Divide And Conquer

Imran Rashid

University of Washington

February 14, 2008

Lecture Outline

Lecture Outline

Lecture Outline

Lecture Outline

Outline

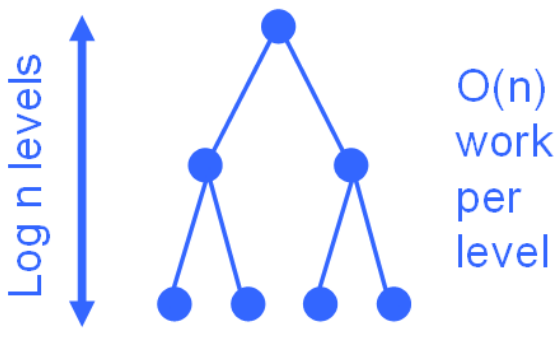
Algorithm Design Techniques

- Divide & Conquer
  - Reduce problem to one or more sub-problems of the same type
  - Typically, each sub-problem is at most a constant fraction of the size of the original problem
    - e.g. Mergesort, Binary Search, Strassen's Algorithm, Quicksort (kind of)

Outline

Mergesort — The Recurrence

- Mergesort: (recursively) sort 2 half-lists, then merge results.
- $T(n) = 2T(n/2) + cn, n > 2$
- $T(1) = 0$
- Solution: $T(n) = O(n \log n)$ (details later)



Merge Sort — Algorithm

Algorithm 1 MergeSort($A[1...n]$)

if $n = 1$ return A
$L = MergeSort(A[1...n/2])$
$U = MergeSort(A[n/2 + 1...n])$
return $Merge(U, L)$

Algorithm 2 Merge($L[1...n], U[1...n]$)

C, new array $[1...2n]$
$a = 1, b = 1$
for $i = 1$ to $2n$ **do**
    $C[i]$ = smaller of $L[a], U[b]$, and correspondingly a++ or b++
end for
return C

Going From Code to Recurrence

- Carefully define what you're counting, and write it down!
  - "Let C(n) be the number of comparisons between sort keys used by MergeSort when sorting a list of length n 1"
- In code, clearly separate base case from recursive case, highlight recursive calls, and operations being counted .
- Write Recurrence(s)

The Recurrence

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + (n-1) & \text{if } n > 1 \end{cases}$$

- Total time: proportional to T(n) (loops, copying data, parameter passing, etc.)

Why Balanced Subdivision?

- Alternative "divide & conquer" algorithm:
  1. Sort n-1
  2. Sort last 1
  3. Merge them
- $T(n) = T(n-1) + T(1) + 2n$ for $n \geq 2$
- $T(1) = 0$
- Solution: $T(n) = 2n + 2(n-1) + 2(n-2) \ldots = \Theta(n^2)$

Another D&C Approach

- Suppose we've already invented DumbSort, taking time n^2
- Try Just One Level of divide & conquer:
  - DumbSort(first $n/2$ elements)
  - DumbSort(last $n/2$ elements)
  - Merge results
- Time:$2(n/2)^2 + n = n^2/2 + n < n^2$
  - Almost twice as fast!

Another D&C Approach, cont.

- Moral 1: "two halves are better than a whole"
  - Two problems of half size are better than one full-size problem, even given the O(n) overhead of recombining, since the base algorithm has super-linear complexity.
- Moral 2: "If a little's good, then more's better"
  - two levels of D&C would be almost 4 times faster, 3 levels almost 8, etc., even though overhead is growing. Best is usually full recursion down to some small constant size (balancing "work" vs "overhead").

Another D&C Approach, cont.

- Moral 3: unbalanced division not as good:
  - $(.1n)^2 + (.9n)^2 + n = .82n^2 + n$
    - The 18% savings compounds significantly if you carry recursion to more levels, actually giving $O(nlogn)$, but with a bigger constant. So worth doing if you can't get 50-50 split, but balanced is better if you can.
    - This is intuitively why Quicksort with random splitter is good – badly unbalanced splits are rare, and not instantly fatal.
  - $(1)^2 + (n-1)^2 + n = n^2 - 2n + 2 + n$
    - Little improvement here.

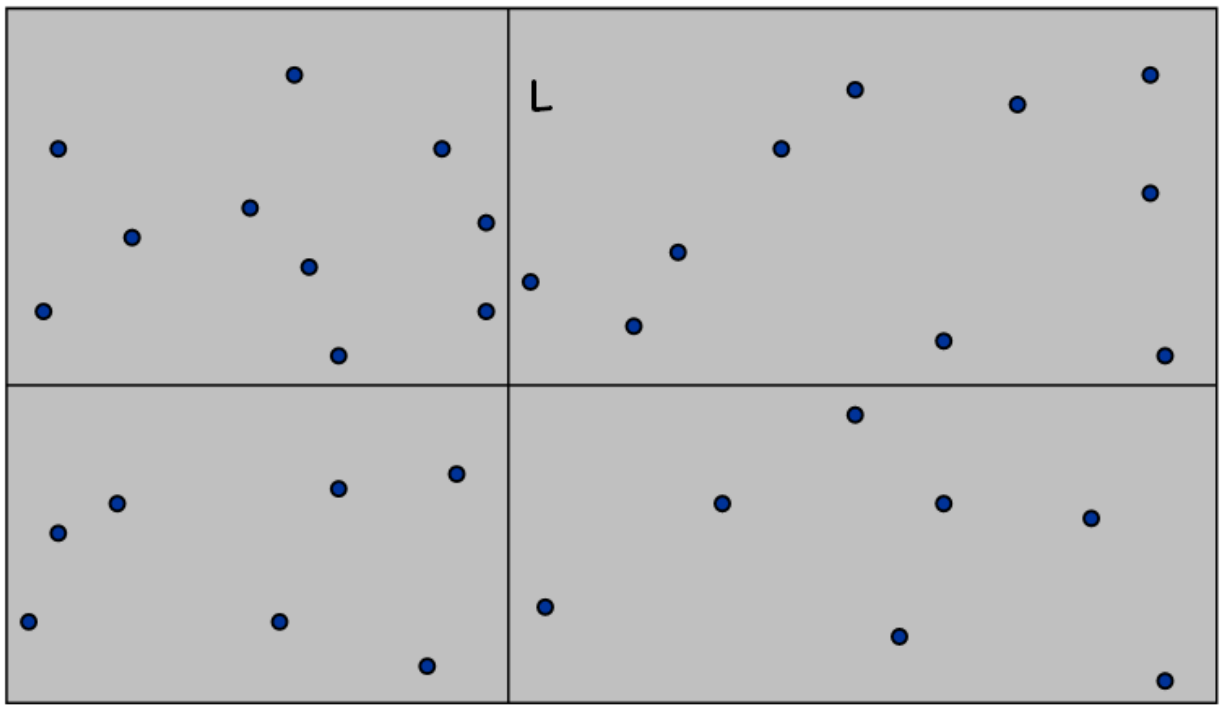
Outline

Closest Pair of Points

- Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.
- Fundamental geometric primitive.
  - Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.
- Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.
- 1-D version. $O(n \log n)$ easy if points are on a line.
- (Assumption: No two points have same x coordinate.)

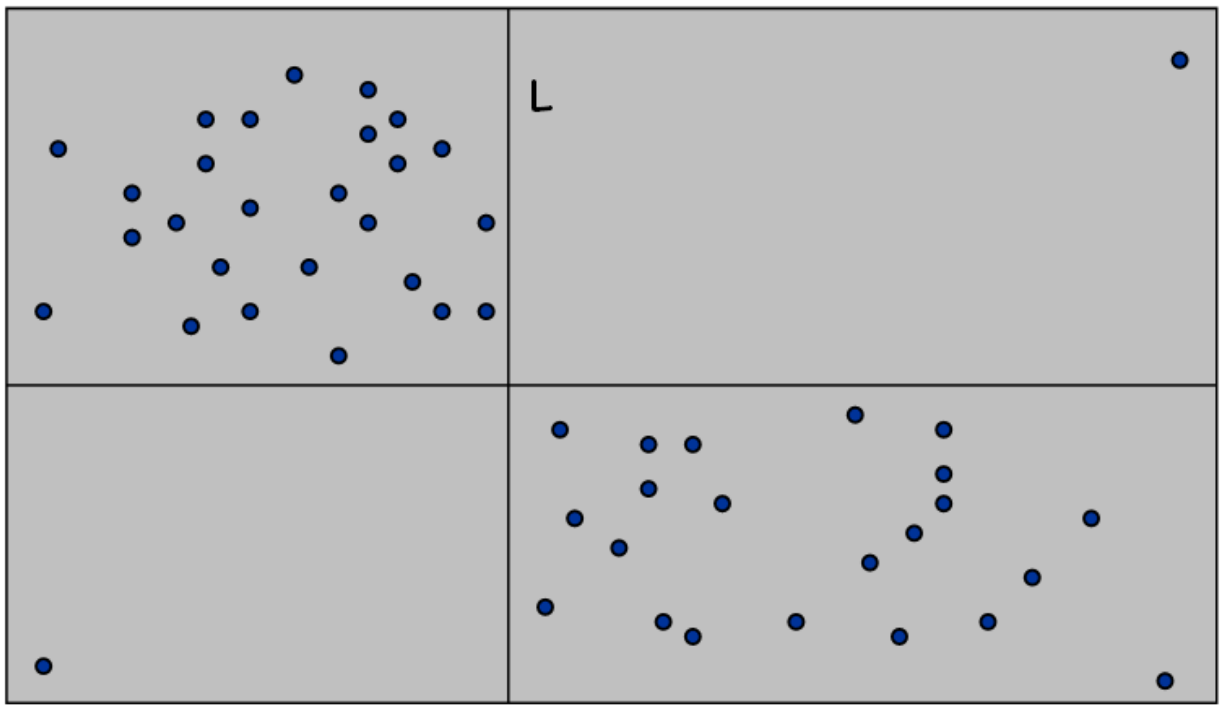
Closest Pair of Points: First Attempt

- Divide. Sub-divide region into 4 quadrants.



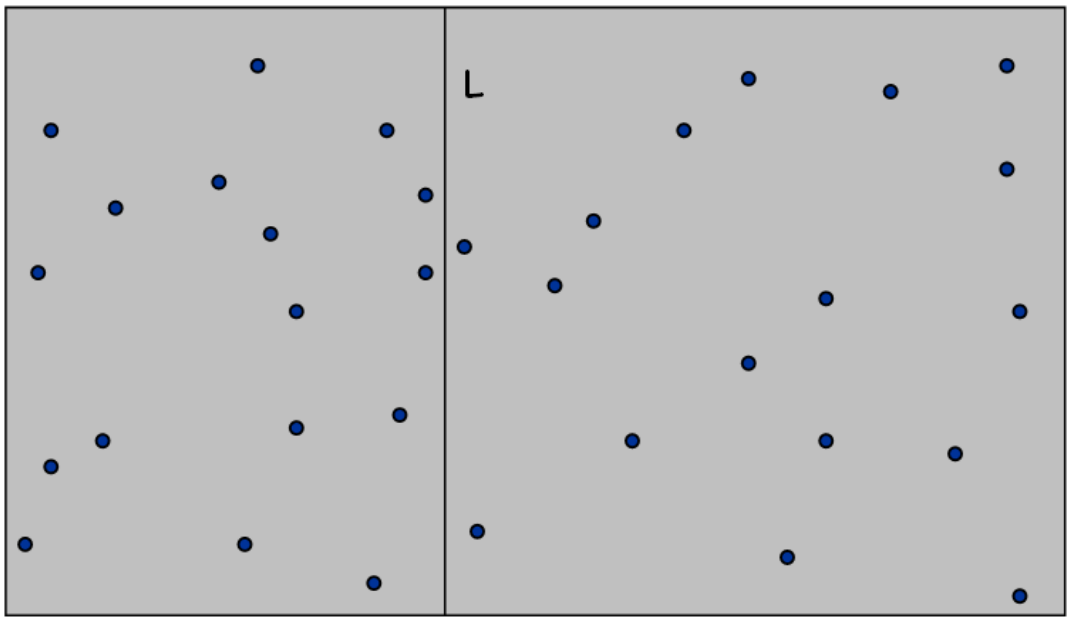
Closest Pair of Points: First Attempt

- Divide. Sub-divide region into 4 quadrants.
- Obstacle. Impossible to ensure n/4 points in each piece.



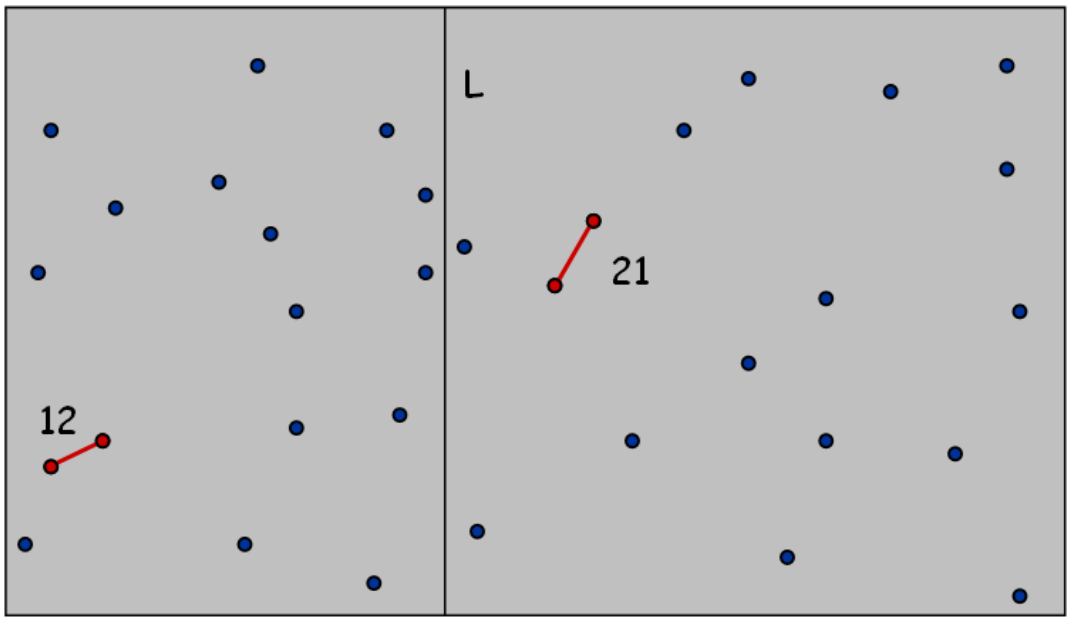
Closest Pair of Points: Correct Algorithm

- Algorithm.
  1. Divide: draw vertical line L so that roughly $n/2$ points on each side.



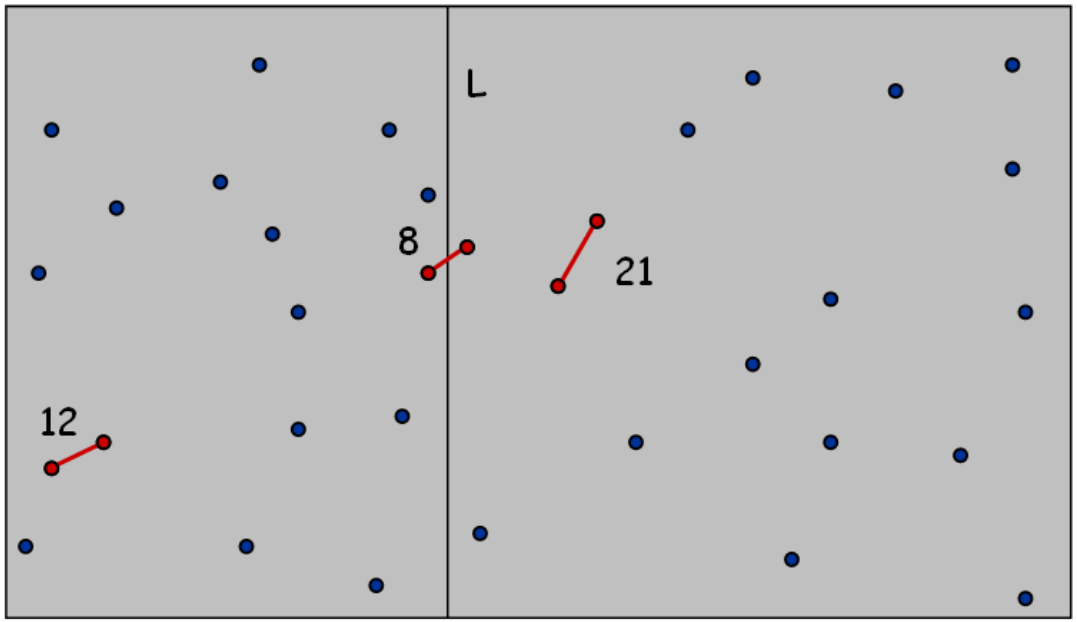
Closest Pair of Points: Correct Algorithm

- Algorithm.
  1. Divide: draw vertical line L so that roughly $n/2$ points on each side.
  2. Conquer: find closest pair in each side recursively.



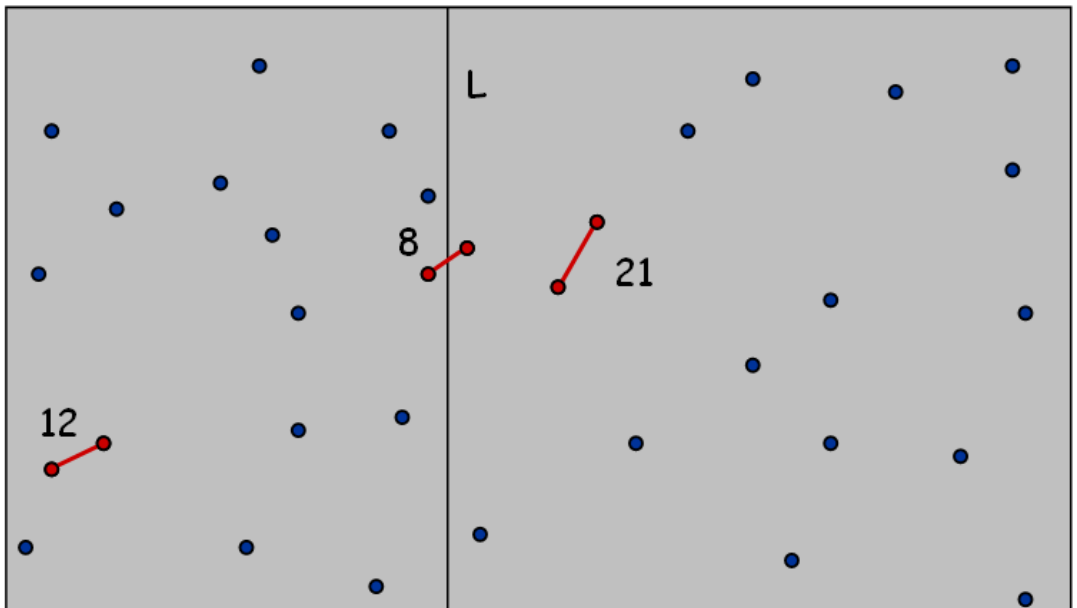
Closest Pair of Points: Correct Algorithm

- Algorithm.
  1. Divide: draw vertical line L so that roughly $n/2$ points on each side.
  2. Conquer: find closest pair in each side recursively.
  3. Combine: find closest pair with one point in each side.



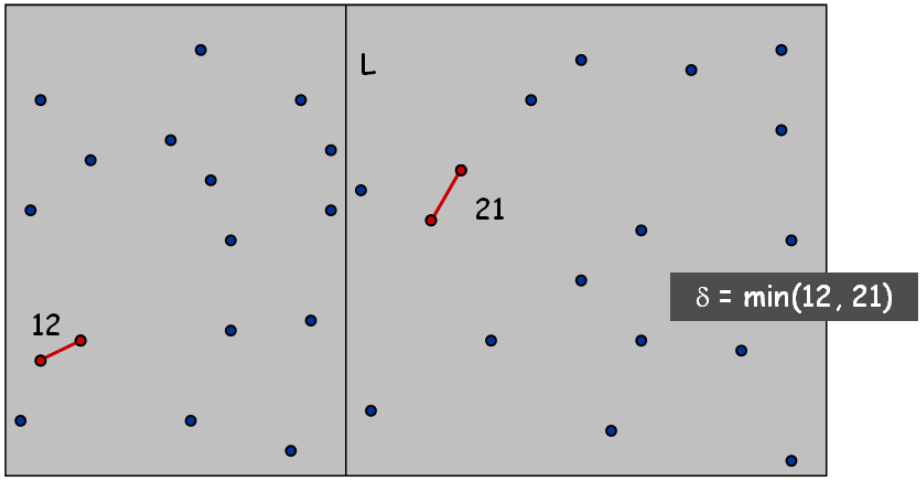
Closest Pair of Points: Correct Algorithm

- Algorithm.
  1. Divide: draw vertical line L so that roughly $n/2$ points on each side.
  2. Conquer: find closest pair in each side recursively.
  3. Combine: find closest pair with one point in each side.
  4. Return best of 3 solutions.
- But isn't the Combine step $\Theta(n^2)$!!



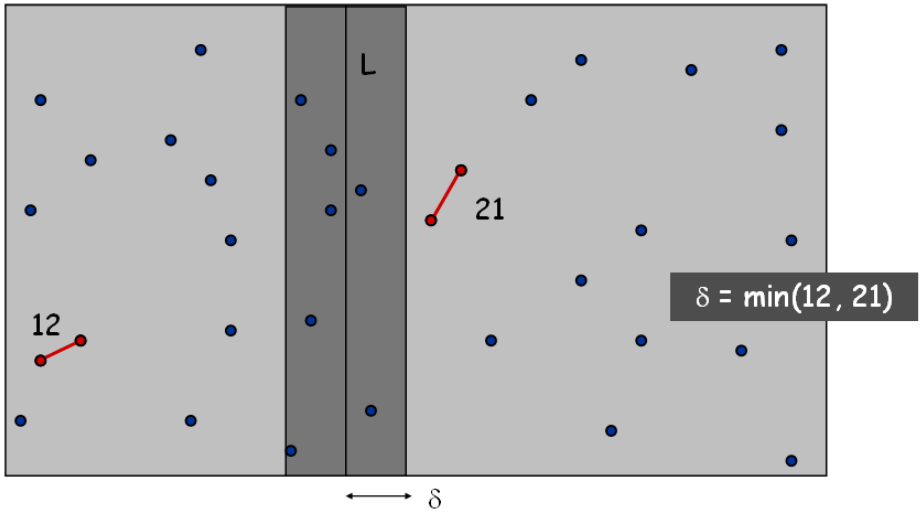
Closest Pair of Points: Combining Efficiently

- Find closest pair with one point in each side, assuming that distance $< \delta = min$(left half, right half).



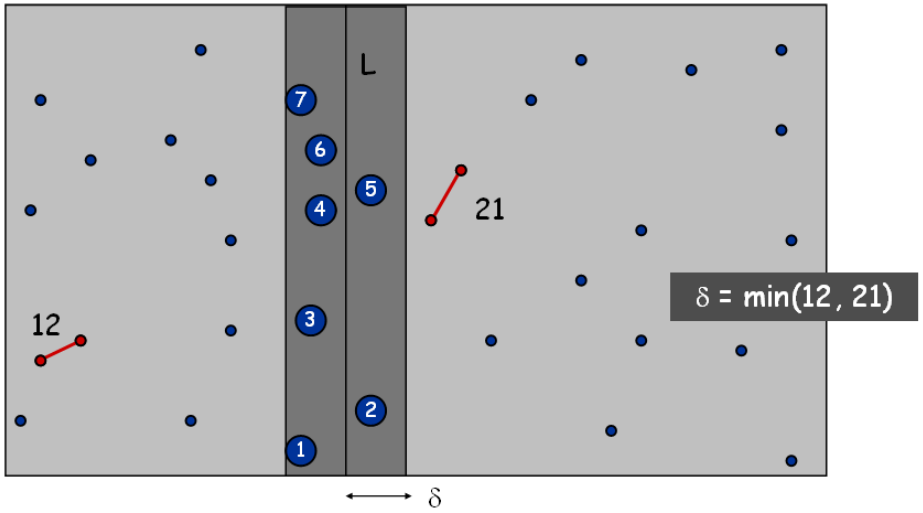
Closest Pair of Points: Combining Efficiently

- Find closest pair with one point in each side, assuming that distance $< \delta = min$(left half, right half).
  - Observation: only need to consider points within δ of line L.



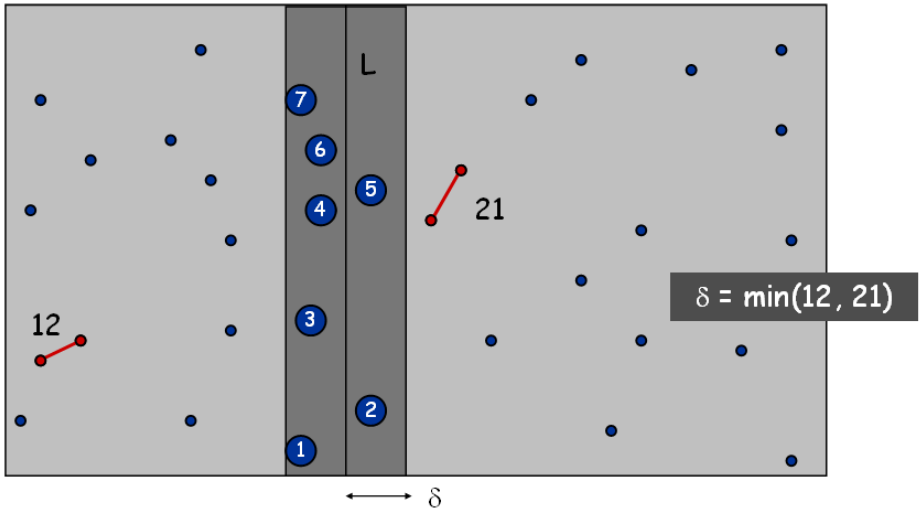
Closest Pair of Points: Combining Efficiently

- Find closest pair with one point in each side, assuming that distance $< \delta = min$(left half, right half).
  - Observation: only need to consider points within δ of line L.
  - Sort points in 2δ-strip by their y coordinate.



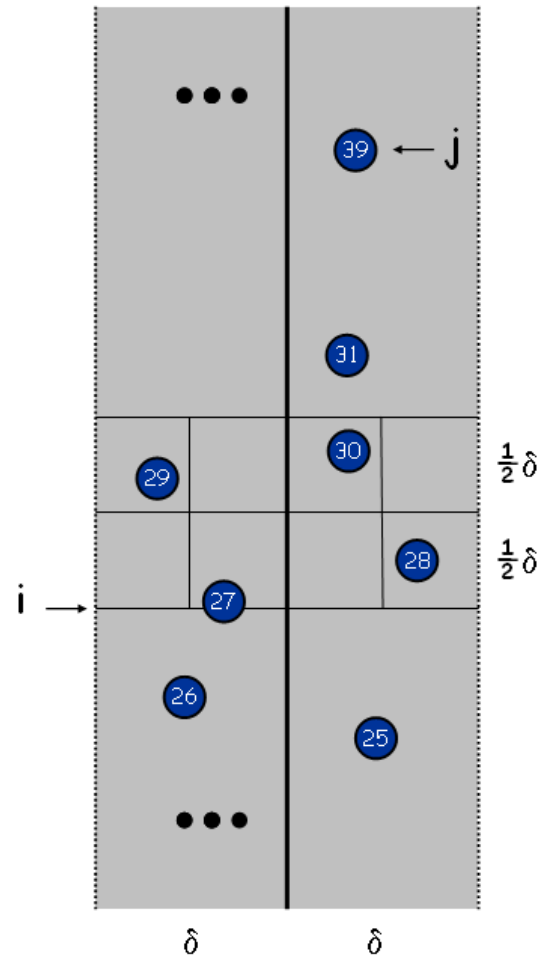
Closest Pair of Points: Combining Efficiently

- Find closest pair with one point in each side, assuming that distance $< \delta = min$(left half, right half).
  - Observation: only need to consider points within δ of line L.
  - Sort points in 2δ-strip by their y coordinate.
  - Only check distances of those within 8 positions in sorted list!



Closest Pair of Points: δ-strip

- Def. Let s_i be the point in the 2δ-strip, with the ith smallest y-coordinate.
- Claim. If $|i - j| > 8$, then the distance between s_i and s_j is at least δ.
- Pf.
  - No two points lie in same $1/2\delta$-by-$1/2\delta$ box.
  - only 8 boxes



Closest Pair Algorithm

---

if $n \leq 1$ **return** ∞
Sort points by x coordinate
Choose line L to divide points in half by x coordinate.
$\delta = min(ClosestPair(\text{left}), ClosestPair(\text{right}))$
Delete all points further than δ from L.
Sort remaining points by y coordinate, say $p[1]$ to $p[m]$
for $i = 1$ to m **do**
    $k \leftarrow 1$
    while $(i + k \leq m)$ && $(p[i + k].y < p[i].y + \delta)$ **do**
        $\delta \leftarrow min(\delta, dist(p[i], p[i + k]))$
        $k \leftarrow k + 1$
    end while
end for

Going From Code to Recurrence

- Carefully define what you're counting, and write it down!
  - "Let C(n) be the number of distance calculations made while finding the closest pair of n points"
- In code, clearly separate **base case** from **recursive case**, highlight **recursive calls,** and **operations being counted .**
- Write Recurrence(s)

Closest Pair of Points: Analysis, distance calcs

- Running time:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + 7n & \text{if } n > 1 \end{cases}$$

$$\Rightarrow T(n) = O(n \log n)$$

- BUT - that's only the number of distance calculations

Closest Pair of Points: Analysis, comparisons

- More comparisons needed, b/c of sort ...
- Running time.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n \log n & \text{if } n > 1 \end{cases}$$

$$\Rightarrow T(n) = O(n \log^2 n)$$

- Q. Can we achieve $O(n \log n)$?
- A. Yes. Don't sort points from scratch each time.
  - Sort by x at top level only.
  - Each recursive call returns δ and list of all points sorted by y
  - Sort by merging two pre-sorted lists.

Integer Arithmetic

- Add. Given two n-digit integers a and b, compute $a + b$.
  - $O(n)$ bit operations.
- Multiply. Given two n-digit integers a and b, compute $a \times b$.
  - Brute force solution: $\Theta(n^2)$ bit operations.

Divide-and-Conquer Multiplication: Warmup

- To multiply two n-digit integers:
  - Multiply four $\frac{n}{2}$-digit integers.
  - Add $\frac{n}{2}$-digit integers, and shift to obtain result.
- $$\begin{aligned} x &= 2^{n/2}x_1 + x_0 \\ y &= 2^{n/2}y_1 + y_0 \\ x \times y &= (2^{n/2}x_1 + x_0)(2^{n/2}y_1 + y_0) \\ &= 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0 \end{aligned}$$
- $$T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$$

Key trick: 2 multiplies for the price of 1:

$$
\begin{aligned}
\alpha &= x_1 + x_0 \\
\beta &= y_1 + y_0 \\
\alpha\beta &= (x_1 + x_0)(y_1 + y_0) \\
&= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
(x_1 y_0 + x_0 y_1) &= \alpha\beta - x_1 y_1 - x_0 y_0
\end{aligned}
$$

Key trick: 2 multiplies for the price of 1:

$$
\begin{aligned}
\alpha &= x_1 + x_0 \\
\beta &= y_1 + y_0 \\
\alpha\beta &= (x_1 + x_0)(y_1 + y_0) \\
&= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
(x_1 y_0 + x_0 y_1) &= \alpha\beta - x_1 y_1 - x_0 y_0
\end{aligned}
$$

$$
\begin{aligned}
x \times y &= (2^{n/2} x_1 + x_0)(2^{n/2} y_1 + y_0) \\
&= 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0 \\
&= 2^n x_1 y_1 + 2^{n/2}(\alpha\beta - x_1 y_1 - x_0 y_0) + x_0 y_0
\end{aligned}
$$

Karatsuba Multiplication

- To multiply two n-digit integers:
  - Add two $\frac{n}{2}$-digit integers.
  - Multiply three $\frac{n}{2}$-digit integers.
  - Add, subtract, and shift $\frac{n}{2}$-digit integers to obtain result.

$$A = x_1 y_1$$
$$B = (x_1 + x_0)(y_1 + y_0)$$
$$C = x_0 y_0$$
$$xy = 2^n A + 2^{n/2}(B - A - C) + C$$

- Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

$$T(n) = 3T(n/2) + cn \Rightarrow T(n) = O(n^{1.585})$$

Multiplication – The Bottom Line

- Naive: $\Theta(n^2)$
- Karatsuba: $\Theta(n^{1.59\ldots})$
- Amusing exercise: generalize Karatsuba to do 5 size $n/3$ subproblems $\Rightarrow \Theta(n^{1.46\ldots})$
- Best known: $\Theta(n \log n \log\log n)$
  - "Fast Fourier Transform"
  - but mostly unused in practice (unless you need really big numbers - a billion digits of π, say)
- High precision arithmetic IS important for crypto

Outline

Recurrences

- Where they come from, how to find them (above)
- Next: how to solve them

Mergesort (review)

- Mergesort: (recursively) sort 2 half-lists, then merge results.
- $T(n) = 2T(n/2) + cn,\ n \geq 2$
- $T(1) = 0$
- Solution: $O(n \log n)$

The Recurrence

- Total time: proportional to C(n)
- (loops, copying data, parameter passing, etc.)

Solve: $T(1) = c,\ T(n) = 2T(n/2) + cn$

- Count work at each level

level	num	size	work
0	$1 = 2^0$	n	cn
1	$2 = 2^1$	$n/2$	$2cn/2$
2	$4 = 2^2$	$n/4$	$4cn/4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	2^i	$n/2^i$	$2^i cn/2^i$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	2^{k-1}	$n/2^{k-1}$	$2^{k-1}cn/2^{k-1}$
k	2^k	$n/2^k = 1$	$2^k T(1)$

- $2^k = n \quad \Rightarrow \quad k = \log n$
- $\log n$ levels, each with $O(n)$ work, $\Rightarrow O(n \log n)$

Solve: $T(1) = c,\ T(n) = 4T(n/2) + cn$

- Count work at each level

level	num	size	work
0	$1 = 4^0$	n	cn
1	$4 = 4^1$	$n/2$	$4cn/2$
2	$16 = 4^2$	$n/4$	$4cn/4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	4^i	$n/2^i$	$4^i cn/2^i$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	4^{k-1}	$n/2^{k-1}$	$4^{k-1}cn/2^{k-1}$
k	4^k	$n/2^k = 1$	$4^k T(1)$

- $\sum_{i=0}^{k} 4^i cn/2^i = O(n^2)$

Solve: $T(1) = c,\ T(n) = 3T(n/2) + cn$

- Count work at each level

level	num	size	work
0	$1 = 3^0$	n	cn
1	$3 = 3^1$	$n/2$	$3cn/2$
2	$9 = 3^2$	$n/4$	$9cn/4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	3^i	$n/2^i$	$3^i cn/2^i$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	3^k	$n/2^k = 1$	$3^k T(1)$

- Total Work: $T(n) = \sum_{i=0}^{k} 3^i cn/2^i$

Solve: $T(1) = c,\ T(n) = 3T(n/2) + cn$ (cont.)

$$T(n) = \sum_{i=0}^{k} 3^i cn/2^i \qquad (1)$$

$$= cn \sum_{i=0}^{k} 3^i/2^i \qquad (2)$$

$$= cn \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \qquad (3)$$

$$= cn \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} \qquad (4)$$

$$= 2cn\left(\left(\frac{3}{2}\right)^{k+1} - 1\right) \qquad (5)$$

Solve: $T(1) = c,\ T(n) = 3T(n/2) + cn$ (cont.)

$$T(n) = \sum_{i=0}^{k} 3^i cn/2^i \qquad (1)$$

$$= cn \sum_{i=0}^{k} 3^i/2^i \qquad (2)$$

$$= cn \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \qquad (3)$$

$$= cn \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} \qquad (4)$$

$$= 2cn\left(\left(\frac{3}{2}\right)^{k+1} - 1\right) \qquad (5)$$

for (3) to (4):

$$\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}$$

(when $x \neq 1$)

Solve: $T(1) = c,\ T(n) = 3T(n/2) + cn$ (cont.)

$$
\begin{aligned}
T(n) &< 2cn\left(\frac{3}{2}\right)^{k+1} && (1)\\
&= 3cn\left(\frac{3}{2}\right)^{k} && (2)\\
&= 3cn\frac{3^k}{2^k} && (3)\\
&= 3cn\frac{3^{\log_2 n}}{2^{\log_2 n}} && (4)\\
&= 3c\left(3^{\log_2 n}\right) && (5)\\
&= 3c\left(n^{\log_2 3}\right) && (6)\\
&= O(n^{1.58\ldots}) && (7)
\end{aligned}
$$

Solve: $T(1) = c,\ T(n) = 3T(n/2) + cn$ (cont.)

$$
\begin{aligned}
T(n) &< 2cn\left(\frac{3}{2}\right)^{k+1} && (1)\\
&= 3cn\left(\frac{3}{2}\right)^{k} && (2)\\
&= 3cn\frac{3^k}{2^k} && (3)\\
&= 3cn\frac{3^{\log_2 n}}{2^{\log_2 n}} && (4)\\
&= 3c\left(3^{\log_2 n}\right) && (5)\\
&= 3c\left(n^{\log_2 3}\right) && (6)\\
&= O(n^{1.58\ldots}) && (7)
\end{aligned}
$$

for (5) to (6)

$$
\begin{aligned}
&= a^{\log_b n}\\
&= \left(b^{\log_b a}\right)^{\log_b n}\\
&= \left(b^{\log_b n}\right)^{\log_b a}\\
&= n^{\log_b a}
\end{aligned}
$$

Master Divide and Conquer Recurrence

- If $T(n) = aT(n/b) + cn^k$ for $n > b$ then
  - if $a > b^k$ then $T(n)$ is $\Theta(n^{\log_b a})$.
    (many subproblems $\Rightarrow$ leaves dominate)
  - if $a < b^k$ then i$T(n)$ is $\Theta(n^k)$
    (few subproblems $\Rightarrow$ top level dominates)
  - if $a = b^k$ then $T(n)$ is $\Theta(n^k log n)$
    (balanced $\Rightarrow$ all $\log n$ levels contribute)
- True even if it is $\lceil n/b \rceil$ instead of n/b.

Divide And Conquer Summary

- If base algorithm is super-linear, dividing into pieces can help. "Two halves better than a whole."
- Very carefully analyze the recurrence. Some constants matter, be careful not to miss anything.
- Solve recurrence with recursion tree or Master Recurrence

More applications

More applications of divide & Conquer in the book:

- Polynomial Multiplication
- Fast Fourier Transform
  - very useful in signal processing