Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions
3

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
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms
- Binary search: decrease-by-half (or degenerate divide&conq.)
General Divide-and-Conquer Recurrence

Examples: 

1. \( T(n) = 2T(n/2) + 1 \Rightarrow T(n) \in ? \)
2. \( T(n) = 4T(n/2) + 1 \Rightarrow T(n) \in ? \)
3. \( T(n) = 4T(n/2) + n \Rightarrow T(n) \in ? \)
4. \( T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ? \)
5. \( T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ? \)
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:**
- \( T(n) = 2T(n/2) + 1 \implies T(n) \in ? \)
- \( T(n) = 4T(n/2) + 1 \implies T(n) \in ? \)
- \( T(n) = 4T(n/2) + n \implies T(n) \in ? \)
- \( T(n) = 4T(n/2) + n^2 \implies T(n) \in ? \)
- \( T(n) = 4T(n/2) + n^3 \implies T(n) \in ? \)
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**  
\[ \text{Merge}(B[0..p-1], C[0..q-1], A[0..p+q-1]) \]

//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 \( C \)

\( i \leftarrow 0; \ j \leftarrow 0; \ k \leftarrow 0 \)

while \( i < p \) and \( j < q \) do

  if \( B[i] \leq C[j] \)
    \( A[k] \leftarrow B[i]; \ i \leftarrow i + 1 \)
  else \( A[k] \leftarrow C[j]; \ j \leftarrow j + 1 \)
  \( k \leftarrow k + 1 \)

if \( i = p \)
  copy \( C[j..q-1] \) to \( A[k..p+q-1] \)
else copy \( B[i..p-1] \) to \( A[k..p+q-1] \)
Mergesort Example

8 3 2 9 7 1 5 4
8 3 2 9
8 3
8
3
3 8
2 9
2 9
1 7
1 7
1
1
1 4 5 7
1 4 5 7
1 2 3 4 5 7 8 9
1 2 3 4 5 7 8 9

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:
  $$\left[\log_2 n!\right] \approx n \log_2 n - 1.44n$$
- 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)

- 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
Hoare’s Partitioning Algorithm

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

// 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

$p \leftarrow A[l]$
$i \leftarrow i; \quad j \leftarrow r + 1$

repeat
    repeat $i \leftarrow i + 1$ until $A[i] \geq p$
    repeat $j \leftarrow j - 1$ until $A[j] \leq p$
    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

3 7 8 5 2 1 9 5 4
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 Tree Algorithms

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

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

Algorithm Inorder(T)

Efficiency:
Binary Tree Algorithms

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

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

Algorithm Inorder(T)

if $T \neq \emptyset$

Inorder($T_{left}$)

print(root of $T$)

Inorder($T_{right}$)

Efficiency: $\Theta(n)$
Binary Tree Algorithms (cont.)

Ex. 2: Computing the height of a binary tree

\[ T = T_L \cup T_R \]
Binary Tree Algorithms (cont.)

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: $\Theta(n)$