CSE 417: Algorithms and Computational Complexity

Winter 2007
Larry Ruzzo

Divide and Conquer Algorithms
The Divide and Conquer Paradigm

Outline:

General Idea

Review of Merge Sort

Why does it work?
  Importance of balance
  Importance of super-linear growth

Two interesting applications
  Polynomial Multiplication
  Matrix Multiplication

Finding & Solving Recurrences
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 (review)

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

\[ T(n) = 2T(n/2) + cn, \ n \geq 2 \]

\[ T(1) = 0 \]

Solution: \( O(n \log n) \) (details later)
Why Balanced Subdivision?

Alternative "divide & conquer" algorithm:
Sort n-1
Sort last 1
Merge them

\[ T(n) = T(n-1) + T(1) + 3n \quad \text{for } n \geq 2 \]
\[ T(1) = 0 \]
Solution: \( 3n + 3(n-1) + 3(n-2) \ldots = \Theta(n^2) \)
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: $2 \left(\frac{n}{2}\right)^2 + n = \frac{n^2}{2} + n \ll n^2$

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 less good:

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

The 18% savings compounds significantly if you carry recursion to more levels, actually giving $O(n \log n)$, 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.
5.4 Closest Pair of Points
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.

to make presentation cleaner
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

Algorithm.

- **Divide**: draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
Closest Pair of Points

Algorithm.

- **Divide:** draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
Closest Pair of Points

**Algorithm.**
- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
- **Combine:** find closest pair with one point in each side. $\rightarrow$ seems like $O(n^2)$
- Return best of 3 solutions.
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < $\delta$. 

$\delta = \min(12, 21)$
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < $\delta$.

- Observation: only need to consider points within $\delta$ of line $L$. 

![Diagram showing close pair of points with one point in each side]
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < \( \delta \).

- Observation: only need to consider points within \( \delta \) of line \( L \).
- Sort points in \( 2\delta \)-strip by their \( y \) coordinate.

\( \delta = \min(12, 21) \)
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < $\delta$.

- Observation: only need to consider points within $\delta$ of line $L$.
- Sort points in $2\delta$-strip by their $y$ coordinate.
- Only check distances of those within 11 positions in sorted list!

$\delta = \min(12, 21)$
Def. Let $s_i$ be the point in the $2\delta$-strip, with the $i^{th}$ smallest y-coordinate.

Claim. If $|i - j| \geq 8$, then the distance between $s_i$ and $s_j$ is at least $\delta$.

Pf.

- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- only 8 boxes
Closest Pair Algorithm

Closest-Pair(p₁, ..., pₙ) {
    if(n <= ??) return ??

    **Compute** separation line L such that half the points are on one side and half on the other side.

    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = min(δ₁, δ₂)

    **Delete** all points further than δ from separation line L

    **Sort** remaining points p[1]...p[m] by y-coordinate.

    for i = 1..m
        k = 1
        while i+k <= m && p[i+k].y < p[i].y + δ
            δ = min(δ, distance between p[i] and p[i+k]);
            k++;

        return δ.
    }
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 \( n \geq 1 \)”

In code, clearly separate **base case** from **recursive case**, highlight **recursive calls**, and **operations being counted**.

Write Recurrence(s)
Closest Pair Algorithm

Closest-Pair(p_1, ..., p_n) {
  if(n <= 1) return \infty

  Compute separation line L such that half the points are on one side and half on the other side.

  \delta_1 = \text{Closest-Pair(left half)}
  \delta_2 = \text{Closest-Pair(right half)}
  \delta = \min(\delta_1, \delta_2)

  Delete all points further than \delta from separation line L

  Sort remaining points p[1]...p[m] by y-coordinate.

  for i = 1..m
    k = 1
    while i+k <= m && p[i+k].y < p[i].y + \delta
      \delta = \min(\delta, \text{distance between } p[i] \text{ and } p[i+k])
      k++;

  return \delta.
}
Closest Pair of Points: Analysis

Running time.

\[ T(n) \leq \begin{cases} 
0 & n = 1 \\
2T(n/2) + 7n & n > 1 
\end{cases} \implies T(n) = O(n \log n) \]

BUT - that’s only the number of distance calculations
Closest Pair Algorithm

```plaintext
Closest-Pair(p₁, ..., pₙ) {
    if(n <= 1) return ∞

    Compute separation line L such that half the points are on one side and half on the other side.

    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = min(δ₁, δ₂)

    Delete all points further than δ from separation line L

    Sort remaining points p[1]...p[m]

    for i = 1..m
        k = 1
        while i+k <= m && p[i+k].y < p[i].y + δ
            δ = min(δ, distance between p[i] and p[i+k]);
            k++;

    return δ.
}
```

Base Case

Recursive calls (2)

Basic operations: comparisons

0

O(n log n)

2T(n / 2)

1

O(n)

O(n log n)

O(n)

Basic operations at this recursive level
Closest Pair of Points: Analysis

Running time.

\[ T(n) \leq \begin{cases} 
0 & n = 1 \\
2T(n/2) + O(n \log n) & n > 1 
\end{cases} \Rightarrow T(n) = O(n \log^2 n) \]

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 \( \delta \) and list of all points sorted by \( y \)
   - Sort by \textit{merging} two pre-sorted lists.

\[ T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]
5.5 Integer Multiplication
**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.

---

**Add**

```
1 1 1 1 1 1 1 0 1
  1 1 0 1 0 1 0 1
+ 0 1 1 1 1 1 0 1
  1 0 1 0 1 0 0 1 0
```

**Multiply**

```
  1 1 0 1 0 1 0 1
* 0 1 1 1 1 1 0 1
  1 1 0 1 0 1 0 1
  0 0 0 0 0 0 0 0 0
  1 1 0 1 0 1 0 1 0
  1 1 0 1 0 1 0 1 0
  1 1 0 1 0 1 0 1 0
  0 0 0 0 0 0 0 0 0
  0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1
```
Divide-and-Conquer Multiplication: Warmup

To multiply two n-digit integers:
- Multiply four \( \frac{1}{2}n \)-digit integers.
- Add two \( \frac{1}{2}n \)-digit integers, and shift to obtain result.

\[
\begin{align*}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) \\
&= 2^n \cdot x_1y_1 + 2^{n/2} \cdot \left(x_1y_0 + x_0y_1\right) + x_0y_0
\end{align*}
\]

\[
T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)
\]

assumes \( n \) is a power of 2
Key trick: 2 multiplies for the price of 1:

\[
x = 2^{n/2} \cdot x_1 + x_0
\]
\[
y = 2^{n/2} \cdot y_1 + y_0
\]
\[
xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0)
\]
\[
= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0
\]

Well, ok, 4 for 3 is more accurate…

\[
\alpha = x_1 + x_0
\]
\[
\beta = y_1 + y_0
\]
\[
\alpha \beta = (x_1 + x_0)(y_1 + y_0)
\]
\[
= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0
\]
\[
(x_1 y_0 + x_0 y_1) = \alpha \beta - x_1 y_1 - x_0 y_0
\]
Karatsuba Multiplication

To multiply two \( n \)-digit integers:

- Add two \( \frac{1}{2}n \) digit integers.
- Multiply three \( \frac{1}{2}n \)-digit integers.
- Add, subtract, and shift \( \frac{1}{2}n \)-digit integers to obtain result.

\[
x = 2^{n/2} \cdot x_1 + x_0
y = 2^{n/2} \cdot y_1 + y_0
xy = 2^n \cdot x_1y_1 + 2^{n/2} \cdot (x_1y_0 + x_0y_1) + x_0y_0
\]

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

\[
T(n) \leq T\left(\left\lfloor n/2 \right\rfloor\right) + T\left(\left\lfloor n/2 \right\rfloor\right) + T\left(1 + \left\lfloor n/2 \right\rfloor\right) + \Theta(n)
\]

Sloppy version: \( T(n) \leq 3T(n/2) + O(n) \)

\[\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})\]
Multiplication – The Bottom Line

Naïve: $\Theta(n^2)$

Karatsuba: $\Theta(n^{1.59...})$

Amusing exercise: generalize Karatsuba to do 5 size $n/3$ subproblems $=>$ $\Theta(n^{1.46...})$

Best known: $\Theta(n \log n \loglog n)$

"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.

\[ T(n) = 2T(n/2) + cn, \quad n \geq 2 \]

\[ T(1) = 0 \]

Solution: \( \Theta(n \log n) \)

(details later)

Log n levels

O(n) work per level

now
Merge Sort

MS(A: array[1..n]) returns array[1..n] {
    if(n=1) return A[1];
    New U: array[1:n/2] = MS(A[1..n/2]);
    New L: array[1:n/2] = MS(A[n/2+1..n]);
    Return(Merge(U,L));
}

Merge(U,L: array[1..n]) {
    New C: array[1..2n];
    a=1; b=1;
    For i = 1 to 2n
        C[i] = “smaller of U[a], L[b] and correspondingly a++ or b++”;
    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 \( n \geq 1 \)”

In code, clearly separate base case from recursive case, highlight recursive calls, and operations being counted.

Write Recurrence(s)
Merge Sort

\[ \text{MS}(A: \text{array}[1..n]) \text{ returns } \text{array}[1..n] \{ \]
\[ \text{If}(n=1) \text{ return } A[1]; \]
\[ \text{New } L: \text{array}[1:n/2] = \text{MS}(A[1..n/2]); \]
\[ \text{New } R: \text{array}[1:n/2] = \text{MS}(A[n/2+1..n]); \]
\[ \text{Return}(\text{Merge}(L,R)); \}
\]

\[ \text{Merge}(A,B: \text{array}[1..n]) \{ \]
\[ \text{New } C: \text{array}[1..2n]; \]
\[ a=1; b=1; \]
\[ \text{For } i = 1 \text{ to } 2n \{ \]
\[ \quad C[i] = \text{“smaller of } A[a], B[b] \text{ and } a++ \text{ or } b++”; \]
\[ \text{Return } C; \}
\]
The Recurrence

\[ C(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2C\left(\frac{n}{2}\right) + (n - 1) & \text{if } n > 1 
\end{cases} \]

Base case

Recursive calls

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

One compare per element added to merged list, except the last.
Solve: $T(1) = c$

$T(n) = 2 \cdot T(n/2) + cn$

<table>
<thead>
<tr>
<th>Level</th>
<th>Num</th>
<th>Size</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$1=2^0$</td>
<td>n</td>
<td>$cn$</td>
</tr>
<tr>
<td>1</td>
<td>$2=2^1$</td>
<td>n/2</td>
<td>$2 \cdot c \cdot n/2$</td>
</tr>
<tr>
<td>2</td>
<td>$4=2^2$</td>
<td>n/4</td>
<td>$4 \cdot c \cdot n/4$</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>i</td>
<td>$2^i$</td>
<td>n/2^i</td>
<td>$2^i \cdot c \cdot n/2^i$</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>k-1</td>
<td>$2^{k-1}$</td>
<td>n/2^{k-1}</td>
<td>$2^{k-1} \cdot c \cdot n/2^{k-1}$</td>
</tr>
<tr>
<td>k</td>
<td>$2^k$</td>
<td>n/2^k=1</td>
<td>$2^k \cdot T(1)$</td>
</tr>
</tbody>
</table>

Total work: add last col
Solve: $T(1) = c$

$$T(n) = 4 \cdot T(n/2) + cn$$

<table>
<thead>
<tr>
<th>Level</th>
<th>Num</th>
<th>Size</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$4^0$</td>
<td>$n$</td>
<td>$cn$</td>
</tr>
<tr>
<td>1</td>
<td>$4^1$</td>
<td>$n/2$</td>
<td>$4 \cdot c \cdot n/2$</td>
</tr>
<tr>
<td>2</td>
<td>$4^2$</td>
<td>$n/4$</td>
<td>$16 \cdot c \cdot n/4$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>$4^i$</td>
<td>$n/2^i$</td>
<td>$4^i \cdot c \cdot n/2^i$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k-1</td>
<td>$4^{k-1}$</td>
<td>$n/2^{k-1}$</td>
<td>$4^{k-1} \cdot c \cdot n/2^{k-1}$</td>
</tr>
<tr>
<td>k</td>
<td>$4^k$</td>
<td>$n/2^k=1$</td>
<td>$4^k \cdot T(1)$</td>
</tr>
</tbody>
</table>

$$\sum_{i=0}^{k} 4^i \cdot cn / 2^i = O(n^2)$$
Solve: \( T(1) = c \)
\( T(n) = 3 \ T(n/2) + cn \)

<table>
<thead>
<tr>
<th>Level</th>
<th>Num</th>
<th>Size</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1(=3^0)</td>
<td>n</td>
<td>cn</td>
</tr>
<tr>
<td>1</td>
<td>3(=3^1)</td>
<td>n/2</td>
<td>3 \ c \ n/2</td>
</tr>
<tr>
<td>2</td>
<td>9(=3^2)</td>
<td>n/4</td>
<td>9 \ c \ n/4</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>i</td>
<td>(3^i)</td>
<td>n/2^i</td>
<td>3^i \ c \ n/2^i</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>k-1</td>
<td>(3^{k-1})</td>
<td>n/2^{k-1}</td>
<td>3^{k-1} \ c \ n/2^{k-1}</td>
</tr>
<tr>
<td>k</td>
<td>(3^k)</td>
<td>n/2^k=1</td>
<td>(3^k \ T(1))</td>
</tr>
</tbody>
</table>

\( n = 2^k \); \( k = \log_2 n \)

Total Work: \( T(n) = \sum_{i=0}^{k} 3^i \ c \ n / 2^i \)
Solve: \( T(1) = c \)
\( T(n) = 3 \ T(n/2) + cn \)  
(cont.)

\[
T(n) = \sum_{i=0}^{k} 3^i \frac{cn}{2^i}
\]

\[
= cn \sum_{i=0}^{k} \frac{3^i}{2^i}
\]

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

\[
= cn \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\left(\frac{3}{2}\right) - 1}
\]

\[
\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1} \quad (x \neq 1)
\]
Solve: \( T(1) = c \)
\( T(n) = 3 \ T(n/2) + cn \)  \( \text{(cont.)} \)

\[
= 2cn \left( \left( \frac{3}{2} \right)^{k+1} - 1 \right) \\
< 2cn \left( \frac{3}{2} \right)^{k+1} \\
= 3cn \left( \frac{3}{2} \right)^k \\
= 3cn \frac{3^k}{2^k}
\]
Solve: \( T(1) = c \)
\( T(n) = 3 \, T(n/2) + cn \) (cont.)

\[
\begin{align*}
a^{\log_b n} &= \left( b^{\log_b a} \right)^{\log_b n} \\
&= \left( b^{\log_b n} \right)^{\log_b a} \\
&= n^{\log_b a}
\end{align*}
\]
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 => leaves dominate]
- if \( a < b^k \) then \( T(n) \) is \( \Theta(n^k) \) [few subproblems => top level dominates]
- if \( a = b^k \) then \( T(n) \) is \( \Theta(n^k \log n) \) [balanced => all log n levels contribute]

True even if it is \([n/b]\) instead of \(n/b\).
Another D&C Approach, cont.

Moral 3: unbalanced division less good:

\[(.1n)^2 + (.9n)^2 + n = .82n^2 + n\]

The 18% savings compounds significantly if you carry recursion to more levels, actually giving $O(n\log n)$, 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.

In contrast:

\[(1)^2 + (n-1)^2 + n = n^2 - 2n + 2 + n\]

Little improvement here.
D & C Summary

“two halves are better than a whole”
if the base algorithm has super-linear complexity.

“If a little's good, then more's better”
repeat above, recursively

Analysis: recursion tree or Master Recurrence