CSE 421: Algorithms and Computational Complexity

Summer 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

\[ T(n)=T(n-1)+T(1)+3n \quad \text{for } n \geq 2 \]
\[ T(1)=0 \]
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: \(2 \left(\frac{n}{2}\right)^2 + n = n^2/2 + n << n^2\)
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 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, 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 8 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

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] 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.
}

Base Case

Closest-Pair Algorithm

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

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)
Closest Pair Algorithm

`Closest-Pair(p_1, ..., p_n) {`
`if(n <= 1) return ∞;` 
`Compute separation line L such that half the points are on one side and half on the other side.`

`δ_1 = Closest-Pair(left half)` 
`δ_2 = Closest-Pair(right half)` 
`δ = min(δ_1, δ_2)` 
`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;` 
`
`}

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 δ and list of all points sorted by y.
- Sort by merging two pre-sorted lists.

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

5.5 Integer Multiplication

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 * b.
- Brute force solution: Θ(n²) bit operations.
Divide-and-Conquer Multiplication: Warmup

To multiply two n-digit integers:
- Multiply four $\frac{1}{2}$-n-digit integers.
- Add two $\frac{3}{2}$-n-digit integers, and shift to obtain result.

$$x = 2^{n/2}x_1 + x_0$$
$$y = 2^{n/2}y_1 + y_0$$
$$xy = \left(2^{n/2}x_1 + x_0\right)\left(2^{n/2}y_1 + y_0\right)$$
$$= 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$$

Karatsuba Multiplication

To multiply two n-digit integers:
- Add two $\frac{3}{2}$-n-digit integers.
- Multiply three $\frac{1}{2}$-n-digit integers.
- Add, subtract, and shift $\frac{1}{2}$-n-digit integers to obtain result.

$$x = 2^{n/2}x_1 + x_0$$
$$y = 2^{n/2}y_1 + y_0$$
$$xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$$

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

Key trick: 2 multiplies for the price of 1:

$$x = 2^{n/2}x_1 + x_0$$
$$y = 2^{n/2}y_1 + y_0$$
$$xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$$

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 => $\Theta(n^{1.46\ldots})$
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 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)

Log \( n \) levels

O(n) work per level

Merge Sort

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

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

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

Base Case

Recursive calls

Base case

Recursive calls

One compare per element added to merged list, except the last.

Total time: proportional to C(n)

(loops, copying data, parameter passing, etc.)

The Recurrence

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

Solve: \( T(1) = c \)

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

The Total work: add last col

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

Solve: \( T(1) = c \)

\[ T(n) = 4 \, T(n/2) + cn \]
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</td>
<td>3(n^0)</td>
<td>(cn)</td>
</tr>
<tr>
<td>1</td>
<td>3(n^1)</td>
<td>(n/2)</td>
<td>3 (c \ n/2)</td>
</tr>
<tr>
<td>2</td>
<td>9(n^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>

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

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

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

\[ T(n) = \sum_{i=0}^{k} 3^i \ cn / 2^i \]
\[ = cn \sum_{i=0}^{k} 3^i / 2^i \]
\[ = cn \sum_{i=0}^{k} \left( \frac{3}{2} \right)^i \]
\[ = cn \left( \frac{\left( \frac{3}{2} \right)^{k+1} - 1}{\left( \frac{3}{2} \right) - 1} \right) \]
\[ = O(n^{1.59...}) \]
**Master Divide and Conquer Recurrence**

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

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

True even if it is \( \lceil n/b \rceil \) 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

**Another Example:**

**Matrix Multiplication –**

**Strassen’s Method**
Multiplying Matrices

\[
\begin{pmatrix}
 a_{11} & a_{12} & a_{13} & a_{14} \\
 a_{21} & a_{22} & a_{23} & a_{24} \\
 a_{31} & a_{32} & a_{33} & a_{34} \\
 a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix} \cdot
\begin{pmatrix}
 h_{11} & h_{12} & h_{13} & h_{14} \\
 h_{21} & h_{22} & h_{23} & h_{24} \\
 h_{31} & h_{32} & h_{33} & h_{34} \\
 h_{41} & h_{42} & h_{43} & h_{44}
\end{pmatrix}
\]

1. \[a_{11}h_{11} + a_{12}h_{21} + a_{13}h_{31} + a_{14}h_{41} \]
2. \[a_{11}h_{12} + a_{12}h_{22} + a_{13}h_{32} + a_{14}h_{42} \]
3. \[a_{11}h_{13} + a_{12}h_{23} + a_{13}h_{33} + a_{14}h_{43} \]
4. \[a_{11}h_{14} + a_{12}h_{24} + a_{13}h_{34} + a_{14}h_{44} \]

\[a_{21}h_{11} + a_{22}h_{21} + a_{23}h_{31} + a_{24}h_{41} \]
2. \[a_{21}h_{12} + a_{22}h_{22} + a_{23}h_{32} + a_{24}h_{42} \]
3. \[a_{21}h_{13} + a_{22}h_{23} + a_{23}h_{33} + a_{24}h_{43} \]
4. \[a_{21}h_{14} + a_{22}h_{24} + a_{23}h_{34} + a_{24}h_{44} \]

\[a_{31}h_{11} + a_{32}h_{21} + a_{33}h_{31} + a_{34}h_{41} \]
2. \[a_{31}h_{12} + a_{32}h_{22} + a_{33}h_{32} + a_{34}h_{42} \]
3. \[a_{31}h_{13} + a_{32}h_{23} + a_{33}h_{33} + a_{34}h_{43} \]
4. \[a_{31}h_{14} + a_{32}h_{24} + a_{33}h_{34} + a_{34}h_{44} \]

\[a_{41}h_{11} + a_{42}h_{21} + a_{43}h_{31} + a_{44}h_{41} \]
2. \[a_{41}h_{12} + a_{42}h_{22} + a_{43}h_{32} + a_{44}h_{42} \]
3. \[a_{41}h_{13} + a_{42}h_{23} + a_{43}h_{33} + a_{44}h_{43} \]
4. \[a_{41}h_{14} + a_{42}h_{24} + a_{43}h_{34} + a_{44}h_{44} \]

\[n^3\text{ multiplications, } n^3-n^2\text{ additions}\]

Simple Matrix Multiply

for \(i = 1\) to \(n\)

for \(j = 1\) to \(n\)

\[C[i,j] = 0\]

for \(k = 1\) to \(n\)

\[C[i,j] = C[i,j] + A[i,k]*B[k,j]\]

\[n^3\text{ multiplications, } n^3-n^2\text{ additions}\]

Multiplying Matrices

\[
\begin{pmatrix}
 a_{11} & a_{12} & a_{13} & a_{14} \\
 a_{21} & a_{22} & a_{23} & a_{24} \\
 a_{31} & a_{32} & a_{33} & a_{34} \\
 a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix} \cdot
\begin{pmatrix}
 h_{11} & h_{12} & h_{13} & h_{14} \\
 h_{21} & h_{22} & h_{23} & h_{24} \\
 h_{31} & h_{32} & h_{33} & h_{34} \\
 h_{41} & h_{42} & h_{43} & h_{44}
\end{pmatrix}
\]

1. \[a_{11}h_{11} + a_{12}h_{21} + a_{13}h_{31} + a_{14}h_{41} \]
2. \[a_{11}h_{12} + a_{12}h_{22} + a_{13}h_{32} + a_{14}h_{42} \]
3. \[a_{11}h_{13} + a_{12}h_{23} + a_{13}h_{33} + a_{14}h_{43} \]
4. \[a_{11}h_{14} + a_{12}h_{24} + a_{13}h_{34} + a_{14}h_{44} \]

\[a_{21}h_{11} + a_{22}h_{21} + a_{23}h_{31} + a_{24}h_{41} \]
2. \[a_{21}h_{12} + a_{22}h_{22} + a_{23}h_{32} + a_{24}h_{42} \]
3. \[a_{21}h_{13} + a_{22}h_{23} + a_{23}h_{33} + a_{24}h_{43} \]
4. \[a_{21}h_{14} + a_{22}h_{24} + a_{23}h_{34} + a_{24}h_{44} \]

\[a_{31}h_{11} + a_{32}h_{21} + a_{33}h_{31} + a_{34}h_{41} \]
2. \[a_{31}h_{12} + a_{32}h_{22} + a_{33}h_{32} + a_{34}h_{42} \]
3. \[a_{31}h_{13} + a_{32}h_{23} + a_{33}h_{33} + a_{34}h_{43} \]
4. \[a_{31}h_{14} + a_{32}h_{24} + a_{33}h_{34} + a_{34}h_{44} \]

\[a_{41}h_{11} + a_{42}h_{21} + a_{43}h_{31} + a_{44}h_{41} \]
2. \[a_{41}h_{12} + a_{42}h_{22} + a_{43}h_{32} + a_{44}h_{42} \]
3. \[a_{41}h_{13} + a_{42}h_{23} + a_{43}h_{33} + a_{44}h_{43} \]
4. \[a_{41}h_{14} + a_{42}h_{24} + a_{43}h_{34} + a_{44}h_{44} \]

\[n^3\text{ multiplications, } n^3-n^2\text{ additions}\]
By Master Recurrence, if

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
8T(n/2) + n^2 & \text{if } n > 1 
\end{cases} \]

Multiplying Matrices

\[
\begin{bmatrix}
    a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
    b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix}
= 
\begin{bmatrix}
    a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\
a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22}
\end{bmatrix}
\]

Counting arithmetic operations:

\[ T(n) = 8T(n/2) + 4(n/2) = 8T(n/2) + n^2 \]

Multiplying Matrices

Strassen’s algorithm

Strassen’s algorithm
Multiply 2x2 matrices using 7 instead of 8 multiplications (and lots more than 4 additions)

\[ T(n) = 7T(n/2) + cn^2 \]

7 > 2^2 so \( T(n) \) is \( \Theta(n^{\log_2 7}) \) which is \( O(n^{2.81}) \)

Fastest algorithms theoretically use \( O(n^{3.76}) \) time

not practical but Strassen’s is practical provided calculations are exact and we stop recursion when matrix has size about 100 (maybe 10)
The algorithm

\[ P_1 = A_{12}(B_{11}+B_{21}) \quad P_2 = A_{21}(B_{12}+B_{22}) \]
\[ P_3 = (A_{11} - A_{12})B_{11} \quad P_4 = (A_{22} - A_{21})B_{22} \]
\[ P_5 = (A_{22} - A_{12})(B_{21} - B_{22}) \]
\[ P_6 = (A_{11} - A_{21})(B_{12} - B_{11}) \]
\[ P_7 = (A_{21} - A_{12})(B_{11} + B_{22}) \]
\[ C_{11} = P_1 + P_3 \quad C_{12} = P_2 + P_3 + P_6 - P_7 \]
\[ C_{21} = P_1 + P_4 + P_5 + P_7 \quad C_{22} = P_2 + P_4 \]