Chapter 4

Divide-and-Conquer
Divide-and-Conquer

The most-well known algorithm design strategy:

2. Divide instance of problem into two or more smaller instances

4. Solve smaller instances recursively

6. Obtain solution to original (larger) instance by combining these solutions
Divide-and-Conquer Technique (cont.)

A problem of size $n$

- Subproblem 1 of size $n/2$
  - A solution to subproblem 1

- Subproblem 2 of size $n/2$
  - A solution to subproblem 2

A solution to the original problem
Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms
General Divide-and-Conquer Recurrence

\[ T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \in \Theta(n^d), \quad d \geq 0 \]

**Master Theorem:**

- If \( a < b^d \), \( T(n) \in \Theta(n^d) \)
- If \( a = b^d \), \( T(n) \in \Theta(n^d \log n) \)
- If \( a > b^d \), \( T(n) \in \Theta(n^{\log_b a}) \)

**Note:** The same results hold with \( O \) instead of \( \Theta \).

**Examples:**

- \( T(n) = 4T(n/2) + n \Rightarrow T(n) \in \? \)
- \( T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \? \)
- \( T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \? \)
Mergesort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.
Pseudocode of Mergesort

ALGORITHM Mergesort(A[0..n − 1])

// Sorts array A[0..n − 1] by recursive mergesort
// Input: An array A[0..n − 1] of orderable elements
// Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌈n/2⌉ − 1] to B[0..⌈n/2⌉ − 1]
    copy A[⌈n/2⌉..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌈n/2⌉ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
Pseudocode of Merge

ALGORITHM \textit{Merge} (B[0..p - 1], C[0..q - 1], A[0..p + q - 1])

//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p - 1] and C[0..q - 1] both sorted
//Output: Sorted array A[0..p + q - 1] of the elements of B and C

i \leftarrow 0; \ j \leftarrow 0; \ k \leftarrow 0

\textbf{while} \ i < p \ \textbf{and} \ j < q \ \textbf{do}

\textbf{if} \ B[i] \leq C[j]

\hspace{1cm} A[k] \leftarrow B[i]; \ i \leftarrow i + 1

\hspace{1cm} \textbf{else} \ A[k] \leftarrow C[j]; \ j \leftarrow j + 1

\hspace{1cm} k \leftarrow k + 1

\textbf{if} \ i = p

\hspace{1cm} \text{copy} \ C[j..q - 1] \text{ to } A[k..p + q - 1]

\textbf{else} \ \text{copy} \ B[i..p - 1] \text{ to } A[k..p + q - 1]
Mergesort Example
Analysis of Mergesort

- All cases have same efficiency: \( \Theta(n \log n) \)

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
  \[
  \lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n
  \]

- Space requirement: \( \Theta(n) \) (not in-place)

- Can be implemented without recursion (bottom-up)
Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first \( s \) positions are smaller than or equal to the pivot and all the elements in the remaining \( n-s \) positions are larger than or equal to the pivot (see next slide for an algorithm)

\[
\begin{align*}
A[i] \leq p \\
A[i] \geq p
\end{align*}
\]

- Exchange the pivot with the last element in the first (i.e., \( \leq \)) subarray — the pivot is now in its final position
- Sort the two subarrays recursively
Partitioning Algorithm

Algorithm \textit{Partition}(A[l..r])

// Partitions a subarray by using its first element as a pivot
// Input: A subarray A[l..r] of A[0..n - 1], defined by its left and right
// indices \( l \) and \( r \) (\( l < r \))
// Output: A partition of A[l..r], with the split position returned as
// this function’s value

\( p \leftarrow A[l] \)
\( i \leftarrow l; \ j \leftarrow r + 1 \)

repeat
    repeat \( i \leftarrow i + 1 \) until \( A[i] \geq p \)
    repeat \( j \leftarrow j - 1 \) until \( A[j] < p \)
    swap(A[i], A[j])
until \( i \geq j \)

swap(A[i], A[j]) // undo last swap when \( i \geq j \)

swap(A[l], A[j])
return \( j \)
Quicksort Example

5 3 1 9 8 2 4 7
Analysis of Quicksort

- **Best case**: split in the middle — $\Theta(n \log n)$
- **Worst case**: sorted array! — $\Theta(n^2)$
- **Average case**: random arrays — $\Theta(n \log n)$

**Improvements:**

- better pivot selection: median of three partitioning
- switch to insertion sort on small subfiles
- elimination of recursion

These combine to 20-25% improvement

**Considered the method of choice for internal sorting of large files** ($n \geq 10000$)
Binary Search

Very efficient algorithm for searching in sorted array:

\[ K \]

vs

\[ A[0] \ldots A[m] \ldots A[n-1] \]

If \( K = A[m] \), stop (successful search); otherwise, continue searching by the same method in \( A[0..m-1] \) if \( K < A[m] \) and in \( A[m+1..n-1] \) if \( K > A[m] \).

\[ l \leftarrow 0; \quad r \leftarrow n-1 \]

while \( l \leq r \) do

\[ m \leftarrow \lfloor (l+r)/2 \rfloor \]

if \( K = A[m] \) return \( m \)

else if \( K < A[m] \) \( r \leftarrow m-1 \)

else \( l \leftarrow m+1 \)

return -1
Analysis of Binary Search

- **Time efficiency**
  - worst-case recurrence: \( C_w(n) = 1 + C_w(\lfloor n/2 \rfloor) \), \( C_w(1) = 1 \)
  - solution: \( C_w(n) = \lceil \log_2(n+1) \rceil \)

  This is VERY fast: e.g., \( C_w(10^6) = 20 \)

- **Optimal for searching a sorted array**

- **Limitations:** must be a sorted array (not linked list)

- **Bad (degenerate) example of divide-and-conquer**

- **Has a continuous counterpart called *bisection method* for solving equations in one unknown \( f(x) = 0 \) (see Sec. 12.4)**
Binary Tree Algorithms

Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

Algorithm *Inorder(T)*

if \( T \neq \emptyset \)

\[
\text{Inorder}(T_{\text{left}}) \quad \text{Inorder}(T_{\text{right}})
\]

\[
\text{print(root of } T) \quad d \quad e
\]

Efficiency: \( \Theta(n) \)
Ex. 2: Computing the height of a binary tree

\[ h(T) = \max\{h(T_L), h(T_R)\} + 1 \text{ if } T \neq \emptyset \text{ and } h(\emptyset) = -1 \]

Efficiency: \( \Theta(n) \)
Multiplication of Large Integers

Consider the problem of multiplying two (large) \(n\)-digit integers represented by arrays of their digits such as:

\[
A = \begin{array}{cccccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 3 & 5 & 7 & 9 & 8 & 6 & 4 & 2 & 9 \\
\end{array} \quad B = \begin{array}{cccccccccccccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 2 & 8 & 4 & 8 & 2 & 0 & 9 & 1 & 2 & 8 & 3 & 6 \\
\end{array}
\]

The grade-school algorithm:

\[
\begin{array}{cccccccccccccccccc}
a_1 & a_2 & \ldots & a_n \\
b_1 & b_2 & \ldots & b_n \\
\hline
(d_{10}) d_{11} d_{12} & \ldots & d_{1n} \\
(d_{20}) d_{21} d_{22} & \ldots & d_{2n} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
(d_{n0}) d_{n1} d_{n2} & \ldots & d_{nn} \\
\end{array}
\]

Efficiency: \(n^2\) one-digit multiplications
First Divide-and-Conquer Algorithm

A small example: \( A \times B \) where \( A = 2135 \) and \( B = 4014 \)

\[
A = (21 \cdot 10^2 + 35), \quad B = (40 \cdot 10^2 + 14)
\]

So, \( A \times B = (21 \cdot 10^2 + 35) \times (40 \cdot 10^2 + 14) \)

\[
= 21 \times 40 \cdot 10^4 + (21 \times 14 + 35 \times 40) \cdot 10^2 + 35 \times 14
\]

In general, if \( A = A_1A_2 \) and \( B = B_1B_2 \) (where \( A \) and \( B \) are \( n \)-digit, \( A_1, A_2, B_1, B_2 \) are \( n/2 \)-digit numbers),

\[
A \times B = A_1 \times B_1 \cdot 10^n + (A_1 \times B_2 + A_2 \times B_1) \cdot 10^{n/2} + A_2 \times B_2
\]

Recurrence for the number of one-digit multiplications \( M(n) \): 

\[
M(n) = 4M(n/2), \quad M(1) = 1
\]

Solution: 

\[
M(n) = n^2
\]
Second Divide-and-Conquer Algorithm

A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2

The idea is to decrease the number of multiplications from 4 to 3:

(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,

I.e., (A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M(n):

M(n) = 3M(n/2), \quad M(1) = 1

Solution: M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}
Example of Large-Integer Multiplication

2135 \times 4014
Strassen’s Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

\[
\begin{pmatrix}
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{pmatrix}
= \begin{pmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{pmatrix}
\times
\begin{pmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\
M_2 + M_4 & M_1 + M_3 - M_2 + M_6
\end{pmatrix}
\]
Formulas for Strassen’s Algorithm

\[ M_1 = (A_{00} + A_{11}) \times (B_{00} + B_{11}) \]

\[ M_2 = (A_{10} + A_{11}) \times B_{00} \]

\[ M_3 = A_{00} \times (B_{01} - B_{11}) \]

\[ M_4 = A_{11} \times (B_{10} - B_{00}) \]

\[ M_5 = (A_{00} + A_{01}) \times B_{11} \]

\[ M_6 = (A_{10} - A_{00}) \times (B_{00} + B_{01}) \]

\[ M_7 = (A_{01} - A_{11}) \times (B_{10} + B_{11}) \]
Analysis of Strassen’s Algorithm

If \( n \) is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

\[
M(n) = 7M(n/2), \quad M(1) = 1
\]

Solution: \( M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807} \) vs. \( n^3 \) of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex.
Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets $S_1$ and $S_2$ by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.
Closest Pair by Divide-and-Conquer (cont.)

Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 Set \( d = \min\{d_1, d_2\} \)

We can limit our attention to the points in the symmetric vertical strip of width \( 2d \) as possible closest pair. Let \( C_1 \) and \( C_2 \) be the subsets of points in the left subset \( S_1 \) and of the right subset \( S_2 \), respectively, that lie in this vertical strip. The points in \( C_1 \) and \( C_2 \) are stored in increasing order of their \( y \) coordinates, which is maintained by merging during the execution of the next step.

Step 4 For every point \( P(x,y) \) in \( C_1 \), we inspect points in \( C_2 \) that may be closer to \( P \) than \( d \). There can be no more than 6 such points (because \( d \leq d_2 \))!
The worst case scenario is depicted below:
Efficiency of the Closest-Pair Algorithm

Running time of the algorithm is described by

\[ T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in O(n) \]

By the Master Theorem (with \( a = 2, b = 2, d = 1 \))

\[ T(n) \in O(n \log n) \]
Quickhull Algorithm

Convex hull: smallest convex set that includes given points

- Assume points are sorted by $x$-coordinate values
- Identify extreme points $P_1$ and $P_2$ (leftmost and rightmost)
- Compute upper hull recursively:
  - find point $P_{\text{max}}$ that is farthest away from line $P_1P_2$
  - compute the upper hull of the points to the left of line $P_1P_{\text{max}}$
  - compute the upper hull of the points to the left of line $P_{\text{max}}P_2$

- Compute lower hull in a similar manner
Efficiency of Quickhull Algorithm

- Finding point farthest away from line $P_1P_2$ can be done in linear time

- Time efficiency:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)

- If points are not initially sorted by $x$-coordinate value, this can be accomplished in $O(n \log n)$ time

- Several $O(n \log n)$ algorithms for convex hull are known