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
- **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 $R$

We call this a spanning pair

Observation 1

- Let $\delta = \min (\text{dist}(p_1, q_1), \text{dist}(p_2, q_2))$
- Then pair $(p, q)$ (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 spanning pair, s?
A: No. Everything in R has dist > δ to p. And we already have a solution with dist = δ.

Observation 2

◆ Say, p (the lowest y valued point in strip) is in pair s.
◆ Then the other point can only lie in this δxδ square.
Q: Why?
Has to be on the opposite side & can’t be > δ higher than p on y axis.

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 \( \delta \)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...

A More Practical Idea

- Don’t differentiate between same and opposite side
- Just search the \( 2\delta \)x\( \delta \) above rectangle each time

Switching sides might complicate code... Turns out it's not needed to get good time complexity.
**Claim:** inner 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).

- $j$-loop performs at most **eight** iterations.
- Each does $\Theta(1)$ work, so entire $j$-loop does $\Theta(1)$ work.
- So entire $i$-loop does $\Theta(n)$ work.
- So, `findMinSpanningPair` does $\Theta(n \log n)$ work.

#### Time complexity (unit cost)

- $j$-loop performs at most **eight** iterations.
- Each does $\Theta(1)$ work, so entire $j$-loop does $\Theta(1)$ work.
- So entire $i$-loop does $\Theta(n)$ work.
- So, `findMinSpanningPair` does $\Theta(n \log n)$ work.

#### Time complexity (unit cost)

- $j$-loop performs at most **eight** iterations.
- Each does $\Theta(1)$ work, so entire $j$-loop does $\Theta(1)$ work.
- So entire $i$-loop does $\Theta(n)$ work.
- So, `findMinSpanningPair` does $\Theta(n \log n)$ work.
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
- Assume for simplicity that x-coordinates are unique

**Shamos’ algorithm (1975)**

1. ShamosClosestPair($P$, $\infty$)
2. $P_x = \text{sort}(P)$ by increasing x values
3. $P_y = \text{sort}(P)$ by increasing y values
4. Recurse($P_x(V)$, $P_y(V)$)
5. // divide & conquer
6. $P_{left} = P_x(\lfloor V/2 \rfloor)$ // $x \leq \text{mid}$
7. $P_{right} = P_x(\lceil V/2 \rceil)$ // $x > \text{mid}$
8. if $n < 4$ then return BruteForce($P_x(V)$)
9. $Py_{left} = \text{select~} p \in P_y$ where $x > \text{mid}$
10. $Py_{right} = \text{select~} p \in P_y$ where $x \leq \text{mid}$
11. $pair_x = \text{Recurse}(P_{left}, Py_{left})$
12. $pair_y = \text{Recurse}(P_{right}, Py_{right})$
13. // combine
14. $B = \min(\text{dist}(pair_x), \text{dist}(pair_y))$
15. $pair = \text{findMinSpanningPair}(B, P_y(V), \text{select\ wid})$
16. return $\text{minDistPair(pair, pair, pair)}$

**Time complexity**

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

Merge sort recurrence, $T(n) = O(n \log n)$

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

**GREEDY ALGORITHMS**
Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.
Problem Instance: Input for the specified problem.
Problem Constraints: Requirements that must be satisfied by any feasible solution.
Feasible Solution: For any problem instance \( f \), feasible \( f \) is the set of all outputs (i.e., solutions) for the instance \( f \) that satisfy the given constraints.
Objective Function: A function \( f : \text{feasible}(f) \to \mathbb{R}^+ \cup \{0\} \). We often think of \( f \) as being a profit or a cost function.
Optimal Solution: A feasible solution \( X \in \text{feasible}(f) \) such that the profit \( f(X) \) is maximized (or the cost \( f(X) \) is minimized).

SOLVING OPTIMIZATION PROBLEMS

- Lots of techniques
- We will study greedy approaches first
- Later, dynamic programming
  Sort of like divide and conquer
  but can sometimes be much more efficient than D&C

Greedy algorithms are usually
  Very fast, but hard to prove optimality for
  Structured as follows...

The Greedy Method

partial solutions
Given a problem instance \( f \), it should be possible to write a feasible solution \( X \) as a tuple \( [x_1, x_2, \ldots, x_n] \) for some integer \( n \), where \( x_i \in X \) for all \( i \). A tuple \( [x_1, \ldots, x_i] \) where \( i \leq n \) is a partial solution if no constraints are violated.
Note: It may be the case that a partial solution cannot be extended to a feasible solution.

choice set
For a partial solution \( X = [x_1, \ldots, x_i] \) where \( i < n \), we define the choice set

\[ \text{choice}(X) = \{ y \in X : [x_1, x_2, \ldots, x_i, y] \text{ is a partial solution} \} \]

CORE CHARACTERISTICS OF GREEDY ALGORITHMS

Greedy algorithms do no looking ahead and no backtracking.
Greedy algorithms can usually be implemented efficiently. Often they consist of a preprocessing step based on the function \( g \), followed by a single pass through the data.
In a greedy algorithm, only one feasible solution is constructed.
The execution of a greedy algorithm is based on local criteria (i.e., the values of the function \( g \)).
Correctness: For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated.

The Greedy Method (cont.)

local evaluation criterion
For any \( y \in X \), \( g(y) \) is a local evaluation criterion that measures the cost or profit of including \( y \) in a (partial) solution.

extension
Given a partial solution \( X = [x_1, \ldots, x_i] \) where \( i < n \), choose \( y \in \text{choice}(X) \) so that \( g(y) \) is as small (or large) as possible. Update \( X \) to be the \((i + 1)\)-tuple \([x_1, \ldots, x_i, y]\).

greedy algorithm
Starting with the “empty” partial solution, repeatedly extend it until a feasible solution \( X \) is constructed. This feasible solution may or may not be optimal.

Interval Selection

PROBLEM: INTERVAL SELECTION

95% CONFIDENCE INTERVAL?

WHY NOT 100% CONFIDENCE?
PROBLEM: INTERVAL SELECTION

- **Input:** a set $A = \{A_1, \ldots, A_n\}$ of time intervals
- Each interval $A_i$ has a start time $s_i$ and a finish time $f_i$
- **Feasible solution:** a subset $X$ of $A$ containing pairwise disjoint intervals
- **Output:** a feasible solution of maximum size
  - i.e., one that maximizes $|X|

Where $s_i$ and $f_i$ are positive integers

POSSIBLE GREEDY STRATEGIES

- **Partial solutions**
  - $X = \{x_1, x_2, \ldots, x_i\}$ where each $x_i$ is an interval for the output
- **Choices**
  - $X = A$ (i.e., all intervals)
  - Choice $X = \{y \in X : x_1, \ldots, x_i, y$ respects all constraints $\}$
    - i.e., where $y \notin X$ and all intervals $(y, x)$
- **Local evaluation function**
  - $g(y) = s_j$ where $y = A[j]$
    - (i.e., $g(y) =$ start time of interval $y$)

POSSIBLE GREEDY STRATEGIES
FOR INTERVAL SELECTION

- 1. Sort the intervals in increasing order of starting times. At any stage, choose the earliest starting interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $s_i$).
- 2. Sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).
- 3. Sort the intervals in increasing order of finishing times. At any stage, choose the earliest finishing interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i$).

Does one of these strategies yield a correct greedy algorithm?

HOW ABOUT STRATEGY 2?

- **Strategy 2**
  Sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).

Consider input: $(0, 5), [6, 10], [4, 7]$. We will show that Strategy 3 (sort in increasing order of finishing times) always yields the optimal solution.

STRATEGY 1: PROVING INCORRECTNESS

-Idea: find one input for which the algorithm gives a non-optimal solution or an infeasible solution

Consider input: $(0, 10), [1, 3], [5, 7]$. Strategy 1

Sort the intervals in increasing order of starting times. At any stage, choose the earliest starting interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $s_i$).

STRATEGY 3

Consider input: $4, 5, 6, 7$

Where is our local evaluation function $g$ in this code?
STRATEGY 3

Time complexity:
Sort + one pass
∈ Θ(n log n)

How to prove this is correct?
(I.e., how can we show the returned solution is both feasible and optimal?)

Feasibility? Easy!
We always choose an interval that starts after all other chosen intervals end

Optimality? Harder...

GREEDY CORRECTNESS PROOFS

Want to prove: greedy solution X is correct (feasible & optimal)

Usually show feasibility directly and optimality by contradiction:

• Suppose solution O is better than X
• Show this necessarily leads to a contradiction

Two broad strategies for deriving this contradiction:

1. Greedy stays ahead: show every choice in X is "at least as good" as the corresponding choice in O
2. Exchange: show O can be improved by replacing some choice in O with a choice in X

Let's demonstrate approach #1 (next time)