interval scheduling

- **Input.** Set of jobs with start times, finish times, and weights.
- **Goal.** Find maximum weight subset of mutually compatible jobs.



step 1 – a recursive algorithm

- Suppose that like ordinary interval scheduling we have first sorted the requests by finish time f_i so
  $f_1 \le f_2 \le \cdots \le f_n$
- Say request **i** comes before request **j** if **i< j**
- For any request **j** let **p(j)** be
  - the largest-numbered request before **j** that is compatible with **j**
  - or **0** if no such request exists
- Therefore **{1,...,p(j)}** is precisely the set of requests before **j** that are compatible with **j**

step 1 – a recursive algorithm

- All subproblems involve requests **{1,.., i }** for some **i**
- For **i=1,...,n** let **OPT(i)** be the weight of the optimal solution to the problem **{1, ..., i}**
- The two cases give
  $\mathbf{OPT}(\boldsymbol{n}) = \max[\boldsymbol{w}_{\boldsymbol{n}} + \mathbf{OPT}(\boldsymbol{p}(\boldsymbol{n})), \mathbf{OPT}(\boldsymbol{n}-\mathbf{1})]$
- Also
  $n \in \mathcal{O}$ iff $w_n + OPT(p(n)) > OPT(n-1)$

step 1 – a recursive algorithm

First, sort requests and compute array **p[i]** for each **i = 1, …, n.**

```
ComputeOpt(n)
    if n=0 then return(0)
    else
        u←ComputeOpt(p[n])
        v←ComputeOpt(n-1)
        if $w_n$+u>v then
            return($w_n$+u)
        else
            return(v)
    endif
```

step 2 – memoization

- ComputeOpt(n) can take exponential time in the worst case
  - 2^n calls if p(i)=i-1 for every i
- There are only n possible parameters to ComputeOpt
- Store these answers in an array OPT[n] and only recompute when necessary
  - Memoization
- Initialize OPT[i]=0 for i=1,…,n

step 2 – memoization

```
ComputeOpt(n):
    if n=0 then return(0)
    else
        u←MComputeOpt(p[n])
        v←MComputeOpt(n-1)
        if $w_n$+u>v then
            return($w_n$+u)
        else return(v)
    endif
```

```
MComputeOpt(n):
    if OPT[n]=0 then
        v←ComputeOpt(n)
        OPT[n]←v
        return(v)
    else
        return(OPT[n])
    endif
```

step 3 – iterative solution

The recursive calls for parameter n have parameter values i that are < n

step 3 – iterative solution

The recursive calls for parameter **n** have parameter values **i** that are < **n**

```
IterativeComputeOpt(n)
    array OPT[0,…,n]
    OPT[0]←0
    for i=1 to n
        if $w_i$+OPT[p[i]] >OPT[i-1] then
            OPT[i] ←$w_i$+OPT[p[i]]
        else
            OPT[i] ←OPT[i-1]
        endif
    endfor
```

producing an optimal solution

```
IterativeComputeOptSolution(n)
  array OPT[0,…,n], Used[1,…,n]
  OPT[0]←0
  for i=1 to n
    if $w_i$+OPT[p[i]] >OPT[i-1] then
      OPT[i] ←$w_i$+OPT[p[i]]
      Used[i]←1
    else
      OPT[i] ← OPT[i-1]
      Used[i] ←0
    endif
  endfor
```

producing an optimal solution

```
IterativeComputeOptSolution(n)
  array OPT[0,…,n], Used[1,…,n]
  OPT[0]←0
  for i=1 to n
    if $w_i$+OPT[p[i]] >OPT[i-1] then
      OPT[i] ←$w_i$+OPT[p[i]]
      Used[i]←1
    else
      OPT[i] ← OPT[i-1]
      Used[i] ←0
    endif
  endfor
```

```
i ←n
S ← ∅
while i > 0 do
    if Used[i]=1 then
        S ← S ∪ {i}
        i ← p[i]
    else
        i ← i - 1
    endif
endwhile
```

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]									
OPT[i]									
Used[i]									

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]									
Used[i]									

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]	3	7	7	8	10	12	14	14	16
Used[i]	1	1	0	1	1	1	1	0	1

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]	3	7	7	8	10	12	14	14	16
Used[i]	1	1	0	1	1	1	1	0	1

S={9,7,2}

segmented least squares

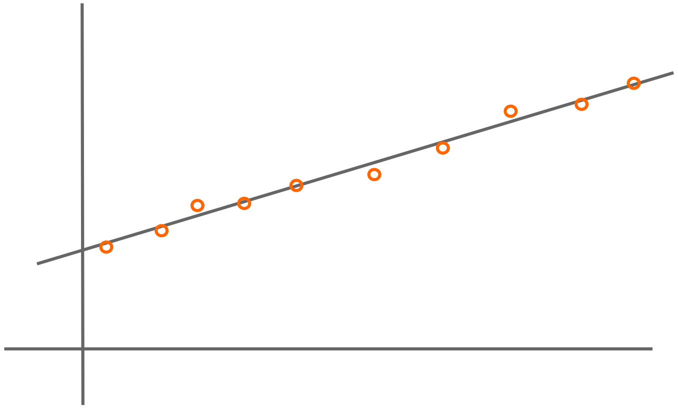
Least Squares

- **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)**

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

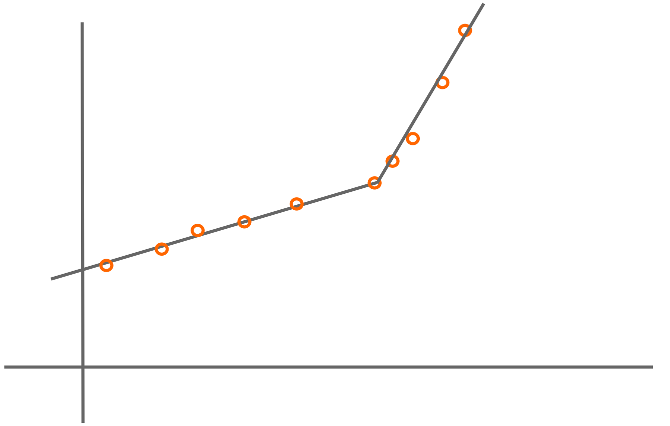
least squares



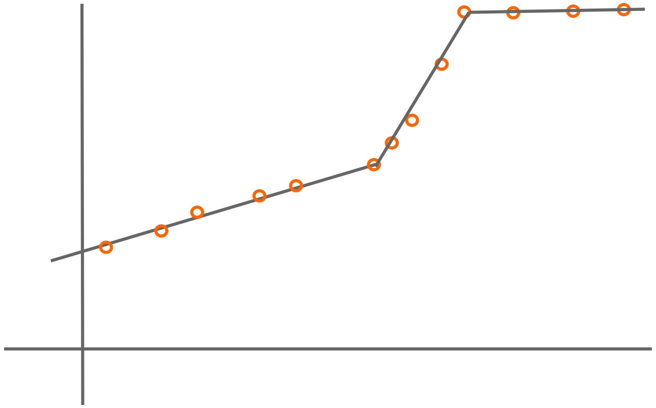
segmented least squares

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

segmented least squares



segmented least squares



segmented least squares

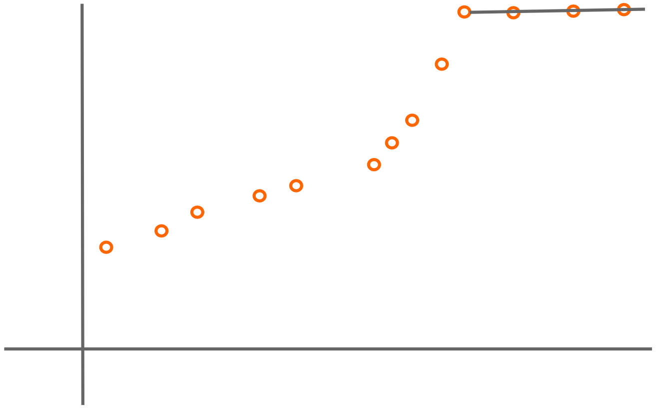
- What if data seems to follow a piece-wise linear model?
- Number of pieces to choose is not obvious
- If we chose **n-1** pieces we could fit with **0** error
  - Not fair
- Add a penalty of **C** times the number of pieces to the error to get a total penalty
- How do we compute a solution with the 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
    - Error($\{p_j,\ldots,p_n\}$) + **C** + 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≤j≤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 $1 \le j \le n$
  - The optimal choice will simply be the best among these possibilities
- Therefore:

$$\mathrm{OPT}(n) = \min_{1\le j\le n} \{\mathrm{Error}(\{p_j,\dots,p_n\}) + C + \mathrm{OPT}(j-1)\}$$

dynamic programming solution

```
SegmentedLeastSquares(n)
   array OPT[0,...,n], Begin[1,...,n]
   OPT[0]←0
   for i=1 to n
    OPT[i]←Error{($p_1$,...,$p_i$)}+C
    Begin[i]←1
    for j=2 to i-1
        e←Error{($p_j$,...,$p_i$)}+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 $x_1, x_2, \dots, x_n$
- Find:
  - Subset **S** of $\{1,\dots, n\}$ 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 $x_1, x_2, \dots, x_n$

recursive algorithm

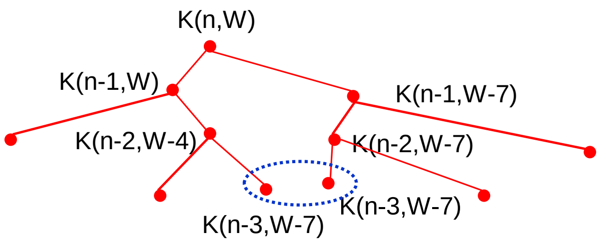
- Let **K(n,W)** denote the problem to solve for **W** and $x_1, x_2, \ldots, x_n$
- For $n>0$,
  - The optimal solution for **K(n,W)** is the better of the optimal solution for either

    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
  - $i = 0,1,\ldots, n$
  - $w = 0,1,\ldots, W$
- 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  $w \ge x_i$ 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?

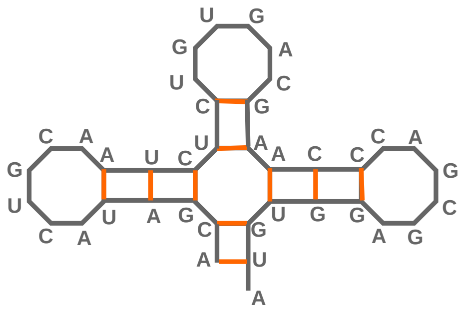
three steps to dynamic programming

- 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

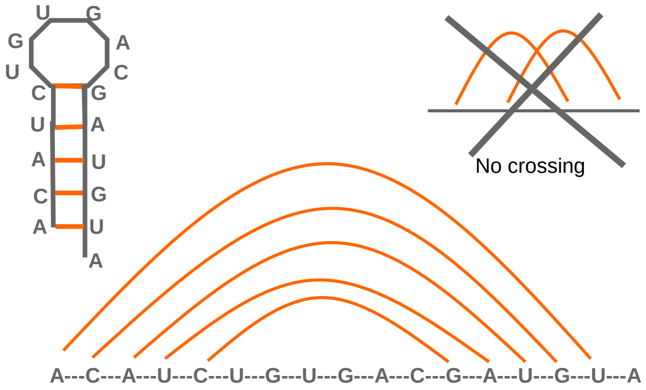
- RNA: sequence of bases
  - String over alphabet {**A**, **C**, **G**, **U**}
    U-G-U-A-C-C-G-G-U-A-G-U-A-C-A
- RNA folds and sticks to itself like a zipper
  - **A** bonds to **U**
  - **C** bonds to **G**
  - Bends can’t be sharp
  - No twisting or criss-crossing
- How the bonds line up is called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

RNA secondary structure



ACGAUACUGCAAUCUCUGUGACGAACCCAGCGAGGUGUA

another view



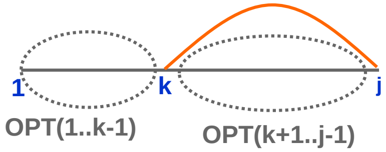
RNA secondary structure

- **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



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

x_k matches x_j

Doesn't start at 1

General form:

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

x_k matches x_j

RNA secondary structure

- 2D Array **OPT(i,j)** for **i≤j** represents optimal # of matches entirely for segment **i..j**
- For **j-i ≤4** set **OPT(i,j)=0** (no sharp bends)
- Then compute **OPT(i,j)** values when **j-i=5,6,…,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