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
We call this a spanning pair

Observation 1
◆ Let δ = min (dist(pair), dist(pair₉))

How to efficiently compute the minimum spanning pair?

Then pair, (if closest globally) lies in the above 2δ-wide green strip

Q: Why?
Example for Observation 1

Q: Can p be part of a globally closest spanning pair?
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.
◆ 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 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...

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

```c
ClosestPair(P[1..n])
1. sort(P) by x values
2. Recurse(P)
3. Recurse(P[1..n3]) // precondition: P sorted by x
4. if n < 4 then compare all pairs and return closest
5. // divide & conquer
6. pairL = Recurse(P[1..n/3])
7. pairR = Recurse(P[n/3+1..n])
8. // combine
9. pair = FindNonSpanningPair(pairL, pairR)
10. return min(|pairL| - dist(pairL)|, |pairR| - dist(pairR)|, dist(pair))
```
**Claim:** inner loop performs $O(1)$ iterations!

### Time complexity

```
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

### 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).

Can only contain eight points! (technically six)

### Observation

If $|S| < 2$, return $-\infty$, $-\infty$, $(\infty, \infty)$

**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.

- $T'(n)$: `ClosestPair(P[1..n])`
- $T(n)$: `Recurse(P[1..n])`
- $T'(n) = \Theta(n \log n) + T(n)$
- $T(n) = 2T(\lceil \frac{n}{2} \rceil) + \Theta(n \log n)$

Lec2 notes using recursion trees showed

- $T(n) \in \Theta(n \log^2 n)$
- $T'(n) \in \Theta(n \log^3 n)$
- So $T'(n) \in \Theta(n \log^3 n)$
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)**

```
ShamosClosestPair(P[0..n])
1. Px = sort(P) by increasing x values
2. Py = sort(P) by increasing y values
3. Recurse(Px[1..n], Py[1..n])

// divide & conquer
4. Split P into P1 and P2
5. S1 = findMinSpanningPair(Px, Py)
6. S2 = findMinSpanningPair(P1, P2)
7. return min(S1, S2)
```

**Time complexity**

$T(n) = 2T(n/2) + \Theta(n)$

Merge sort recurrence...

So runtime for Shamos' algorithm is in $O(n \log n)$.
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 solution:

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 \( i < n \) is a partial solution if no constraints are violated.

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).

The Greedy (cont.)

Local evaluation criterion:

For any \( y \in X \), \( \phi(y) \) 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 \( y \in \text{choose}(X) \) so that \( \phi(y) \) 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.

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 \( A = \{A_1, \ldots, A_n\} \) 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

- **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 \( s_i \)).
- **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 \( f_i - s_i \)).
- **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?

POSSIBLE GREEDY STRATEGIES FOR INTERVAL SELECTION

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

HOW ABOUT STRATEGY 2?

**Strategy 2**

- **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

**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 \( s_i \)).

Consider input: \([0, 10], [1, 3], [5, 7] \)

**Strategy 3**

- **Consider input:** \([4, 10], [3, 7] \)

Where is our local evaluation function \( g \) in this code?
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

We give an induction proof.
Let $X$ be the greedy solution,
$X = (A_{i1}, \ldots, A_{ik})$.

Optimal solution is $O = (A_{j1}, \ldots, A_{jl})$.

Recall
Greedy solution is $X = (A_{i1}, \ldots, A_{ik})$.
Optimal solution is $O = (A_{j1}, \ldots, A_{jl})$.

Now we complete the proof.
From the Lemma, we have $f_{ik} \leq f_{jk}$.
Suppose that $k > i$.

To obtain a contradiction:

Correctness Proof (cont.)

We are merely imagining reordering the intervals chosen by the optimal algorithm so we can easily compare their finish times to intervals in $X$.

CRUCIAL: We are NOT assuming the optimal algorithm uses the same sort order!
A DIFFERENT PROOF

"Slick" ad-hoc approaches are sometimes possible...

Let $F = \{f_i : i \leq n\}$ be the finishing times of the intervals in $X$.

No interval finishes strictly to the left of $f_i$.

So, in addition to the intervals in $X$, only the following types of intervals are possible:

- Contains $f_i$.
- Contains $f_i$ and $f_j$.
- Contains $f_i$ and $f_j$.
- Contains $f_i$.

Thus, every interval contains some finishing time in $F$.

And, two intervals cannot contain the same element of $F$.

So, there must be as many finishing times in $F$ as there are intervals in $X$. QED.