ALGORITHM DESIGN USING DIVIDE & CONQUER METHOD: II

Partha P Chakrabarti

Indian Institute of Technology Kharagpur

Algorithm Design by Recursion Transformation

- Algorithms and Programs
- Pseudo-Code
- Algorithms + Data Structures = Programs
- Initial Solutions + Analysis + Solution Refinement + Data Structures = Final Algorithm
- Use of Recursive Definitions as Initial Solutions
- Recurrence Equations for Proofs and Analysis
- Solution Refinement through Recursion Transformation and Traversal
- Data Structures for saving past computation for future use

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 DIVIDE & CONQUER
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

Basics of Divide & Conquer Method

$f(X)$

{ Base $B(X) \rightarrow J(X)$

Recursive

1. Decomposition $D(X)$

$\langle X_1, X_2, X_3, \ldots, X_K \rangle = D(X)$

2. Recursive call

$\langle Y_1, Y_2, \ldots, Y_K \rangle = f(X_1, X_2, \ldots X_K)$

3. Recomposition $R(Y)$

$Z = \langle Z_1, Z_2, \ldots, Z_p \rangle = R(Y_1, Y_2, \ldots Y_K)$

Return (Z)

}

Decomposition & Recomposition

→ costs involved

→ Recurrence Relation for Time Complexity

→ Recurrence / Recursion Tree where nodes have costs corresponding to D, R

→ properties of D and R

Recursion Structure : Required insights on how the problem is solved by D&C and try to optimize our CHOICES in D and R

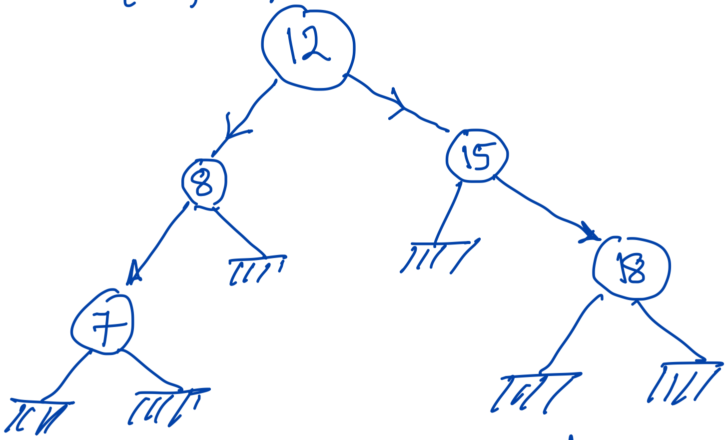
DATA STRUCTURES

Sorting & Searching Problems

Search (L, S)
where L is a set/List of elements from which we try to search.
S is the set of elements which we try to find in L.
L and S may be ordered or unordered
L is ordered and S is a single element $\longrightarrow$ BINARY SEARCH

Recursion Tree in ordered (L) based search $\longrightarrow$ Binary search Tree (BST) structure

{7, 8, 12, 15, 18}



Height Balanced BST that minimizes the length of the longest path is optimal in the worst case.

Another Data Structure called HEAP

Sorting by Max Removal

Sort1(L)
{ if |L| ≤ 1 return (L)

D $\boxed{\begin{array}{l} x = \text{Max}(L) \\ L_1 = L - \{x\} \end{array}}$ /Remove x from L/

$M_1 = \text{Sort1}(L_1)$

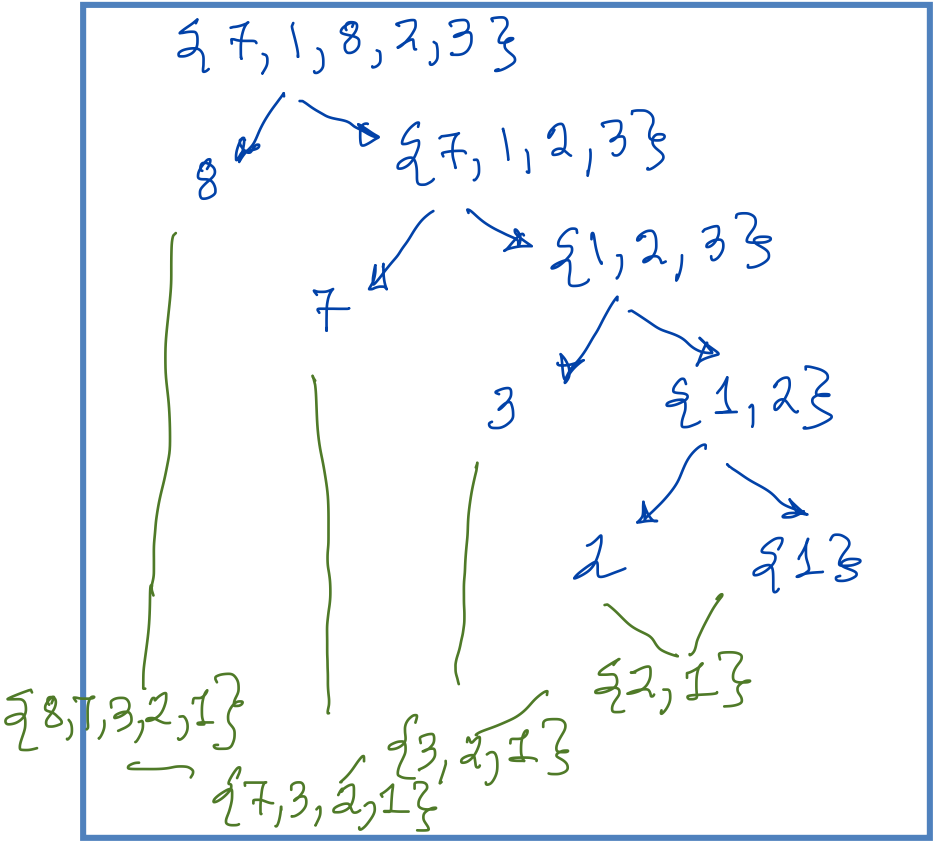
R $\boxed{M = \{x\} \,||\, M_1 \}}$

concat puts x in front of M_1

return(M)

}

$$T(n) = T(n+1) + O(n)$$
$$= O(n^2)$$



Sorting by Max Removal: Data Structure

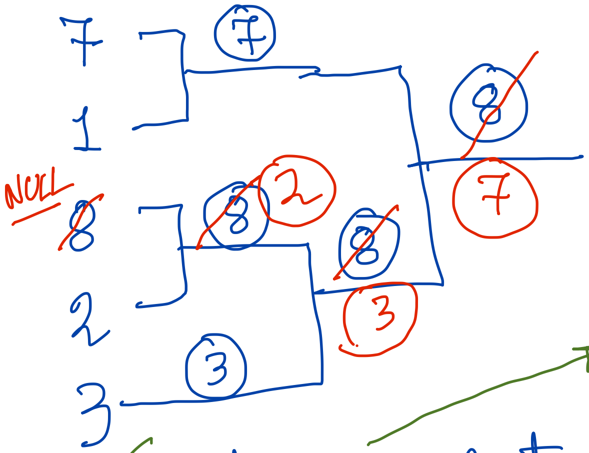
Max1-Max2

We can form a Data structure during the first Max (Tournament, HEAP) → knock-out comparison

Then onwards finding the rest of the max values is done on the HEAP / Tournament Data structure

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

$\{7, 1, 8, 2, 3\}$



first max → Create HEAP → $O(n)$

Next max onwards → Manipulate this HEAP → $O(\log n)$ time

Sorting by Max Removal: Finalization

sort_1_New (L)

{ if $|L| \leq 1$ return (L)

H = max1+HEAP (L)

(first max which creates HEAP H)

M = { }

for $i = 1$ to $|L|$ do

{ x_i = rem_max (H)

$M = M \| x_i$ (enqueue)

}

}

$O(n \log n)$

Insertion Sort

Sort 2 (L)

{ Let $L = \{x_1, x_2, \ldots, x_n\}$

B if $|L| \leq 1$ return (L)

D choose x_i from L

$L_1 = L - \{x_i\}$

$O(1)$

$O(n)$?

$M_1 = \text{sort 2}(L_1)$

R $M = \text{Insert}(M_1, x_i)$

↳ inserts x_i in its proper place in M_1 so that M_1 is sorted

return (M)

}

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

linear insertion $O(n)$

↳ $O(n^2)$

But can say that since M_1 is sorted, why don't we use Binary search?

Suppose M_1 is an array

↳ $O(\log n)$ [Find]

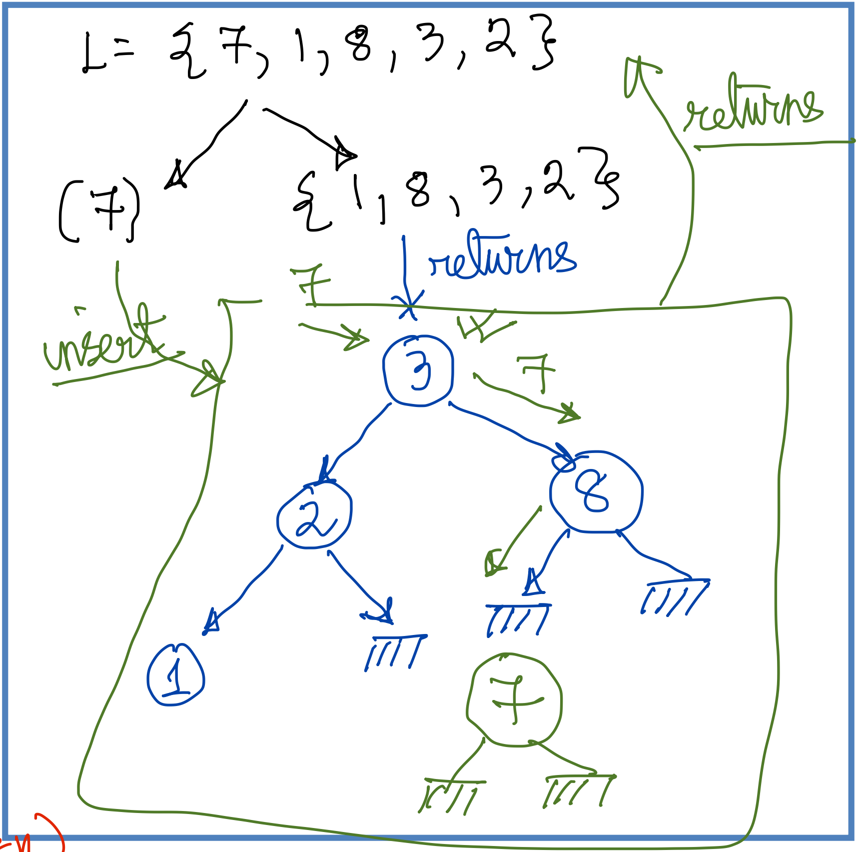
But in an array the insertion is an issue

Insertion Sort: Data Structure

How are we going to store the ~~em~~ elements of M_1 so that insertion can be done in $O(\log n)$ time?

→ Binary Search Tree Data structure (for M_1)

Traversal of the BST will provide us with the sorted list → only once at the end $O(n)$ ✓

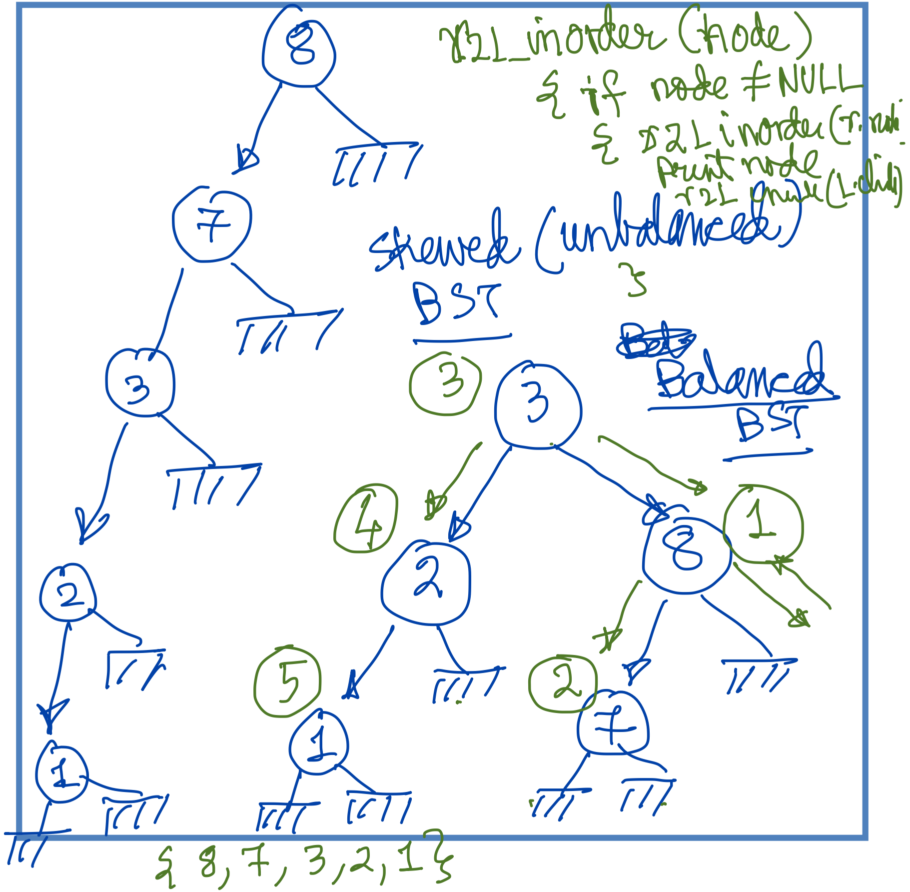


$T(n) = T(n-1) + O(n) = O(n \log n)$

Insertion Sort: Finalization

sort_M2_final (L)

1. BST data structure
   - ↳ insert (can be written recursively)
   - → traversal (Recursive)
     - pre-order
     - ~~and~~ inorder
     - post order
   - left to right or right to left
2. Balanced BST → height O(log n)



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

Thank you

Any Questions?