Lecture 3: Divide-and-Conquer Algorithms

We just derived an $O(n \log n)$ divide-and-conquer algorithm for solving the Maximum Contiguous Subarray problem.

In COMP171 you already saw Mergesort, an $O(n \log n)$ time divide-and-conquer sorting algorithm.

**Divide-and-Conquer** is not a trick. It is a very useful general purpose tool for designing efficient algorithms.
The Basic Divide-and-Conquer Approach

**Divide:** Divide a given problem into two subproblems (ideally of approximately equal size).

**Conquer:** Solve each subproblem (directly or recursively), and

**Combine:** Combine the solutions of the two subproblems into a global solution.

**Note:** the hard work and cleverness is usually in the Combine step.
**MERGESORT**

**Mergesort**(A, i, j) : Sort A[i...j]

If (i ≠ j)

{  
    Mergesort(A, i, \left\lfloor \frac{i+j}{2} \right\rfloor)  
    \quad M(\frac{j-i}{2})  
    \ 
    Mergesort(A, 1 + \left\lfloor \frac{i+j}{2} \right\rfloor, j)  
    \quad M(\frac{j-i}{2})  
    \ 
    Merge the two sorted lists  
    A[i...\left\lfloor \frac{i+j}{2} \right\rfloor] and A[1 + \left\lfloor \frac{i+j}{2} \right\rfloor, j]  
    \quad O(j - i)  
    \ 
    and return complete sorted list

}

The algorithm sorts an array of size N by splitting it into two parts of (almost) equal size, recursively sorting each of them, and then merging the two sorted subarrays back together into a fully sorted list in \(O(N)\) time (how?).

The running time of the algorithm satisfies

\[
\forall N > 1, \quad M(N) \leq 2M(N/2) + O(N)
\]

which we previously saw implies

\[
M(N) = O(N \log N).
\]
Mergesort Example

12 23 3 13 8 4 11 24

split

12 23 3 13

8 4 11 24

sort each sublist

3 12 13 23

4 8 11 24

Merge

3 4 8 11 12 13 23 24
A More General Divide-and-Conquer Approach

**Divide:** Divide a given problem into subproblems (ideally of approximately equal size).
No longer only TWO subproblems

**Conquer:** Solve each subproblem (directly or recursively), and

**Combine:** Combine the solutions of the subproblems into a global solution.
The Polynomial Multiplication Problem

another divide-and-conquer algorithm

Problem:
Given two polynomials of degree $n - 1$

$$A(x) = a_0 + a_1x + \cdots + a_n x^{n-1}$$

$$B(x) = b_0 + b_1x + \cdots + b_n x^{n-1},$$

compute the product $A(x)B(x)$.

Example:

$$A(x) = 1 + 2x + 3x^2$$

$$B(x) = 3 + 2x + 2x^2$$

$$A(x)B(x) = 3 + 5x + 15x^2 + 10x^3 + 6x^4$$

Question: How can we efficiently calculate the coefficients of $A(x)B(x)$?

Assume that the coefficients $a_i$ and $b_i$ are stored in arrays $A[0 \ldots n - 1]$ and $B[0 \ldots n - 1]$.

Cost of any algorithm is number of scalar multiplications and additions performed.
Convolutions

Let \( A(x) = \sum_{i=1}^{n} a_i x_i \) and \( B(x) = \sum_{i=0}^{m} b_i x_i \).

Set \( C(x) = \sum_{k=0}^{n+m} c_i x^i = A(x)B(x) \).

Then

\[
c_k = \sum_{i=0}^{k} a_i b_{k-i}
\]

for all \( 0 \leq k \leq m + n \).

**Definition:** The vector \((c_0, c_1, \ldots, c_{m+n})\)

is the convolution of the vectors

\((a_0, a_1, \ldots, a_n)\) and \((b_0, b_1, \ldots, b_m)\).

Calculating convolutions (and thus polynomial multiplication) is a major problem in digital signal processing.
The Direct (Brute Force) Approach

Let $A(x) = \sum_{i=1}^{n-1} a_i x_i$ and $B(x) = \sum_{i=0}^{n-1} b_i x_i$.

Set $C(x) = \sum_{k=0}^{2n-2} c_i x^i = A(x)B(x)$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

for all $0 \leq k \leq 2n - 2$.

The direct approach is to compute all $c_k$ using the formula above. The total number of multiplications and additions needed are $n^2$ and $(n - 1)^2$ respectively. Hence the complexity is $\Theta(n^2)$.

**Questions:** Can we do better?
Can we apply the divide-and-conquer approach to develop an algorithm?
The Divide-and-Conquer Approach

The Divide Step: Define

\[ A_0(x) = a_0 + a_1 x + \cdots + a_{\lfloor n/2 \rfloor - 1} x^{\lfloor n/2 \rfloor - 1}, \]
\[ A_1(x) = a_{\lfloor n/2 \rfloor} + a_{\lfloor n/2 \rfloor + 1} x + \cdots + a_n x^{n - \lfloor n/2 \rfloor}. \]

Then \( A(x) = A_0(x) + A_1(x) x^{\lfloor n/2 \rfloor}. \)

Similarly we define \( B_0(x) \) and \( B_1(x) \) such that

\[ B(x) = B_0(x) + B_1(x) x^{\lfloor n/2 \rfloor}. \]

Then

\[ A(x) B(x) = A_0(x) B_0(x) + A_0(x) B_1(x) x^{\lfloor n/2 \rfloor} + A_1(x) B_0(x) x^{\lfloor n/2 \rfloor} + A_1(x) B_1(x) x^{2 \lfloor n/2 \rfloor}. \]

Remark: The original problem of size \( n \) is divided into 4 problems of input size \( n/2 \).
Example:

\[
\begin{align*}
A(x) &= 2 + 5x + 3x^2 + x^3 \\
B(x) &= 1 + 2x + 2x^2 + 3x^3 \\
A(x)B(x) &= 2 + 9x + 17x^2 + 23x^3 + 23x^4 + 11x^5 + 3x^6
\end{align*}
\]

\[
\begin{align*}
A_0(x) &= 2 + 5x, \quad A_1(x) = 3 + x, \quad A(x) = A_0(x) + A_1(x)x^2 \\
B_0(x) &= 1 + 2x, \quad B_1(x) = 2 + 3x, \quad B(x) = B_0(x) + B_1(x)x^2
\end{align*}
\]

\[
\begin{align*}
A_0(x)B_0(x) &= 2 + 9x + 10x^2 \\
A_1(x)B_1(x) &= 6 + 11x + 3x^2 \\
A_0(x)B_1(x) &= 4 + 16x + 15x^2 \\
A_1(x)B_0(x) &= 3 + 7x + 2x^2 \\
A_0(x)B_1(x) + A_1(x)B_0(x) &= 7 + 23x + 17x^2
\end{align*}
\]

\[
\begin{align*}
&= A_0(x)B_0(x) \\
+ (A_0(x)B_1(x) + A_1(x)B_0(x))x^2 \\
+ A_1(x)B_1(x)x^4
\end{align*}
\]

\[
= 2 + 9x + 17x^2 + 23x^3 + 23x^4 + 11x^5 + 3x^6
\]
The Divide-and-Conquer Approach

The Conquer Step: Solve the four subproblems, i.e., computing

\[ A_0(x)B_0(x), \quad A_0(x)B_1(x), \]
\[ A_1(x)B_0(x), \quad A_1(x)B_1(x) \]

by recursively calling the algorithm 4 times.
The Divide-and-Conquer Approach

The Combining Step: Adding the following four polynomials

\[ A_0(x)B_0(x) + A_0(x)B_1(x)x^{[\frac{n}{2}]} + A_1(x)B_0(x)x^{[\frac{n}{2}]} + A_1(x)B_1(x)x^{2[\frac{n}{2}]} \]

takes \( \Theta(n) \) operations. Why?
The First Divide-and-Conquer Algorithm

PolyMulti1\( (A(x), B(x)) \)
\[\{\]
\[A_0(x) = a_0 + a_1x + \cdots + a_{\lfloor n/2 \rfloor - 1}x^{\lfloor n/2 \rfloor - 1};\]
\[A_1(x) = a_{\lfloor n/2 \rfloor} + a_{\lfloor n/2 \rfloor + 1}x + \cdots + a_nx^{n - \lfloor n/2 \rfloor};\]
\[B_0(x) = b_0 + b_1x + \cdots + b_{\lfloor n/2 \rfloor - 1}x^{\lfloor n/2 \rfloor - 1};\]
\[B_1(x) = b_{\lfloor n/2 \rfloor} + b_{\lfloor n/2 \rfloor + 1}x + \cdots + b_nx^{n - \lfloor n/2 \rfloor};\]
\[U(x) = PolyMulti1(A_0(x), B_0(x));\]
\[V(x) = PolyMulti1(A_0(x), B_1(x));\]
\[W(x) = PolyMulti1(A_1(x), B_0(x));\]
\[Z(x) = PolyMulti1(A_1(x), B_1(x));\]
\[\text{return}\left(U(x) + [V(x) + W(x)]x^{\lfloor n/2 \rfloor} + Z(x)x^{2\lfloor n/2 \rfloor}\right);\]
\[\}\]
Running Time of the Algorithm

Assume \( n \) is a power of 2, \( n = 2^h \). By substitution (expansion),

\[
T(n) = 4 T\left(\frac{n}{2}\right) + c n
\]
\[
= 4 \left[ 4 T\left(\frac{n}{2^2}\right) + c \frac{n}{2} \right] + c n
\]
\[
= 4^2 T\left(\frac{n}{2^2}\right) + (1 + 2) c n
\]
\[
= 4^2 \left[ 4 T\left(\frac{n}{2^3}\right) + c \frac{n}{2^2} \right] + (1 + 2) c n
\]
\[
= 4^3 T\left(\frac{n}{2^3}\right) + (1 + 2 + 2^2) c n
\]
\[
\vdots
\]
\[
= 4^i T\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i-1} 2^j c n \quad \text{(induction)}
\]
\[
\vdots
\]
\[
= 4^h T\left(\frac{n}{2^h}\right) + \sum_{j=0}^{h-1} 2^j c n
\]
\[
= n^2 T(1) + c n(n - 1)
\]

(since \( n = 2^h \) and \( \sum_{j=0}^{h-1} 2^j = 2^h - 1 = n - 1 \))

\[
= \Theta(n^2).
\]

The same order as the brute force approach!
Comments on the Divide-and-Conquer Algorithm

**Comments:** The divide-and-conquer approach makes no essential improvement over the brute force approach!

**Question:** Why does this happen.

**Question:** Can you improve this divide-and-conquer algorithm?
**Problem:** Given 4 numbers

\[ A_0, A_1, B_0, B_1 \]

how many multiplications are needed to calculate the three values

\[ A_0B_0, A_0B_1 + A_1B_0, A_1B_1? \]

This can obviously be done using 4 multiplications but there is a way of doing this using only the following 3:

\[
\begin{align*}
U &= (A_0 + A_1)(B_0 + B_1) \\
V &= A_0B_0 \\
W &= A_1B_1
\end{align*}
\]

\(Y\) and \(Z\) are what we originally wanted and

\[ A_0B_1 + A_1B_0 = U - V - W. \]
Improving the Divide-and-Conquer Algorithm

Define

\[ U(x) = (A_0(x) + A_1(x)) \times (B_0(x) + B_1(x)) \]
\[ V(x) = A_0(x)B_0(x) \]
\[ W(x) = A_1(x)B_1(x) \]

Then

\[ U(x) - V(x) - W(x) = A_0(x)B_1(x) + A_1(x)B_0(x). \]

Hence \( A(x)B(x) \) is equal to

\[ V(x) + [U(x) - V(x) - W(x)]x^{n/2} + W(x) \times x^{2[n/2]} \]

**Conclusion:** You need to call the multiplication procedure 3, rather than 4 times.
**The Second Divide-and-Conquer Algorithm**

\[
\text{PolyMulti2}(A(x), B(x)) \\
\begin{align*}
A_0(x) &= a_0 + a_1 x + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1}; \\
A_1(x) &= a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + a_n x^{n - \lfloor \frac{n}{2} \rfloor}; \\
B_0(x) &= b_0 + b_1 x + \cdots + b_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1}; \\
B_1(x) &= b_{\lfloor \frac{n}{2} \rfloor} + b_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + b_n x^{n - \lfloor \frac{n}{2} \rfloor}; \\
U(x) &= \text{PolyMulti2}(A_0(x) + A_1(x), B_0(x) + B_1(x)) \\
V(x) &= \text{PolyMulti2}(A_0(x), B_0(x)); \\
W(x) &= \text{PolyMulti2}(A_1(x), B_1(x)); \\
\text{return } &\left(V(x) + [U(x) - V(x) - W(x)] x^{\lfloor \frac{n}{2} \rfloor} + W(x) x^{2 \lfloor \frac{n}{2} \rfloor}\right);
\end{align*}
\]
Assume $n = 2^h$. Let $\lg x$ denote $\log_2 x$.
By the substitution method,

$$T(n) = 3 T\left(\frac{n}{2}\right) + cn$$

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

$$= 3^2 T\left(\frac{n}{2^2}\right) + \left(1 + \frac{3}{2}\right) cn$$

$$= 3^2 \left[ 3 T\left(\frac{n}{2^3}\right) + c \frac{n}{2^2} \right] + \left(1 + \frac{3}{2} + \frac{3^2}{2^2}\right) cn$$

$$= 3^3 T\left(\frac{n}{2^3}\right) + \left(1 + \frac{3}{2} + \left[\frac{3}{2}\right]^2\right) cn$$

$$\vdots$$

$$= 3^h T\left(\frac{n}{2^h}\right) + \sum_{j=0}^{h-1} \left[\frac{3}{2}\right]^j c n.$$ 

We have

$$3^h = (2^{\lg 3})^h = 2^{h \lg 3} = (2^h)^{\lg 3} = n^{\lg 3} \approx n^{1.586},$$

and

$$\sum_{j=0}^{h-1} \left[\frac{3}{2}\right]^j = \frac{(3/2)^h - 1}{3/2 - 1} = 2 \cdot \frac{3^h}{2^h} - 2 = 2n^{\lg 3-1} - 2.$$ 

Hence

$$T(n) = \Theta(n^{\lg 3} T(1) + 2 cn^{\lg 3}) = \Theta(n^{\lg 3}).$$
• The divide-and-conquer approach doesn’t always give you the best solution. Our original D-A-C algorithm was just as bad as brute force.

• There is actually an $O(n \log n)$ solution to the polynomial multiplication problem. It involves using the Fast Fourier Transform algorithm as a subroutine. The FFT is another classic D-A-C algorithm (Chapt 30 in CLRS gives details).

• The idea of using 3 multiplications instead of 4 is used in large-integer multiplications. A similar idea is the basis of the classic Strassen matrix multiplication algorithm (CLRS, Chapter 28).