# Divide And Conquer

Imran Rashid

University of Washington

February 14, 2008

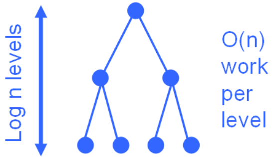
# Lecture Outline

# 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)

# Mergesort — The Recurrence

- Mergesort: (recursively) sort 2 half-lists, then merge results.



# Merge Sort — Algorithm

# 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

# 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

# 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:
  - $(.1n)^2 + (.9n)^2 + n = .82n^2 + n$
  - $(1)^2 + (n-1)^2 + n = n^2 - 2n + 2 + n$

# 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.

# Closest Pair of Points: First Attempt

- Divide. Sub-divide region into 4 quadrants.

# Closest Pair of Points: Correct Algorithm

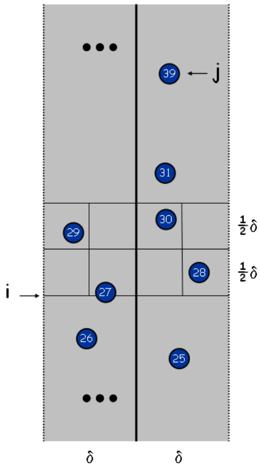
- Algorithm.
  1. Divide: draw vertical line $L$ so that roughly $n/2$ points on each side.

# Closest Pair of Points: Combining Efficiently

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

# Closest Pair of Points: $\delta$-strip

- Def. Let $s_i$ be the point in the $2\delta$-strip, with the ith smallest y-coordinate.



# Closest Pair Algorithm

```
if n ≤ 1 return ∞
Sort points by x coordinate
Choose line L to divide points in half by x coordinate.
δ = min(ClosestPair(left), ClosestPair(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 ← 1
    while (i + k ≤ m) && (p[i + k].y < p[i].y + δ) do
        δ ← min(δ, dist(p[i], p[i + k]))
        k ← 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

# Closest Pair of Points: Analysis, comparisons

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

# 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.

# Key trick: 2 multiplies for the price of 1:

# 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.

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

- 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: $\Theta(n^{1.59\cdots})$
- Amusing exercise: generalize Karatsuba to do 5 size $n/3$ subproblems $\Rightarrow \Theta(n^{1.46\cdots})$
- 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 $\pi$, say)
- High precision arithmetic IS important for crypto

# 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 |
| --- | --- | --- | --- |
| $0$ | $1 = 2^0$ | $n$ | $cn$ |
| $1$ | $2 = 2^1$ | $n/2$ | $2cn/2$ |
| $2$ | $4 = 2^2$ | $n/4$ | $4cn/4$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $i$ | $2^i$ | $n/2^i$ | $2^i cn/2^i$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $k-1$ | $2^{k-1}$ | $n/2^{k-1}$ | $2^{k-1}cn/2^{k-1}$ |
| $k$ | $2^k$ | $n/2^k = 1$ | $2^k T(1)$ |

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

- Count work at each level

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

# 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.)

# 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 $\Theta(n^k logn)$
    (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