lecture 16

Divide and Conquer I

- review mergesort (COMP 250)
- Closest pair of points

Divide and conquer:
- partition problem into independent smaller problems
  (different from dynamic programming)
- solve each of smaller problems
  (recursion)
- combine solutions

COMP 250 examples

- binary search
- merge sort ← most relevant for today's lecture
- quicksort

merge sort (list) {
  if (list.size == 1)
    return list
  else {
    partition list into two approximately equal size lists l1, l2
    l1 ← mergesort(l1)
    l2 ← mergesort(l2)
    return merge(l1, l2)
  }
}

Given two sorted lists
of size \( \frac{n}{2} \),
merge them to form
a single sorted
list of size \( n \).
This can be done
in time \( O(n) \).
Subproblems of mergesort  
(divide and conquer)

\[ t(n) = 2 \cdot t \left( \frac{n}{2} \right) + c \cdot n \]

Assume \( n \) is a power of 2.
\[
t(n) = 2 \cdot t \left( \frac{n}{2} \right) + c \cdot n
= 2 \left\{ 2 \cdot t \left( \frac{n}{4} \right) + c \cdot \frac{n}{2} \right\} + c \cdot n
= 4 \cdot t \left( \frac{n}{4} \right) + c \cdot n + c \cdot n
= 4 \left\{ 2 \cdot t \left( \frac{n}{8} \right) + c \cdot \frac{n}{4} \right\} + 2 \cdot c \cdot n
= 8 \cdot t \left( \frac{n}{8} \right) + 3 \cdot c \cdot n
\]

Thus, \( t(n) \) is \( O(n \log n) \).
Recurrents were covered in COMP 250.
If you did not learn them well enough to be able to do the above by yourself then you need to review.

e.g. see my COMP 250 lectures 13 & 14 (+ Exercises 4)

Problem: find the closest pair of points.

Minimize the distance \( d(i,j) \) where \( i \neq j \).

Solution ("brute force"):
- \( d = \infty \)
- For each \( i = 1 \) to \( n \)
  - For each \( j = i+1 \) to \( n \)
    - If \( d(i,j) < d \)
      - \( d = d(i,j) \)
      - \( \text{closest pair} = (i,j) \)
- Return closest pair.

1D version of problem \( \rightarrow O(n \log n) \)

- Sort the points e.g. mergesort \( \rightarrow O(n \log n) \)
- Check distances between successive points \( \rightarrow O(n) \).

Can we solve the 2D problem in \( O(n \log n) \)?

Solution for 2D (Shamos & Hoey 1972):

Begin by sorting points by \( x \) value, and sorting points by \( y \) value, giving two sorted arrays \( X \) and \( Y \).
I will start explaining the algorithm using $X$ only.

We'll see later how the $Y$ ordering is used.

Partition $X$ into two sets:

- $X_L$ has $\frac{N}{2}$ smallest $x$ values ('left')
- $X_R$ has $\frac{N}{2}$ largest $x$ values ('right')

Define a vertical line $q$ that separates $X_L$ and $X_R$

Find closest pair $(X)$

- Compute $X_L \times X_R$
- Find closest pair $(X_l)$ // recursive
- Find closest pair $(X_r)$ // recursive
- Find the closest pair such that one point is in $X_L$ and the other point is in $X_R$.
- Return the closest of the three pairs.

Note: this will be too slow. Later I will modify it slightly to allow speedup.

What are the subproblems?

Find closest pair is recursive.

What is the base case?

What is the recurrence?

What are the subproblems?

Find closest pair $(X)$

- If $|X| \leq 3$ then compute closest pair by brute force and return it
- Else
  - Compute $X_L \times X_R$
  - Find closest pair $(X_l)$
  - Find closest pair $(X_r)$
  - Find the closest pair such that one point is in $X_L$ and the other point is in $X_R$.
  - Return the closest of the three pairs.
What is the recurrence?

\[ t(n) \]

Find closest pair \((X)\) \{ 
  \[ C \]
  Compute \(X_L X_R\)
  \[ t(n/2) \]
  Find closest pair \((X_L)\)
  Find closest pair \((X_R)\)
  \[ ? \]
  Find the closest pair such that one point is in \(X_L\) and the other point is in \(X_R\).
  \[ c \]
  Return the closest of the three pairs.
\]

\[ t(n) = 2 \cdot t(n/2) + ? + C \]

Let the closest pair in \(X_L\) have distance \(d_L\).
Let the closest pair in \(X_R\) have distance \(d_R\).

There are the pairs returned by the two recursive calls

\[ d_L \]
\[ d_R \]
\[ \min(d_L, d_R) = \delta \]

Observation doesn't necessarily reduce the number of points we need to consider.
All points might be a distance \(< \delta\) from line \(L\).

Consider a point \(p\) that lies between the two green lines.
Is there another point between the green lines that has a \(y\) value greater than that of \(p\)
and is a distance less than \(\delta\) from \(p\)?
It is sufficient to check those points whose y values are between $y_i$ and $y_i + \delta$.

Q: How many points do we need to check (worst case)?

A: at most 7.

To see why, notice that square cells of width $\frac{\delta}{2}$ can contain at most 1 point.

We also now see why we need the points to be sorted by their y coordinate.

Find closest pair $(X, Y)$

1. Compute $X_L, X_R, Y_L, Y_R$

2. Find closest pair $(X_L, Y_L)$
3. Find closest pair $(X_R, Y_R)$
4. Find the closest pair such that one point is in $X_L$ and the other point is in $X_R$.

Use $Y$.

Return the closest of the three pairs.

More specifically (examine this at home)

middle = empty list

for $i = 1$ to $n$ // find points between if $Y[i]$ is between green lines // green lines: $O(n)$
    middle = add ($Y[i]$)

for $i = 1$ to middle.size {} // $O(n)$
    for $j = i + 1$ to $7$ // ignore out of bounds error
        $tmp = d(middle[i], middle[i + j])$
        if $tmp < \delta$
            closest pair = ($middle[i], middle[i + j]$)
            $\delta = tmp$

See exercise Q2

$t(n) = 2 \cdot t(\frac{n}{2}) + c_3n + c$

which is the same as mergesort.