CS 473ug: Algorithms

Mahesh Viswanathan
vmahesh@cs.uiuc.edu
3232 Siebel Center

University of Illinois, Urbana-Champaign

Spring 2008

Part I

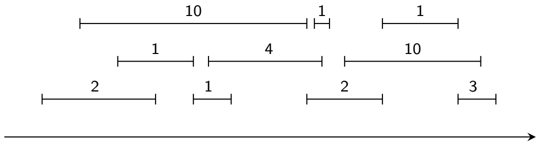
Dynamic Programming: An Introduction

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and weights

Goal Schedule jobs so that total weight of jobs is maximized

- Two jobs with overlapping intervals cannot both be scheduled!

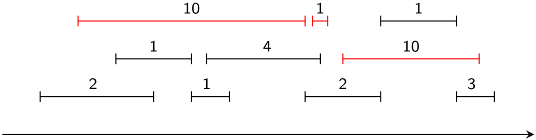


Weighted Interval Scheduling

Input A set of jobs with start times, finish times and weights

Goal Schedule jobs so that total weight of jobs is maximized

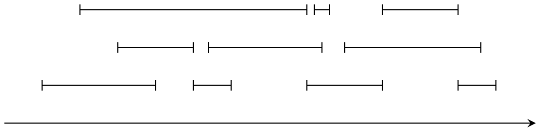
- Two jobs with overlapping intervals cannot both be scheduled!



Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

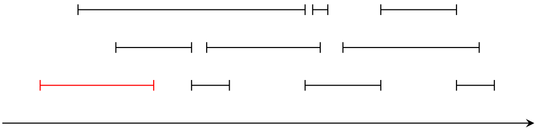


Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule

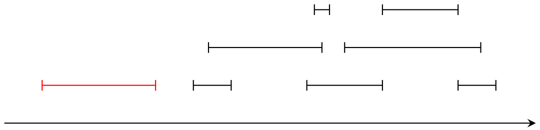


Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule

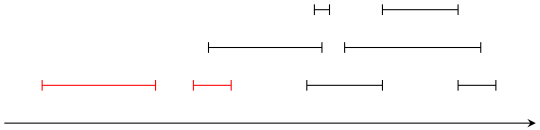


Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule

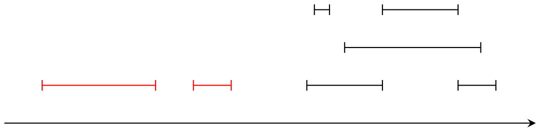


Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule

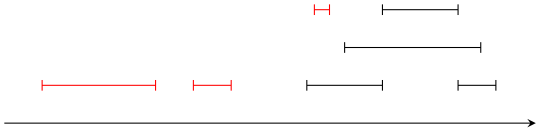


Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule

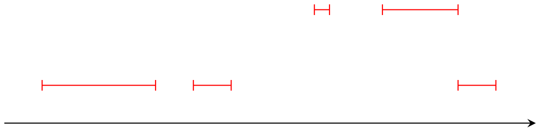


Interval Scheduling

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1

Goal Schedule as many jobs as possible

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule

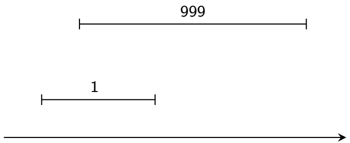


Greedy Strategy for Weighted Interval Scheduling

- Pick jobs in order of finishing times
- Add job to schedule if it does not conflict with current schedule

Greedy Strategy for Weighted Interval Scheduling

- Pick jobs in order of finishing times
- Add job to schedule if it does not conflict with current schedule



Moral: Greedy strategies often don't work!

Conventions

Definition

Conventions

Definition

- Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$

Conventions

Definition

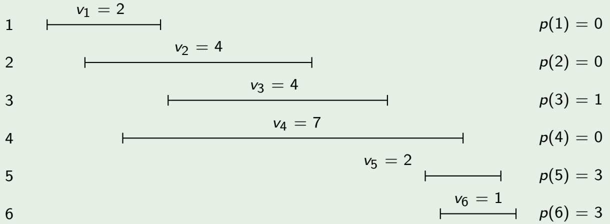
- Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$
- Define $p(j)$ to be the largest i (less than j) such that job i and job j are not in conflict

Conventions

Definition

- Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$
- Define $p(j)$ to be the largest i (less than j) such that job i and job j are not in conflict

Example



Towards a Recursive Solution

Observation

Consider an optimal schedule $\mathcal{O}$

Case $n \in \mathcal{O}$ *None of the jobs between n and $p(n)$ can be scheduled. Moreover $\mathcal{O}$ must contain an optimal schedule for the first $p(n)$ jobs.*

Case $n \notin \mathcal{O}$ *$\mathcal{O}$ is an optimal schedule for the first $n-1$ jobs!*

A Recursive Algorithm

```
Let O_1 be optimal schedule for the first p(n) jobs, computed
recursively
Let O_2 be optimal schedule for the first n-1 jobs, computed
recursively
If (O_1 + v_n < O_2) then
    optimal schedule is O_2
else
    optimal schedule is O_1 ∪ {n}
```

A Recursive Algorithm

```
Let O_1 be optimal schedule for the first p(n) jobs, computed
recursively
Let O_2 be optimal schedule for the first n−1 jobs, computed
recursively
If (O_1 + v_n < O_2) then
    optimal schedule is O_2
else
    optimal schedule is O_1 ∪ {n}
```

Time Analysis

Running time is $T(n) = T(p(n)) + T(n-1) + O(1)$ which is ...

Bad Example

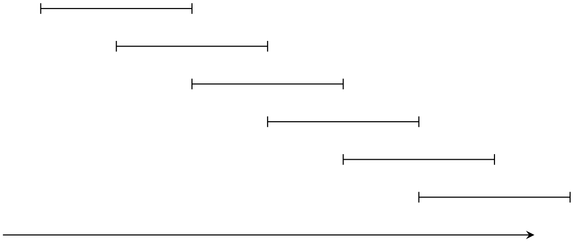


Figure: Bad instance for recursive algorithm

Running time on this instance is

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Bad Example

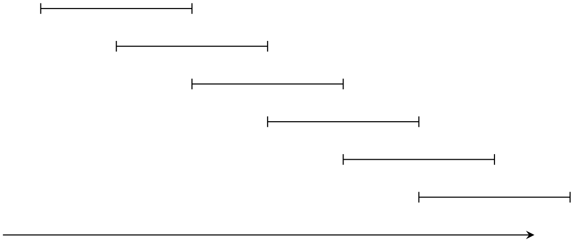


Figure: Bad instance for recursive algorithm

Running time on this instance is

$$T(n) = T(n-1) + T(n-2) + O(1) = \Theta(\phi^n)$$

where $\phi \approx 1.618$ is the golden ratio.

Analysis of the Problem

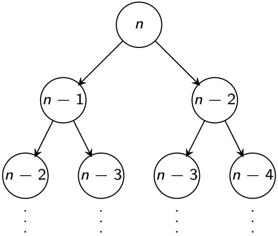


Figure: Label of node indicates size of sub-problem. Tree of sub-problems grows very quickly

Memo(r)ization

Observation

Memo(r)ization

Observation

- *Number of different sub-problems in recursive algorithm is*

Memo(r)ization

Observation

- *Number of different sub-problems in recursive algorithm is* $O(n)$

Memo(r)ization

Observation

- *Number of different sub-problems in recursive algorithm is* $O(n)$
- *Exponential time is due to recomputation of solutions to sub-problems*

Memo(r)ization

Observation

- *Number of different sub-problems in recursive algorithm is* $O(n)$
- *Exponential time is due to recomputation of solutions to sub-problems*

Solution

Store optimal solution to different sub-problems, and perform recursive call only if not already computed.

Recursive Solution with Memoization

```
computeOpt(int j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max(v_j + computeOpt(p(j)), computeOpt(j-1))
        return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one `M[·]`

Recursive Solution with Memoization

```
computeOpt(int j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max(v_j + computeOpt(p(j)), computeOpt(j-1))
        return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one `M[·]`
- Initially no entry of `M[]` is filled

Recursive Solution with Memoization

```
computeOpt(int j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max(v_j + computeOpt(p(j)), computeOpt(j-1))
        return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one $\texttt{M}[\cdot]$
- Initially no entry of `M[]` is filled; at the end all entries of `M[]` are filled

Recursive Solution with Memoization

```
computeOpt(int j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max(v_j + computeOpt(p(j)), computeOpt(j-1))
        return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one `M[·]`
- Initially no entry of `M[]` is filled; at the end all entries of `M[]` are filled
- So total time is $O(n)$

Automatic Memoization

Fact

Many functional languages (like LISP) automatically do memoization for recursive function calls!

Back to Weighted Interval Scheduling

Iterative Solution

```
M[0] = 0
for i = 1 to n
    M[i] = max(v_i + M[p(i)], M[i-1])
```

Computing Solutions

- Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

Computing Solutions

- Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n
    M[i] = max(v_i + M[p(i)], M[i-1])
    S[i] = v_i + M[p(i)] < M[i-1] ?  S[i-1] :  S[p(i)] U i
```

Computing Solutions

- Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n
    M[i] = max(v_i + M[p(i)], M[i-1])
    S[i] = v_i + M[p(i)] < M[i-1] ?  S[i-1] :  S[p(i)] U i
```

Computing Solutions

- Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n
    M[i] = max(v_i + M[p(i)], M[i-1])
    S[i] = v_i + M[p(i)] < M[i-1] ?  S[i-1] :  S[p(i)] U i
```

- Naïvely updating S[] takes $O(n)$ time

Computing Solutions: First Attempt

- Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n
    M[i] = max(v_i + M[p(i)], M[i-1])
    S[i] = v_i + M[p(i)] < M[i-1] ?  S[i-1] :  S[p(i)] U i
```

- Naïvely updating `S[]` takes $O(n)$ time
- Total running time is $O(n^2)$

Computing Implicit Solutions

Observation

Solution can be obtained from `M[]` *in* $O(n)$ *time, without any additional information*

```
findSolution(int j)
    if (j=0) then return empty schedule
    if (v_j + M[p(j)] > M[j-1]) then
        return findSolution(p(j)) U {j}
    else
        return findSolution(j-1)
```

Makes $O(n)$ *recursive calls, so* `findSolution` *runs in* $O(n)$ *time.*

Dynamic Programming

Dynamic Programming = Recursion + Memoization

Dynamic Programming

Dynamic Programming = Recursion + Memoization

Pattern

Dynamic Programming

Dynamic Programming = Recursion + Memoization

Pattern

1. Formulate problem recursively in terms of solutions to polynomially many sub-problems

Dynamic Programming

Dynamic Programming = Recursion + Memoization

Pattern

1. Formulate problem recursively in terms of solutions to polynomially many sub-problems
2. Solve sub-problems bottom-up, storing optimal solutions

Historical Perspective

- Systematic study pioneered by Bellman in the 1950s

Historical Perspective

- Systematic study pioneered by Bellman in the 1950s
- Name formulated to avoid confrontation with Secretary of Defense

Historical Perspective

- Systematic study pioneered by Bellman in the 1950s
- Name formulated to avoid confrontation with Secretary of Defense
    - "It is impossible to use dynamic in a pejorative sense"

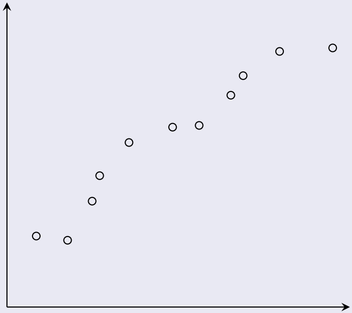
Historical Perspective

- Systematic study pioneered by Bellman in the 1950s
- Name formulated to avoid confrontation with Secretary of Defense
  - "It is impossible to use dynamic in a pejorative sense"
  - "... something not even a Congressman could object to"

Interpreting Data

Problem

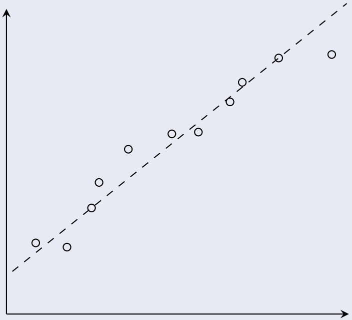
Given a sequence of observations, find a line that best describes the data.



Interpreting Data

Problem

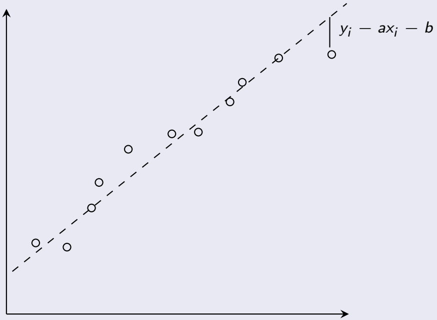
Given a sequence of observations, find a line that best describes the data.



Interpreting Data

Problem

Given a sequence of observations, find a line that best describes the data.



- For a sequence of data points $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ and a line $y = ax + b$, the squared error is

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

Best Fit Line

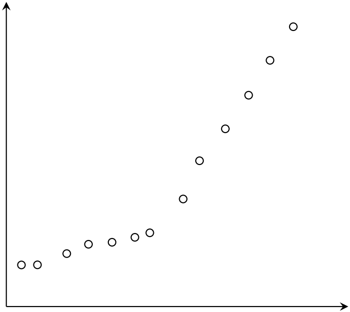
Proposition

For a set of points $(x_1, y_2), (x_2, y_2), \ldots (x_n, y_n)$ the best fit line is given by $y = ax + b$, where

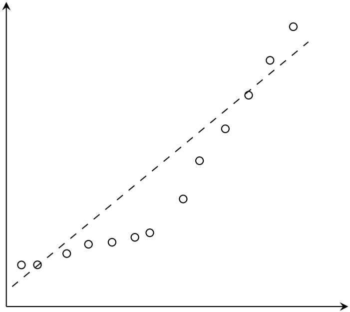
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}$$

$$b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Best fit line



Best fit line



Best fit line segments

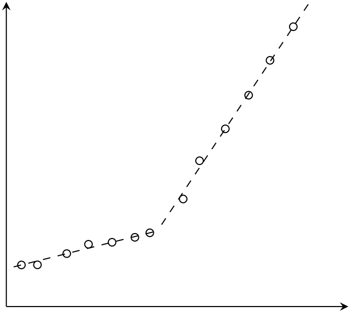


Figure: Points lie on two lines not one

Segmented Least Squares

Problem

Given a set of points $\mathbb{D} = \{p_1, p_2, \dots p_n\}$ ($p_i = (x_i, y_i)$), find a set of line segments such that $\mathrm{Error}(\mathbb{D})$ is minimized

Segmented Least Squares

Problem

Given a set of points $\mathbb{D} = \{p_1, p_2, \ldots p_n\}$ $(p_i = (x_i, y_i))$, find a set of line segments such that $\mathrm{Error}(\mathbb{D})$ is minimized

Error Metric

What is $\mathrm{Error}(\mathbb{D})$?

Segmented Least Squares

Problem

Given a set of points $\mathbb{D} = \{p_1, p_2, \ldots p_n\}$ ($p_i = (x_i, y_i)$), find a set of line segments such that $\mathrm{Error}(\mathbb{D})$ is minimized

Error Metric

What is $\mathrm{Error}(\mathbb{D})$? Needs to balance "goodness of fit" and "as few lines as possible"

Segmented Least Squares

Problem

Given a set of points $\mathbb{D} = \{p_1, p_2, \ldots p_n\}$ $(p_i = (x_i, y_i))$, find a set of line segments such that $\mathrm{Error}(\mathbb{D})$ is minimized

Error Metric

What is $\mathrm{Error}(\mathbb{D})$? Needs to balance "goodness of fit" and "as few lines as possible"

- Add the squared error of each line segment

Segmented Least Squares

Problem

Given a set of points $\mathbb{D} = \{p_1, p_2, \ldots p_n\}$ $(p_i = (x_i, y_i))$, find a set of line segments such that $\mathrm{Error}(\mathbb{D})$ is minimized

Error Metric

What is $\mathrm{Error}(\mathbb{D})$? Needs to balance "goodness of fit" and "as few lines as possible"

- Add the squared error of each line segment
- If L lines are used then add cL to the error

Partitioning and Line Segments

Observation

Partitioning and Line Segments

Observation

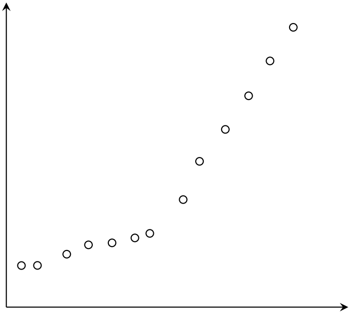
- *Let the list of points* $\mathbb{D} = \{p_1, p_2, \ldots p_n\}$ *be sorted according to the x-coordinate, i.e.,* $x_i < x_j$ *for* $i < j$

Partitioning and Line Segments

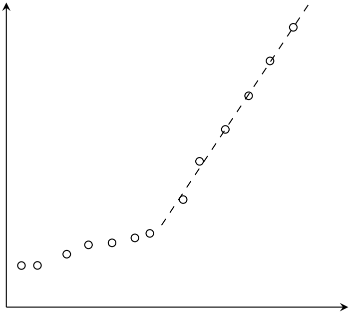
Observation

- *Let the list of points* $\mathbb{D} = \{p_1, p_2, \ldots p_n\}$ *be sorted according to the x-coordinate, i.e.,* $x_i < x_j$ *for* $i < j$
- *A line segment must pass through a contiguous subset of* $\mathbb{D}$ *in the sorted order*

Structure of Optimal Solution

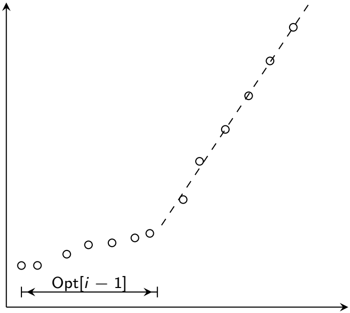


Structure of Optimal Solution



- Suppose the last point $p_n = (x_n, y_n)$ is part of a segment that starts at $p_i = (x_i, y_i)$

Structure of Optimal Solution



- Suppose the last point $p_n = (x_n, y_n)$ is part of a segment that starts at $p_i = (x_i, y_i)$
- Then optimal solution is optimal solution for $\{p_1, \ldots p_{i-1}\}$ plus (best) line through $\{p_i, \ldots p_n\}$

Cost of Optimal Solution

- Suppose the last point p_n is part of a segment that starts at p_i
- If $\mathrm{Opt}(j)$ denotes the cost of the first j points and $e(j,k)$ the error of the best line through points j to k then

$$\mathrm{Opt}(n) = e(i,n) + C + \mathrm{Opt}(i-1)$$

Cost of Optimal Solution

- Suppose the last point p_n is part of a segment that starts at p_i
- If $\mathrm{Opt}(j)$ denotes the cost of the first j points and $e(j,k)$ the error of the best line through points j to k then

$$\mathrm{Opt}(n) = e(i,n) + C + \mathrm{Opt}(i-1)$$

- How do we find where the last segment ends?

Cost of Optimal Solution

- Suppose the last point p_n is part of a segment that starts at p_i
- If $\mathrm{Opt}(j)$ denotes the cost of the first j points and $e(j,k)$ the error of the best line through points j to k then

$$\mathrm{Opt}(n) = e(i,n) + C + \mathrm{Opt}(i-1)$$

- How do we find where the last segment ends? We find the i that minimizes the above equation!

Dynamic Programming Solution

let $\{(x_1, y_1), \ldots (x_n, y_n)\}$ be list in sorted order
M[0] = 0
for all pairs (i, j) where $i \leq j$
    compute e(i, j) the least squares error for $\{(x_i, y_i), \ldots (x_j, y_j)\}$
for j = 1 to n
    M[j] = $\min_{1 \leq i \leq j}$ (e(i, j) + C + M[j-1])

Analysis

Dynamic Programming Solution

let $\{(x_1, y_1), \ldots (x_n, y_n)\}$ be list in sorted order
M[0] = 0
for all pairs (i, j) where $i \leq j$
    compute e(i, j) the least squares error for $\{(x_i, y_i), \ldots (x_j, y_j)\}$
for j = 1 to n
    M[j] = $\min_{1 \leq i \leq j}$ (e(i, j) + C + M[j-1])

Analysis

- $O(n^2)$ values of $e(i, j)$; each value takes $O(n)$ time

Dynamic Programming Solution

```
let {(x_1, y_1), ... (x_n, y_n)} be list in sorted order
M[0] = 0
for all pairs (i,j) where i ≤ j
    compute e(i,j) the least squares error for {(x_i, y_i), ... (x_j, y_j)}
for j = 1 to n
    M[j] = min_{1≤i≤j} (e(i,j) + C + M[j-1])
```

Analysis

- $O(n^2)$ values of $e(i,j)$; each value takes $O(n)$ time
- Computing `M[]` after $e(i,j)$ is $O(n^2)$

Dynamic Programming Solution

let $\{(x_1, y_1), \ldots (x_n, y_n)\}$ be list in sorted order
M[0] = 0
for all pairs (i, j) where $i \le j$
    compute e(i, j) the least squares error for $\{(x_i, y_i), \ldots (x_j, y_j)\}$
for j = 1 to n
    M[j] = $\min_{1 \le i \le j}$ (e(i, j) + C + M[j-1])

Analysis

- $O(n^2)$ values of $e(i, j)$; each value takes $O(n)$ time
- Computing M[] after $e(i, j)$ is $O(n^2)$
- Total time $O(n^3) + O(n^2)$

Improving the Running Time

- Recall variance and covariance of a set of points $\{p_1, p_2, \ldots p_n\}$, where $p_i = (x_i, y_i)$, is defined as follows

$$\sigma_x^2 = \frac{\sum_i x_i^2 - n\bar{x}^2}{n} \qquad \sigma_y^2 = \frac{\sum_i y_i^2 - n\bar{y}^2}{n}$$

$$\sigma_{x,y} = \frac{\sum_i x_i y_i - n\bar{x}\bar{y}}{n}$$

  where $\bar{x}$ and $\bar{y}$ are the means of $\{x_i\}_{i=1}^n$ and $\{y_i\}_{i=1}^n$, respectively.

Improving the Running Time

- Recall variance and covariance of a set of points $\{p_1, p_2, \ldots p_n\}$, where $p_i = (x_i, y_i)$, is defined as follows

$$\sigma_x^2 = \frac{\sum_i x_i^2 - n\bar{x}^2}{n} \qquad \sigma_y^2 = \frac{\sum_i y_i^2 - n\bar{y}^2}{n}$$

$$\sigma_{x,y} = \frac{\sum_i x_i y_i - n\bar{x}\bar{y}}{n}$$

where $\bar{x}$ and $\bar{y}$ are the means of $\{x_i\}_{i=1}^n$ and $\{y_i\}_{i=1}^n$, respectively.

- The squared error of the best-fitting line can be expressed as

$$n(\sigma_y^2 - \frac{\sigma_{x,y}^2}{\sigma_x^2})$$

Improving the Running Time

- Recall variance and covariance of a set of points $\{p_1, p_2, \ldots p_n\}$, where $p_i = (x_i, y_i)$, is defined as follows

$$\sigma_x^2 = \frac{\sum_i x_i^2 - n\bar{x}^2}{n} \qquad \sigma_y^2 = \frac{\sum_i y_i^2 - n\bar{y}^2}{n}$$
$$\sigma_{x,y} = \frac{\sum_i x_i y_i - n\bar{x}\bar{y}}{n}$$

  where $\bar{x}$ and $\bar{y}$ are the means of $\{x_i\}_{i=1}^n$ and $\{y_i\}_{i=1}^n$, respectively.
- The squared error of the best-fitting line can be expressed as

$$n(\sigma_y^2 - \frac{\sigma_{x,y}^2}{\sigma_x^2})$$

- The variance and covariance of $\{p_i, \ldots p_{i+k+1}\}$ can be obtained from the variance and covariance of $\{p_i, \ldots p_{i+k}\}$ using constantly many operations.

Faster Calculation of $e(i,j)$

Faster Calculation of $e(i, j)$

```
for all i varX[i,i] = x_i
for all i varY[i,i] = y_i
for all i covar[i,i] = 0
for k = 1 to n-1
    for i = 1 to n-k
        compute varX[i,i+k] from varX[i,i+k-1]
        compute varY[i,i+k] from varY[i,i+k-1]
        compute covar[i,i+k] from covar[i,i+k-1]
        compute e[i,i+k] from varX[i,i+k], varY[i,i+k], covar[i,i+k]
```

Running Time $O(n^2)$

Computing the Segments

Recall `M[j]` stores the cost of the best way to partition $\{(x_1, y_1), \ldots (x_j, y_j)\}$ into line segments

Computing the Segments

Recall `M[j]` stores the cost of the best way to partition $\{(x_1, y_1), \ldots (x_j, y_j)\}$ into line segments

```
findSegments(int j)
    if j = 0 then return empty sequence
    else
        find i such that (e(i,j) + C + M[i-1]) is minimized
        return findSegments(i-1) plus {(x_i, y_i), ... (x_j, y_j)}
```