A design paradigm

Divide and conquer:

(When) does decomposing a problem into smaller parts help?
INPUT: Two pairs of integers, (a,b), (c,d) representing complex numbers, a+ib, c+id, respectively.
OUTPUT: The pair [(ac-bd),(ad+bc)] representing the product (ac-bd) + i(ad+bc)

Naïve approach: 4 multiplications, 2 additions. Suppose a multiplication costs $1 and an addition cost a penny. The naïve algorithm costs $4.02.

Q: Can you do better?
Gauss’ idea

- \( m_1 = ac \)
- \( m_2 = bd \)
- \( A_1 = m_1 - m_2 = ac-bd \)
- \( m_3 = (a+b)(c+d) = ac + ad + bc + bd \)
- \( A_2 = m_3 - m_1 - m_2 = ad+bc \)
- **Saves 1 multiplication! Uses more additions. The cost now is $3.03.**
- This is good (saves 25% multiplications), but it leads to more dramatic asymptotic improvement elsewhere!

*(aside: look for connections to known algorithms)*

Q: How fast can you multiply two n-bit numbers?
How to multiply two n-bit numbers.

Elementary School algorithm

\( \times \)

\[
\begin{array}{c}
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\end{array}
\]

\( n^2 \)

\[
\begin{array}{c}
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\end{array}
\]

EECS 3101
Elementary School algorithm

\[ \begin{array}{c}
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\times \\
\end{array} \]

Q: Is there a faster algorithm?

A: YES! Use divide-and-conquer.
Divide and Conquer

Intuition:

• **DIVIDE** my instance to the problem into smaller instances to the same problem.
• Recursively solve them.
• **GLUE** the answers together so as to obtain the answer to your larger instance.
• Sometimes the last step may be trivial.
Multiplication of two n-bit numbers

\[
\begin{aligned}
X &= a 2^{n/2} + b \\
Y &= c 2^{n/2} + d \\
XY &= ac 2^n + (ad+bc) 2^{n/2} + bd
\end{aligned}
\]

MULT(X,Y):
If \(|X| = |Y| = 1\) then RETURN XY
Break X into a;b and Y into c;d
RETURN
\[
MULT(a,c) 2^n + (MULT(a,d) + MULT(b,c)) 2^{n/2} + MULT(b,d)
\]
Time complexity of MULT

• T(n) = time taken by MULT on two n-bit numbers
• What is T(n)? Is it \( \theta(n^2) \)?
• Hard to compute directly
• Easier to express as a recurrence relation!
• T(1) = k for some constant k
• T(n) = 4 T(n/2) + c_1 n + c_2 for some constants c_1 and c_2
• How can we get a \( \theta() \) expression for T(n)?

MULT(X,Y):

If \(|X| = |Y| = 1\) then RETURN XY

Break X into a;b and Y into c;d

RETURN

\[
\text{MULT}(a,c) 2^n + (\text{MULT}(a,d) + \text{MULT}(b,c)) 2^{n/2} + \text{MULT}(b,d)
\]
Time complexity of MULT

Make it concrete

- $T(1) = 1$
- $T(n) = 4 \cdot T(n/2) + n$

**Technique 1: Guess and verify**

$T(n) = 2n^2 - n$

Holds for $n=1$

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

$= 2n^2 - n$
Time complexity of MULT

- \( T(1) = 1 \) \& \( T(n) = 4 \ T(n/2) + n \)

**Technique 2:** Expand recursion

\[
T(n) = 4 \ T(n/2) + n \\
= 4 (4T(n/4) + n/2) + n = 4^2T(n/4) + n + 2n \\
= 4^2(4T(n/8) + n/4) + n + 2n \\
= 4^3T(n/8) + n + 2n + 4n \\
= \ldots \\
= 4^kT(1) + n + 2n + 4n + \ldots + 2^{k-1}n \text{ where } 2^k = n
\]

**GUESS**

\[
= n^2 + n \ (1 + 2 + 4 + \ldots + 2^{k-1}) \\
= n^2 + n \ (2^k-1) \\
= 2 \ n^2 - n \ [\text{NOT FASTER THAN BEFORE}]
\]
Gaussified MULT (Karatsuba 1962)

MULT(X,Y):

If |X| = |Y| = 1 then RETURN XY

Break X into a;b and Y into c;d

e = MULT(a,c) and f = MULT(b,d)

RETURN e2^n + (MULT(a+b, c+d) - e - f) 2^{n/2} + f

• T(n) = 3 T(n/2) + n
  • Actually: T(n) = 2 T(n/2) + T(n/2 + 1) + kn
Time complexity of Gaussified MULT

- \( T(1) = 1 \) & \( T(n) = 3 \, T(n/2) + n \)

Technique 2: Expand recursion

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

\[
= 3 \left( 3T(n/4) + n/2 \right) + n = 3^2T(n/4) + n + 3/2n
\]

\[
= 3^2(3T(n/8) + n/4) + n + 3/2n
\]

\[
= 3^3T(n/8) + n + 3/2n + (3/2)^2n
\]

\[
= \ldots \ldots
\]

\[
= 3^kT(1) + n + 3/2n + (3/2)^2n + \ldots + (3/2)^{k-1}n \text{ where } 2^k = n
\]

\[
= 3 \log_2 n + n(1 + 3/2 + (3/2)^2 + \ldots + (3/2)^{k-1})
\]

\[
= n \log_2 3 + 2n \left( (3/2)^k - 1 \right)
\]

\[
= n \log_2 3 + 2n \left( n \log_2 3 / n - 1 \right)
\]

\[
= 3n \log_2 3 - 2n
\]

Not just 25% savings!
\[ \Theta(n^2) \text{ vs } \Theta(n^{1.58..}) \]
## Multiplication Algorithms

<table>
<thead>
<tr>
<th>Kindergarten ?</th>
<th>n(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*4=3+3+3+3</td>
<td></td>
</tr>
<tr>
<td>Grade School</td>
<td>n(^2)</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>n(^{1.58...})</td>
</tr>
<tr>
<td>Fastest Known</td>
<td>n \log n \log \log n</td>
</tr>
</tbody>
</table>
Next...

1. Covered basics of a simple design technique (Divide-and-conquer) – Ch. 2 of the text.
2. Next, Strassen’s algorithm for matrix multiplication
Matrix multiplication

• Fundamental operation in Linear Algebra
• Used for numerical differentiation, integration, optimization etc
Naïve matrix multiplication

SimpleMatrixMultiply (A,B)

1. n ← A.rows
2. C ← CreateMatrix(n,n)
3. for i ← 1 to n
4.   for j ← 1 to n
5.     C[i,j] ← 0
6.   for k ← 1 to n
8. return C

- Argue that the running time is $\theta(n^3)$
Faster Algorithm?

- Idea: Similar to multiplication in $\mathbb{N}$, $\mathbb{C}$
- Divide and conquer approach provides unexpected improvements
First attempt and Divide & Conquer

Divide A, B into 4 \( \frac{n}{2} \times \frac{n}{2} \) matrices

- \( C_{11} = A_{11}B_{11} + A_{12}B_{21} \)
- \( C_{12} = A_{11}B_{12} + A_{12}B_{22} \)
- \( C_{21} = A_{21}B_{11} + A_{22}B_{21} \)
- \( C_{22} = A_{21}B_{12} + A_{22}B_{22} \)

Simple Recursive implementation. Running time is given by the following recurrence.

- \( T(1) = C \), and for \( n > 1 \)
- \( T(n) = 8T(\frac{n}{2}) + \theta(n^2) \)
- \( \theta(n^3) \) time-complexity
Strassen’s algorithm

Avoid one multiplication (details on page 80) (but uses more additions)

Recurrence:
• $T(1) = C$, and for $n > 1$
• $T(n) = 7T(n/2) + \theta(n^2)$

• How can we solve this?
• Will see that $T(n) = \theta(n^{\lg 7})$, $\lg 7 = 2.8073\ldots$
The maximum-subarray problem

• Given an array of integers, find a contiguous subarray with the maximum sum.
• Very naïve algorithm:

• Brute force algorithm:

• At best, \( \theta(n^2) \) time complexity
Can we do divide and conquer?

• Want to use answers from left and right half subarrays.
• Problem: The answer may not lie in either!

• Key question: What information do we need from (smaller) subproblems to solve the big problem?

• Related question: how do we get this information?
A divide and conquer algorithm

Algorithm in Ch 4.1:

Recurrence:
• \( T(1) = C \), and for \( n > 1 \)
• \( T(n) = 2T(n/2) + \theta(n) \)
• \( T(n) = \theta(n \log n) \)
More divide and conquer: Merge Sort

- **Divide**: If $S$ has at least two elements (nothing needs to be done if $S$ has zero or one elements), remove all the elements from $S$ and put them into two sequences, $S_1$ and $S_2$, each containing about half of the elements of $S$. (i.e. $S_1$ contains the first $\left\lfloor n/2 \right\rfloor$ elements and $S_2$ contains the remaining $\left\lceil n/2 \right\rceil$ elements).

- **Conquer**: Sort sequences $S_1$ and $S_2$ using Merge Sort.

- **Combine**: Put back the elements into $S$ by merging the sorted sequences $S_1$ and $S_2$ into one sorted sequence.
Merge Sort: Algorithm

```
Merge-Sort(A, p, r)
    if p < r then
        q ← (p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
    Take the smallest of the two topmost elements of sequences A[p..q] and A[q+1..r] and put into the resulting sequence. Repeat this, until both sequences are empty. Copy the resulting sequence into A[p..r].
```
Merge Sort: example
Merge Sort: example
Merge Sort: example
Merge Sort: example

EECS 3101
Merge Sort: example

EECS 3101
Merge Sort: example

EECS 3101
Merge Sort: example
Merge Sort: example
Merge Sort: example
Merge Sort: example
Merge Sort: example

EECS 3101
Merge Sort: example

EECS 3101
Merge Sort: example
Merge Sort: example
Merge Sort: example

EECS 3101
Merge Sort: example

EECS 3101
Merge Sort: example

24  45  64  85  17  31  96  50
Merge Sort: example

24  45  64  85

17  31  96  50

EECS 3101
Merge Sort: example

EECS 3101
Merge Sort: example

EECS 3101
Merge Sort: example
Merge Sort: summary

- To sort $n$ numbers
  - if $n=1$ done!
  - recursively sort 2 lists of numbers $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ elements
  - merge 2 sorted lists in $\Theta(n)$ time

- Strategy
  - break problem into similar (smaller) subproblems
  - recursively solve subproblems
  - combine solutions to answer
Recurrences

• Running times of algorithms with Recursive calls can be described using recurrences

• A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

Example: Merge Sort

\[
T(n) = \begin{cases} 
\text{solving_trivial_problem} & \text{if } n = 1 \\
\text{num_pieces } T(n / \text{subproblem_size_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 
\end{cases}
\]

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

• Repeated substitution method
  – Expanding the recurrence by substitution and noticing patterns

• Substitution method
  – guessing the solutions
  – verifying the solution by the mathematical induction

• Recursion-trees

• Master method
  – templates for different classes of recurrences
Repeated Substitution Method

• Let’s find the running time of merge sort (let’s assume that \( n=2^b \), for some \( b \)).

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

\[
T(n) = 2T(n/2) + n \quad \text{substitute}
\]

\[
= 2\left(2T(n/4) + n/2\right) + n \quad \text{expand}
\]

\[
= 2^2T(n/4) + 2n \quad \text{substitute}
\]

\[
= 2^2\left(2T(n/8) + n/4\right) + 2n \quad \text{expand}
\]

\[
= 2^3T(n/8) + 3n \quad \text{observe the pattern}
\]

\[
T(n) = 2^i T(n/2^i) + in
\]

\[
= 2^{\lg n}T(n/n) + n\lg n = n + n\lg n
\]
Repeated Substitution Method

• The procedure is straightforward:
  – Substitute
  – Expand
  – Substitute
  – Expand
  – ...
  – Observe a pattern and write how your expression looks after the $i$-th substitution
  – Find out what the value of $i$ (e.g., $\lg n$) should be to get the base case of the recurrence (say $T(1)$)
  – Insert the value of $T(1)$ and the expression of $i$ into your expression
**Substitution method**

Solve $T(n) = 4T(n/2) + n$

1) Guess that $T(n) = O(n^3)$, i.e., that $T$ of the form $cn^3$

2) Assume $T(k) \leq ck^3$ for $k \leq n/2$ and

3) Prove $T(n) \leq cn^3$ by induction

$$T(n) = 4T(n/2) + n \text{ (recurrence)}$$

$$\leq 4c(n/2)^3 + n \text{ (ind. hypoth.)}$$

$$= \frac{c}{2} n^3 + n \text{ (simplify)}$$

$$= cn^3 - \left( \frac{c}{2} n^3 - n \right) \text{ (rearrange)}$$

$$\leq cn^3 \text{ if } c \geq 2 \text{ and } n \geq 1 \text{ (satisfy)}$$

Thus $T(n) = O(n^3)$!

Subtlety: Must choose $c$ big enough to handle $T(n) = \Theta(1)$ for $n < n_0$ for some $n_0$
Substitution method

- Achieving tighter bounds

Try to show $T(n) = O(n^2)$

Assume $T(k) \leq ck^2$

$$T(n) = 4T(n/2) + n$$
$$\leq 4c(n/2)^2 + n$$
$$= cn^2 + n$$
$$\leq cn^2 \text{ for no choice of } c > 0.$$
**Substitution method**

The problem: We could not rewrite the equality

\[ T(n) = cn^2 + \text{(something positive)} \]

as:

\[ T(n) \leq cn^2 \]

in order to show the inequality we wanted

- Sometimes to prove inductive step, try to strengthen your hypothesis
  - \( T(n) \leq (\text{answer you want}) - (\text{something} > 0) \)
Substitution method

• Corrected proof: the idea is to strengthen the inductive hypothesis by subtracting lower-order terms!

Assume \( T(k) \leq c_1k^2 - c_2k \) for \( k < n \)

\[
T(n) = 4T(n/2) + n \\
\leq 4(c_1(n/2)^2 - c_2(n/2)) + n \\
= c_1n^2 - 2c_2n + n \\
= c_1n^2 - c_2n - (c_2n - n) \\
\leq c_1n^2 - c_2n \text{ if } c_2 \geq 1
\]
A recursion tree is a convenient way to visualize what happens when a recurrence is iterated. Construction of a recursion tree

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

\[
\begin{align*}
&n^2 \\
&\quad \frac{1}{4}n^2 \\
&\quad \quad \frac{1}{16}n^2 \\
&\quad \frac{1}{8}n^2 \\
&\quad \frac{1}{16}n^2 \\
&\quad \frac{1}{8}n^2 \\
&\quad \frac{1}{4}n^2 \\
&\quad \frac{5}{16}n^2 \\
&\quad \frac{25}{256}n^2 \\
&\quad \vdots \\
&\Theta(n^2)
\end{align*}
\]
Recursion Tree

\[ T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \]

Total: \( O(n \log n) \)
Master Method

• The idea is to solve a class of recurrences that have the form

\[ T(n) = aT(n/b) + f(n) \]

• \( a \geq 1 \) and \( b > 1 \), and \( f \) is asymptotically positive!

• Abstractly speaking, \( T(n) \) is the runtime for an algorithm and we know that
  
  – a subproblems of size \( n/b \) are solved recursively, each in time \( T(n/b) \)
  
  – \( f(n) \) is the cost of dividing the problem and combining the results. In merge-sort

\[ T(n) = 2T(n/2) + \Theta(n) \]
Master method

Split problem into $a$ parts at $\log_b n$ levels. There are $a^{\log_b n} = n^{\log_b a}$ leaves.

Total: $\Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$
Master method

- Number of leaves:
- Iterating the recurrence, expanding the tree yields

\[ a^{\log_b n} = n^{\log_b a} \]

\[ T(n) = f(n) + aT(n/b) \]
\[ = f(n) + af(n/b) + a^2T(n/b^2) \]
\[ = f(n) + af(n/b) + a^2T(n/b^2) + ... \]
\[ + a^{\log_b n-1} f(n/b^{\log_b n-1}) + a^{\log_b n} T(1) \]

Thus,

\[ T(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) + \Theta(n^{\log_b a}) \]

- The first term is a division/recombination cost (totaled across all levels of the tree)
- The second term is the cost of doing all \( n^{\log_b a} \) subproblems of size 1 (total of all work pushed to leaves)
Master method intuition

• Three common cases:
  – Running time dominated by cost at leaves
  – Running time evenly distributed throughout the tree
  – Running time dominated by cost at root
• Consequently, to solve the recurrence, we need only to characterize the dominant term
• In each case compare $f(n)$ with $O(n^{\log_b a})$
Master method Case 1

• $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
  – $f(n)$ grows polynomially (by factor $n^\varepsilon$) slower than $n^{\log_b a}$

• The work at the leaf level dominates
  – Summation of recursion-tree levels $O(n^{\log_b a})$
  – Cost of all the leaves $\Theta(n^{\log_b a})$
  – Thus, the overall cost $\Theta(n^{\log_b a})$
Master method Case 2

• \( f(n) = \Theta(n^{\log_b a} \lg n) \)
  \(-f(n)\) and \( n^{\log_b a} \) are asymptotically the same

• The work is distributed equally throughout the tree \( T(n) = \Theta(n^{\log_b a} \lg n) \)
  \(-\) (level cost) \(\times\) (number of levels)
Master method Case 3

• \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \) for some constant \( \varepsilon > 0 \)
  – Inverse of Case 1
  – \( f(n) \) grows polynomially faster than \( n^{\log_b a} \)
  – Also need a regularity condition
  \( \exists c < 1 \) and \( n_0 > 0 \) such that \( af(n/b) \leq cf(n) \ \forall n > n_0 \)

• The work at the root dominates

\[ T(n) = \Theta(f(n)) \]
Master Theorem Summarized

- Given a recurrence of the form \( T(n) = aT(n/b) + f(n) \)
  1. \( f(n) = O\left(n^{\log_b a - \varepsilon}\right) \)
      \[ \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right) \]
  2. \( f(n) = \Theta\left(n^{\log_b a}\right) \)
      \[ \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right) \]
  3. \( f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \) and \( af(n/b) \leq cf(n) \), for some \( c < 1, n > n_0 \)
      \[ \Rightarrow T(n) = \Theta\left(f(n)\right) \]

- The master method cannot solve every recurrence of this form; there is a gap between cases 1 and 2, as well as cases 2 and 3
Using the Master Theorem

• Extract \( a, b, \) and \( f(n) \) from a given recurrence
• Determine \( n^{\log_b a} \)
• Compare \( f(n) \) and \( n^{\log_b a} \) asymptotically
• Determine appropriate MT case, and apply
• Example merge sort

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

\( a = 2, \ b = 2; \ n^{\log_b a} = n^{\log_2 2} = n = \Theta(n) \)

Also \( f(n) = \Theta(n) \)

\[
\Rightarrow \text{Case 2: } T(n) = \Theta\left(n^{\log_b a} \log n\right) = \Theta(n \log n)
\]
Examples

\( T(n) = T(n/2) + 1 \)

\( a = 1, b = 2; \ n^{\log_2 1} = 1 \)

also \( f(n) = 1, f(n) = \Theta(1) \)

\( \Rightarrow \text{Case 2: } T(n) = \Theta(\log n) \)

\( T(n) = 9T(n/3) + n \)

\( a = 9, b = 3; \)

\( f(n) = n, f(n) = O(n^{\log_3 9 - \varepsilon}) \text{ with } \varepsilon = 1 \)

\( \Rightarrow \text{Case 1: } T(n) = \Theta(n^2) \)

\[\text{Binary-search}(A, p, r, s) : \]
\[q \leftarrow (p+r)/2\]
\[
\text{if } A[q] = s \text{ then return } q
\]
\[
\text{else if } A[q] > s \text{ then }
\quad \text{Binary-search}(A, p, q-1, s)
\]
\[
\text{else Binary-search}(A, q+1, r, s)
\]
Examples

\[ T(n) = 3T(n/4) + n \log n \]
\[ a = 3, b = 4; \quad n^{\log_4 3} = n^{0.793} \]
\[ f(n) = n \log n, \quad f(n) = \Omega(n^{\log_4 3 + \varepsilon}) \text{ with } \varepsilon \approx 0.2 \]
\[ \Rightarrow \text{Case 3:} \]

Regularity condition
\[ af(n/b) = 3(n/4) \log(n/4) \leq (3/4)n \log n = cf(n) \text{ for } c = 3/4 \]
\[ T(n) = \Theta(n \log n) \]

\[ T(n) = 2T(n/2) + n \log n \]
\[ a = 2, b = 2; \quad n^{\log_2 2} = n^1 \]
\[ f(n) = n \log n, \quad f(n) = \Omega(n^{1 + \varepsilon}) \text{ with } \varepsilon? \]
also \( n \log n / n^1 = \log n \)
\[ \Rightarrow \text{neither Case 3 nor Case 2!} \]
Examples

\[ T(n) = 4T(n/2) + n^3 \]
\[ a = 4, b = 2; \quad n^{\log_2 4} = n^2 \]
\[ f(n) = n^3; \quad f(n) = \Omega(n^2) \]

⇒ Case 3: \[ T(n) = \Theta(n^3) \]

Checking the regularity condition
\[ 4f(n/2) \leq cf(n) \]
\[ 4n^3 / 8 \leq cn^3 \]
\[ n^3 / 2 \leq cn^3 \]
\[ c = 3 / 4 < 1 \]
1. Covered basics of a simple design technique (Divide-and-conquer) – Ch. 4 of the text.
2. Next, more sorting algorithms.