CS 341: ALGORITHMS

Lecture 6: divide & conquer III

Readings: see website

Trevor Brown (co-taught with Anna Lubiw)

https://www.student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
THE SELECTION PROBLEM

NATURAL SELECTION in progress...
THE SELECTION PROBLEM

• Input: An array $A$ containing $n$ distinct integer values, and an integer $k$ between 1 and $n$
• Output: The $k$-th smallest integer in $A$
• Minimum is a special case where $k = 1$
• Median is a special case where $k = \frac{n}{2}$
• Maximum is a special case where $k = n$
• Simple algorithm for solving selection?
Suppose we choose a **pivot** element \( y \) in the array \( A \), and we **restructure** \( A \) so that all elements less than \( y \) precede \( y \) in \( A \), and all elements greater than \( y \) occur after \( y \) in \( A \). (This is exactly what is done in *Quicksort*, and it takes **linear time**.)

<table>
<thead>
<tr>
<th>Restructure ((A, y))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
</tr>
<tr>
<td>12 4 6 27 23 17 40 9</td>
</tr>
<tr>
<td>12 4 6 27 23 17 40 9</td>
</tr>
</tbody>
</table>

Number of elements on each side depend on the **value** \( y \)...
Recursive calls

\[ A \text{ after Restructure}(A, y) \]

Number of elements in this range = \( i_y \)

- **What's the \( k \)-th smallest element of \( A \)?**
  - If \( k = i_y \) then \( y \)
  - If \( k < i_y \) then the \( k \)th smallest in \( A_L \)
  - If \( k > i_y \) then the \((k - i_y)\)th smallest in \( A_R \)
QuickSelect(k, A[1..n])
    if n = 1 then return A[1]  // base case

    y = A[1]  // pick an arbitrary pivot
    (AL, AR, iy) = Restructure(A, y)

    if k == iy return y
    else if k < iy then return QuickSelect(k, AL)
    else /* k > iy */ return QuickSelect(k - iy, AR)

Restructure(A[1..n], y)
    AL = new array[1..n]  // allocate more than enough
    AR = new array[1..n]  // to avoid need for expansion
    nL = 0, nR = 0

    for i = 1 .. n
        if A[i] < y then AL[nL++] = A[i]
        else A[i] > y then AR[nR++] = A[i]

    return (AL, AR, nL+1)  // nL+1 is the new index of y
OVERLY OPTIMISTIC ANALYSIS 😊

\( A \) after \( \text{Restructure}(A, y) \)

- If \( i_y = \frac{n}{2} \), then we recurse on \( \sim \frac{n}{2} \) elements.
- If we could always recurse on \( \frac{n}{2} \) elements then
  - We would get \( T(n) = T \left( \frac{n}{2} \right) + \Theta(n) \)
  - Which would yield \( a = 1, b = 2, y = 1, x = \log_2 1 = 0 \), \( y > x \) and \( T(n) \in \Theta(n^y) = \Theta(n) \) by the Master theorem.
WORST-CASE ANALYSIS

\[ A \text{ after } Restructure(A, y) \]
\[
\begin{array}{cccccccc}
12 & 4 & 6 & 17 & 9 & 23 & 27 & 40 \\
\end{array}
\]

• If we always get \( i_y = 1 \) and recurse on the right, then
• We get \( T(n) = T(n - 1) + \Theta(n) \)
• By the substitution method this is \( \Theta(n^2) \)

• So, sometimes the pivot is good, sometimes it’s bad…
• **What about the average case?**
**AVERAGE-CASE ANALYSIS**

- Definition: we say a pivot $y$ is **good** if $i_y \in \left(\frac{n}{4}, \frac{3n}{4}\right)$

\[
\begin{array}{cccccccccc}
1 & \ldots & \ldots & \ldots & \ldots & \ldots & y & \ldots & \ldots & \ldots & \ldots & n
\end{array}
\]

\[
\begin{array}{cccccccccc}
\sim \frac{n}{4} \text{ elements} & \sim \frac{n}{2} \text{ elements} & \sim \frac{n}{4} \text{ elements}
\end{array}
\]

- For any **good pivot**, we recurse on at most $\frac{3n}{4}$ elements

- Probability of an arbitrary pivot being **good**? 

Reducing the size of the subproblem by at least $1/4$
• Probability of a good pivot is $\frac{1}{2}$, so

• On average, every two recursive calls, we will encounter a **good pivot**

• Cost of two recursive calls:
  • $O(n)$ for two calls to Restructure (pivoting)
  • $O(1)$ for other steps

• Encountering a good pivot reduces problem size by at least $\frac{n}{4}$

• So, problem size is reduced by $\frac{n}{4}$ after **expected linear work**

Let’s consider the **average-case** recurrence relation:

$$T(n) = T\left(\frac{3n}{4}\right) + \Theta(n).$$

Apply the **Master Theorem** with $a = 1$, $b = 4/3$ and $y = 1$. Here $x = \log_{4/3} 1 = 0 < 1 = y$ so we are in case 3.

This yields $T(n) \in \Theta(n)$ **on average**.
Here is a more rigorous proof of the average-case complexity: We say the algorithm is in phase \( j \) if the current subarray has size \( s \), where

\[
 n \left( \frac{3}{4} \right)^{j+1} < s \leq n \left( \frac{3}{4} \right)^j.
\]

Let \( X_j \) be a random variable that denotes the amount of computation time occurring in phase \( j \). If the pivot is in the middle half of the current subarray, then we transition from phase \( j \) to phase \( j+1 \). This occurs with probability 1/2, so the expected number of recursive calls in phase \( j \) is 2. The computing time for each recursive call is linear in the size of the current subarray, so \( E[X_j] \leq 2cn(3/4)^j \) (where \( E[\cdot] \) denotes the expectation of a random variable). The total time of the algorithm is given by \( X = \sum_{j \geq 0} X_j \). Therefore

\[
 E[X] = \sum_{j \geq 0} E[X_j] \leq 2cn \sum_{j \geq 0} (3/4)^j = 8cn \in O(n).
\]

\[
 \sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}, \text{ for } |r| < 1.
\]
TAKING SELECTION FURTHER

• We just showed:
  • QuickSelect with **average case** runtime in \(O(n)\)

• Next up:
  • Median-of-medians QuickSelect (MOMQuickSelect)
  • **worst case** runtime in \(O(n)\)

Relies on getting a **good pivot** within \(O(1)\) recursive calls **on average**

Must get a **good pivot** within \(O(1)\) recursive calls **always**

The algorithm we will see picks a **good pivot** in **every** recursive call
HIGH LEVEL ALGORITHM

• Similar to QuickSelect
  • Choose a pivot
  • Move smaller elements to the left of the pivot, and larger elements to the right of the pivot
  • Recursively call MOMQuickSelect on one subarray
• Only difference is how we choose the pivot
  • Always want to pick a good pivot
### Example input

A[1...50]:

11, 38, 6, 21, 20, 17, 14, 9, 7, 5, 8, 34, 49, 47, 28, 18, 44, 31, 46, 48, 27, 4, 2, 50, 23, 45, 3, 13, 43, 22, 10, 32, 35, 41, 24, 1, 30, 12, 15, 26, 16, 19, 36, 33, 37, 39, 25, 40, 29, 42

### Group into rows of 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>38</td>
<td>6</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>49</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>44</td>
<td>31</td>
<td>46</td>
<td>48</td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td>2</td>
<td>50</td>
<td>23</td>
</tr>
<tr>
<td>45</td>
<td>3</td>
<td>13</td>
<td>43</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>35</td>
<td>41</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>12</td>
<td>15</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>36</td>
<td>33</td>
<td>37</td>
</tr>
<tr>
<td>39</td>
<td>25</td>
<td>40</td>
<td>29</td>
<td>42</td>
</tr>
</tbody>
</table>

### Find median of each row

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>38</td>
<td>6</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>49</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>44</td>
<td>31</td>
<td>46</td>
<td>48</td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td>2</td>
<td>50</td>
<td>23</td>
</tr>
<tr>
<td>45</td>
<td>3</td>
<td>13</td>
<td>43</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>35</td>
<td>41</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>12</td>
<td>15</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>36</td>
<td>33</td>
<td>37</td>
</tr>
<tr>
<td>39</td>
<td>25</td>
<td>40</td>
<td>29</td>
<td>42</td>
</tr>
</tbody>
</table>

### Build array of medians

20, 9, 34, 44, 23, 22, 32, 15, 33, 39

### Time complexity for this step?

- Group into rows of 5:
  - Time complexity: \( O(n) \)

### Time complexity for this step?

- Find median of each row:
  - Time complexity: \( O(n) \)

### Recursive problem size?

- Recursively find the median of these medians: \( 23 \)
**Recall:** median of each row

<table>
<thead>
<tr>
<th>11</th>
<th>38</th>
<th>6</th>
<th>21</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>49</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>44</td>
<td>31</td>
<td>46</td>
<td>48</td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td>2</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>3</td>
<td>13</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>35</td>
<td>41</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>12</td>
<td>15</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>36</td>
<td>33</td>
<td>37</td>
</tr>
<tr>
<td>39</td>
<td>25</td>
<td>40</td>
<td>29</td>
<td>42</td>
</tr>
</tbody>
</table>

**Imagine** sorting each row:

<table>
<thead>
<tr>
<th>6</th>
<th>11</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>34</td>
</tr>
<tr>
<td>18</td>
<td>31</td>
<td>44</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>25</td>
<td>29</td>
<td>39</td>
</tr>
</tbody>
</table>

**Then** ordering rows by medians:

<table>
<thead>
<tr>
<th>5</th>
<th>7</th>
<th>9</th>
<th>14</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>15</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>20</td>
<td>21</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>22</td>
<td>43</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>23</td>
<td>27</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>32</td>
<td>35</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>33</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>34</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
<td>42</td>
<td>46</td>
<td>48</td>
</tr>
</tbody>
</table>

**# elements ≤ 23** is at least 3(5). This is at least 3/10ths of our 50-element input, or 3n/10.

**# elements ≥ 23** is at least 3(6). This is at least 3/10ths of our 50-element input.

So, after restructuring, pivot 23 must have at least 3n/10 elements before and after it.

This is a good pivot!

We recurse on $A_L$ or $A_R$, and both have size at most 7n/10.
**MOMQuickSelect**($k = 11, n = 14, A$)

1. **base case**
   
   if $n \leq 14$ then sort($A$) and return $A[k]$

2. **divide and conquer to find medians**
   
   $r = (n-5) / 10$
   
   medians[1..(2*r+1)] = new array
   
   for $i = 1..(2*r+1)$
   
   $B[1..5] = A[(5*(i-1)+1)..(5*i)]$
   
   sort($B$)
   
   medians[i] = $B[3]$

3. $y = MOMQuickSelect(r+1, 2*r+1, medians)$

4. **divide and conquer to find rank $k$**
   
   $(AL, AR, iy) = Restructure(A, y)$
   
   if $k == iy$ then return $y$
   
   else if $k < iy$ then return $MOMQuickSelect(k, iy-1, AL)$
   
   else /* $k > iy */ then return $MOMQuickSelect(k-iy, n-iy, AR)$
MOMQuickSelect(k, n, A)

// base case
if n <= 14 then sort(A) and return A[k]

// divide and conquer to find medians
r = (n-5) / 10
medians[1..(2*r+1)] = new array
for i = 1..(2*r+1)
    B[1..5] = A[(5*(i-1)+1)..(5*i)]
sort(B)
    medians[i] = B[3]
y = MOMQuickSelect(r+1, 2*r+1, medians)

// divide and conquer to find rank k
(AL, AR, iy) = Restructure(A, y)
if k == iy then return y
else if k < iy then return MOMQuickSelect(k, iy-1, AL)
else /* k > iy */ then return MOMQuickSelect(k-iy, n-iy, AR)

r = \left\lceil \frac{21 - 5}{10} \right\rceil = 1

Not considering at most 9 elements

B sort(B)

\begin{array}{cccccccc}
5 & 7 & 9 & 14 & 17 & 8 & 34 & 49 & 47 & 28 & 18 & 44 & 31 & 46 & 48 & 27 \\
8 & 28 & 34 & 47 & 49 & 18 & 44 & 31 & 46 & 48 & 27 \\
\end{array}

medians = 20, 9, 34

y = MOMQuickSelect(2, 3, [20, 9, 34]) \Rightarrow 20
MOMQuickSelect(k, n, A)

// base case
if n <= 14 then sort(A) and return A[k]

// divide and conquer to find medians
r = (n-5) / 10
medians[1...(2*r+1)] = new array
for i = 1...(2*r+1)
    B[1..5] = A[(5*(i-1)+1)..(5*i)]
sort(B)
    medians[i] = B[3]
y = MOMQuickSelect(r+1, 2*r+1, medians)

// divide and conquer to find rank k
(AL, AR, iy) = Restructure(A, y)
if k == iy then return y
else if k < iy then return MOMQuickSelect(k, iy-1, AL)
else /* k > iy */ then return MOMQuickSelect(k-iy, n-iy, AR)

Restructure(A, y = 20) ⇒

\[ A_L = [11, 6, 17, 14, 9, 7, 5, 8, 18] \]
\[ A_R = [38, 21, 34, 49, 47, 28, 44, 31, 46, 48, 27] \]

\[ i_y = |A_L| + 1 = 10 \]

\[ k = 11 > i_y = 10 \]
\[ k - i_y = 1 \]
\[ n - i_y = 10 \]

MOMQuickSelect(1, 10, A_R) ⇒ 21
Time complexity?

// base case
if n <= 14 then sort(A) and return A[k]

// divide and conquer to find medians
r = (n-5) / 10
medians[1..(2*r+1)] = new array
for i = 1..(2*r+1)
    B[1..5] = A[(5*(i-1)+1)..(5*i)]
    sort(B)
    medians[i] = B[3]

y = MOMQuickSelect(r+1, 2*r+1, medians)

// divide and conquer to find rank k
(AL, AR, iy) = Restructure(A, y)
if k == iy then return y
else if k < iy then return MOMQuickSelect(k, iy-1, AL)
else /* k > iy */ then return MOMQuickSelect(k-iy, n-iy, AR)

So problem size shrinks by at least 3(r + 1)
Observe n = 10r + 5
HOW MUCH DOES THE PROBLEM SHRINK?

• Shrinks by at least $3(r + 1)$
• Problem size $\sim n = 10r + 5$
• Subproblem size $\leq n - Shrink = n - 3(r + 1)$
  • $= 10r + 5 - 3r - 3 = 7r + 2$
• Express in terms of $n$ using $r = \left\lfloor \frac{n-5}{10} \right\rfloor$
  • Subproblem size $\leq 7 \left\lfloor \frac{n-5}{10} \right\rfloor + 2 \leq 7 \frac{n-5}{10} + 2$
  • $= \frac{7n}{10} - 7 \left( \frac{5}{10} \right) + 2 = \frac{7n}{10} - \frac{3}{2} \leq \frac{7n}{10}$
// base case
if n <= 14 then sort(A) and return A[k]

// divide and conquer to find medians
r = (n-5) / 10
medians[1..(2*r+1)] = new array
for i = 1..(2*r+1)
    B[1..5] = A[(5*(i-1)+1)..<(5*i)]
sort(B)
medians[i] = B[3]
y = MOMQuickSelect(r+1, 2*r+1, medians)

// divide and conquer to find rank k
(AL, AR, iy) = Restructure(A, y)
if k == iy then return y
else if k < iy then return MOMQuickSelect(k, iy-1, AL)
else /* k > iy */ then return MOMQuickSelect(k-iy, n-iy, AR)

T(n) ∈ O(n) + T(n/5) + T(7n/10)  if n ≥ 15
T(n) ∈ O(1)  if n ≤ 14
The key fact is that $1/5 + 7/10 = 19/20 < 1$.

The fact that $T(n) \in \Theta(n)$ can be proven formally using guess-and-check (induction) or informally using the recursion tree method.

\[
T(n) \in O(n) + T(n/5) + T(7n/10) \quad \text{if } n \geq 15 \\
T(n) \in O(1) \quad \text{if } n \leq 14
\]

\[
\sum_{i=0}^{8} n \left( \frac{9}{10} \right)^i = 10n \in \Theta(n)
\]
• Let $T(n) = c'n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$ where $c' > 0$

• Want to prove: $T(n) = cn$ for some $c > 0$

• Note $c$ and $c'$ are independent constants
  • $c'$ comes from the work at each level of recursion being $O(n)$
  • $c$ is a positive constant we are trying to show exists

• I.H.: Suppose $\exists c > 0 : T(n') = cn'$ for $15 \leq n' < n$

• $T(n) = c'n + c \frac{n}{5} + c \frac{7n}{10}$
  (by inductive hypoth.)

• $T(n) = cn$
  (want this to be true)

• $\iff c'n + c \frac{n}{5} + c \frac{7n}{10} = cn$
  (equivalently)

• $\iff c' + c \frac{1}{5} + c \frac{7}{10} = c \iff c = 10c'$
  (by algebra)

Guess & check

$T(n) = cn$
THE CLOSEST PAIR PROBLEM

Hopefully not anti-vaxxer

When someone near you

COUGHS
The Closest Pair Problem

- **Input:** Set P of n 2D points
- **Output:** pair p and q s.t. \( \text{dist}(p, q) \) minimum over all pairs
- Break ties arbitrarily

\[ \text{dist}(p, q) = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2} \]
Can we Divide & Conquer?

- Like non-dominated points: sort by x-axis & divide in half

\[ p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8 \]

- L
- R

Claim that doesn’t require a proof: closest pair \((p, q)\):

1. \((p, q)\) both in L or
2. \((p, q)\) both in R or
3. One of \((p, q)\) in L and one of \((p, q)\) in R

We call this a spanning pair.
How to efficiently compute the minimum spanning pair?

```plaintext
ClosestPair(P[1..n])
    sort(P) by x values
    Recurse(P)

Recurse(P[1..n]) // precondition: P sorted by x
    // base case
    if n < 4 then compare all pairs and return closest

    // divide & conquer
    pairL = Recurse(P[1..(n/2)])
    pairR = Recurse(P[(n/2)+1..n])

    // combine
    pairS = findMinSpanningPair(P)
    return minDistPair(pairL, pairR, pairS)
```
Observation 1

◆ Let $\delta = \min (\text{dist}(\text{pair}_L), \text{dist}(\text{pair}_R))$

◆ Then $\text{pair}_s$ (if closest globally) lies in the above $2\delta$-wide green strip

Q: Why?
Q: Can $p$ be part of a globally closest pair $s$?
A: No. Everything in $R$ has $\text{dist} > \delta$ to $p$. And we already have a solution with $\text{dist} = \delta$. 
Observation 2

◆ Say, p (the lowest y valued point in strip) is in pairs

◆ Then the other point can only lie in this $\delta \times \delta$ square.

Q: Why?

Has to be on the opposite side & can’t be $> \delta$ higher than p on y axis.
1. Start from lowest y valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 for the next lowest y-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 for the next lowest y-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta x \delta$ square points on the opposite side
3. Repeat 1 & 2 for the next lowest y-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta x \delta$ square points on the opposite side
3. Repeat 1 & 2 for the next lowest y-valued point
4. So on and so forth...

\[ \delta \quad \delta \]
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 for the next lowest y-valued point
4. So on and so forth...

Switching sides might complicate code... Turns out it’s not needed to get good time complexity.
A More Practical Idea

- Don’t differentiate between same and opposite side
- Just search the $2\delta \times \delta$ above rectangle each time
A More Practical Idea

◆ Don’t differentiate between same and opposite side
◆ Just search the $2\delta \times \delta$ above rectangle each time
A More Practical Idea

- Don’t differentiate between same and opposite side
- Just search the $2\delta \times \delta$ above rectangle each time
A More Practical Idea

◆ Don’t differentiate between same and opposite side
◆ Just search the $2\delta \times \delta$ above rectangle each time
ClosestPair(P[1..n])
   sort(P) by x values
   Recurse(P)

Recurse(P[1..n]) // precondition: P sorted by x
   // base case
   if n < 4 then compare all pairs and return closest

   // divide & conquer
   pairL = Recurse(P[1..(n/2)])
   pairR = Recurse(P[(n/2)+1..n])

   // combine
   δ = min(dist(pairL), dist(pairR))
   pairS = findMinSpanningPair(P, δ)
   return minDistPair(pairL, pairR, pairS)
Claim: loop performs $O(1)$ iterations!
For a particular $i$, how many $j$ iterations occur?

\begin{verbatim}
for i = 1..len(S)
  for j = (i+1)..<len(S)
    if S[j].y - S[i].y > \delta then break
\end{verbatim}

**Obs:** as many as there are **points** in the $2\delta \times \delta$ rectangle.

**Q:** How many points can be in a $2\delta \times \delta$ rectangle?

**A:** As many as in the left $\delta \times \delta$ square + right $\delta \times \delta$ square.
POINTS IN A $\delta \times \delta$ SQUARE

• Recall $\delta$ is the smallest distance between any pair of points that are both in $L$ or both in $R$

• Note this square is entirely in $L$ or entirely in $R$

So, $\delta$ is the smallest distance between any pair of points in this square!

A point in the middle would rule out any other points

So, most efficient packing of points puts one in each corner (4 total)
For a particular $i$, how many $j$ iterations occur?

```python
for i = 1..len(S)
    for j = (i+1)..len(S)
        if S[j].y - S[i].y > $\delta$ then break
```

**Obs:** as many as there are **points** in the $2\delta \times \delta$ rectangle.

**Q:** How many points can be in a $2\delta \times \delta$ rectangle?

**A:** As many as in the left $\delta \times \delta$ square + right $\delta \times \delta$ square.

---

Can only contain **eight** points!
```
findMinSpanningPair(δ, P[1..n]) // P sorted by x
S = { p in P : abs(P[n/2].x - p.x) <= δ }
sort(S) by increasing y values
minPair = (S[1], S[2]) // arbitrary pair to start
for i = 1..len(S)
    for j = (i+1)..len(S)
        if S[j].y - S[i].y > δ then break
        minPair = minDistPair(minPair, (S[i], S[j]))
return minPair
```

- Loop performs at most eight iterations
- Each does $\Theta(1)$ work, so entire loop does $\Theta(1)$ work!
- So, findMinSpanningPair does $\Theta(n \log n)$ work
Time complexity

1. ClosestPair(P[1..n])
   - sort(P) by x values
   - Recurse(P)
     - $T(n)$

2. Recurse(P[1..n]) // precondition: P sorted by x
   - if n < 4 then compare all pairs and return closest
     - $\Theta(1)$

3. // divide & conquer
   - pairL = Recurse(P[1..(n/2)])
     - $\Theta(n) + T\left(\frac{n}{2}\right)$
   - pairR = Recurse(P[(n/2)+1..n])
     - $\Theta(n) + T\left(\frac{n}{2}\right)$

4. // combine
   - $\delta = \min(\text{dist(pairL)}, \text{dist(pairR)})$
     - $\Theta(1)$
   - pairS = findMinSpanningPair(P, $\delta$)
     - $\Theta(n \log n)$
   - return minDistPair(pairL, pairR, pairS)
     - $\Theta(1)$

- Let $T'(n)$ be runtime of ClosestPair(P[1..n])
- Let $T(n)$ be runtime of Recurse(P[1..n])
- $T'(n) \in \Theta(n \log n) + T(n)$
- $T(n) \in 2T\left(\frac{n}{2}\right) + \Theta(n \log n)$
- In Lec4, we used recursion trees to show
  - $T(n) \in \Theta(n \log^2 n)$
  - $T'(n) \in \Theta(n \log n) + \Theta(n \log^2 n)$
  - So $T'(n) \in \Theta(n \log^2 n)$
IMPROVING THIS RESULT FURTHER
IMPROVING THE PREVIOUS ALGORITHM

• Sorting by \( y \)-values causes findMinSpanningPair to take \( O(n \log n) \) time instead of \( O(n) \) time.

• This happens in each recursive call, and dominates the running time.

• Avoid sorting \( P \) over and over by creating another copy of \( P \) that is pre-sorted by \( y \)-values.
ShamosClosestPair(P[1..n])
  Px = sort(P) by increasing x values
  Py = sort(P) by increasing y values
  Recurse(Px, Py)

Recurse(Px[1..n], Py[1..n])
  // base case
  if n < 4 then return BruteForce(Px)

  // divide & conquer
  xmid = Px[n/2].x
  PxL = Px[1..(n/2)]  // x <= xmid
  PxR = Px[(n/2+1)..n]  // x > xmid
  PyL = select p from Py where p.x <= xmid
  PyR = select p from Py where p.x > xmid
  pairL = Recurse(PxL, PyL)
  pairR = Recurse(PxR, PyR)

  // combine
  δ = min(dist(pairL), dist(pairR))
  pairS = findMinSpanningPair(δ, Py, xmid)
  return minDistPair(pairL, pairR, pairS)

This selection step preserves the y-sort order

Observe PxL and PyL contain the same points
  (specifically the points with x <= xmid)

Moreover PxL is sorted by x while PyL is sorted by y

And similarly for PxR, PyR...

No need to sort in Recurse!
findMinSpanningPair(δ, Py[1..n], xmid) // Py sorted by y
S = { p in Py : abs(xmid - p.x) <= δ }

minPair = (S[1], S[2]) // arbitrary pair to start
for i = 1..len(S)
    for j = (i+1)..len(S)
        if S[j].y - S[i].y > δ then break
    minPair = minDistPair(minPair, (S[i], S[j]))

return minPair

Θ(𝑛) and preserves the y-sort order

Total Θ(𝑛) for this function
ShamosClosestPair(P[1..n])
    Px = sort(P) by increasing x values
    Py = sort(P) by increasing y values
    Recurse(Px, Py)

Recurse(Px[1..n], Py[1..n])
    // base case
    if n < 4 then return BruteForce(Px)

    // divide & conquer
    xmid = Px[n/2].x
    PxL = Px[1..(n/2)]  // x ≤ xmid
    PxR = Px[(n/2+1)..n]  // x > xmid
    PyL = select p from Py where p.x ≤ xmid
    PyR = select p from Py where p.x > xmid
    pairL = Recurse(PxL, PyL)
    pairR = Recurse(PxR, PyR)

    // combine
    δ = min(dist(pairL), dist(pairR))  // Θ(1)
    pairS = findMinSpanningPair(δ, Py, xmid)  // Θ(n)
    return minDistPair(pairL, pairR, pairS)  // Θ(1)

Time complexity:

\[ T(n) = 2T \left( \frac{n}{2} \right) + \Theta(n) \]

Merge sort recurrence...
\[ T(n) \in \Theta(n \log n) \]

So runtime for Shamos' algorithm is in \( \Theta(n \log n) \).