CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca
THIS TIME

• Finishing closest pair of points in 2D
• Two loop analysis examples
• Starting **greedy algorithm** design paradigm
  • Interval selection problem
  • Greedy solution
  • Correctness (optimality) proof
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
Can we Divide & Conquer?

- Like non-dominated points: sort by x-axis & divide in half

Recursive: get closest pair on the left

Recursive: get closest pair on the right

What if closest pair spans both sides?
**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\(_S\))
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?
Example for Observation 1

Q: Can $p_5$ be part of a globally closest pair $s$?
A: No. *Everything in R has dist $> \delta$ to $p_5$. And we already have a solution with dist $= \delta$. 
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point $p$ in the strip
2. Search the $2\delta \times \delta$ rectangle above the point $p$
3. Repeat for the next lowest y-valued point...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point $p$ in the strip
2. Search the $2\delta \times \delta$ rectangle above the point $p$
3. Repeat for the next lowest y-valued point...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point \( p \) in the strip
2. Search the \( 2\delta \times \delta \) rectangle above the point \( p \)
3. Repeat for the next lowest y-valued point...
1. Start from lowest y valued point in the strip
2. Search the \(2\delta \times \delta\) rectangle above the point
3. Repeat for the next lowest y-valued point...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point $p$ in the strip
2. Search the $2\delta \times \delta$ rectangle above the point $p$
3. Repeat for the next lowest y-valued point...
**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])
   δ = min(dist(pair\_L, pair\_R))
   pair\_s = findMinSpanningPair(δ, P)
   return min(pair\_L, pair\_R, pair\_s)
procedure findMinSpanningPair (δ, P):
    S = select each p in P s.t \(|p_{n/2}.x-p.x| \leq \delta\) \(\rightarrow O(n)\)
    sort(S by increasing y values) \(\rightarrow O(n \log n)\)
    minDist = +∞
    minPair = null;
    for i = 1 to S.length: \(\rightarrow O(n)\)
        j = i+1 \(\text{(compare } S[i] \text{ to points above it)}\)
        while \(|S[j].y - S[i].y| \leq \delta\):
            if \(\text{dist}(S[i], S[j]) < \text{minDist}\):
                minPair = (S[i], S[j]);
                minDist = \(\text{dist}(S[i], S[j])\)
            j++;
    return minPair

Q: How many times does the while loop body execute?
For a point $p$, how many times does while loop execute?

Observation: 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.
# 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

Why?

Because $\delta$ is the smallest distance between any pair of points that are both in L, or both in R.
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 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)  \( \rightarrow O(n \log n) \)
    minDist = +∞
    minPair = null;
    for i = 1 to S.length:  \( \rightarrow O(n) \)
        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; \( \rightarrow O(1) \)
   pair\(_L\) = DC-CP1(P[1,...,n/2]) \( \rightarrow T(n/2) \)
   pair\(_R\) = DC-CP1(P[n/2+1,...,n]) \( \rightarrow T(n/2) \)
   \( \delta = \min(\text{dist}(\text{pair}\_L, \text{pair}\_R)) \) \( \rightarrow O(1) \)
   pair\(_s\) = findMinSpanningPair(\( \delta \), P) \( \rightarrow O(n \log n) \)
   return min(pair\(_L\), pair\(_R\), pair\(_s\)) \( \rightarrow O(1) \)

Cost of pre-sorting by x-coords: \( O(n \log n) \)
Cost of DC-CP1 calls: \( T(n) = 2T(n/2) + O(n \log n) \)

Exercise: Show by induction or recursion tree that \( T(n) \) is \( O(n \log^2 n) \).

Then total cost is \( O(n \log n) + O(n \log^2 n) = O(n \log^2 n) \).
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:

\[\text{procedure findMinSpanningPair}(\delta, P):\]
\[\text{S = select each } p \text{ in } P \text{ s.t } |P[n/2].x - p.x| \leq \delta\]
\[\text{sort}(S \text{ by increasing } y \text{ values}) \rightarrow O(n \log n)\]
\[\minDist = +\infty\]
\[\minPair = \text{null};\]
\[\text{for } i = 1 \text{ to } S.\text{length}: \rightarrow O(n)\]
\[j = i+1\]
\[\text{while } (|S[j].y - S[i].y| \leq \delta):\]
\[\text{if } (\text{dist}(S[i], S[j]) < \minDist):\]
\[\minPair = (S[i], S[j])\]
\[j++;\]
\[\text{return } \minPair\]
Shamos’ DC Algorithm (1975) (1)

**procedure** Algorithm(P of n points):

- \( P_x = \) sort \( P \) by \( x \) values in increasing order
- \( P_y = \) sort \( P \) by \( y \) values in increasing order
- DC-Shamos(\( P_x, P_y \))

**procedure** DC-Shamos(\( P_x, P_y \)):

- if \( (P_x$.size \leq 3) \) ...;
- \( P_{yL} = \) select from \( P_y \) points with \( x \leq P_x[n/2].x \)
- \( P_{yR} = \) select from \( P_y \) points with \( x > P_x[n/2].x \)
- \( pair_{L} = \) DC-Shamos(\( P_x[1,...,n/2], P_{yL} \))
- \( pair_{R} = \) DC-Shamos(\( P_x[n/2+1,...,n], P_{yR} \))
- \( \delta = \min(\text{dist}(pair_{L}, pair_{R})) \)
- \( pair_{s} = \) findMinSpanningPairShamos(\( \delta, P_x, P_y \))
- return \( \min(pair_{L}, pair_{R}, pair_{s}) \)

Sorted by \( y \) already!
Shamos’ DC Algorithm (1975) (2)

Don’t need to sort by $y$!

```
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 ($|S[j].y - S[i].y| \leq \delta$):
            if (dist($S[i]$, $S[j]$) < minDist):
                minPair = ($S[i]$, $S[j]$)
        $j++$;
    return minPair
```

Sorted by $y$ already!

After this point, code is really the same as the earlier alg.

Total: $O(n)$
Runtime Analysis of Shamos’ Algorithm

Cost of pre-sorting by x-coords: $O(n \log n)$
Cost of pre-sorting by y-coords: $O(n \log n)$
Cost of DC-Shamos calls: $T(n) = 2T(n/2) + O(n)$

By Master Theorem, total: $O(n \log n)$
(Alternatively, note: recurrence is same as for merge sort → immediately get $O(n \log n)$)

Total cost = $O$(sorting) + $O$(cost of DC-Shamos calls)
= $O(n \log n) + O(n \log n) = O(n \log n)$. 
TWO LOOP ANALYSIS EXAMPLES
The **while** loop is executed $\Theta(\log i)$ times. So the overall complexity is

$$\sum_{i=1}^{n} \Theta(\log i) = \Theta \left( \sum_{i=1}^{n} \log i \right).$$

We have that

$$\sum_{i=1}^{n} \log i = \log \prod_{i=1}^{n} i = \log n!$$

and we know from an earlier slide that $\log n! \in \Theta(n \log n)$.

**Algorithm:** *LoopAnalysis3*(n : integer)

- $sum \leftarrow 0$
- for $i \leftarrow 1$ to $n$
  - $j \leftarrow i$
  - while $j \geq 1$
  -     - $sum \leftarrow sum + i/j$
  -     - $j \leftarrow \left\lfloor \frac{i}{2} \right\rfloor$
- return $sum$
Algorithm: **LoopAnalysis2** $(A: array; n: integer)$

1. $max \leftarrow 0$
2. **for** $i \leftarrow 1$ to $n$
   1. **for** $j \leftarrow i$ to $n$
      1. $sum \leftarrow 0$
      2. **for** $k \leftarrow i$ to $j$
      3. $sum \leftarrow sum + A[k]$
      4. **if** $sum > max$
         1. $max \leftarrow sum$
3. **return** $(max)$

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} \Theta(1)$$

$$= \Theta \left( \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 \right)$$

$$= \Theta \left( \sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1) \right)$$

[remainder on the blackboard]
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 $I$, $\text{feasible}(I)$ is the set of all outputs (i.e., solutions) for the instance $I$ that satisfy the given constraints.

**Objective Function:** A function $f : \text{feasible}(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of $f$ as being a profit or a cost function.

**Optimal Solution:** A feasible solution $X \in \text{feasible}(I)$ 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 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 $[x_1, x_2, \ldots, x_n]$ for some integer $n$, where $x_i \in \mathcal{X}$ for all $i$. A tuple $[x_1, \ldots, x_i]$ where $i < n$ 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

$$\text{choice}(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$
Local evaluation means we cannot consider future choices when deciding whether to include $y$ in our solution.

We irrevocably decide to include $y$ (or not). We do not reconsider.

This may or may not be a good idea...

We choose the next element to include greedily by taking the $y$ that gives the maximum local improvement.
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 $B$ of $A$ containing pairwise disjoint intervals
- **Output:** a feasible solution of maximum size
  - i.e., one that maximizes $|B|$

Where $s_i$ and $f_i$ are positive integers

Bad solution. Not optimal!
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?
PROVING INCORRECTNESS

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

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 1** does not always yield the optimal solution:

\[[0, 10), [1, 3), [5, 7)\].

Here we choose \([0, 10)\) and we’re done. But the other two intervals comprise an optimal solution.
HOW ABOUT STRATEGY 2?

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 \( f_i - s_i \)).

**Strategy 2** does not always yield the optimal solution:

\[ [0, 5), [6, 10), [4, 7) \].

Here we choose \([4, 7)\) and we’re done. But the other two intervals comprise an optimal solution.

We will show that **Strategy 3** (sort in increasing order of finishing times) always yields the optimal solution.
A CORRECT GREEDY ALGORITHM

Algorithm: \textit{GreedyIntervalSelection}(A)
renaming the intervals, by sorting if necessary, so that \( f_1 \leq \cdots \leq f_n \)
\( B \leftarrow \{A_1\} \)
\( \text{prev} \leftarrow 1 \)
\textbf{comment:} \( \text{prev} \) is the index of the last selected interval
\textbf{for} \( i \leftarrow 2 \text{ to } n \)
\hspace{1em} \textbf{if} \( s_i \geq f_{\text{prev}} \)
\hspace{2em} \textbf{do} \quad \{ \)
\hspace{3em} \text{then} \quad \{ \)
\hspace{4em} \( B \leftarrow B \cup \{A_i\} \)
\hspace{4em} (prev \leftarrow i \)
\hspace{2em} \}\)
\textbf{return} \( (B) \)

But is it optimal?

Time complexity?
- Sort + one pass
- \( O(n \log n) + O(n) \)
- Total \( O(n \log n) \)
We give an induction proof.
Let \( \mathcal{B} \) be the greedy solution,
\[
\mathcal{B} = (A_{i_1}, \ldots, A_{i_k}),
\]
where \( i_1 < \cdots < i_k \).
Let \( \mathcal{O} \) be any optimal solution,
\[
\mathcal{O} = (A_{j_1}, \ldots, A_{j_\ell}),
\]
where \( j_1 < \cdots < j_\ell \).
Observe that \( \ell \geq k \) since \( \mathcal{O} \) is optimal.
We want to prove that \( \ell = k \).
Lemma 4.2 (Greedy stays ahead)

\[ f_{i_m} \leq f_{j_m} \text{ for } m = 1, 2, \ldots. \]

**Proof.**

Initial case \( m = 1 \). We have \( f_{i_1} \leq f_{j_1} \) since the greedy algorithm begins by choosing \( i_1 = 1 \). (\( A_1 \) has the earliest finishing time.)

Induction assumption: \( f_{i_{m-1}} \leq f_{j_{m-1}} \). Consider \( A_{i_m} \) and \( A_{j_m} \). We have

\[ s_{j_m} \geq f_{j_{m-1}} \geq f_{i_{m-1}}. \]

Why?

\[ \leq f_{j_{m-1}} \text{ (by I.H.)} \]

\[ \geq f_{j_{m-1}} \text{ (since optimal solution is feasible, so intervals are disjoint)} \]

\( A_{i_m} \) has the earliest finishing time of any interval that starts after \( f_{j_{m-1}} \) finishes. Therefore \( f_{i_m} \leq f_{j_m} \). \( \square \)
Recall

Greedy solution is \( B = (A_{i_1}, \ldots, A_{i_k}) \).
Optimal solution is \( O = (A_{j_1}, \ldots, A_{j_\ell}) \).

Now we complete the proof.

From the Lemma, we have \( f_{i_k} \leq f_{j_k} \).

Suppose that \( \ell > k \).

(to obtain a contradiction)

This completes the inductive proof.

Note: induction is the standard way to prove greedy algs correct.

1. \( A_{j_{k+1}} \) starts after \( A_{j_k} \) finishes (by disjointness)

2. \( A_{i_k} \) finishes before \( A_{j_k} \) (by lemma)

3. so \( A_{j_{k+1}} \) starts after \( A_{i_k} \) finishes!

4. so \( A_{j_{k+1}} \) would be chosen by greedy!
   Contradiction!