ALGORITHM DESIGN USING DIVIDE & CONQUER METHOD: III

Partha P Chakrabarti

Indian Institute of Technology Kharagpur

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

1. Core Methods
   a. Divide and Conquer ✓
   b. Greedy Algorithms ✓
   c. Dynamic Programming ✓
   d. Branch-and-Bound ✓
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

Sorting by Divide & Conquer Method

Sort (L)
{ If(|L| ≤ 1) return (L)
Decomposition(/choice points/)
split L into non-empty L_1, L_2
Appropriately choose an element
Recursion
$M_1 =$ Sort(L_1)
$M_2 =$ sort(L_2)
Recomposition
Mechanism to combine M_1 & M_2 to get M
return (M)
}

Method 1 :-
Decomposition :- Remove Max $O(n)$
$L_1 = max(L)$
$L_2 = L_1 - max(L)$ $O(1)$
Recomposition :- concat.(M_1.M_2)
$O(1)$
Method 2 :-
Decomposition :- Remove an element
$L_1 = \{x_i\}$, $L_2 = L - L_1$ $O(n)$
Recomposition :- Insert x_i in M_2

$$\boxed{T(n) = T(n-1) + O(n)} = O(n^2)$$

Data Structures → HEAP, Balanced BST } $O(n\log n)$

MergeSort

Mergesort (L)
{ if (|L| ≤ 1) return (L)
split L into L_1, L_2 which are non-empty
M_1 = Mergesort (L_1)
M_2 = Mergesort (L_2)
M = Merge (L_1, L_2)
return (M)
}

$$T(n) = T(n/k) + T\left(n - \frac{n}{k}\right) + O(n)$$
$$= O(n \log n)$$

choose k = 2 → optimal performance

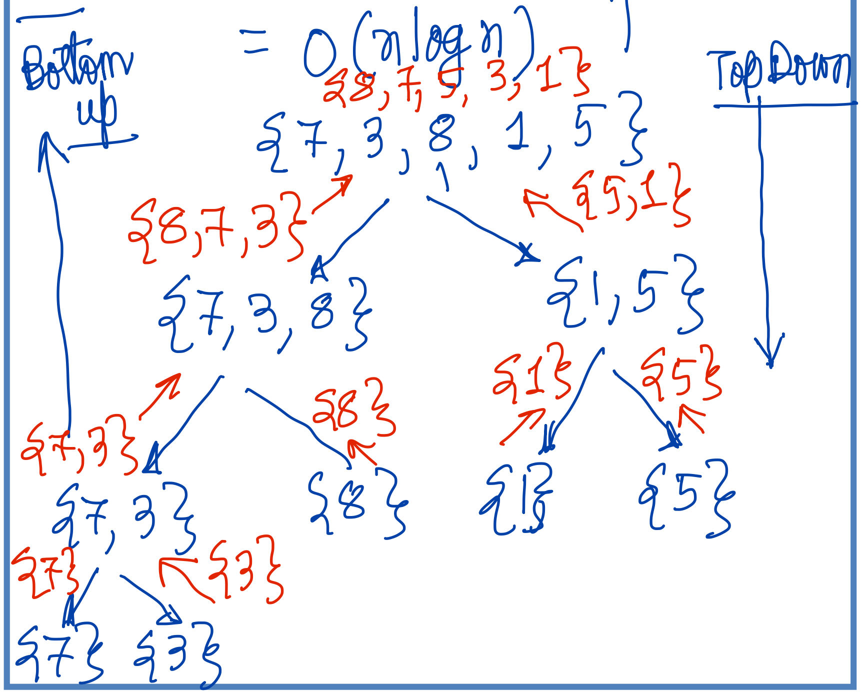
Merge (L_1, L_2)
{ if (L_1 = NULL) return (L_2)
if (L_2 = NULL) return (L_1)
let $L_1 = \{x_1, x_2, \ldots, x_n\}$
$L_2 = \{y_1, y_2, \ldots, y_m\}$
if ($x_1 \geq y_1$)
$L = \{x_1\} \,||\,$ Merge$(L_1 - \{x_1\}, L_2)$
↑concat
else
$L = \{y_1\} \,||\,$ Merge$(L_1, L_2 - \{y_1\})$
return (L)
}

$$T(n) = T(n-1) + 1$$
$$= O(n)$$

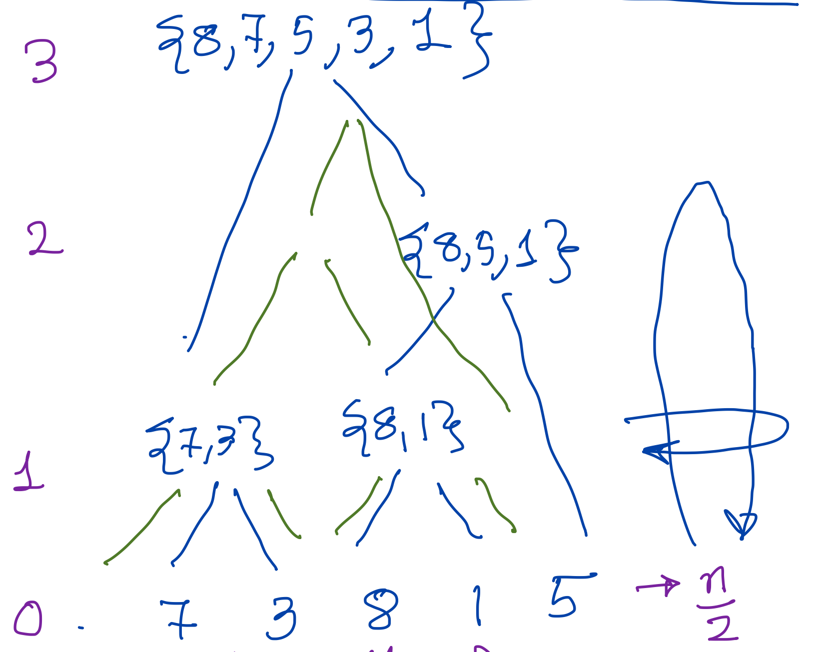
MergeSort: Analysis

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

$$= O(n \log n)$$



Bottom up evaluation (iteration)



No. of levels: $O(\log n)$

MergeSort: Finalization

→ Use a global array A to store the elements

→ pass array A indices during recursion

→ Merge → We use pointers / indices on Array A

TOP-DOWN ALGORITHM

BOTTOM UP ITERATIVE ALGO

→ inner loop of Merge calls and an outer loop which will go on for $O(\log n)$ steps

Available in any standard book

$O(n)$ Level 0 : $\frac{n}{2}$ merges of size 1 each

$O(n)$ level 1 : $\frac{n}{4}$ merges of size 2 each

$O(n)$ Level $O(\log n)$:

$\boxed{O(n \log n)}$

Optimal Merge Sequence: Problem

MergeSequence (L)

{ $L = \{L_1, L_2, \ldots, L_n\}$

each L_i has $|L_i| = l_i$ elements
which are already sorted

GENERAL RECURSIVE METHOD

for each pair (L_i, L_j)

$L_{ij} = \text{Merge}(L_i, L_j)$

$L_R = L - \{L_i\} - \{L_j\} + L_{ij}$

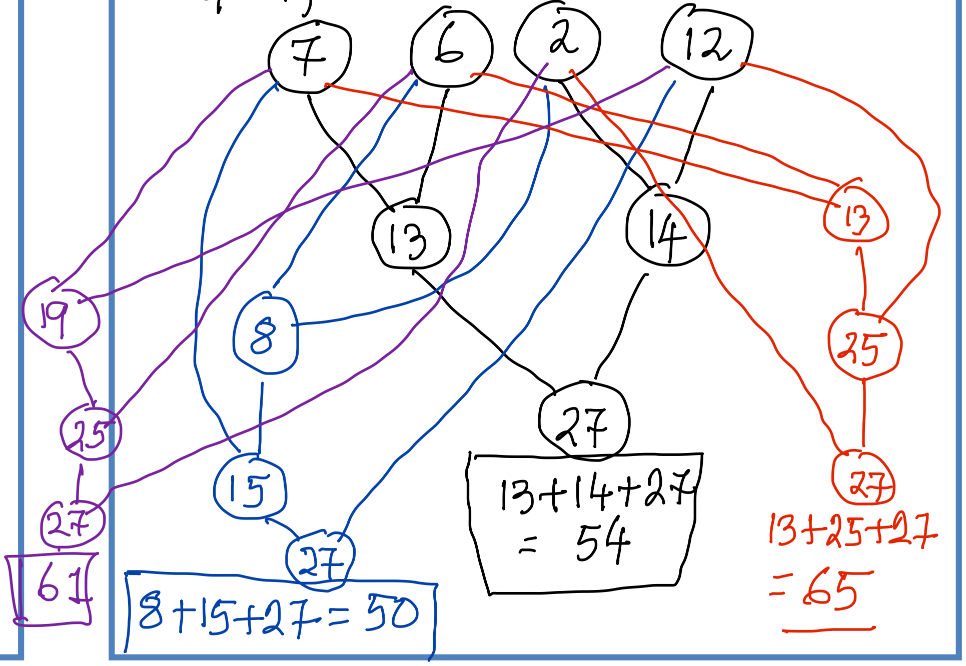
$M_{ij} = \text{MergeSequence}(L_R)$

Best Solution / option of choosing L_i & L_j

Example

Four Lists L_1, L_2, L_3, L_4

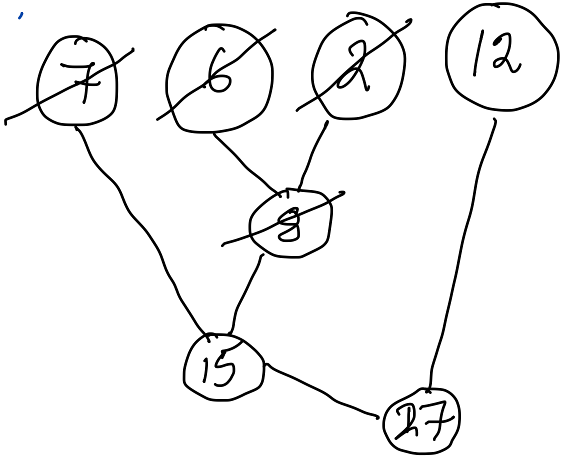
$l_1 = 7, \; l_2 = 6, \; l_3 = 2, \; l_4 = 12$



Optimal Merge Sequence: Algorithm & Choice

$L = \{L_1, L_2, \ldots, L_n\}$

choose: L_i and L_j such that l_i and l_j ($|L_i| = l_i$) ($|L_j| = l_j$) are the smallest sized sets in L.



1. Prove that this GREEDY choice always yields the optimal value.

2. Analyze the Time Complexity of this Algorithm

[Assignments / Homework / Tutorial]

Use of a GREEDY CHOICE METHOD

↳ is also another way of looking at bottom-up evaluation in Mergesort

Optimal Merge Sequence: Alternative

$L_1 = \{x_1, x_2, \ldots x_{l_1}\}$

$L_2 = \{y_1, y_2, \ldots, y_{l_2}\}$

$L_3 = \{z_1, z_2, \ldots, z_{l_3}\}$

$\vdots$

$L_n = \{ \quad \}$

$T(n) = T(n-1) + O(\log n) = O(n \log n)$

each of $O(\log n)$

1. Find the largest element from all the elements at the head of each list
2. Remove it and put it on the result
3. Recurse

$T(n) = T(n-1) + \boxed{O(n)}$

If we use a simple Max Data structure

Balanced BST to store only these header elements → REMOVE-MAX, INSERT.

Optimal Merge Sequence: Finalization

1. Greedy Algorithm → Data structure
2. Balanced BST based algo

(A) Leave it as a home work to determine which/whether any of them is better than the other

(B) Write down the final versions of both these algorithms

QuickSort

Quicksort (L)
{ if (|L| ≤ 1) return (L)
Let $L = \{x_1, x_2, \dots, x_n\}$
D y = choose (L)
/choose an element from L/
$L_1 = \{ z \mid z \le y \}$
$L_2 = \{ z \mid z > y \}$
M_1 = Quicksort (L_1)
M_2 = Quicksort (L_2)
R $M = M_2 \,||\, M_1$ (concatenation)
return (M)
}

$T(n) = T(k) + T(n-k) + O(n)$

In the worst case $k = 1$
↳ $O(n^2)$

How do we choose y from L to reduce the complexity from $O(n^2)$

Case 1: If we wish to divide L into almost equal halves
then y = MEDIAN (L)

Question: How to find Median in $O(n)$ time
↳ classical Algo.

Case 2: Randomly choose y from L.
↳ Average case: $O(n \log n)$

QuickSort: Analysis

→ Worst Case : $O(n^2)$

↳ Median Finding in $O(n)$ time : $O(n \log n)$

→ Average Case : $O(n \log n)$

$T(n) = \boxed{}$

Exercise / Look up the book for this analysis.

→ Average Case Analysis of Binary Search if we choose a random point

→ Average Case Analysis of Height of a BST which does not apply sophisticated balancing techniques

Time-Space Relationship

1. Max Removal Sorting	$O(n^2)$ Time	$O(1)$ Add^n space
2. Insertion Sort		BST
3. Merge Sort		
4. Quick Sort		

Space: → The additional space requirement beyond what we need to store the input

HEAP

In-place merge in $O(n)$

Other Approaches to Sorting

1. When elements are from a known domain

2. Alternative Data Structures
   ↳ Hash Tables

3. External Sorting

Summary

1. Sorting based on D&C Method

2. O(n log n) HEAP BST

3. Median Finding in O(n)

4. Optimal Merge Sequence Problem → Greedy Algorithm

5. Alternative Data Structures

6. Special Sorting $\boxed{\text{KNUTH'S BOOK}}$

Thank you

Any Questions?