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≥2
- T(1)=0
- Solution: Θ(n log n)
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

1. 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”

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

3. Write Recurrence(s)

The Recurrence

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

Total time: proportional to C(n)
(loops, copying data, parameter passing, etc.)
Why Balanced Subdivision?

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

- \( T(n) = T(n-1) + T(1) + 3n \) 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: \( (n/2)^2 + (n/2)^2 + n = n^2/2 + n \)
- Almost twice as fast!

Another D&C Approach, cont.

- Moral 1:
  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.
Another D&C Example: Multiplying Faster

- On the first HW you analyzed our usual algorithm for multiplying numbers
  - $\Theta(n^2)$ time
- We can do better!
  - We'll describe the basic ideas by multiplying polynomials rather than integers
  - Advantage is we don't get confused by worrying about carries at first

Notes on Polynomials

- These are just formal sequences of coefficients so when we show something multiplied by $x^k$ it just means shifted $k$ places to the left – basically no work

Usual Polynomial Multiplication:

<p>| 3x^2 + 2x + 2 |</p>
<table>
<thead>
<tr>
<th>x^2 - 3x + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x^2 + 2x + 2</td>
</tr>
<tr>
<td>-9x^3 - 6x^2 - 6x</td>
</tr>
<tr>
<td>3x^4 + 2x^3 + 2x^2</td>
</tr>
<tr>
<td>3x^4 - 7x^3 - x^2 - 4x + 2</td>
</tr>
</tbody>
</table>

Polynomial Multiplication

- Given:
  - Degree $m-1$ polynomials $P$ and $Q$
  - $P = a_0 + a_1x + a_2x^2 + ... + a_{m-2}x^{m-2} + a_{m-1}x^{m-1}$
  - $Q = b_0 + b_1x + b_2x^2 + ... + b_{m-2}x^{m-2} + b_{m-1}x^{m-1}$
- Compute:
  - Degree $2m-2$ Polynomial $PQ$
  - $PQ = a_0b_0 + (a_0b_1+a_1b_0)x + (a_0b_2+a_1b_1+a_2b_0)x^2 + ... + (a_0b_{m-2}+a_{m-2}b_0+a_{m-1}b_{m-2})x^{2m-3} + a_{m-1}b_{m-1}x^{2m-2}$
- Obvious Algorithm:
  - Compute all $a_ib_j$ and collect terms
  - $\Theta(m^2)$ time

Naive Divide and Conquer

- Assume $m=2k$
  - $P = (a_0 + a_1x + a_2x^2 + ... + a_{k-2}x^{k-2} + a_{k-1}x^{k-1}) + (a_k + a_{k+1}x + ... + a_{m-2k}x^{k-2} + a_{m-1}x^{k-1})x^k$
    - $P_0 + P_1x^k$ = $P_0Q_0 + (P_1Q_0 + P_0Q_1)x^k + P_1Q_1x^{2k}$
  - 4 sub-problems of size $k=m/2$ plus linear combining
  - $T(m)=4T(m/2)+cm$
  - Solution $T(m) = O(m^2)$
Karatsuba’s Algorithm

- A better way to compute terms
  - Compute
    - $P_0Q_0$
    - $P_1Q_1$
    - $(P_0+P_1)(Q_0+Q_1)$ which is $P_0Q_0+P_0Q_1+P_1Q_0+P_1Q_1$
  - Then
    - $P_0Q_1+P_1Q_0 = (P_0+P_1)(Q_0+Q_1) - P_0Q_0 - P_1Q_1$
  - 3 sub-problems of size $m/2$ plus $O(m)$ work
    - $T(m) = 3T(m/2) + cm$
    - $T(m) = O(m^{\alpha})$ where $\alpha = \log_2 3 = 1.59...$

PolyMul($P, Q$):

```plaintext
// $P, Q$ are length $m = 2k$ vectors, with $P[i], Q[i]$ being
// the coefficient of $x^i$ in polynomials $P, Q$ respectively.
if ($m==1$) return ($P[0]*Q[0]$);

Let $P_{zero}$ be elements 0..k-1 of $P$; $P_{one}$ be elements k..m-1
$Q_{zero}$, $Q_{one}$: similar

Prod1 = PolyMul($P_{zero}$, $Q_{zero}$); // result is a (2k-1)-vector
Prod2 = PolyMul($P_{one}$, $Q_{one}$); // ditto
Pzo = $P_{zero}$ + $P_{one}$; // add corresponding elements
Qzo = $Q_{zero}$ + $Q_{one}$; // ditto

Mid = Prod3 - Prod1 - Prod2; // subtract corr. elements
R = Prod1 + Shift(Mid, m/2) + Shift(Prod2, m) // a (2m-1)-vector
Return( R );
```

Multiplication – The Bottom Line

- Polynomials
  - Naïve: $\Theta(n^2)$
  - Karatsuba: $\Theta(n^{1.59...})$
  - Best known: $\Theta(n \log n)$
    - "Fast Fourier Transform"

- Integers
  - Similar, but some ugly details re: carries, etc. gives $\Theta(n \log n \log \log n)$,
    - but mostly unused in practice

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

\[
\begin{array}{|c|c|c|}
\hline
\text{Level} & \text{Num} & \text{Size} \\ \hline
0 & 1 = 2^0 & n \phantom{\frac{1}{2}} \\ 1 & 2 = 2^1 & \frac{n}{2} \phantom{\frac{1}{2}} c \frac{n}{2} \\ 2 & 4 = 2^2 & \frac{n}{4} \phantom{\frac{1}{2}} c \frac{n}{4} \\ \vdots & \vdots & \vdots \\ i & 2^i & \frac{n}{2^i} \phantom{\frac{1}{2}} c \frac{n}{2^i} \\ \vdots & \vdots & \vdots \\ k-1 & 2^{k-1} & \frac{n}{2^{k-1}} \phantom{\frac{1}{2}} c \frac{n}{2^{k-1}} \\ k & 2^k & \frac{n}{2^k} = 1 \phantom{\frac{1}{2}} 4^k T(1) \\
\hline
\end{array}
\]

Total work: add last col

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

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

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

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

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

\[
T(n) = 3 T(n/2) + cn
\]

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.)

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

\( \Sigma_{i=0}^{k} x^i = 1 
(\text{x} \neq 1) \)

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

\[
\log_b a
= \frac{\log_a b \cdot \log_b n}{\log_a n}
\]

\[
= O(n^{1.59\ldots})
\]

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 \( \Theta(n^{\log_b a}) \)
  - if \( a < b^k \) then \( T(n) \) is \( \Theta(n^k) \)
  - if \( a = b^k \) then \( T(n) \) is \( \Theta(n^k \log n) \)
- Works even if it is \([n/b]\) instead of \(n/b\).
Another Example:
Matrix Multiplication –
Strassen’s Method

Simple Matrix Multiply
for i = 1 to n
for j = 1 to n
    C[i,j] = 0
for k = 1 to n

n^3 multiplications, n^3-n^2 additions

Multiplying Matrices

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

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

- n^3 multiplications, n^3-n^2 additions

Multiplying Matrices

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

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

- n^3 multiplications, n^3-n^2 additions
Multiplying Matrices

\begin{align*}
\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}
\end{align*}

Counting arithmetic operations:

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

Multiplying Matrices

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

By Master Recurrence, if

\[ T(n) = aT(n/b) + cn^k & \text{ and } a > b^k \text{ then} \]

\[ T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_b 8}) = \Theta(n^3) \]
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 \( T(n) = \Theta(n^{\log_2 7}) \) which is \( O(n^{2.81}) \)
- Fastest algorithms theoretically use \( O(n^{2.376}) \) 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

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

Another D&C Example: Fast exponentiation

- Power(a, n)
  - **Input**: integer n and number a
  - **Output**: \( a^n \)
- Obvious algorithm
  - \( n-1 \) multiplications
- Observation:
  - if \( n \) is even, \( n=2m \), then \( a^n = a^m \cdot a^m \)

Divide & Conquer Algorithm

- Power(a, n)
  - if \( n=0 \) then
    - return(1)
  - else
    - \( x \leftarrow \text{Power}(a, \lfloor n/2 \rfloor) \)
    - if \( n \) is even then
      - return(\( x \cdot x \))
    - else
      - return(\( a \cdot x \cdot x \))
Analysis

- Worst-case recurrence
  
  \[ T(n) = T(\lfloor n/2 \rfloor) + 2 \]

- By master theorem
  
  \[ T(n) = O(\log n) \quad (a=1, b=2, k=0) \]

- More precise analysis:
  
  \[ T(n) = \lceil \log_2 n \rceil + \# \text{ of 1's in } n \text{'s binary representation} \]

A Practical Application- RSA

- Instead of \( a^n \) want \( a^n \mod N \)
  
  \[ a^{i+j} \mod N = (a^i \mod N \cdot a^j \mod N) \mod N \]

- Same algorithm applies with each \( x \cdot y \) replaced by
  
  \[ (x \mod N) \cdot (y \mod N) \mod N \]

- In RSA cryptosystem (widely used for security)
  
  \[ \text{need } a^n \mod N \text{ where } a, n, N \text{ each typically have 1024 bits} \]

- Power: at most 2048 multiplies of 1024 bit numbers
  
  - relatively easy for modern machines
  
  - Naive algorithm: \( 2^{1024} \) multiplies

Another Example: Binary search for roots (bisection method)

- Given:
  
  - continuous function \( f \) and two points \( a \leq b \) with \( f(a) < 0 \) and \( f(b) > 0 \)

- Find:
  
  - approximation to \( c \) s.t. \( f(c) = 0 \) and \( a < c < b \)

Divide and Conquer Summary

- Powerful technique, when applicable

- Divide large problem into a few smaller problems of the same type

- Choosing subproblems of roughly equal size is usually critical

- Examples:
  
  - Merge sort, quicksort (sort of), polynomial multiplication, FFT, Strassen’s matrix multiplication algorithm, powering, binary search, root finding by bisection, …