Announcements

- Homework 3 online, due Friday
- Akhila's OH now online at https://ucsd.zoom.us/j/93415925061

Last Time

- Divide and Conquer
- Sorting
- Closest Pair of Points

Divide and Conquer

This is the first of our three major algorithmic techniques.

1. Break problem into pieces
2. Solve pieces recursively
3. Recombine pieces to get answer

Today

- Closest Pair of Points
- Greedy Algorithms
  - Making change
  - Interval scheduling

Closest Pair of Points (Ex 2.32)

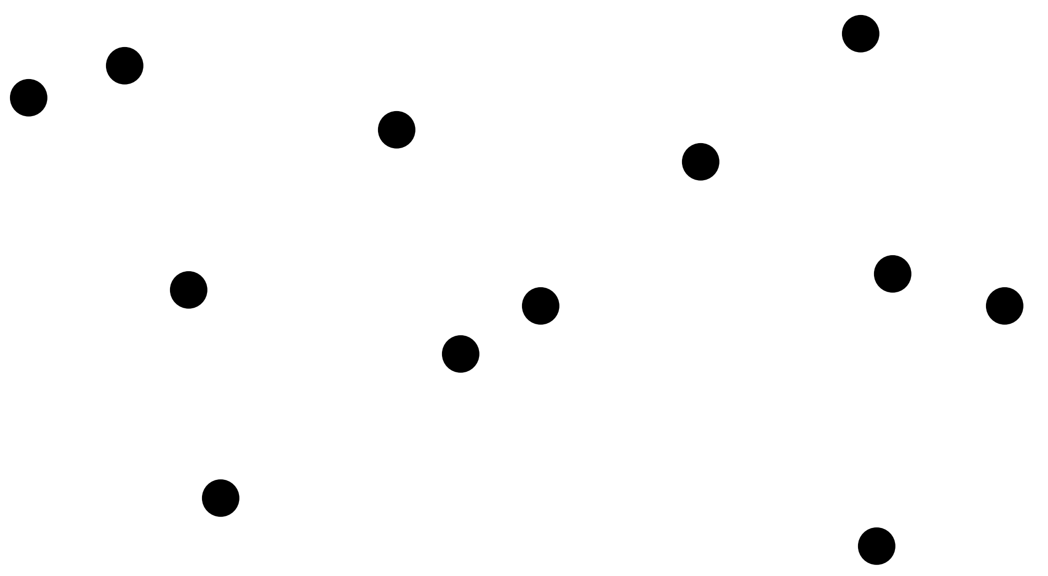
Problem: Given n points in the plane $(x_1,y_1)\ldots(x_n,y_n)$ find the pair (x_i,y_i) and (x_j,y_j) whose Euclidean distance is as small as possible.

Closest Pair of Points (Ex 2.32)

Problem: Given n points in the plane $(x_1,y_1)\ldots(x_n,y_n)$ find the pair (x_i,y_i) and (x_j,y_j) whose Euclidean distance is as small as possible.

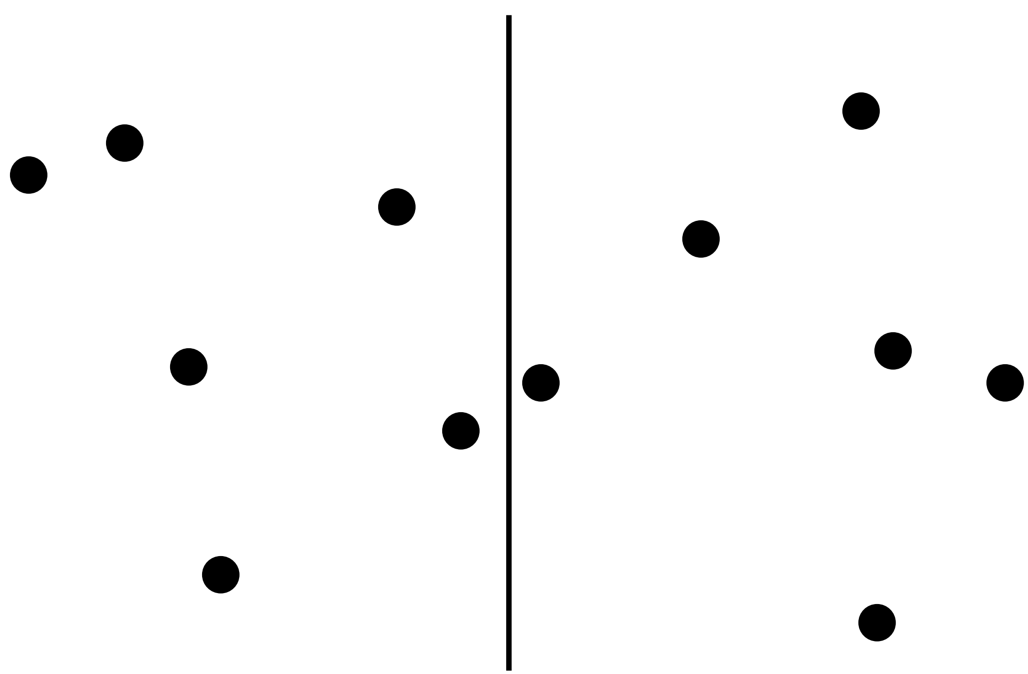
Naïve Algorithm: Try every pair of points. $O(n^2)$ time.

Divide and Conquer Outline



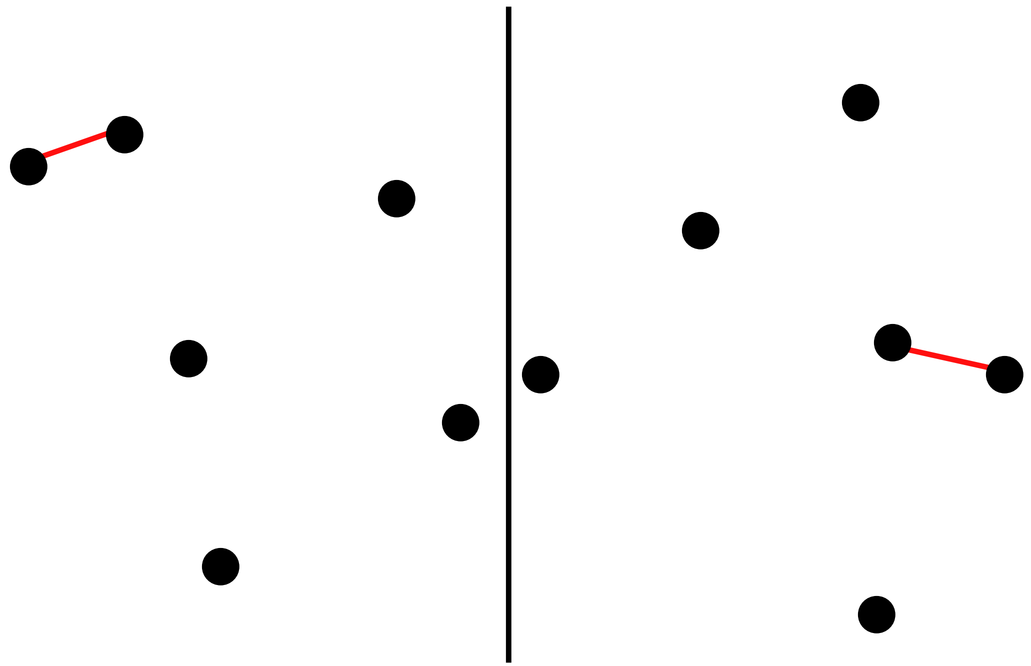
Divide and Conquer Outline

- Divide points into two sets by drawing a line.



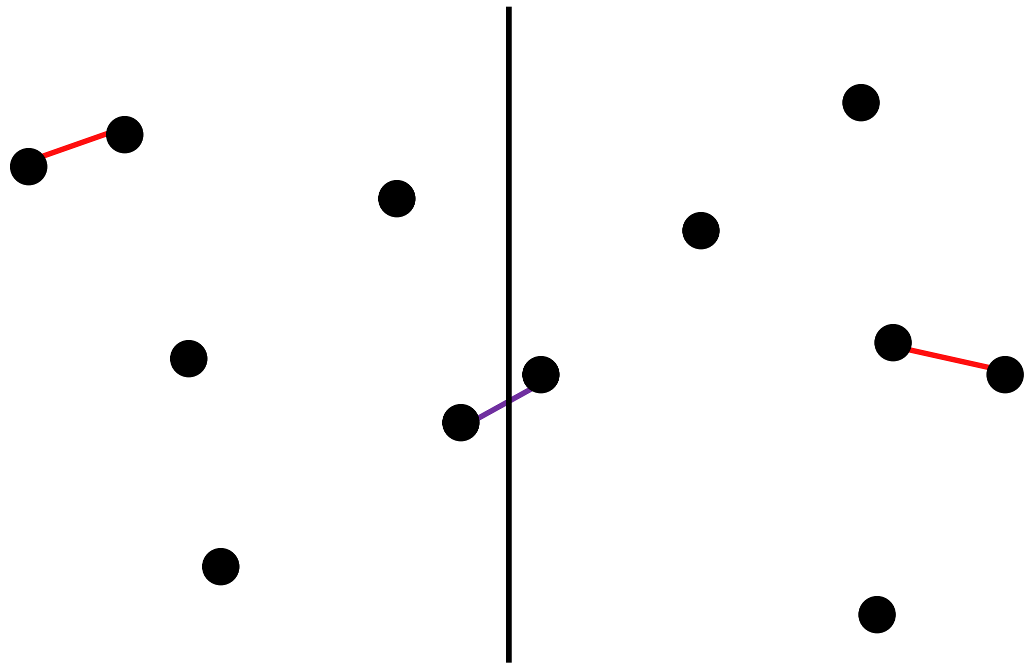
Divide and Conquer Outline

- Divide points into two sets by drawing a line.
- Compute closest pair on each side.



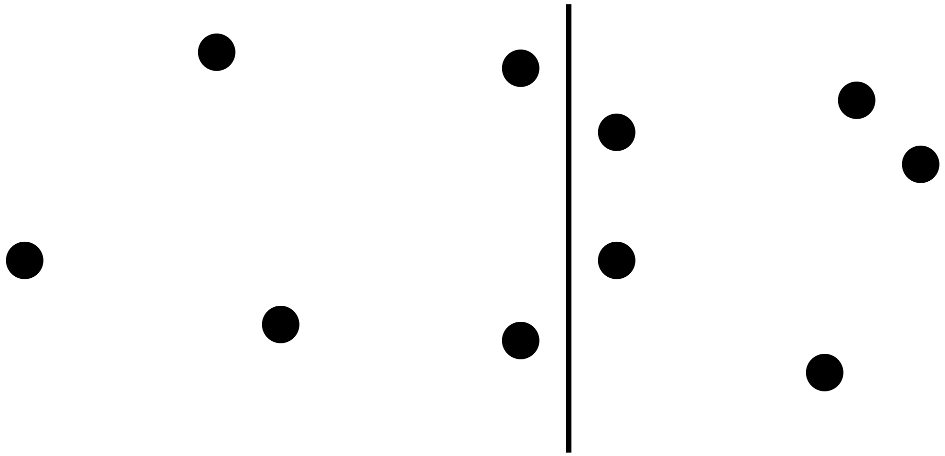
Divide and Conquer Outline

- Divide points into two sets by drawing a line.
- Compute closest pair on each side.
- What about pairs that cross the divide?



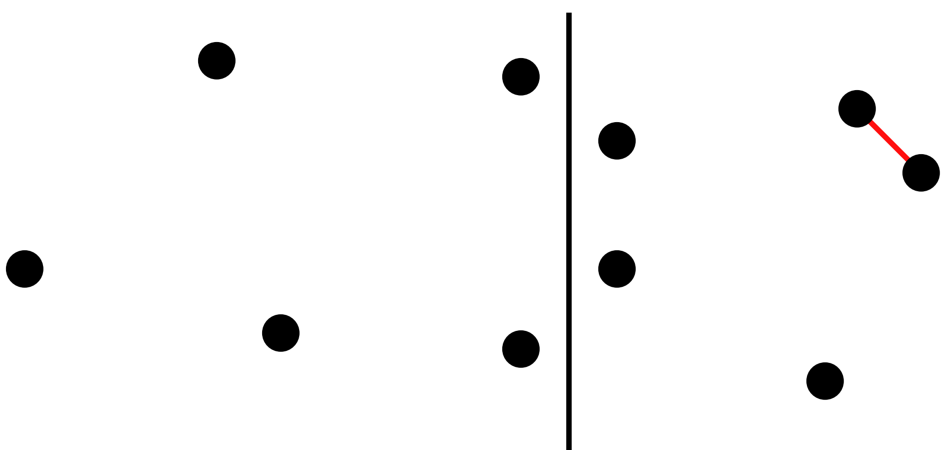
Observation

- Suppose closest pair on either side at distance δ.



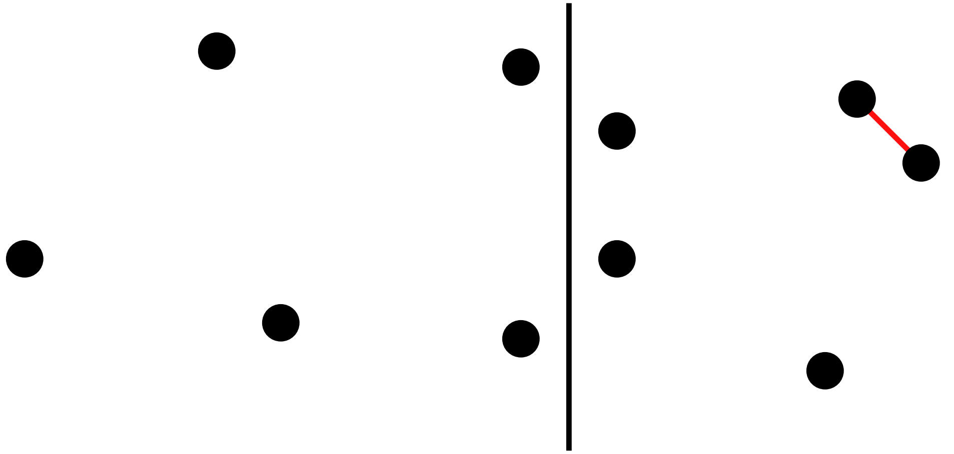
Observation

- Suppose closest pair on either side at distance δ.



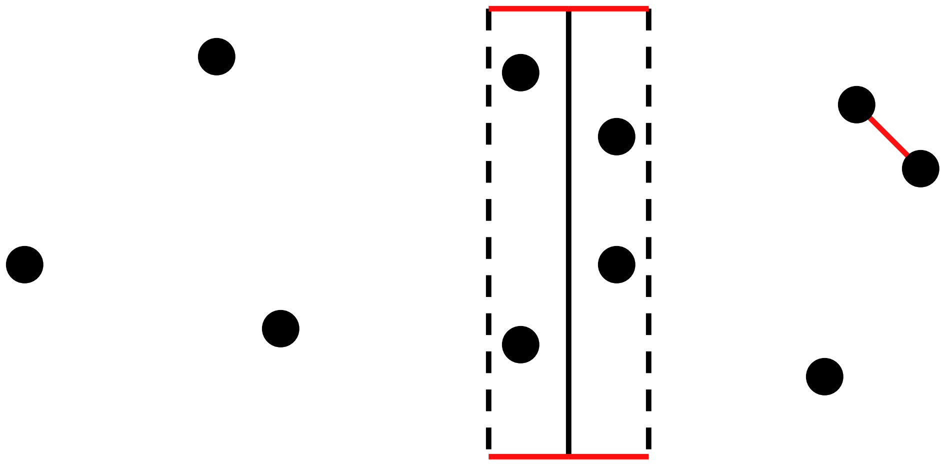
Observation

- Suppose closest pair on either side at distance δ.
- Only need to care about points within δ of dividing line.



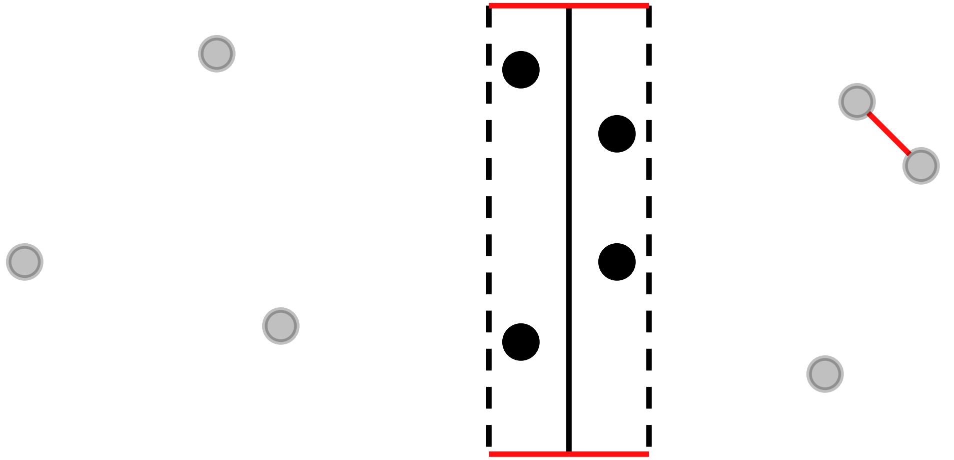
Observation

- Suppose closest pair on either side at distance δ.
- Only need to care about points within δ of dividing line.



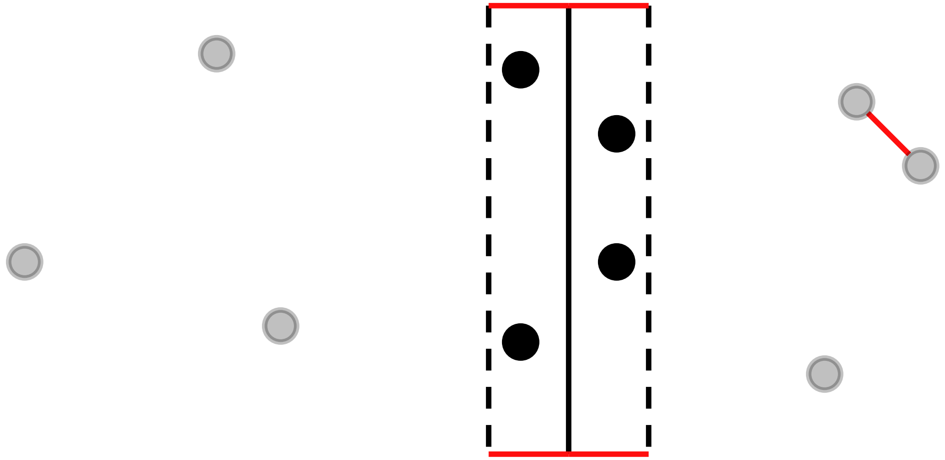
Observation

- Suppose closest pair on either side at distance δ.
- Only need to care about points within δ of dividing line.



Observation

- Suppose closest pair on either side at distance δ.
- Only need to care about points within δ of dividing line.
- Need to know if some pair closer than δ.



Main Idea

Proposition: Take the points within δ of the dividing line and sort them by y-coordinate. Any one of these points can only be within δ of the 8 closest points on either side of it.

Main Idea

<u>Proposition:</u> Take the points within δ of the dividing line and sort them by y-coordinate. Any one of these points can only be within δ of the 8 closest points on either side of it.

This means that we only need to check a few pairs of points crossing the line.

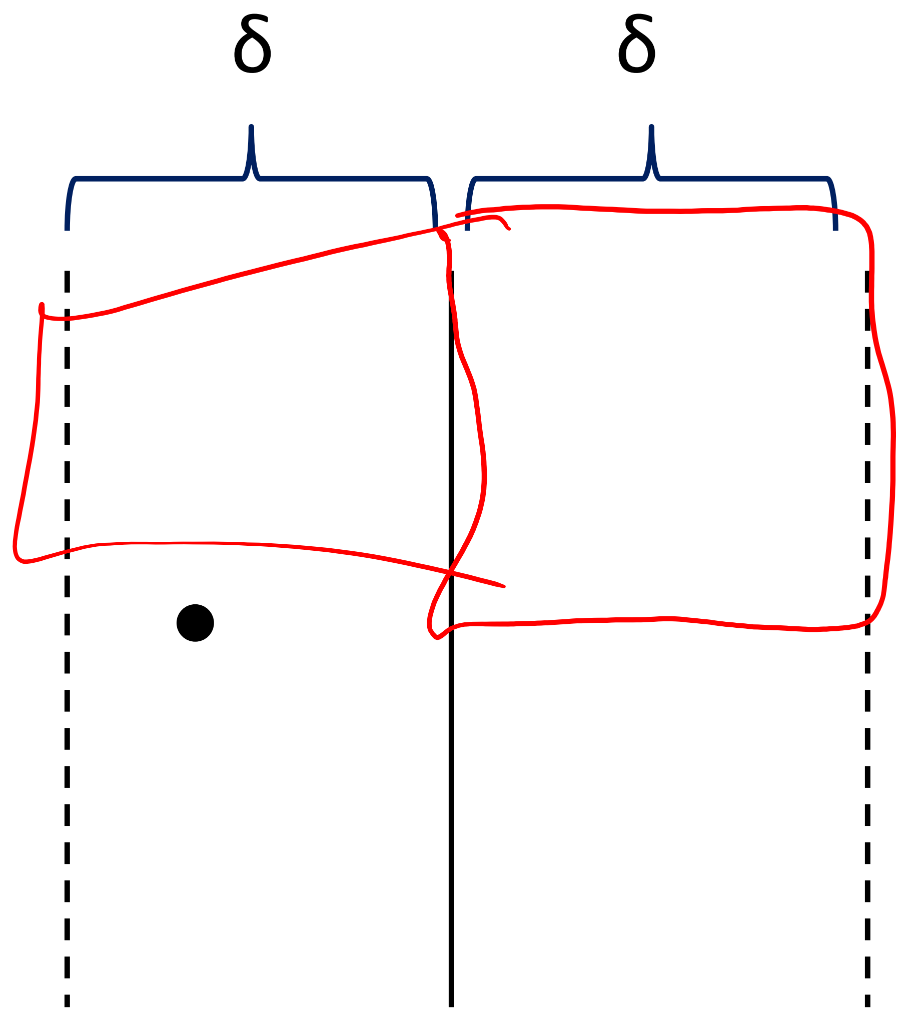
Main Idea

<u>Proposition:</u> Take the points within δ of the dividing line and sort them by y-coordinate. Any one of these points can only be within δ of the 8 closest points on either side of it.

This means that we only need to check a few pairs of points crossing the line.

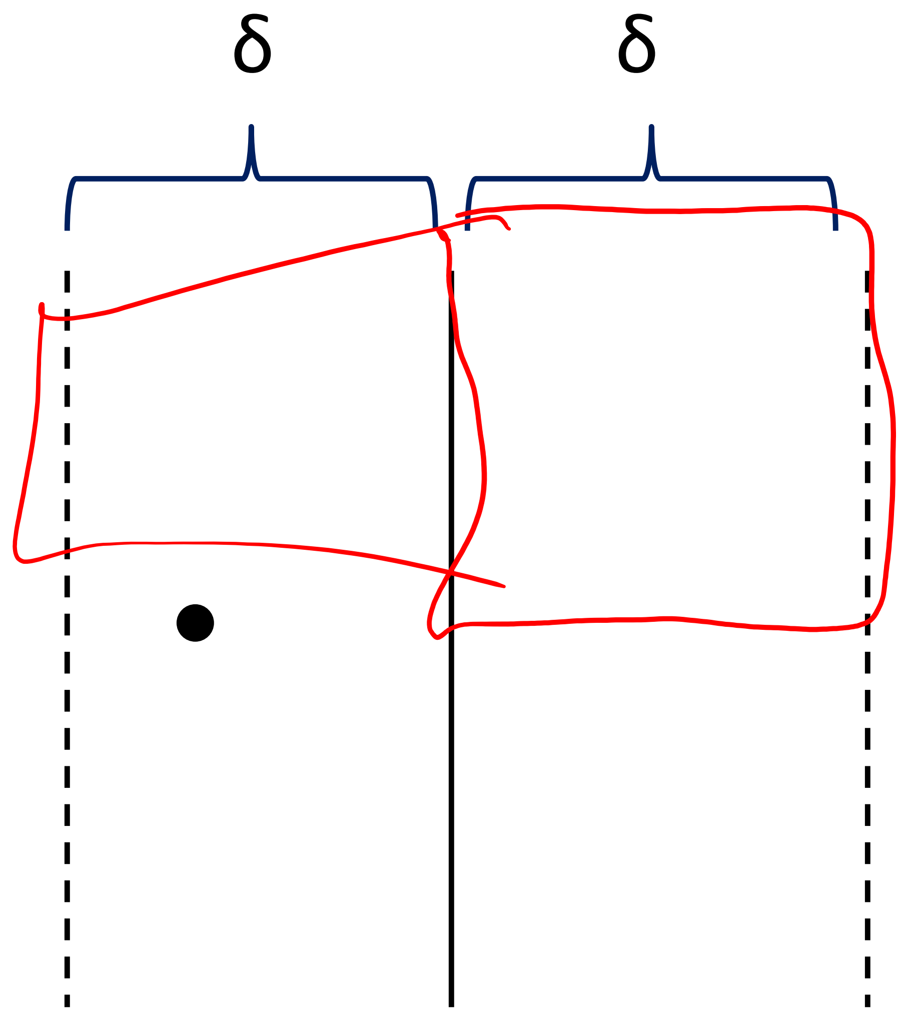
<u>Idea:</u> Points on each side separated by δ. Not enough room for many of them nearby.

Proof



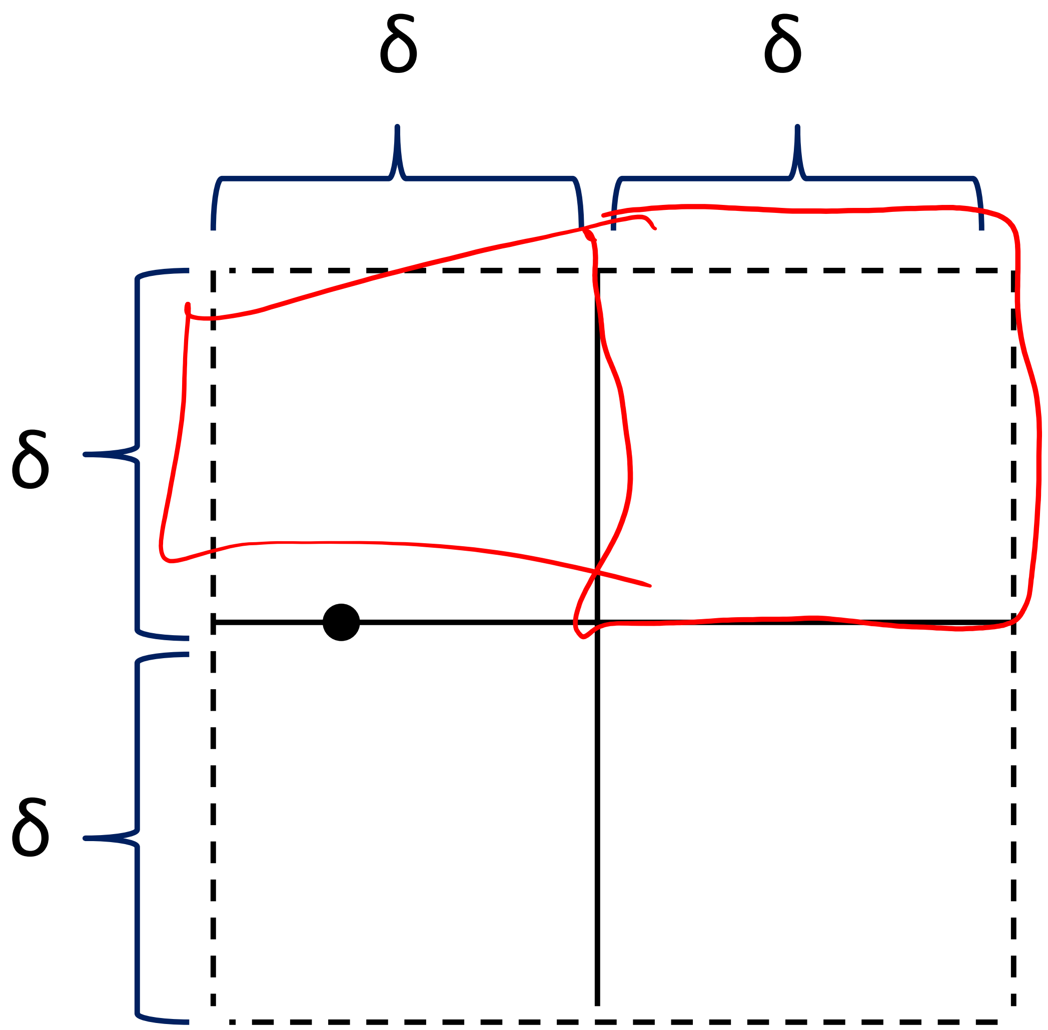
Proof

- Nearby points must have y-coordinate within δ.



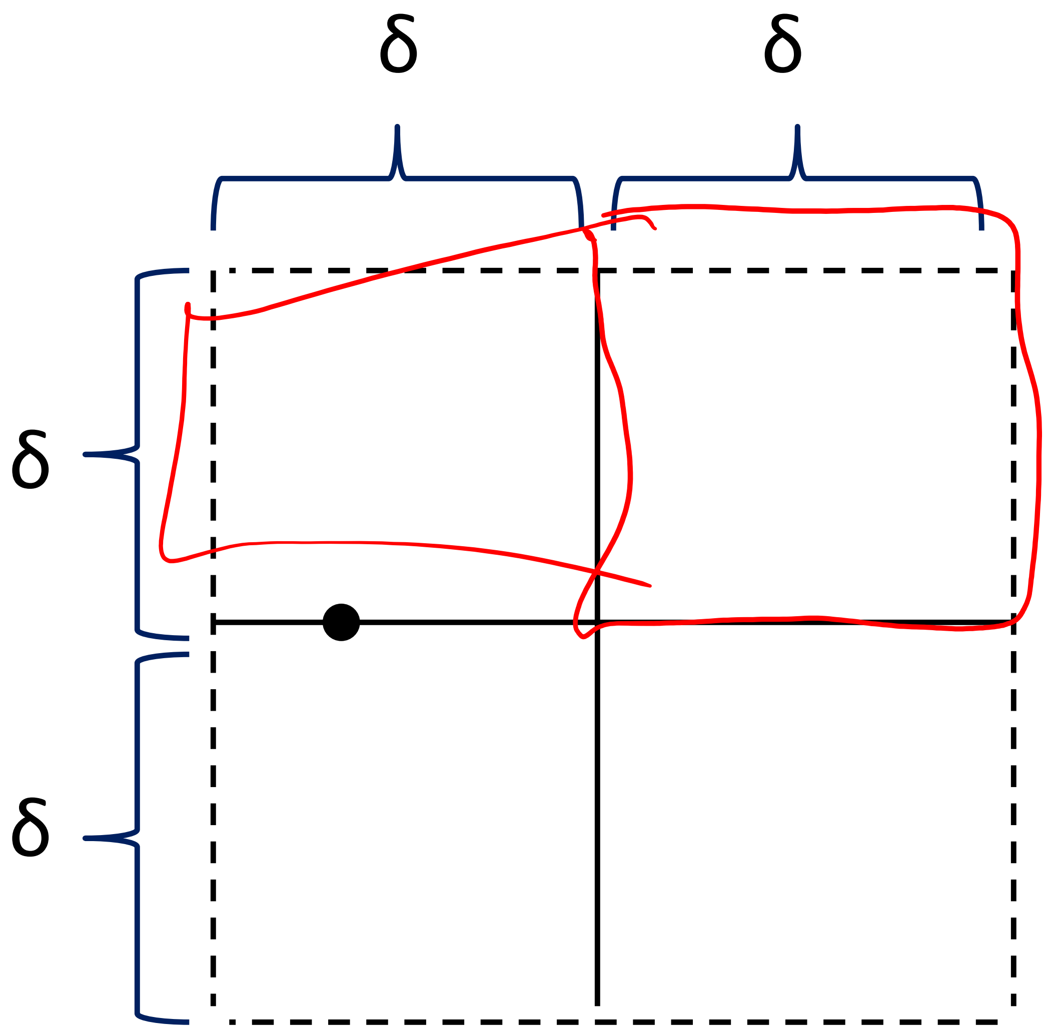
Proof

- Nearby points must have y-coordinate within δ.



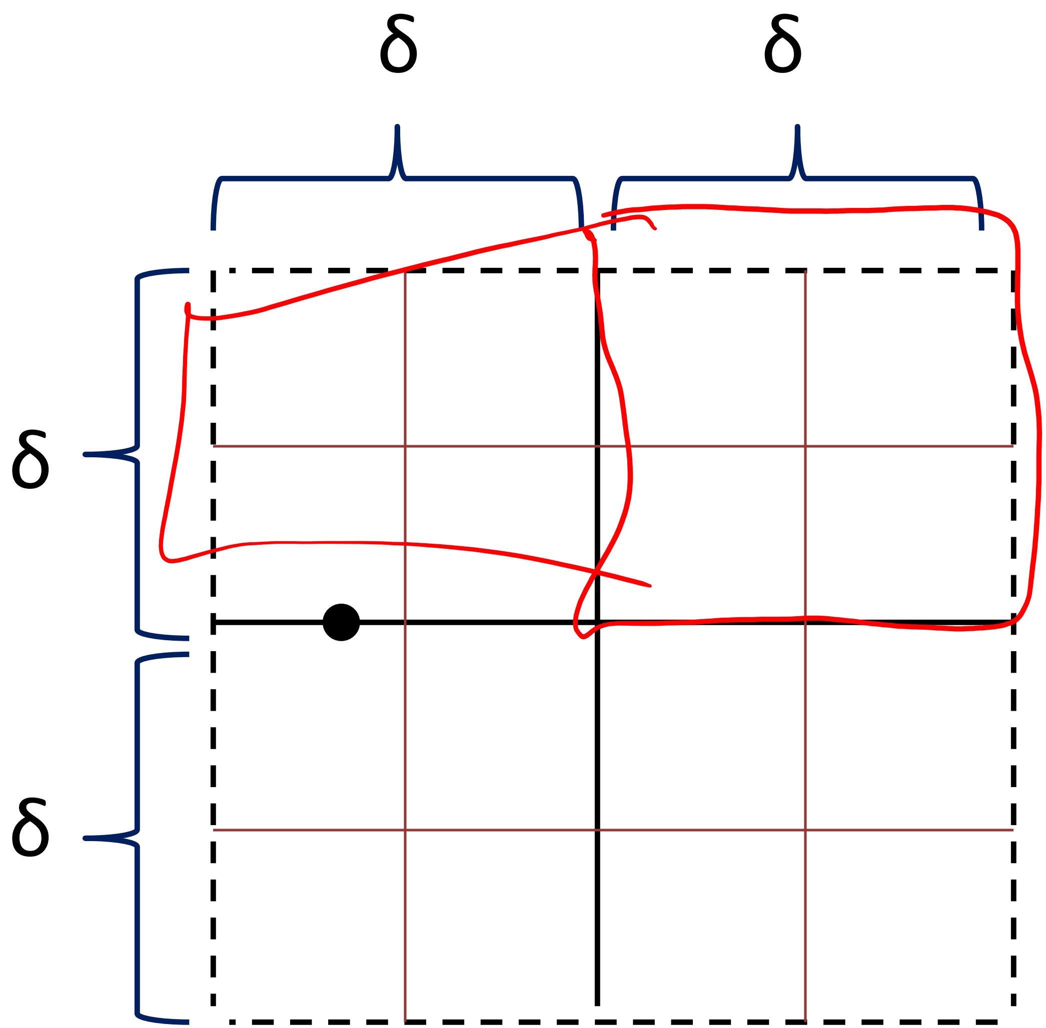
Proof

- Nearby points must have y-coordinate within δ.
- Subdivide region into δ/2-sided squares.



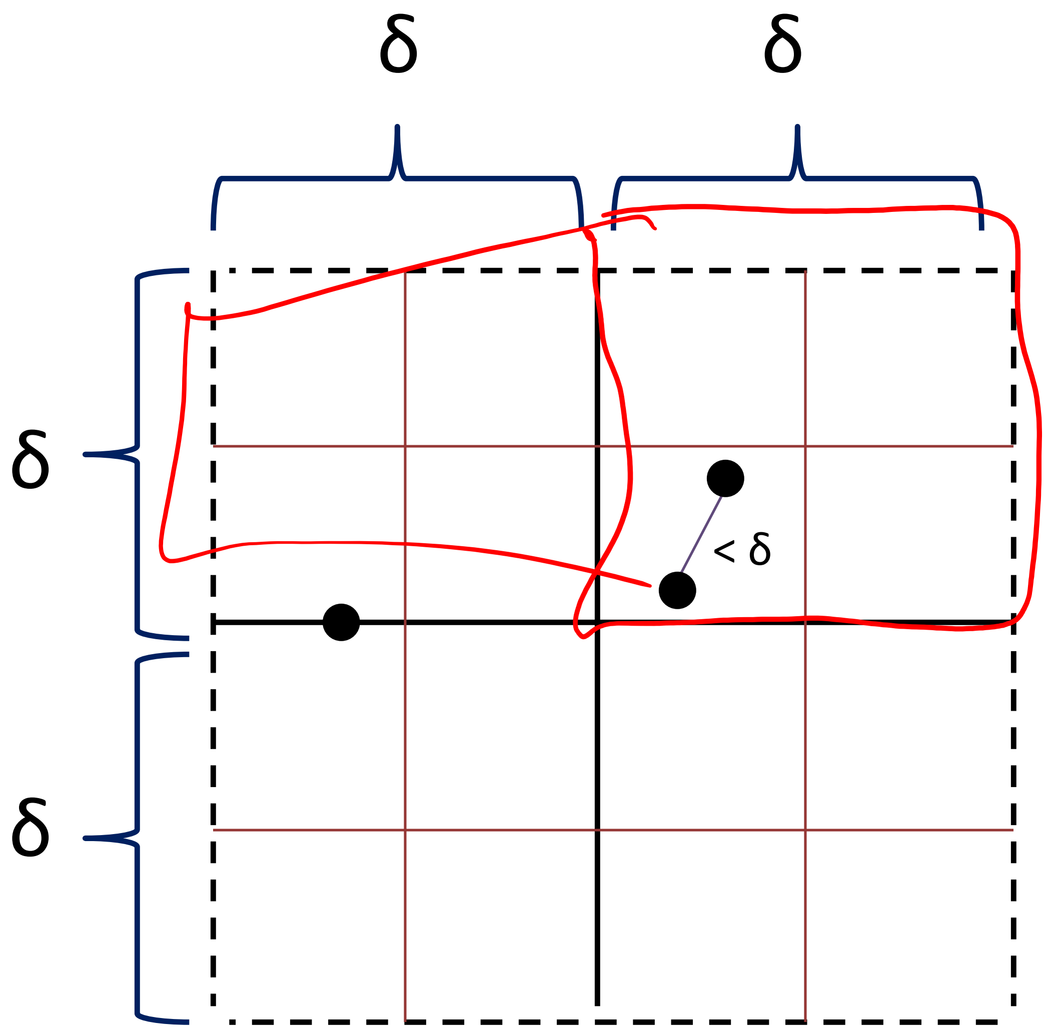
Proof

- Nearby points must have y-coordinate within δ.
- Subdivide region into δ/2-sided squares.



Proof

- Nearby points must have y-coordinate within δ.
- Subdivide region into δ/2-sided squares.
- At most one point in each square.



Algorithm

```
CPP(S)
  If |S| ≤ 3
    Return closest distance
  Find line L evenly dividing points
  Sort S into S_left, S_right
  δ ← min(CPP(S_left),CPP(S_right))
  Let T be points within δ of L
  Sort T by y-coordinate
  Compare each element of T to 8 closest
    on either side. Let min dist be δ'.
  Return min(δ,δ')
```

Algorithm

```
CPP(S)
  If |S| ≤ 3
    Return closest distance
  Find line L evenly dividing points
  Sort S into S_left, S_right
  δ ← min(CPP(S_left),CPP(S_right))
  Let T be points within δ of L
  Sort T by y-coordinate
  Compare each element of T to 8 closest
    on either side. Let min dist be δ'.
  Return min(δ,δ')
```

O(n) (median of x-coordinates)

Algorithm

```
CPP(S)
  If |S| ≤ 3
    Return closest distance
  Find line L evenly dividing points
  Sort S into S_left, S_right
  δ ← min(CPP(S_left),CPP(S_right))
  Let T be points within δ of L
  Sort T by y-coordinate
  Compare each element of T to 8 closest
    on either side. Let min dist be δ'.
  Return min(δ,δ')
```

O(n) (median of x-coordinates)

O(n)

Algorithm

```
CPP(S)
  If |S| ≤ 3
    Return closest distance
  Find line L evenly dividing points
  Sort S into S_left, S_right
  δ ← min(CPP(S_left),CPP(S_right))
  Let T be points within δ of L
  Sort T by y-coordinate
  Compare each element of T to 8 closest
    on either side. Let min dist be δ'.
  Return min(δ,δ')
```

O(n) (median of x-coordinates)

O(n)

2T(n/2)

Algorithm

CPP(S)

  If $|S| \leq 3$

    Return closest distance

  Find line L evenly dividing points O(n) (median of x-coordinates)

  Sort S into S_{left}, S_{right} O(n)

  $\delta \leftarrow \min(\mathrm{CPP}(S_{left}), \mathrm{CPP}(S_{right}))$ 2T(n/2)

  Let T be points within δ of L

  Sort T by y-coordinate O(n log(n))

  Compare each element of T to 8 closest

    on either side. Let min dist be δ'.

  Return $\min(\delta, \delta')$

Algorithm

```
CPP(S)
  If |S| ≤ 3
    Return closest distance
  Find line L evenly dividing points
  Sort S into S_left, S_right
  δ ← min(CPP(S_left),CPP(S_right))
  Let T be points within δ of L
  Sort T by y-coordinate
  Compare each element of T to 8 closest
    on either side. Let min dist be δ'.
  Return min(δ,δ')
```

O(n) (median of x-coordinates)

O(n)

2T(n/2)

O(n log(n))

O(n)

Runtime

We have a recurrence

$$T(n) = O(n \log(n)) + 2\, T(n/2).$$

Runtime

We have a recurrence

$$T(n) = O(n \log(n)) + 2\,T(n/2).$$

This is not quite covered by the Master Theorem, but can be shown to give

$$T(n) = O(n \log^2(n)).$$

Runtime

We have a recurrence

$$T(n) = O(n \cancel{\log(n)}) + 2\ T(n/2).$$

This is not quite covered by the Master Theorem, but can be shown to give

$$T(n) = O(n \log^{\cancel{2}}(n)).$$

Alternatively, if you are more careful and have `CPP` take points already sorted by y-coordinate, you can reduce to O(n log(n)).

Fast Fourier Transform

- We will *not* cover this in class.

Fast Fourier Transform

- We will *not* cover this in class.
- It is a great example of a divide and conquer algorithm.

Fast Fourier Transform

- We will *not* cover this in class.
- It is a great example of a divide and conquer algorithm.
- Very important algorithm. Allows you to:
  - Multiply n-bit numbers in $O(n \log^2(n))$ time.
  - Decompose a signal into frequencies.
  - Remove noise from signals.

Fast Fourier Transform

- We will *not* cover this in class.
- It is a great example of a divide and conquer algorithm.
- Very important algorithm. Allows you to:
  - Multiply n-bit numbers in $O(n \log^2(n))$ time.
  - Decompose a signal into frequencies.
  - Remove noise from signals.
- It is *highly* recommended that you look it up in the book if you:
  - Have familiarity with complex numbers
  - Already know what a Fourier transform is
  - Are motivated to put in some extra time

Greedy Algorithms (Ch 5)

- Basics
- Change making
- Interval scheduling
- Exchange arguments
- Optimal caching
- Huffman codes
- Minimal spanning trees

Greedy Algorithms

Often when trying to find the optimal solution to some problem you need to consider all your possible choices and how they might interact with other choices down the line.

Greedy Algorithms

Often when trying to find the optimal solution to some problem you need to consider all your possible choices and how they might interact with other choices down the line.

But sometimes you don’t. Sometimes you can just take what looks like the best option for now and repeat.

Greedy Algorithms

General Algorithmic Technique:

1. Find decision criterion
2. Make best choice according to criterion
3. Repeat until done

Greedy Algorithms

General Algorithmic Technique:

1. Find decision criterion
2. Make best choice according to criterion
3. Repeat until done

Surprisingly, this sometimes works.

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Example: Making Change

Problem: How do you make exact change for \$12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
	$12.83

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
$10	$2.83

Example: Making Change

<u>Problem:</u> How do you make exact change for \$12.83 using the fewest number of bills/coins?

<u>Idea:</u> Want to use biggest denominations possible.

Bills used:	Amount Left:
\$10 \$1	\$1.83

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
$10 $1 $1	$0.83

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
\$10 \$1 \$1 ¢25	\$0.58

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
$10 $1 $1 ¢25 ¢25	$0.33

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
$10 $1 $1 ¢25 ¢25 ¢25	$0.08

Example: Making Change

<u>Problem:</u> How do you make exact change for $12.83 using the fewest number of bills/coins?

<u>Idea:</u> Want to use biggest denominations possible.

Bills used:	Amount Left:
$10 $1 $1 ¢25 ¢25 ¢25 ¢5	$0.03

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:

$10 $1 $1 ¢25 ¢25
¢25 ¢5 ¢1

Amount Left:

$0.02

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:

$10 $1 $1 ¢25 ¢25 ¢25 ¢5 ¢1 ¢1

Amount Left:

$0.01

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:	Amount Left:
$10 $1 $1 ¢25 ¢25 ¢25 ¢5 ¢1 ¢1 ¢1	$0.00

Example: Making Change

Problem: How do you make exact change for $12.83 using the fewest number of bills/coins?

Idea: Want to use biggest denominations possible.

Bills used:

$10 $1 $1 ¢25 ¢25 ¢25 ¢5 ¢1 ¢1 ¢1

10 bills/coins

Amount Left:

$0.00

This is the best possible!

Change Making

Note: This technique *always* works for making change when using American currency.

Change Making

Note: This technique *always* works for making change when using American currency.

Note: It does not always work when using other currencies.

Change Making

Note: This technique *always* works for making change when using American currency.

Note: It does not always work when using other currencies.

Example: Weirdtopia has \$1, \$5, \$7 bills.

Change Making

<u>Note:</u> This technique *always* works for making change when using American currency.

<u>Note:</u> It does not always work when using other currencies.

<u>Example:</u> Weirdtopia has \$1, \$5, \$7 bills.

<u>Problem:</u> Make change for \$10 in Weirdtopia.

Change Making

<u>Note:</u> This technique *always* works for making change when using American currency.

<u>Note:</u> It does not always work when using other currencies.

<u>Example:</u> Weirdtopia has \$1, \$5, \$7 bills.

<u>Problem:</u> Make change for \$10 in Weirdtopia.

<u>Greedy:</u>	<u>Optimal:</u>
\$7+\$1+\$1+\$1 = \$10	\$5+\$5 = \$10

Things to Keep in Mind about Greedy Algorithms

- Algorithms are very natural and easy to write down.
- However, not all greedy algorithms work.
- Proving correctness is important.

Interval Scheduling

Imagine that you are trying to schedule *as many* classes as possible without any conflicting lectures.

Interval Scheduling

Imagine that you are trying to schedule *as many* classes as possible without any conflicting lectures.

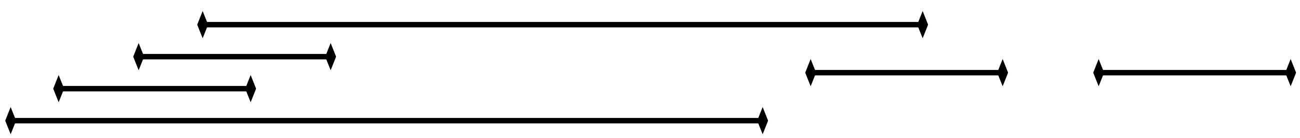
Problem: Given a collection C of intervals, find a subset $S \subseteq C$ so that:

1. No two intervals in S overlap.
2. Subject to (1), $|S|$ is as large as possible.

Question: Interval Scheduling

What is the greatest number of non-overlapping intervals that can be picked from the below?

A) 1
B) 2
C) 3
D) 4
E) 5



Question: Interval Scheduling

What is the greatest number of non-overlapping intervals that can be picked from the below?

A) 1
B) 2
C) 3
D) 4
E) 5

First Interval

- Note: S must consist of intervals $J_1 = [x_1,y_1]$, $J_2 = [x_2,y_2]$, …
  where $y_i < x_{i+1}$.

First Interval

- Note: S must consist of intervals $J_1 = [x_1,y_1]$, $J_2 = [x_2,y_2]$, … where $y_i < x_{i+1}$.
- What is the best J_1?
  - Only way J_1 matters is that $y_1 < x_2$.
  - Want y_1 as small as possible.

First Interval

- Note: S must consist of intervals
  $J_1 = [x_1,y_1]$, $J_2 = [x_2,y_2]$, …
  where $y_i < x_{i+1}$.
- What is the best J_1?
  - Only way J_1 matters is that $y_1 < x_2$.
  - Want y_1 as small as possible.
- Once, we've picked J_1, have another interval cover problem among the intervals that don't overlap J_1.

First Interval

- Note: S must consist of intervals $J_1 = [x_1,y_1]$, $J_2 = [x_2,y_2]$, … where $y_i < x_{i+1}$.
- What is the best J_1?
  - Only way J_1 matters is that $y_1 < x_2$.
  - Want y_1 as small as possible.
- Once, we've picked J_1, have another interval cover problem among the intervals that don't overlap J_1.
- Algorithm: repeatedly pick non-overlapping interval with smallest max.

Algorithm

```
IntervalScheduling(C)
  S ← {}
  While(some interval in C
       doesn't overlap any in S)
    Let J be the non-overlapping
      interval with smallest max
    Add J to S
  Return S
```

Algorithm

```
IntervalScheduling(C)
  S ← {}
  While(some interval in C
        doesn't overlap any in S)
    Let J be the non-overlapping
      interval with smallest max
    Add J to S
  Return S
```

O(n)
iterations

Algorithm

```
IntervalScheduling(C)
  S ← {}
  While(some interval in C
        doesn't overlap any in S)
    Let J be the non-overlapping
      interval with smallest max
    Add J to S
  Return S
```

O(n) iterations

O(n) time

Algorithm

```
IntervalScheduling(C)
  S ← {}
  While(some interval in C
        doesn't overlap any in S)
    Let J be the non-overlapping
      interval with smallest max
    Add J to S
  Return S
```

O(n) iterations

O(n) time

Runtime: $O(n^2)$

Proof of Correctness

- Algorithm produces J_1, J_2,…,J_s with $J_i = [x_i,y_i]$.
- Consider some other solution
  K_1, K_2,…,K_t with $K_i = [w_i,z_i]$.

Proof of Correctness

- Algorithm produces J_1, J_2,…,J_s with $J_i = [x_i,y_i]$.
- Consider some other solution
  K_1, K_2,…,K_t with $K_i = [w_i,z_i]$.

Claim: For each $m \leq t$, $y_m \leq z_m$.

In particular, $s \geq t$.

Proof of Claim

Use Induction on m.

Proof of Claim

Use Induction on m.

<u>Base Case:</u> m = 1.

J_1 is the interval with smallest max, so $y_1 \leq z_1$.

Proof of Claim

Use Induction on m.

Base Case: $m = 1$.

J_1 is the interval with smallest max, so $y_1 \leq z_1$.

Inductive Step: Assume $y_m \leq z_m$.

Proof of Claim

Use Induction on m.

<u>**Base Case:**</u> $m = 1$.

J_1 is the interval with smallest max, so $y_1 \leq z_1$.

<u>**Inductive Step:**</u> Assume $y_m \leq z_m$.

- J_{m+1} has smallest y for any $[x,y]$ with $x > y_m$.

Proof of Claim

Use Induction on m.

Base Case: $m = 1$.

J_1 is the interval with smallest max, so $y_1 \leq z_1$.

Inductive Step: Assume $y_m \leq z_m$.

- J_{m+1} has smallest y for any [x,y] with $x > y_m$.
- $K_{m+1} = [w_{m+1}, z_{m+1}]$ has

$$w_{m+1} > z_m \geq y_m$$

Proof of Claim

Use Induction on m.

Base Case: $m = 1$.

J_1 is the interval with smallest max, so $y_1 \le z_1$.

Inductive Step: Assume $y_m \le z_m$.

- J_{m+1} has smallest y for any $[x,y]$ with $x > y_m$.
- $K_{m+1} = [w_{m+1}, z_{m+1}]$ has

$$w_{m+1} > z_m \ge y_m$$

- Therefore, $y_{m+1} \le z_{m+1}$.

Optimization

- Original algorithm checks all intervals every time.
- Only need to consider intervals in increasing order of y.
- Sort once.

Optimized Algorithm

```
IntervalScheduling(C)
  Sort C by y-value
  S ← {}
  y_max ← -∞
  For J = [x,y] in C
    If x > y_max
      Add J to S
      y_max ← y
  Return S
```

Optimized Algorithm

```
IntervalScheduling(C)
  Sort C by y-value
  S ← {}
  y_max ← -∞
  For J = [x,y] in C
    If x > y_max
      Add J to S
      y_max ← y
  Return S
```

O(n log(n))

Optimized Algorithm

```
IntervalScheduling(C)
  Sort C by y-value
  S ← {}
  ymax ← -∞
  For J = [x,y] in C
    If x > ymax
      Add J to S
      ymax ← y
  Return S
```

O(n log(n))

O(n)

Optimized Algorithm

```
IntervalScheduling(C)
  Sort C by y-value
  S ← {}
  y_max ← -∞
  For J = [x,y] in C
    If x > y_max
      Add J to S
      y_max ← y
  Return S
```

O(n log(n))

O(n)

Runtime:
O(n log(n))

Question: Other Greedy Candidates

What are other possible candidate greedy decision procedures for interval scheduling?

Question: Other Greedy Candidates

What are other possible candidate greedy decision procedures for interval scheduling?

- Smallest max
- Shortest
- Fewest overlaps
- Largest min
- Smallest min
- Largest max

Question: Other Greedy Candidates

What are other possible candidate greedy decision procedures for interval scheduling?

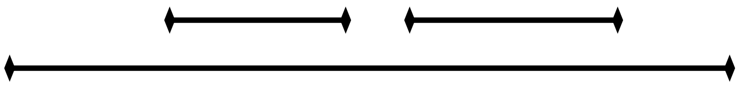
- Smallest max
- Shortest
- Fewest overlaps
- Largest min
- Smallest min
- Largest max

Only these work!

Smallest Min

Greedy

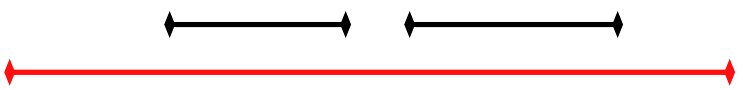
Optimal



Smallest Min

Greedy

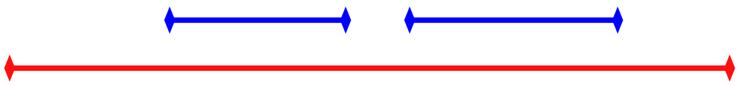
Optimal



Smallest Min

Greedy

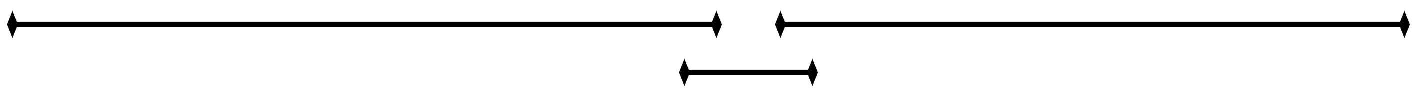
Optimal



Shortest

Greedy

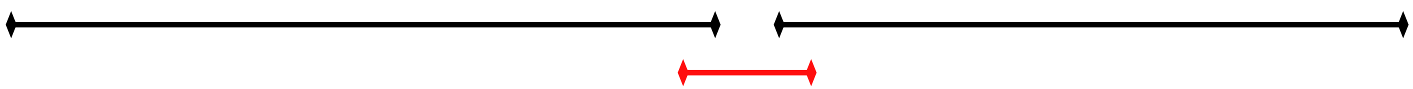
Optimal



Shortest

Greedy

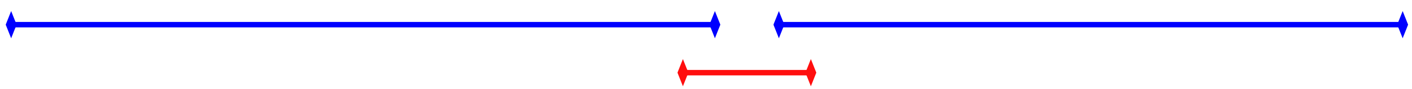
Optimal



Shortest

Greedy

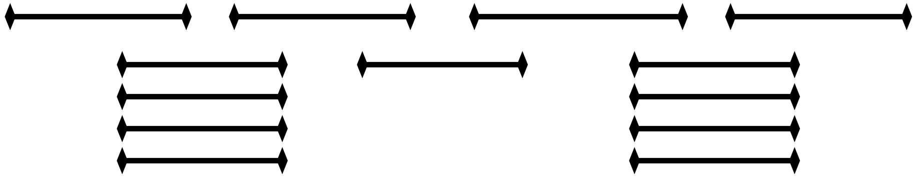
Optimal



Fewest Overlaps

Greedy

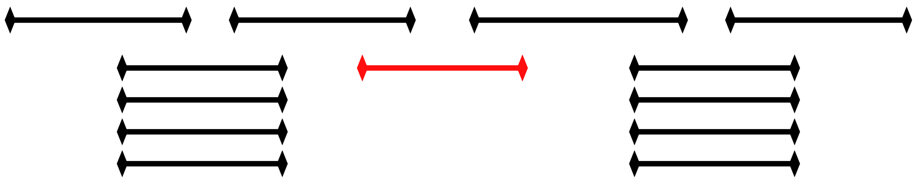
Optimal



Fewest Overlaps

Greedy

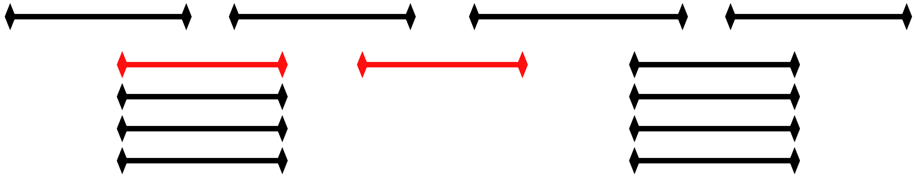
Optimal



Fewest Overlaps

Greedy

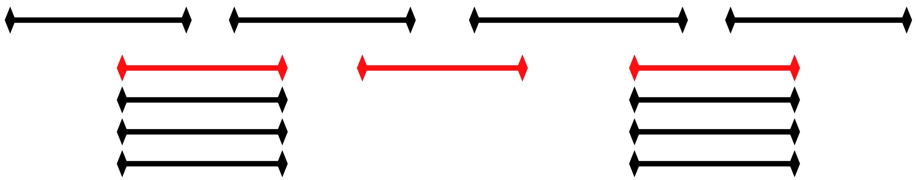
Optimal



Fewest Overlaps

Greedy

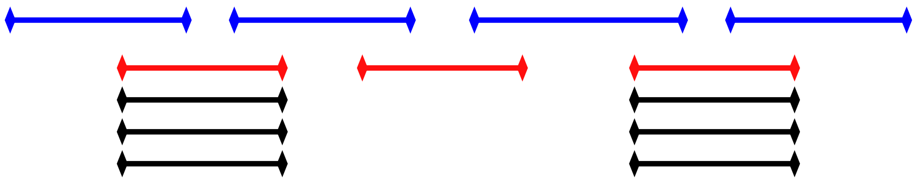
Optimal



Fewest Overlaps

Greedy

Optimal



Proofs

As it is very easy to write down plausible greedy algorithms for problems, but more difficult to find correct ones, it is very important to be able to *prove* that your algorithm is correct.

Proofs

As it is very easy to write down plausible greedy algorithms for problems, but more difficult to find correct ones, it is very important to be able to *prove* that your algorithm is correct.

Fortunately, there is a standard proof technique for greedy algorithms.