CSE 101, WINTER 2018

DESIGN AND ANALYSIS OF ALGORITHMS

LECTURES 4 + 5: DIVIDE AND CONQUER

CLASS URL:
HTTP://VLSICAD.UCSD.EDU/COURSES/CSE101-W18/
MINIMUM DISTANCE

- Given a list of coordinates in the plane, find the distance between the closest pair.
distance\((x_i, y_i), (x_j, y_j)\) = \sqrt{(x_i - y_i)^2 + (x_j - y_j)^2}
Given a list of coordinates, \([(x_1, y_1), \ldots, (x_n, y_n)]\), find the distance between the closest pair.

Brute force solution?
Given a list of coordinates, \([(x_1, y_1), \ldots, (x_n, y_n)]\), find the distance between the closest pair.

Brute force solution?

\[
\begin{align*}
\text{min} &= 0 \\
\text{for } i \text{ from } 1 \text{ to } n-1: \\
&\quad \text{for } j \text{ from } i+1 \text{ to } n: \\
&\quad \quad \text{if } \text{min} > \text{distance}(x_i, y_i, x_j, y_j) \\
&\quad \quad \text{min} := \text{distance}(x_i, y_i, x_j, y_j) \\
\text{return } \text{min}
\end{align*}
\]
MINIMUM DISTANCE

- Base case.
- Break the problem up.
- Recursively solve each problem.
  “Assume the algorithm works for the subproblems”
- Combine the results.
BASE CASE

- if \( n=2 \) then return \( \text{distance}((x_1, y_1), (x_2, y_2)) \)
BREAK THE PROBLEM INTO SMALLER PIECES
We will break the problem in half. Sort the points by their $x$ values. Then find a value $x_m$ such that half of the $x$ values are on the left and half are on the right.
breaker the problem into smaller pieces

- Usually the smaller pieces are each of size n/2.

- We will break the problem in half. Sort the points by their $x$ values.
- Then find a value $x_m$ such that half of the $x$ values are on the left and half are on the right.
- Perform the algorithm on each side.
- “Assume our algorithm works!!”
- What does that give us?
Usually the smaller pieces are each of size $n/2$.

We will break the problem in half. Sort the points by their $x$ values.

Then find a value $x_m$ such that half of the $x$ values are on the left and half are on the right.

Perform the algorithm on each side.

“Assume our algorithm works!!”

What does that give us?

It gives us the distance of the closest pair on the left and on the right and lets call them $d_L$ and $d_R$.
EXAMPLE

\[ y \]

\[ d_L \]

\[ d_R \]

\[ x_m \]
How will we use this information to find the distance of the closest pair in the whole set?

\[ \text{Naive: } (\frac{n}{2})(\frac{n}{2}) = O(n^2) \]
How will we use this information to find the distance of the closest pair in the whole set?

We must consider if there is a closest pair where one point is in the left half and one is in the right half.

How do we do this?
How will we use this information to find the distance of the closest pair in the whole set?

We must consider if there is a closest pair where one point is in the left half and one is in the right half.

How do we do this?

Let $d = \min(d_L, d_R)$ and compare only the points $(x_i, y_i)$ such that $x_m - d \leq x_i$ and $x_i \leq x_m + d$.

Worst case, how many points could this be?
Let $P_m$ be the set of points within $d$ of $x_m$. Then $P_m$ may contain as many as $n$ different points. So, to compare all the points in $P_m$ with each other would take $\binom{n}{2}$ many comparisons. So the runtime recursion is:
Let $P_m$ be the set of points within $d$ of $x_m$.
Then $P_m$ may contain as many as $n$ different points.
So, to compare all the points in $P_m$ with each other would take $\binom{n}{2}$ many comparisons.
So the runtime recursion is:
\[
T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)
\]
Can we improve the combine term?
EXAMPLE

\[ y \]

\[ d_L \]

\[ d_R \]

\[ d \]

\[ x_m \]
Given a point \((x, y) \in P_m\), let’s look in a \(2d \times d\) rectangle with that point at its upper boundary:

How many points could possibly be in this rectangle?
Given a point \((x, y) \in P_m\), let’s look in a \(2d \times d\) rectangle with that point at its upper boundary:

There could not be more than 8 points total because if we divide the rectangle into 8 \(\frac{d}{2} \times \frac{d}{2}\) squares then there can never be more than one point per square.

Why???
So instead of comparing \((x, y)\) with every other point in \(P_m\) we only have to compare it with the next 7 points lower than it.

To gain quick access to these points, let’s sort the points in \(P_m\) by \(y\) values.

Now, if there are \(k\) vertices in \(P_m\) we have to sort the vertices in \(O(k \log k)\) time and make at most \(7k\) comparisons in \(O(k)\) time for a total combine step of \(O(k \log k)\).

But we said in the worst case, there are \(n\) vertices in \(P_m\) and so worst case, the combine step takes \(O(n \log n)\) time.
But we said in the worst case, there are $n$ vertices in $P_m$ and so worst case, the combine step takes $O(n \log n)$ time.

Runtime recursion:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$
\[ T(n) = O \left( n^d \sum_{k=1}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right) \]
But we said in the worst case, there are $n$ vertices in $P_m$ and so worst case, the combine step takes $O(n \log n)$ time.

Runtime recursion:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

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

Can anyone improve on this runtime?

```
// preprocess: sort by x's and by y's.
T(n) = 2T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n \log n).
```
The median of a list of numbers is the middle number in the list.

If the list has \( n \) values and \( n \) is odd, then the middle element is clear. It is the \( \lfloor n/2 \rfloor \)th smallest element.

Example:

\[
med(8,2,9,11,4) = 8
\]
because \( n = 5 \) and 8 is the 3rd = \( \lfloor 5/2 \rfloor \)th smallest element of the list.
The median of a list of numbers is the middle number in the list.

If the list has \( n \) values and \( n \) is even, then there are two middle elements. Let's say that the median is the \( n/2 \)th smallest element. Then in either case the median is the \([n/2]\)th smallest element.

Example:

\[
\text{med}(10, 23, 7, 26, 17, 3) = 10
\]

because \( n = 6 \) and 10 is the 3rd = \([6/2]\)th smallest element of the list.
The purpose of the median is to summarize a set of numbers. The *average* is also a commonly used value. The median is more typical of the data.

For example, suppose in a company with 20 employees, the CEO makes 1 million and all the other workers each make 50,000.

Then the average is 97,500 and the median is 50,000, which is much closer to the typical worker’s salary.
Can you think of an efficient way to find the median?
How long would it take?
Is there a lower bound on the runtime of all median selection algorithms?
Can you think of an efficient way to find the median?
How long would it take?
Is there a lower bound on the runtime of all median selection algorithms?

Sort the list then find the \([n/2]\)th element \(O(n \log n)\).
You can never have a faster runtime than \(O(n)\) because you at least have to look at every element.
All selection algorithms are \(\Omega(n)\)
What if we designed an algorithm that takes as input, a list of numbers of length \( n \) and an integer \( 1 \leq k \leq n \) and outputs the \( k \)th smallest integer in the list.

Then we could just plug in \( \lceil n/2 \rceil \) for \( k \) and we could find the median!!

**Quickselect.**
Let's think about selection in a divide and conquer type of way.

- Break a problem into similar subproblems
  - Split the list into two sublists
- Solve each subproblem recursively
  - recursively select from one of the sublists
- Combine
  - determine how to split the list again.
How would you split the list?

Just splitting the list down the middle does not help so much.

What we will do is pick a random “pivot” and split the list into all integers greater than the pivot and all that are less than the pivot.

Then we can determine which list to look in to find the $k$th smallest element. (Note that the value of $k$ may change depending on which list we are looking in.)

After this is done, the pivot is in its correct position.
Example:

Selection([40, 31, 6, 51, 76, 58, 97, 37, 86, 31, 19, 30, 68], 7)

= Selection(S_k, 2)

pick a random pivot.....

S_L: 6 19 30
S_U: 31 31
S_R: 40 51 76 58 97 37 86 68
EXAMPLE!!!

- Selection([40,31,6,51,76,58,97,37,86,31,19,30,68], 7)
Input: list of integers and integer $k$
Output: the $k$th smallest number in the set of integers.

```
function Selection(a[1…n], k)
  if $n = 1$:
    return $a[1]$
  pick a random integer in the list $v$. 
  Split the list into sets $SL$, $Sv$, $SR$.
  if $k \leq |SL|$:
    return Selection($SL$, $k$)
  if $k \leq |SL| + |Sv|$:
    return $v$
  else:
    return Selection($SR$, $k - |SL| - |Sv|$)
```
The runtime is dependent on how big are $|SL|$ and $|SR|$.

If we were so lucky as to choose $v$ to be close to the median every time, then $|SL| \approx |SR| \approx n/2$. And so, no matter which set we recurse on,

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

And by the Master Theorem:

$$T(n) = O(n).$$
The runtime is dependent on how big are $|SL|$ and $|SR|$.

Conversely, if we were so unlucky as to choose $v$ to be the maximum (resp. minimum) then $|SL|$ (resp. $|SR|$) = n-1 and

$$T(n) = T(n - 1) + O(n)$$

Which is ……………?

$$T(n) = O(n^2)$$
The runtime is dependent on how big are $|SL|$ and $|SR|$.

Conversely, if we were so unlucky as to choose $v$ to be the maximum (resp. minimum) then $|SL|$ (resp. $|SR|$) = $n-1$ and

$$T(n) = T(n - 1) + O(n)$$

Which is $O(n^2)$, worse than sorting then finding.

So is it worth it even though there is a chance of having a high runtime?
If you randomly select the \( i \)th element, then your list will be split into a list of length \( i \) and a list of length \( n-i \).

So when we recurse on the smaller lists, it will take time proportional to

\[
\max(i, n - i)
\]
Clearly, the split with the smallest maximum size is when $i=\lfloor n/2 \rfloor$ and worst case is $i=n$ or $i=1$. 
What is the expected runtime?

Well what is our random variable?

For each input and sequence of random choices of pivots, the random variable is the runtime of that particular outcome.
So if we want to find the expected runtime, we must sum over all possibilities of choices.

Let $ET(n)$ be the expected runtime. Then

$$ET(n) = \frac{1}{n} \sum_{i=1}^{n} ET(\max(i, n - i)) + O(n)$$
What is the probability of choosing a value from 1 to $n$ in the interval $\left[ \frac{n}{4}, \frac{3n}{4} \right]$ if all values are equally likely?
If you did choose a value between \( n/4 \) and \( 3n/4 \) then the sizes of the subproblems would both be \( \leq \frac{3n}{4} \).

Otherwise, the subproblems would be \( \leq n \).

So we can compute an upper bound on the expected runtime.

\[
ET(n) \leq \frac{1}{2} ET \left( \frac{3n}{4} \right) + \frac{1}{2} ET(n) + O(n)
\]
\[ ET(n) \leq \frac{1}{2} ET \left( \frac{3n}{4} \right) + \frac{1}{2} ET(n) + O(n) \]

Plugging into the master theorem with \( a=1, b=\frac{4}{3}, d=1 \)

\( a < b^d \) so

\[ ET(n) \leq O(n) \]
What have we noticed about the partitioning part of Selection?

After partitioning, the “pivot” is in its correct position in sorted order.

Quicksort takes advantage of that.
Let’s think about selection in a divide and conquer type of way.

- Break a problem into similar subproblems
  - Split the list into two sublists by partitioning a pivot
- Solve each subproblem recursively
  - recursively sort each sublist
- Combine
  - concatenate the lists.
procedure quicksort(a[1…n])
if n ≤ 1:
  return a
set v to be a random element in a.
partition a into SL, Sv, SR
return quicksort(SL) ∘ Sv ∘ quicksort(SR)
QUICKSORT (RUNTIME)

- procedure quicksort(a[1...n])
- if n \leq 1:
  - return a
- set v to be a random element in a.
- partition a into SL, Sv, SR
- return quicksort(SL) \circ Sv \circ quicksort(SR)

Expected Runtime of QSort
\[ O(n \log n) \]
(Exercise)
procedure quicksort(a[1…n])
if n ≤ 1:
  ▪ return a
set v to be a random element in a.
partition a into SL, Sv, SR
return quicksort(SL) • Sv • quicksort(SR)
quicksort(60, 82, 20, 10, 7, 85, 89, 94, 33, 53, 14, 75)
Sometimes this algorithm we have described is called quick select because generally it is a very practical linear expected time algorithm. This algorithm is used in practice.

For theoretic computer scientists, it is unsatisfactory to only have a randomized algorithm that could run in quadratic time.

Blum, Floyd, Pratt, Rivest, and Tarjan have developed a deterministic approach to finding the median (or any kth biggest element.)

They use a divide and conquer strategy to find a number close to the median and then use that to pivot the values.
The strategy is to split the list into sets of 5 and find the medians of all those sets. Then find the median of the medians using a recursive call $T(n/5)$.

Then partition the set just like in quickselect and recurse on SR or SL just like in quickselect.

By construction, it can be shown that $|SR|<7n/10$ and $|SL|<7n/10$ and so no matter which set we recurse on, we have

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

You cannot use the master theorem to solve this, but you can use induction to show that if $T(n) \leq cn$ for some $c$, then $T(n + !) \leq cn$.

And so we have a linear time selection algorithm!!!!!
The MaxMin Problem

- MaxMin: Given list of n numbers, return largest and smallest
- Naïve: how many comparisons?

Recurse to find median

As pivot

Medium
THE MAXMIN PROBLEM

- **MaxMin**: Given list of n numbers, return largest and smallest
- **Naïve**: 2(n-1) comparisons (two passes)
MaxMin: Given list of n numbers, return largest and smallest

Naïve: 2(n-1) comparisons (two passes)

DC approach:
- Divide the problem
- Recursively solve each subproblem
- “Combine”
THE MAXMIN PROBLEM

- **MaxMin**: Given list of \( n \) numbers, return largest and smallest
- **Naïve**: \( 2(n-1) \) comparisons (two passes)
- **DQ approach**
  
  - \( n = 1 \rightarrow 0 \) comparisons needed
  - \( n = 2 \rightarrow 1\) comparison needed
  - else: bisect list
    - make recursive calls
    - return \( \max(\max_1, \max_2), \min(\min_1, \min_2) \)
THE MAXMIN PROBLEM

- **MaxMin**: Given list of \( n \) numbers, return largest and smallest
- **Naïve**: \( 2(n-1) \) comparisons (two passes)
- **DQ approach**
  
  \( n = 1 \rightarrow 0 \) comparisons needed
  
  \( n = 2 \rightarrow 1 \) comparison needed
  
  else: bisect list
    
    make recursive calls
    
    return \( \max(\max_1, \max_2), \min(\min_1, \min_2) \)

- **#comparisons**: \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, \ n > 2 \)
“Information argument”
- Start: Nothing known about $n$ elements
- End: “Neither Max nor Min” known about all but 2 elements

Four “buckets”
- Know Nothing
- Not Max
- Not Min
- Neither Max nor Min

$$\left(\frac{n}{2}\right) + 2\left[\frac{n}{2} - 1\right] = \frac{3n}{2} - 2$$
DQ FOR THE MAXMIN PROBLEM

- $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2$, $n > 2$
- Transform with $S(k) = T(2^k)$

\[
S(k) = 2S(k-1) + 2
\]

\[
S(k) - 2S(k-1) = 2
= 1^n \cdot 2
\]

C.P. = $(x - 2)(x - 1)$ with roots 2, 1

\[
S(k) = c_12^k + c_21^k
\]
DQ FOR THE MAXMIN PROBLEM

- \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, \; n > 2 \)
- Transform with \( S(k) = T(2^k) \)

\[
S(k) = c_1 2^k + c_2 1^k
\]

Initial Conditions

\[
S(1) = c_1 \cdot 2 + c_2 = 1
\]
\[
S(2) = c_1 \cdot 4 + c_2 = 4
\]

\( \Rightarrow c_1 = 3/2, \; c_2 = -2 \)

\[
T(n) = T(2^k) = S(k) = \frac{3}{2} \cdot 2^{\log_2 n} - 2 = \frac{3n}{2} - 2
\]
DQ FOR THE MAXMIN PROBLEM

- \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, \ n > 2 \)
- Transform with \( S(k) = T(2^k) \rightarrow S(k) = 2S(k-1) + 2 \)

**Note:** The recurrence \( a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \) has solution \( t_n = \sum_{i=1}^{k} c_i r_i^n \) where \( r_i \) are roots of the C.P.: \( (a_0 x^k + a_1 x^{k-1} + \ldots + a_k) (x - b)^d = 0 \)

\[ S(n) - 2S(n-1) = 1^n \cdot 2 \]
\[ \rightarrow a_0 = 1, \ a_1 = -2, \ b = 1, \ p(n) = 2, \ d = 0 \]
\[ \rightarrow C.P. = (x-2)(x-1)^1 \]
\[ \rightarrow S(k) = c_1 2^k + c_2 1^k \]

- Initial conditions: \( S(1) = c_1 \cdot 2 + c_2 = 1 \);
- \( S(2) = c_1 \cdot 4 + c_2 = 4 \Rightarrow c_1 = 3/2, \ c_2 = -2 \)
- \( T(n) = T(2^k) = S(k) = 3/2 \cdot 2^{\log_2 n - 2} = 3n/2 - 2 \)

i.e., \( n \) a power of 2

\[
\begin{align*}
S(1) &= T(2) = 1 \\
S(2) &= T(4) = T(2) + T(2) + 2 = 4
\end{align*}
\]

\[
S(1) = T(2) = 1, \quad S(2) = T(4) = T(2) + T(2) + 2 = 4
\]
Given a list of intervals \([a_1, b_1], \ldots, [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

An interval \([a, b]\) is a set of integers starting at \(a\) and ending at \(b\). For example: \([16, 23]\) = \{16, 17, 18, 19, 20, 21, 22, 23\}

An overlap between two intervals \([a, b]\) and \([c, d]\) is their intersection.

Given two intervals \([a, b]\) and \([c, d]\), how would you compute the length of their overlap?
Given two intervals \([a, b]\) and \([c, d]\), how would you compute the length of their overlap?

**procedure overlap**([a, b], [c, d]) [Assume that \(a \leq c\)]

- if \(b < c\):
  - return ?????????
- else:
  - if \(b \leq d\):
    - return ?????????
  - if \(b > d\):
    - return ?????????
Given two intervals \([a,b]\) and \([c,d]\), how would you compute the length of their overlap?

procedure overlap([a,b],[c,d]) [Assume that \(a \leq c\)]

- if \(b < c\):
  - return 0

- else:
  - if \(b \leq d\):
    - return \(b - c + 1\)
  - if \(b > d\):
    - return \(d - c + 1\)
DIVIDE AND CONQUER EXAMPLES (GREATEST OVERLAP.)

- Given a list of intervals \([a_1,b_1], \ldots [a_n,b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

- Example: What is the greatest overlap of the intervals:
  - \([45,57],[17,50],[10,29],[12,22],[23,51],[31,32],[10,44],[27,35]\)
DIVIDE AND CONQUER EXAMPLES
(GREATEST OVERLAP.)

- Given a list of intervals \([a_1, b_1], \ldots, [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

- Example: What is the greatest overlap of the intervals:
  - \([45, 57], [17, 50], [10, 29], [12, 22], [23, 51], [31, 32], [10, 15], [23, 35]\)
Given a list of intervals \([a_1, b_1], \ldots [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

Simple solution:
Given a list of intervals \([a_1,b_1],...[a_n,b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

Simple solution:

\[
\text{olap}:=0
\]

\[\text{for } i \text{ from } 1 \text{ to } n-1\]

\[\text{for } j \text{ from } i+1 \text{ to } n\]

\[\text{if overlap}([a_i,b_i],[a_j,b_j]) > \text{olap} \text{ then}\]

\[\text{olap}:=\text{overlap}([a_i,b_i],[a_j,b_j])\]

\[\text{return } \text{olap}\]
DIVIDE AND CONQUER EXAMPLES
(GREATEST OVERLAP.)

- Given a list of intervals \([a_1,b_1],…[a_n,b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

- Simple solution
  - olap:=0
  - for i from 1 to n-1
    - for j from i+1 to n
      - if overlap([a_i,b_i],[a_j,b_j])>olap then
        - olap:=overlap([a_i,b_i],[a_j,b_j])
  - return olap

What is the runtime? \(O(n^2)\)
Can we do better? maybe...
Given a list of intervals \([a_1, b_1], \ldots [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

- Compose your base case
- Break the problem into smaller pieces
- Recursively call the algorithm on the smaller pieces
- Combine the results
What happens if there is only one interval?

return 0
What happens if there is only one interval?

if $n=1$ then return 0
The question here is “Would knowing the result on smaller problems help with knowing the solution on the original problem?”

(In this stage, let’s keep the combine part in mind.)

How would you break the problem into smaller pieces?
Would it be helpful to break the problem into two depending on the starting value?

Can you determine the greatest overlap where one interval is red and one is blue.

And what is right is 35?
Break the problem into smaller pieces.

- Sort the list and break it into lists each of size \( n/2 \).

- \([10,15],[10,29],[12,22],[17,50],[23,51],[27,35],[31,32],[45,57]\)
BREAK THE PROBLEM INTO SMALLER PIECES

- Sort the list and break it into lists each of size $n/2$.
  - $[10,15],[10,29],[12,22],[17,50],[23,51],[27,35],[31,32],[45,57]$
- Let’s assume we can get a DC algorithm to work. Then what information would it give us to recursively call each subproblem?
Let’s assume we can get a DC algorithm to work. Then what information would it give us to recursively call each subproblem?

- overlapDC([10,15],[10,29],[12,22],[17,50])=12
- overlapDC([23,51],[27,35],[31,32],[45,57])=8
BREAK THE PROBLEM INTO SMALLER PIECES

- overlapDC([10,15],[10,29],[12,22],[17,50])=12
- overlapDC([23,51],[27,35],[31,32],[45,57])=8

Is this enough information to solve the problem? What else must we consider?

29-17=12
35-27=8
BREAK THE PROBLEM INTO SMALLER PIECES

- \( \text{overlapDC}([10,15],[10,29],[12,22],[17,50]) = 12 \)
- \( \text{overlapDC}([23,51],[27,35],[31,32],[45,57]) = 8 \)

- The greatest overlap overall may be contained entirely in one sublist or it may be an overlap of one interval from either side.

29 - 17 = 12
35 - 27 = 8
So far we have split up the set of intervals and recursively called the algorithm on both sides. The runtime of this algorithm satisfies a recurrence that looks something like this.

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

What goes into the \( O(\ldots) \)?
So far we have split up the set of intervals and recursively called the algorithm on both sides. The runtime of this algorithm satisfies a recurrence that looks something like this.

\[ T(n) = 2T \left( \frac{n}{2} \right) + O(\text{??}) \]

What goes into the \( O(\text{??}) \)?

How long does it take to “combine.” In other words, how long does it take to check if there is not a bigger overlap between sublists?
What is an efficient way to determine the greatest overlap of intervals where one is red and the other is blue?
Let’s formalize our algorithm that finds the greatest overlap of two intervals such that they come from different sets sorted by starting point.

procedure overlapbetween (\([a_1, b_1], \ldots [a_\ell, b_\ell], [c_1, d_1], \ldots [c_k, d_k]\))

\((a_1 \leq a_2 \leq \cdots \leq a_\ell \leq c_1 \leq c_2 \leq \cdots \leq c_k)\)

- if \(k=0\) or \(\ell = 0\) then return 0.
Let’s formalize our algorithm that finds the greatest overlap of two intervals such that they come from different sets sorted by starting point.

**procedure overlapbetween** ([[[a₁, b₁], ... [a₁, b₁]], [[c₁, d₁], ... [c₁, d₁]])

(a₁ ≤ a₂ ≤ ... ≤ aᵢ ≤ c₁ ≤ c₂ ≤ ... ≤ cᵢ)

- if k==0 or ℓ == 0 then return 0.
- minc = c₁
- maxb = 0
- olap = 0
- for i from 1 to ℓ:
  - if maxb < bᵢ:
    - maxb = bᵢ
- for j from 1 to k:
  - if olap < overlap([minc, maxb], [cᵢ, dᵢ]):
    - olap = overlap([minc, maxb], [cᵢ, dᵢ])
- return olap
Given a list of intervals \([a_1, b_1], \ldots, [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

procedure overlapdc \(([a_1, b_1], \ldots, [a_n, b_n])\)
  ▪ if n==1 then return 0.
Given a list of intervals \([a_1, b_1], \ldots, [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

procedure overlapdc ([\(a_1, b_1\], \ldots, \([a_n, b_n]\))
  ▪ if n==1 then return 0.
  ▪ sort([\(a_1, b_1\], \ldots, \([a_n, b_n]\)) by a values.
  ▪ mid = \(\lfloor n/2 \rfloor\)
  ▪ LS = \([a_1, b_1], \ldots, [a_{mid}, b_{mid}]\]
  ▪ RS = \([a_{mid+1}, b_{mid+1}], \ldots, [a_n, b_n]\]
  ▪ olap1 = overlapdc(LS)
  ▪ olap2 = overlapdc(RS)
  ▪ olap3 = overlapbetween(LS,RS)
  ▪ return max(olap1,olap2,olap3)
GREATEST OVERLAP RUNTIME.

\[ T(n) = 2T \left( \frac{n}{2} \right) + O(n \log n) \]
Given a list of intervals \([a_1, b_1], \ldots [a_n, b_n]\) write pseudocode for a D/C algorithm that outputs the length of the greatest overlap between two intervals.

**procedure overlapdc** (sort([\([a_1, b_1], \ldots [a_n, b_n]\)])
- if \(n=1\) then return 0.
- \(\text{mid} = \lfloor n/2 \rfloor\)
- \(\text{LS} = [[a_1, b_1], \ldots [a_{\text{mid}}, b_{\text{mid}}]]\)
- \(\text{RS} = [[a_{\text{mid}+1}, b_{\text{mid}+1}], \ldots [a_n, b_n]]\)
- \(\text{olap1} = \text{overlapdc}(\text{LS})\)
- \(\text{olap2} = \text{overlapdc}(\text{RS})\)
- \(\text{olap3} = \text{overlapbetween}(\text{LS}, \text{RS})\)
- return \(\max(\text{olap1}, \text{olap2}, \text{olap3})\)
If you sort first then run the overlapdc algorithm you will have

Sorting first:

\[ S(n) = O(n \log n) \]
\[ T(n) = 2T \left( \frac{n}{2} \right) + O(n) \]

Total runtime = \( O(n \log n) + O(n \log n) = O(n \log n) \)
It is clear that the greatest overlap will be from two intervals in the left half, two from the right half, or one from the left and one from the right.

Our algorithm finds all three of these values and outputs the max.