CS 341: ALGORITHMS

Lecture 7: divide and conquer

Slides by Trevor Brown (some material from Doug Stinson and Semih Salihoglu)

trevor.brown@uwaterloo.ca
(http://tbrown.pro)
TAKING SELECTION FURTHER

• Last time we showed:
  • QuickSelect with average case runtime in $O(n)$
• Next up:
  • Median-of-medians QuickSelect (MOM-QuickSelect)
  • worst case runtime in $O(n)$

The algorithm we will see picks a good pivot in every recursive call

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

Must get a good pivot within $O(1)$ recursive calls always
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 MOM-QuickSelect on one subarray (left OR right)
• Only difference is how we choose the pivot
• Always want to pick a good pivot
**ALWAYS PICKING 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

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

**Time complexity** for this step?  
**Time complexity** for this step?  
Recursive problem size?
**HOW GOOD IS THE PIVOT 23?**

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

- # elements ≤ 23 is at least 3(5). This is at least 3/10ths of our 50-element input, or 3\(n/10\).
- # 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, because it is trapped between two fractions of \(n\)!

We recurse on \(A_L\) or \(A_R\), and both have size at most \(7n/10\).
Algorithm: \texttt{MOM-QuickSelect}(k, n, A)
1. if $n \leq 14$ then sort $A$ and return $(A[k])$
2. write $n = 10r + 5 + \theta$, where $0 \leq \theta \leq 9$
3. construct $B_1, \ldots, B_{2r+1}$ (subarrays of $A$, each of size 5)
4. find medians $m_1, \ldots, m_{2r+1}$ non-recursively
5. $M \leftarrow [m_1, \ldots, m_{2r+1}]$
6. $y \leftarrow \texttt{MOM-QuickSelect}(r + 1, 2r + 1, M)$
7. $(A_L, A_R, \text{posn}) \leftarrow \texttt{Restructure}(A, y)$
8. if $k = \text{posn}$ then return $(y)$
9. else if $k < \text{posn}$ then return $(\texttt{MOM-QuickSelect}(k, \text{posn} - 1, A_L))$
10. else return $(\texttt{MOM-QuickSelect}(k - \text{posn}, n - \text{posn}, A_R))$

$T(n) \leq O(n) + T(n/5) + T(7n/10)$ if $n \geq 15$

$T(n) = O(1)$ if $n \leq 14$
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.

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

The sum can be simplified as follows:

\[ \sum_{i=0}^{8} n \left( \frac{9}{10} \right)^i = 10n \in \Theta(n) \]
THE CLOSEST PAIR PROBLEM
**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

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
DC Algorithm Template:

**procedure** Algorithm(P of n points):
   sort P by x values
   DC-CP(P)

**procedure** DC-CP(P sorted by x values):
   if (P.size ≤ 3) compare all & return closest;
   pair_L = DC-CP(P[1,…,n/2])
   pair_R = DC-CP(P[n/2+1,…,n])
   pair_s = findMinSpanningPair(P)
   return minDistPair(pair_L, pair_R, pair_s)

Q: How can we find the spanning pair quickly?
Observation 1

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

Then 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 pair, \( \delta \)

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.

◆ Then the other point can only lie in this \( \delta \times \delta \) square.
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 $\delta \times \delta$ 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…
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
DC-CP

**procedure** Algorithm(P of n points):
    sort P by x values
    DC-CP1(P)

**procedure** DC-CP(P sorted by x values):
    if (P.size ≤ 3) compare all & return closest;
    pair\textsubscript{L} = DC-CP(P[1,...,n/2])
    pair\textsubscript{R} = DC-CP(P[n/2+1,...,n])
    δ = dist(minDistPair(pair\textsubscript{L}, pair\textsubscript{R}))
    pair\textsubscript{s} = findMinSpanningPair(δ, P)
    **return** minDistPair(pair\textsubscript{L}, pair\textsubscript{R}, pair\textsubscript{s})
procedure findMinSpanningPair (δ, P):
  S = select each p in P s.t |p_{n/2}.x-p.x| \leq \delta \rightarrow O(n)
  sort(S by increasing y values) \rightarrow O(n \log n)
  minDist = +\infty
  minPair = null;
  for i = 1 to S.length: \rightarrow O(n)
    j = i+1 (compare S[i] to points above it)
    while (|S[j].y - S[i].y| \leq \delta):
      if (dist(S[i], S[j]) < minDist):
        minPair = (S[i], S[j]);
        minDist = dist(S[i], S[j])
    j++;
  return minPair

Q: How many times does the while loop execute? Claim: O(1) times
For a point p, how many times does while loop execute?

Obs: as many times 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.
Recall: Each point in the square is at least at distance $\delta$.

Q1: How many can fit the lower triangle?

A: 3

Why?

Because $\delta$ is the smallest distance between any pair of points that are both in L, or both in R.

no other point can be inside the triangle except the other two corners
Recall: Each point in the square is at least at distance $\delta$.

Q1: How many can fit the lower triangle?
A: 3

Q2: How many can fit the square?
A: 4
For a point p, how many times does while loop execute?

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

$\# \text{ points in the } 2\delta \times \delta \text{ rectangle} \leq 4 + 4 = 8$
procedure findMinSpanningPair (δ, P):
S = select each p in P s.t |P[n/2].x-p.x| ≤ δ
sort(S by increasing y values) → O(n log n)
minDist = +∞
minPair = null;
for i = 1 to S.length: → O(n)
    j = i+1
    while (|S[j].y - S[i].y| ≤ δ):
        if (dist(S[i], S[j]) < minDist):
            minPair = (S[i], S[j])
            j++;
return minPair

Total for this procedure: O(n log n)
DC-CP: Runtime Analysis

**procedure** DC-CP1(P sorted by x values):

1. if (P.size ≤ 3) compare all & return closest;
2. \( \text{pair}_L = \text{DC-CP}(P[1,\ldots,n/2]) \)
3. \( \text{pair}_R = \text{DC-CP}(P[n/2+1,\ldots,n]) \)
4. \( \delta = \text{dist}(	ext{minDistPair}(\text{pair}_L, \text{pair}_R)) \)
5. \( \text{pair}_s = \text{findMinSpanningPair}(\delta, P) \)
6. return \( \text{minDistPair}(\text{pair}_L, \text{pair}_R, \text{pair}_s) \)

**Recursive part:** Outside Recursive Calls: \( n \log n \) work.

\[
T(n) = 2T(n/2) + n \log n
\]

We showed last class that \( T(n) = O(n \log^2 n) \).

**Total Alg Work:** \( O(n \log n) + O(n \log^2 n) = O(n \log^2 n) \).
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.
procedure Algorithm(P of n points):
    \( P_x \) = sort P by x values in increasing order
    \( P_y \) = sort P by y values in increasing order
    DC-Shamos(\( P_x \), \( P_y \))

procedure DC-Shamos(\( P_x \), \( P_y \)):
    if (\( P_x \).size \leq 3) …;
    \( P_yL \) = select from \( P_y \) points with \( x \leq P_x[n/2].x \)
    \( P_yR \) = select from \( P_y \) points with \( x > P_x[n/2].x \)
    \( \text{pair}_L \) = DC-Shamos(\( P_x[1,\ldots,n/2] \), \( P_yL \))
    \( \text{pair}_R \) = DC-Shamos(\( P_x[n/2+1,\ldots,n] \), \( P_yR \))
    \( \delta \) = dist(minDistPair(\( \text{pair}_L \), \( \text{pair}_R \)))
    \( \text{pair}_S \) = findMinSpanningPairShamos(\( \delta \), \( P_x \), \( P_y \))
    return minDistPair(\( \text{pair}_L \), \( \text{pair}_R \), \( \text{pair}_S \))

Sorted by y already!
Shamos’ DC Algorithm (1975) (2)

Don’t need to sort by \( y \)!

\[
\text{procedure } \text{findMinSpanningPairShamos}(\delta, P_x, P_y):
\]

\[
S = \text{select each } p \text{ in } P_y \text{ s.t } |P_x[n/2].x - p.x| \leq \delta
\]

\[
\text{minDist} = +\infty
\]

\[
\text{minPair} = \text{null};
\]

\[
\text{for } i = 1 \text{ to } S.\text{length}:
\]

\[
j = i+1
\]

\[
\text{while } (|S[j].y - S[i].y| \leq \delta):
\]

\[
\text{if } (\text{dist}(S[i], S[j]) < \text{minDist}):
\]

\[
\text{minPair} = (S[i], S[j])
\]

\[
j++;
\]

\[
\text{return } \text{minPair} \quad \text{Total: } O(n)
\]

Sorted by \( y \) already!

After this point, code is really the same as the earlier alg.
Runtime Analysis of Shamos’ Algorithm

Outside Recursive Calls: $O(n)$ work.

$$T(n) = 2T(n/2) + O(n)$$

By Master Theorem, total: $O(n \log n)$

(Also note: recurrence is the same as the recurrence for merge sort $\rightarrow$ immediately get $O(n \log n)$)

Total Work for Shamos

$= O(\text{time for sort}) + O(\text{time for DC-Shamos call})$
$= O(n \log n) + O(n \log n) = O(n \log n)$. 