CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca
THIS TIME

• Divide and conquer algorithms
  • (Finishing) Strassen matrix multiplication
• Selection (k-th smallest element)
  • Average case $O(n)$ time algorithm
  • Worst case $O(n)$ time algorithm
• Closest pair of points in 2D
**STRAßEN** MATRIX MULTIPLICATION

Define

\[ A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \]

\[ P_1 = a(f - h) \quad P_2 = (a + b)h \]
\[ P_3 = (c + d)e \quad P_4 = d(g - e) \]
\[ P_5 = (a + d)(e + h) \quad P_6 = (b - d)(g + h) \]
\[ P_7 = (a - c)(e + f). \]

Then, compute

\[ r = P_5 + P_4 - P_2 + P_6 \quad s = P_1 + P_2 \]
\[ t = P_3 + P_4 \quad u = P_5 + P_1 - P_3 - P_7. \]

We now require only 7 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C = AB \).
The verifications are somewhat tedious but straightforward.

For example,

\[ P_5 + P_4 - P_2 + P_6 \]
\[ = (a + d)(e + h) + d(g - e) - (a + b)h + (b - d)(g + h) \]
\[ = ae + ah + de + dh + dg - de - ah - bh + bg + bh - dg - dh \]
\[ = ae + bg, \]

The recurrence is \( T(k) = 7T(k/2) + \Theta(k^2) \), so
\( T(k) \in \Theta(k^{\log_2 7}) = \Theta(n^{2.81}) \) by the **Master Theorem**.

Details: \( a = 7, b = 2 \), so \( x = \log_2 7 = 2.81, y = 2, x > y \) so we are in case 1 and \( T(n) = \Theta(n^x) = \Theta(n^{2.81}) \).

**Strassen's algorithm** was improved in 1990 by Coppersmith-Winograd. Their algorithm has complexity \( O(n^{2.376}) \). Some slight improvements have been found more recently.
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 $A$
• When $k=1$: output is smallest integer in $A$
• When $k=2$: output is 2nd smallest integer in $A$
• Median is a special case of Selection where $k = \lceil n/2 \rceil$
• Simple algorithms for solving selection?
QuickSelect

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.)


Then the $k$th smallest element of $A$ is

$$
\begin{cases}
  y & \text{if } k = posn \\
  \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\
  \text{the } (k - posn)\text{th smallest element of } A_R & \text{if } k > posn.
\end{cases}
$$

We make (at most) one recursive call at each level of the recursion.
QuickSelect(k, n, A)
   if n = 1 then return A[1]
   y = A[1] // pick an arbitrary pivot
   (AL, AR, posn) = Restructure(A, y)
   if k == posn return y
   else if k < posn then return QuickSelect(k, posn - 1, AL)
   else return QuickSelect(k - posn, n - posn, 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
   sizeL = 0, sizeR = 0
   for i = 1 .. n
      if A[i] < y then
         AL[sizeL] = A[i] ; sizeL++
      else if A[i] > y then
         AR[sizeR] = A[i] ; sizeR++
   position_of_y = sizeL // alias included for clarity
   return (AL, AR, position_of_y)
Average-case Analysis of QuickSelect

We say that a pivot is good if \( \text{posn} \) is in the middle half of \( A \), i.e., \( n/4 \leq \text{posn} \leq 3n/4 \).

The probability that a pivot is good is \( 1/2 \).

On average, after two iterations, we will encounter a good pivot.

If a pivot is good, then \( |A_L| \leq 3n/4 \) and \( |A_R| \leq 3n/4 \).

With an expected linear amount of work, the size of the subproblem is reduced by at least 25%.

Let’s consider the average-case recurrence relation:
\[ T(n) = T(3n/4) + \Theta(n). \]

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 \( T(n) \in \Theta(n) \) in 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 (MOM-QuickSelect)
  • worst case runtime in $O(n)$

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

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

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

**Time complexity** for this step?

**Time complexity** for this step?

**Recursively** find median of this smaller array: 23

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

<table>
<thead>
<tr>
<th>Then sorting rows by medians:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 7 9</td>
</tr>
<tr>
<td>1 12 15</td>
</tr>
<tr>
<td>6 11 20</td>
</tr>
<tr>
<td>3 13 22</td>
</tr>
<tr>
<td>2 4 23</td>
</tr>
<tr>
<td>10 24 32</td>
</tr>
<tr>
<td>16 19 33</td>
</tr>
<tr>
<td>25 29 39</td>
</tr>
</tbody>
</table>

# elements < 23 is at least 3 × 5 − 1. This represents ~3/10ths of the input, or ~3\(n/10\).

# elements > 23 is at most 10 × 5 − 3 × 5 − 1 = 7 × 5 − 1. This represents ~7/10ths of the input, or ~7\(n/10\).

Argument generalizes easily to input of size \(n\) instead of size 50. We recurse on input size 7\(n/10\) in the worst case.
Algorithm: 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. \( y \leftarrow \text{MOM-QuickSelect}(r + 1, 2r + 1, M) \)
7. \((A_L, A_R, \text{posn}) \leftarrow \text{Restructure}(A, y)\)
8. if \( k = \text{posn} \) then return \((y)\)
9. else if \( k < \text{posn} \) then return \((\text{MOM-QuickSelect}(k, \text{posn} - 1, A_L))\)
10. else return \((\text{MOM-QuickSelect}(k - \text{posn}, n - \text{posn}, A_R))\)

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

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

\[
T(n) = O(1) \quad \text{if } n \leq 14
\]

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

◆ Input: Set P of n 2-D points

◆ Output: pair p and q s.t. 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 $p_5$ be part of a globally closest pair?
A: No. Everything in $R$ has dist $> \delta$ to $p_5$. And we already have a solution with dist $= \delta$. 
Observation 2

◆ Say, $p_7$ (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 $p_7$ on y axis.*
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 δxδ 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 δxδ 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...

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 1 (1)**

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

**procedure** DC-CP1(P sorted by x values):
  if (P.size ≤ 3) compare all & return closest;
  pair\_L = DC-CP1(P[1,...,n/2])
  pair\_R = DC-CP1(P[n/2+1,...,n])
  \( \delta = \min(\text{dist}(\text{pair\_L, pair\_R})) \)
  pair\_S = \text{findMinSpanningPair}(\delta, P)
  return \( \min(\text{pair\_L, pair\_R, pair\_S}) \)
procedure findMinSpanningPair (δ, P):
    S = select each p in P s.t \(|p_{n/2}.x-p.x| \leq \delta\) → O(n)
    sort(S by increasing y values) → O(n log n)
    minDist = +∞
    minPair = null;
    for i = 1 to S.length: → 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
# Points in a $\delta \times \delta$ Square

Recall: Each point in the square is at least at distance $\delta$.

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

Q2: How many can fit the square?
   A: 4
For a point $p$, how many times does the 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)
procedure DC-CP1(P sorted by x values):
  if (P.size ≤ 3) compare all & return closest;
  pair_L = DC-CP1(P[1,…,n/2])
  pair_R = DC-CP1(P[n/2+1,…,n])
  δ = min(dist(pair_L, pair_R))
  pair_s = findMinSpanningPair(δ, P)
  return min(pair_L, pair_R, pair_s)

Recursive part: Outside Recursive Calls: n log n work.
\[ T(n) = 2T(n/2) + n \log n \]

Exercise: Show by induction or recursion tree that
total work of recursive part is \( O(n \log^2 n) \).

Total Alg Work: \( O(n \log n) + O(n \log^2 n) = O(n \log^2 n) \).
AN IMPROVEMENT WE DID NOT COVER

... but which is easy to understand and interesting
IMPROVING THIS: SHAMOS’ 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
Recall:

```plaintext
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)
    minDist = +∞
    minPair = null;
    for i = 1 to S.length:
        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
```

This step needs to go!
Shamos’ DC Algorithm (1975) (1)

**procedure** Algorithm(P of n points):
\[ P_x = \text{sort P by x values in increasing order} \]
\[ P_y = \text{sort P by y values in increasing order} \]
\[ \text{DC-Shamos}(P_x, P_y) \]

**procedure** DC-Shamos(P\_x, P\_y):
\[ \text{if (} P_x.\text{size} \leq 3 \text{) } \ldots; \]
\[ P_{yL} = \text{select from } P_y \text{ points with } x \leq P_x[n/2].x \]
\[ P_{yR} = \text{select from } P_y \text{ points with } x > P_x[n/2].x \]
\[ \text{pair}_L = \text{DC-Shamos}(P_x[1,\ldots,n/2], P_{yL}) \]
\[ \text{pair}_R = \text{DC-Shamos}(P_x[n/2+1,\ldots,n], P_{yR}) \]
\[ \delta = \min(\text{dist}((\text{pair}_L, \text{pair}_R)) \]
\[ \text{pair}_s = \text{findMinSpanningPairShamos}(\delta, P_x, P_y) \]
\[ \text{return } \min(\text{pair}_L, \text{pair}_R, \text{pair}_s) \]
procedure findMinSpanningPairShamos(\(\delta, P_x, P_y\)):

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

\(minDist = +\infty\)

\(minPair = null;\)

for \(i = 1\) to \(S.length\):

\(j = i + 1\)

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

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

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

\(j++;\)

return \(minPair\)

Total: \(O(n)\)
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 → immediately get $O(n \log n)$)

Total Work for Shamos

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

$= O(n \log n) + O(n \log n) = O(n \log n)$. 