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.)

$A$

$y$

Restructure($A$, $y$)

$A$

Number of elements on each side depend on the value $y$...

$A$ after Restructure($A$, $y$)

Recursive calls

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 $A_R$
OVERLY OPTIMISTIC ANALYSIS 😅

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

But we don't always recurse on \( \frac{n}{2} \) elements!

WORST-CASE ANALYSIS

- If we always get \( y = 1 \) 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 \( y \in \left( \frac{3n}{4}, \frac{3n}{4} \right) \)
- For any good pivot we recurse on at most \( \frac{3n}{4} \) elements
- Probability of an arbitrary pivot being good?

Here is a more rigorous proof of the average-case complexity. We say the algorithm is in phase 1 if the current subarray has size \( x \), where \( n \left( \frac{3}{4} \right)^{j+1} < x \leq n \left( \frac{3}{4} \right)^j \).

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 \( E[X_j] \leq 3n/4 \) (where \( E[X] \) denotes the expectation of a random variable). The total time of the algorithm is given by \( T = \sum_{j=0}^{\infty} X_j \). Therefore

\[
E[X] = \sum_{j=0}^{\infty} E[X_j] \leq 2m \sum_{j=0}^{\infty} (3/4)^j \text{ if } E[X] \in \Theta(n).
\]

TAKING SELECTION FURTHER

- We just showed:
  - QuickSelect with average case runtime in \( \Theta(n) \)

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

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 MOMQuickSelect on one subarray
- Only difference is how we choose the pivot
- **Always** want to pick a **good pivot**

**HOW GOOD IS THE PIVOT 23?**

Recall median of each row

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

Imagine sorting each row:

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

If elements ≤ 23 is at least 3/5, this is at least 3/10ths of our 30-element input.

This is a good pivot

We recurse on $A_d$ or $A_u$, and both have size at most $\lceil n/2 \rceil$

**ALWAYS PICKING A GOOD PIVOT**

Example input: $A[0..50]: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]$

**Group into rows of 5**

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

**Find median of each row**

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

**Build array of medians**

$[11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]$

**Time complexity for this step?**

$\mathcal{O}(n)$

**Recursive find the median of these medians: 23**

**Recursive pivot used**

**MOMQuickSelect(k, n, A)**

```plaintext
// base case
if n <= 14 then sort(A) and return A[k]
// divide and conquer to find medians
r = (n/3) / 10
medians[0...r-1] = new array for i = 1...r-1
B[i, 1] = A[(n/2-1)*r + 1],...,(n*r)]
sort(B)
medians[r] = B[r]
y = MOMQuickSelect(r, r, medians)
// divide and conquer to find rank k
if k <= y then return y
else if k <= y then return MOMQuickSelect(k, y, y, A)
else if k > y then return MOMQuickSelect(k, y, y, A)
```

**Restructure(k, y, y)**

```plaintext
A_d = [20, 21, 11, 12, 13, 14, 18, 19, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
y = 15, y = 16
A_u = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
k = 15 > y = 16
```

**MOMQuickSelect(11, 14, A)**

```plaintext
// base case
if n <= 14 then sort(A) and return A[k]
// divide and conquer to find medians
r = (n/3) / 10
medians[0...r-1] = new array for i = 1...r-1
B[i, 1] = A[(n/2-1)*r + 1],...,(n*r)]
sort(B)
medians[r] = B[r]
y = MOMQuickSelect(r, r, medians)
// divide and conquer to find rank k
if k <= y then return y
else if k <= y then return MOMQuickSelect(k, y, y, A)
else if k > y then return MOMQuickSelect(k, y, y, A)
```
HOW MUCH DOES THE PROBLEM SHRINK?

- Shrinks by at least $3(r+1)$
- Problem size $\approx n = 10r + 5$
- Subproblem size $\leq n - \text{Shrink} = n - 3(r+1)
  \quad = 10r + 5 - 3r - 3 = 7r + 2$
- Express in terms of $n$ using $r = \frac{n-5}{10}$

\[
\text{Subproblem size} \leq \frac{n-5}{10} + 2 \leq \frac{n-5}{10} + 2
\]

\[
= \frac{5}{10} - 2 = \frac{5}{10}
\]

\[
T(n) \in O(n) + T(n/5) + T(7n/10)
\]

The key fact is that $1/5 + 7/10 = 10/20 = 1/2 < 1$

\[
T(n) \in O(n) + T(n/5) + T(7n/10)
\]

if $n \geq 15$

$T(n) \in O(1)$

if $n \leq 14$

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

\[
T(n) \in O(n) + T(n/5) + T(7n/10)
\]

if $n \geq 15$

$T(n) \in O(1)$

if $n \leq 14$

The closest pair problem

When someone near you
coughs
THE CLOSEST PAIR PROBLEM

- Input: Set $P$ of $n$ 2D points
- Output: pair $p$ and $q$ s.t. dist$(p, q)$ minimum over all pairs
- Break ties arbitrarily
- dist$(p, q) = (p.x - q.x)^2 + (p.y - q.y)^2$

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 $R$

Observation 1

- Let $\delta = \min(\text{dist}(\text{pair}_L), \text{dist}(\text{pair}_R))$
- Then pair, (if closest globally) lies in the above $2\delta$-wide green strip
- Q: Why?

Example for Observation 1

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

Observation 2

- Say, $p$ (the lowest $y$ valued point in strip) is in pair
- 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 $3\delta \times 3\delta$ square.
- Q: Why?
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...

A More Practical Idea

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

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 2δxδ above rectangle each time

Claim: loop performs $O(1)$ iterations!
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).

43

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.

So, 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

- Let $T'(n)$ be runtime of findMinSpanningPair($P[1..n]$)
  - Sort $P$ by $y$ values causes findMinSpanningPair to take $\Theta(n \log n)$ 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

IMPROVING THIS RESULT FURTHER

- 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

IMPROVING THE PREVIOUS ALGORITHM
Shamos’ algorithm (1975)

This selection step preserves the y-sort order

Observe $P_L$ and $P_U$ contain the same points (specifically the points with $x \leq x_{mid}$)

Moreover, $P_L$ is sorted by $x$ while $P_U$ is sorted by $y$

And similarly for $P_R$, $P_U$

No need to sort in Recurse!

Time complexity

$T(n) = 2T(n/2) + O(n)$$\Rightarrow T(n) \in \Theta(n \log n)$

So runtime for Shamos’ algorithm is in $\Theta(n \log n)$

Total $\Theta(n)$ for this function