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.

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

\[ T(1) = 0 \]

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

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

\[ T(n) = T(n-1) + T(1) + 3n \text{ for } n \geq 2 \]
\[ T(1) = 0 \]
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: \( 2 \left( \frac{n}{2} \right)^2 + n = \frac{n^2}{2} + n = n^2/2 + n \ll n^2 \)
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.

Fast closest pair inspired fast algorithms for these problems.

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.
- **Conquer**: find closest pair in each side recursively.
- **Combine**: find closest pair with one point in each side. Assume the size of a rectangle $\Theta(n^2)$.
  - Return best of 3 solutions.

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 \( L \).
- Sort points in 2\( \delta \)-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!

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

```c
Closest-Pair(p_1, ..., p_n) {
    if(n <= ??) return ??
    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 \( p[1..m] \) 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 \( \delta \).
}
```

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 of Points: Analysis

Running time.

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

BUT - that's only the number of distance calculations
Closest Pair of Points: Analysis

Running time.

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

\[ T(n) = O(n \log^2 n) \]

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 merging two pre-sorted lists.

\[ T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]

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 two \( \frac{n}{2} \)-digit integers, and shift to obtain result.

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

Key trick: 2 multiplies for the price of 1:

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

Karatsuba Multiplication

To multiply two n-digit integers:

- Add two \( \frac{n}{3} \)-digit integers.
- Multiply three \( \frac{n}{3} \)-digit integers.
- Add, subtract, and shift \( \frac{n}{3} \)-digit integers to obtain result.

\[
x = 2^n \cdot x_1 + x_0 \\
y = 2^n \cdot y_1 + y_0 \\
x y = 2^n \cdot x_1 y_1 + 2^{n/3} (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
= 2^n \cdot x_1 y_1 + 2^{n/3} (x_1 y_0 + x_0 y_1) - x_0 y_0 + x_0 y_0
\]

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in \( O(n^{\log_2 3}) \) bit operations.

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

"Fast Fourier Transform" but mostly unused in practice (unless you need really big numbers - a billion digits of \( n \), 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)

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

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

`Merge(A,B: array[1..n]) {
  New C: array[1..2n];
  a=1;
  b=1;
  For i = 1 to 2n {
    C[i] = "smaller of A[a], B[b] and a++ or b++";
  }
  Return C;
}`

Recursive calls
Base Case
Recursive calls
Recursive case
Operations being counted

The Recurrence

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

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

<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 c n/2</td>
</tr>
<tr>
<td>2</td>
<td>4=2^2</td>
<td>n/4</td>
<td>4 c 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 c 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} c n/2^{k-1}</td>
</tr>
<tr>
<td>k</td>
<td>2^k</td>
<td>n/2^k=1</td>
<td>2^k T(1)</td>
</tr>
</tbody>
</table>

Total work: add last col

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

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

\[ \sum_{i=0}^{k} 4^i cn/2^i = O(n^2) \]
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>...</td>
<td>...</td>
<td>...</td>
<td>...</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>...</td>
<td>...</td>
<td>...</td>
<td>...</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>

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

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

\( \text{Solve: } T(1) = c \)
\( T(n) = 3 \ T(n/2) + cn \) (cont.)

\( T(n) = \sum_{i=0}^{k} 3^i cn / 2^i \)
\( = cn \sum_{i=0}^{k} 3^i / 2^i \)
\( = cn \sum_{i=0}^{k} \left( \frac{3}{2} \right)^i \)
\( = cn \left( \frac{\left( \frac{3}{2} \right)^{k+1} - 1}{\frac{3}{2} - 1} \right) \)
\( = \frac{cn}{x} \frac{3^{k+1} - 1}{x - 1} \)
\( (x \neq 1) \)

\( \text{Solve: } T(1) = c \)
\( T(n) = 3 \ T(n/2) + cn \) (cont.)

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

\( \text{Solve: } T(1) = c \)
\( T(n) = 3 \ T(n/2) + cn \) (cont.)

\( = 3cn \frac{3^{\log_2 n}}{2^{\log_2 n}} \)
\( = 3cn \frac{3^{\log_2 n}}{n} \)
\( = 3cn \frac{3^{\log_2 n}}{n} \)
\( = 3c3^{\log_2 n} \)
\( = 3c \left( n^{\log_2 3} \right) \)
\( = O(n^{1.59...}) \)
Master Divide and Conquer Recurrence

If \( T(n) = aT(n/b) + cn^k \) for \( n > b \) then

- if \( a > b^k \) then \( T(n) = \Theta(n^{\log_b a}) \) \[many subproblems \Rightarrow leaves dominate\]
- if \( a < b^k \) then \( T(n) = \Theta(n^k) \) \[few subproblems \Rightarrow top level dominates\]
- if \( a = b^k \) then \( T(n) = \Theta(n^k \log n) \) \[balanced \Rightarrow all \log n levels contribute\]

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 \( \Theta(n \log n) \), 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