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
◆ \( \text{dist}(p,q) = (p.x - q.x)^2 + (p.y - q.y)^2 \)

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 \( \delta = \min (\text{dist(pair}_1), \text{dist(pair}_2)) \)

◆ Then \( \text{pair}_i \) (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 \(\text{dist} > \delta\) 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, \(s\).
- Has to be on the opposite side & can’t be > \(\delta\) higher than p on y axis.

Why? 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 $2\delta \times \delta$ 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

```python
def ClosestPair(P, n)
    if n < 4
        return closest
    else
        divide & conquer
        L = P[0 : n/2]
        R = P[n/2 : n]
        (pair1, dist1) = ClosestPair(L, n/2)
        (pair2, dist2) = ClosestPair(R, n/2)
        return min(pair1, dist1, pair2, dist2)
    else
        return closest
```

Return minDistance(pair1, pair2, dist1, dist2)
Claim: inner loop performs $O(1)$ iterations!

Points in $S$:

\[ \Theta(n) \]

Time complexity?

For a particular $i$, how many $j$ iterations occur?

For $i = 1, \ldots, |S|$, for $j = (i+1), \ldots, |S|$ if $S[j] \notin S[i] \cup S[i+1]$ then break

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

Q: How many points can be in the $2\delta \times \delta$ rectangle?

A: As many as in the left $\delta \times \delta$ square + right $\delta \times \delta$ square.

Can only contain eight points! (technically six)

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).

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$

$\delta \delta$
IMPROVING THE PREVIOUS ALGORITHM

- Sorting by $y$-values causes $\text{findMinSpanningPair}$ 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)

- This selection step preserves the $y$-sort order.
- Observe $P_{xL}$ and $P_{yL}$ contain the same points (specifically the points with $x \leq x_{\text{mid}}$).
- Moreover $P_{xL}$ is sorted by $x$ while $P_{yL}$ is sorted by $y$.
- No need to sort in $\text{Recurs}$.

Time complexity

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

Merge sort recurrence. $T(n) \in \Theta(n \log n)$.

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

GREEDY ALGORITHMS
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 $I$, it should be possible to write a feasible solution $X$ as a tuple $[x_1, x_2, ..., x_n]$ for some integer $n$, where $x_i \in X$ for all $i$. A tuple $[x_1, ..., x_k]$ where $k \leq n$ is a **partial solution** if no constraints are violated.

**choice set**

For a partial solution $X = [x_1, ..., x_k]$ where $k < n$, we define the choice set

$\text{choice}(X) = \{ y \in X : [x_1, ..., x_k, y] \text{ is a partial solution} \}$.

The Greedy Method (cont.)

**local evaluation criterion**

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

**extension**

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

**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.

**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!
**PROBLEM: INTERVAL SELECTION**

- **Input:** a set $A = \{ A_1, ..., A_n \}$ 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 $|X|$

**POSSIBLE GREEDY STRATEGIES**

- **Partial solutions**
  - $X = \{ x_1, x_2, ..., x_i \}$ where each $x_i$ is an interval for the output
- **Choices**
  - $X = A$ [i.e., all intervals]
  - $X = \{ y \in X : \{ x_1, ..., x_i, y \} \text{ respects all constraints} \}$
    - i.e., where $y \notin X$ and $X \cup \{ y \}$ is feasible
- **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 finish time. At any stage, choose the interval of minimum finish time 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?

**STRATEGY 1: PROVING INCORRECTNESS**

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

**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: $[0, 10), [1, 3), [3, 7)$.

We will show that Strategy 3 (sort in increasing order of finishing times) always yields the optimal solution.
STRATEGY 3

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)