

ALGORITHM DESIGN USING **DIVIDE** & **CONQUER** METHOD



Partha P Chakrabarti

Indian Institute of Technology Kharagpur

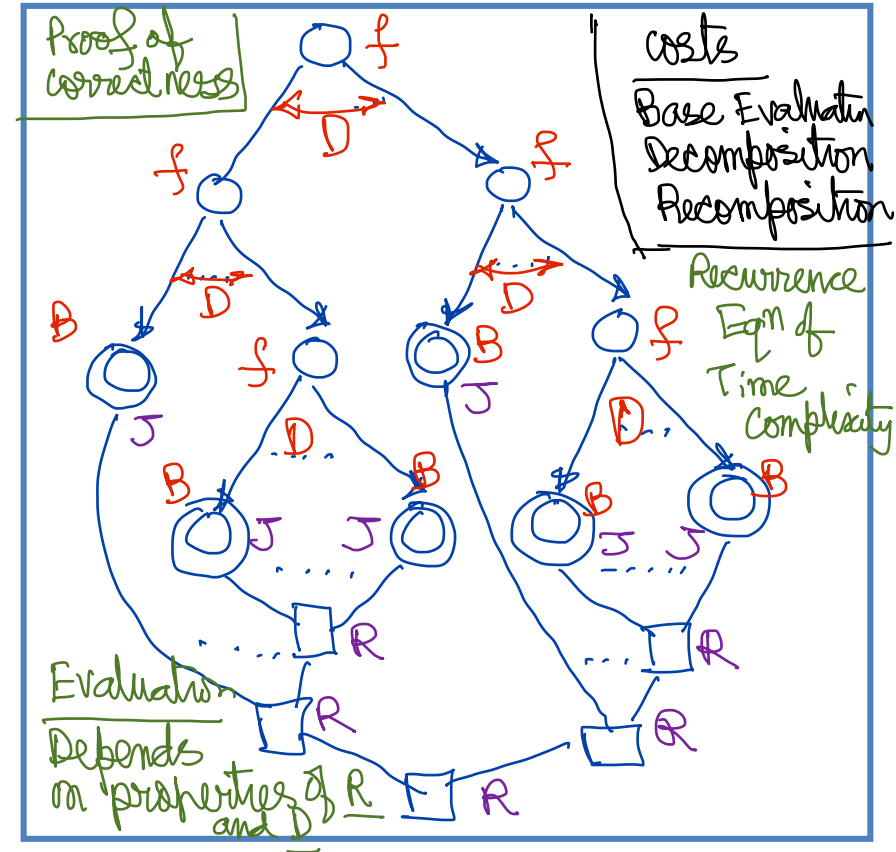
Algorithm Design by Recursion Transformation

- ❑ 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
3. Choice of Solution & Complexity Analysis
 - a. Balancing the Split, Choosing Paths *Divide & Conquer*
 - 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

Structure of Recursive Definition

- $f(x)$
1. Base Condition $B(x)$
 if $B(x)$ return $(J(x))$
 2. Decomposition $D(x)$
 $\langle x_1, x_2, \dots, x_k \rangle = D(x)$
 3. Recursive Call
 for each i , $y_i = f(x_i)$
 4. Recomposition $R(y)$
 $J = R(y_1, y_2, \dots, y_k)$
 5. Returning the Result
 Return (J)



Basics of Divide & Conquer Method

1. Mechanism of Decomposition and the CHOICE POINTS

$$T(n) = T(n_1) + T(n_2) + \dots + T(n_k) \\ + \text{decomposition}(n) \\ + \text{recomposition}(k)$$

various alternatives of

D and R

properties of D and R

choice points : how to optimally split the problem

2. Evaluation

3. Dynamic Programming

→ solving identical subproblems once and memorizing the past solutions

4. Data Structuring

Organizing data of past computation for future use

5. Branch & Bound

Pruning of unnecessary paths in the computation of the recursion tree

Basics of Divide & Conquer Method

CHOICE OF D (Decomposition)
and R (Recomposition)

→ Time Complexity
 ↳ optimize this time complexity

Analysis

1. By analysis of the recurrence eqn formed

2. By Analysis of the Recursion Tree and costs (computation) incurred during D and R

→ split of D
(max, max-min, max-2nd max)
(coins)

3. what structures are formed which enable us to develop alternative insights for an iterative solution to the recursive defⁿ

Sorting & Searching Problems

Searching

1. unordered set or list
2. ordered list or sequence
3. A combination
4. static / dynamic

Sorting

1. Ascending or Descending Order
2. In-Memory Sorting
3. Secondary Storage Sorting

Searching an Unordered List

Search-U(L, x)

{ Let $L = \{s_1, s_2, \dots, s_n\}$

if ($|L| = 0$) return (false)

if ($|L| = 1$)
if ($s_1 = x$) return (true)
else return (false)

split L into non-empty L_1, L_2

if (Search-U(L_1, x))
return (true)

else return (Search-U(L_2, x))

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

choice: Any split is equal in complexity

choose $\underline{1}, \underline{n-1}$

Solution:

An iterative loop

for $i = 1$ to n do

{ if ($x = s_i$) return (true)

} return (false)

Searching an Unordered List: Finalization

Finalization

1. choice
2. check whether this is optimal
↳ unless every element is checked in the WORST CASE we may not find the correct answer by any algo.

VARIANT

1. Searchall(L, S) ✓✓

/where we have

$$L = \{l_1, l_2, \dots, l_n\}$$

$$S = \{s_1, s_2, \dots, s_m\}$$

Both unordered
and we have to return
the elements of S that are
in L .

$\{L \cap S\} \rightarrow$ how do
we solve
this problem

Searching Ordered Lists

```

search_0(L, x)
{
  L = {s1, s2, ..., sn}
  [si ≤ sj for j > i]
  B: if |L| = 0 return (False, 0)
  D: choose (si)
      if (si = x) return (true, i)
      else
        if (si < x)
          {
            L' = {si+1, ..., sn}
            <y, z> = search_0(L', x)
            if (y) return (true, z+i)
            else return (False, 0)
          }
        else
          {
            L' = {s1, ..., si-1}

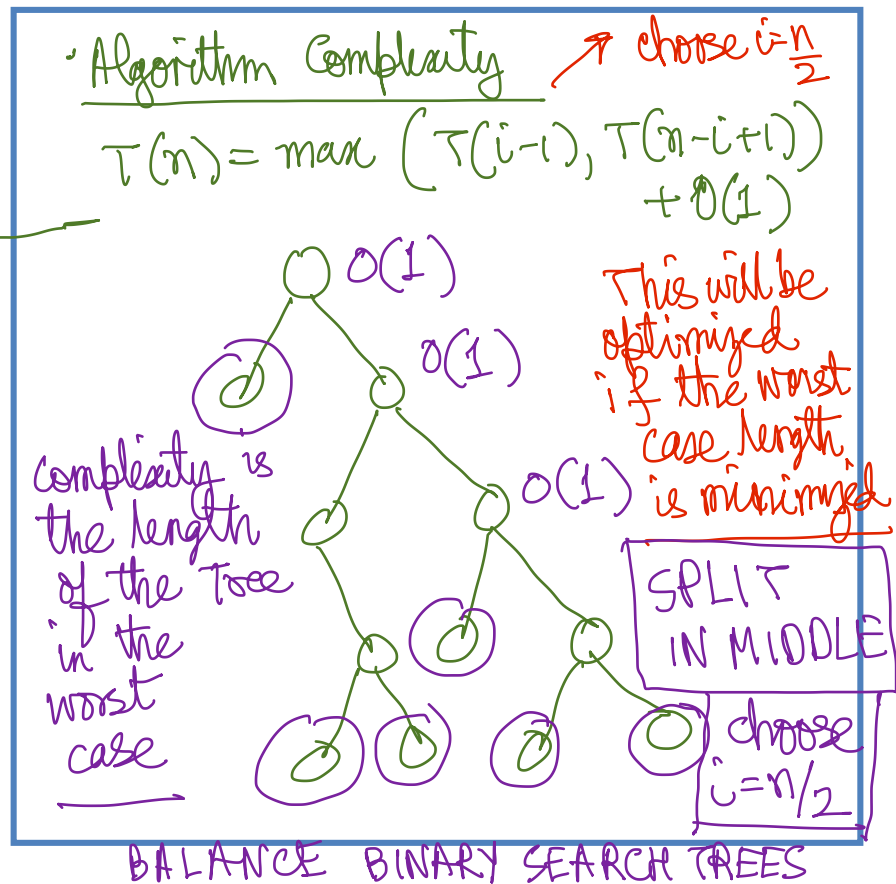
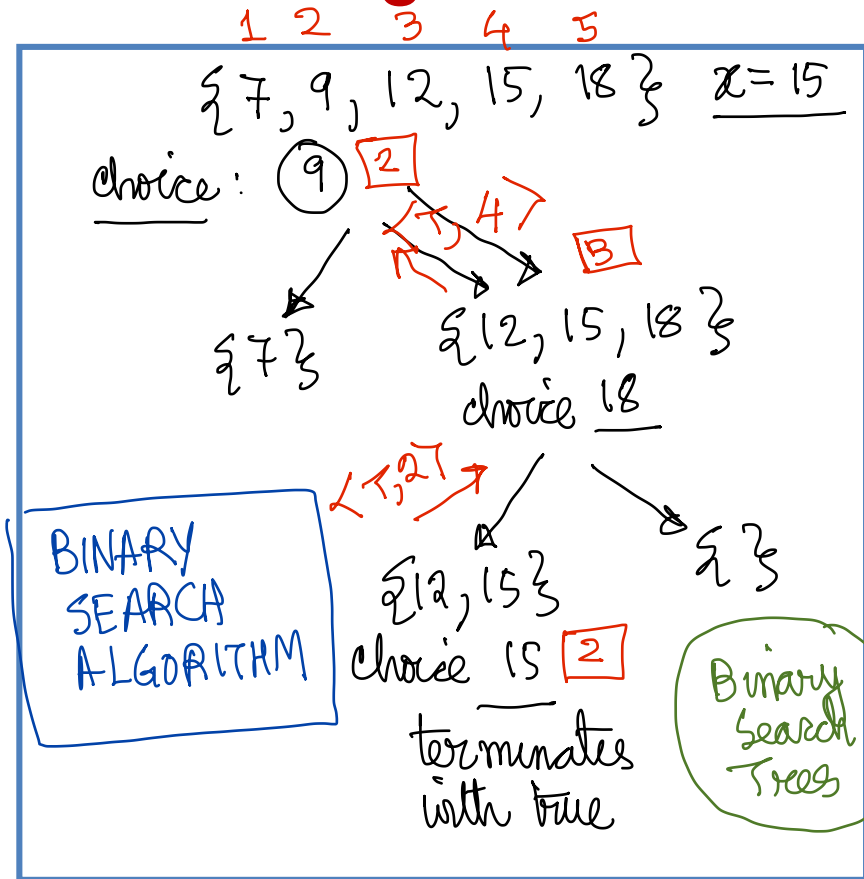
```

```

  <y, z> = search_0(L', x)
  if (y) return (<true, z>)
  else return (False, ∞)
}
}
D: choice of i
R: OR → compute only in
    one direction
T(n) = max{T(i-1), T(n-i+1)} + O(1)

```

Searching Ordered Lists: Data Structure View



Problem Variations and Approaches

$$T(n) = T(n/2) + O(1) \\ = O(\log n) \quad \checkmark$$

which is the height of the
Balanced Recursion Tree in
the worst case

Other options

$$T(n) = \max \left\{ T\left(\frac{1}{2}n\right), T\left(\frac{2}{3}n\right) \right\} + O(1) \\ = T\left(\frac{2}{3}n\right) + O(1) \\ = O(\log n)$$

$T(n) \rightarrow$ into 3 parts and
done 2 comparisons

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

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

VARIATIONS

- 1. Search-0 (L, S) $\nearrow$ ordered
- 2. Dynamic Search $\nearrow$ ordered or unordered
 - $\hookrightarrow$ interleaved queries
 - (insert, find) into L in L (insert, delete, find)

Sorting & Searching Problems

Search (L, S)

where L is a set/list of elements from which we try to search.

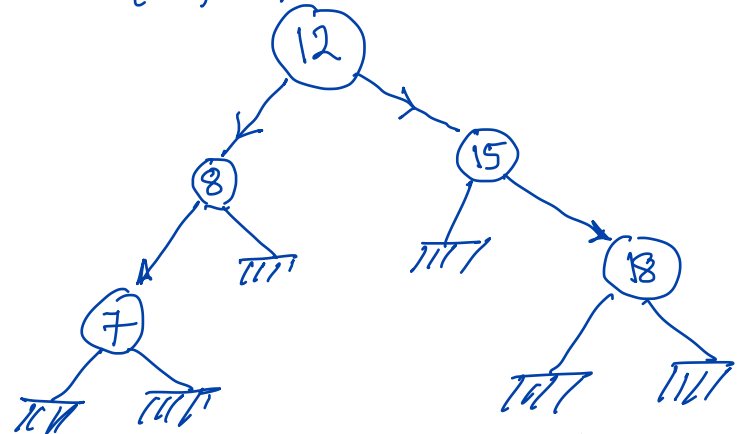
S is the set of elements which we try to find in L.

L and S may be ordered or unordered

L is ordered and S is a single element $\rightarrow$ BINARY SEARCH

Recursion Tree in ordered (L) based search $\rightarrow$ 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 $|L| \leq 1$ return(L)

$x = \text{Max}(L)$

$L_1 = L - \{x\}$ / Remove x from L

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

$M = \{x\} \parallel M_1$

concat puts x in front of M_1

return(M)

}

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

$$= O(n^2)$$

{ 7, 1, 8, 2, 3 }

8 → { 7, 1, 2, 3 }

7 → { 1, 2, 3 }

3 → { 1, 2 }

2 → { 1 }

{ 2, 1 }

{ 8, 7, 3, 2, 1 }

{ 7, 3, 2, 1 }

{ 3, 2, 1 }

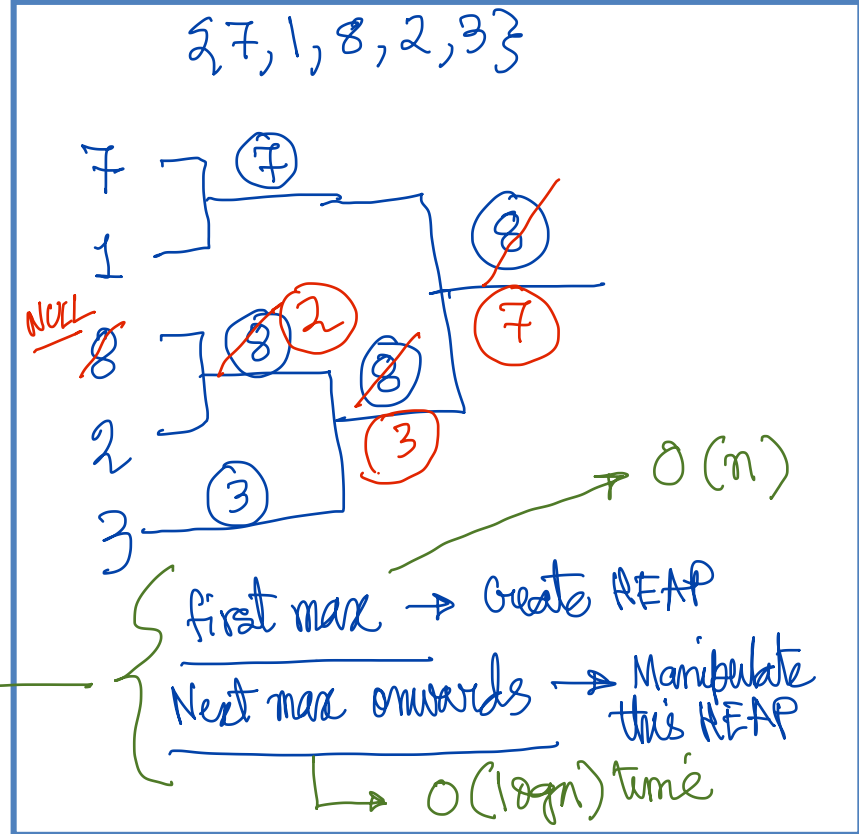
Sorting by Max Removal: Data Structure

Max 1 - Max 2

We can form a Data Structure during the first Max (Tournament, HEAP) $\rightarrow$ knock-out comparison

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

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



Sorting by Max Removal: Finalization

```
sort-by-New (L)
{
  if  $|L| \leq 1$  return (L)
  H = max-1+HEAP (L)
    (first max)
  M = { } which creates HEAP H
  for  $i = 1$  to  $|L|$  do
    {  $x_i = \text{rem-max}(H)$ 
      M = M ||  $x_i$  (enqueue)
    }
}
```

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

Insertion Sort

sort 2(L)

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

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

D choose x_i from L $O(1)$
 $L_1 = L - \{x_i\}$

$\rightarrow M_1 = \text{sort 2}(L_1)$ $O(n)?$

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

$\rightarrow$ inserts x_i in its proper place in M_1 so that M is sorted

} return (M)

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

linear insertion $O(n)$

$\rightarrow O(n^2)$

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

Suppose M_1 is an array

$\rightarrow O(\log n)$ [Find]

But in an array the insertion is an issue

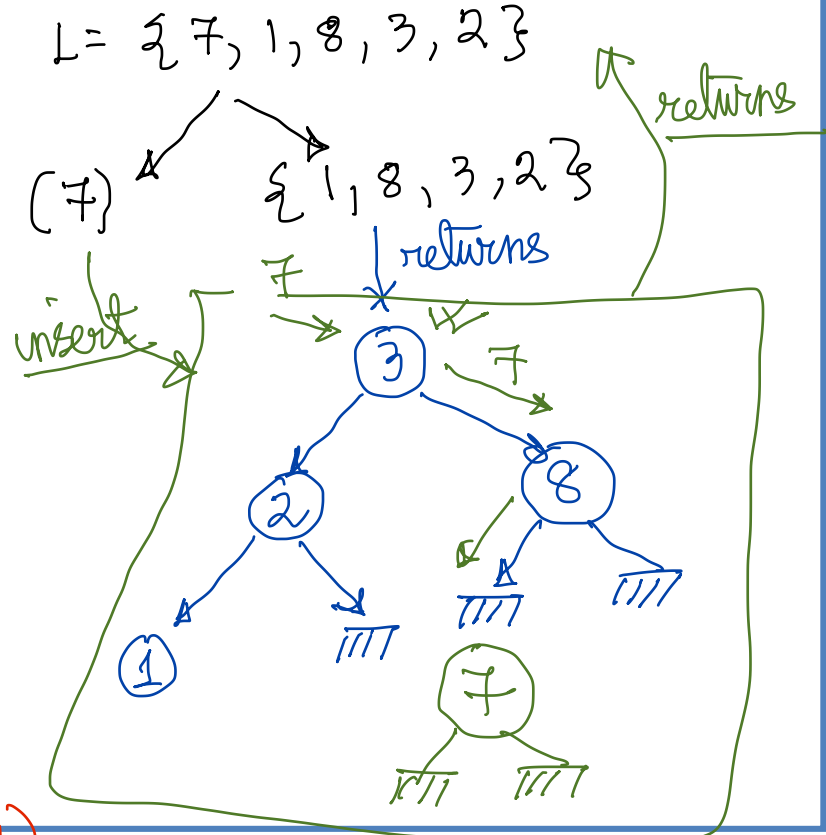
Insertion Sort: Data Structure

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

→ Binary Search Tree
Data structure (for M)

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

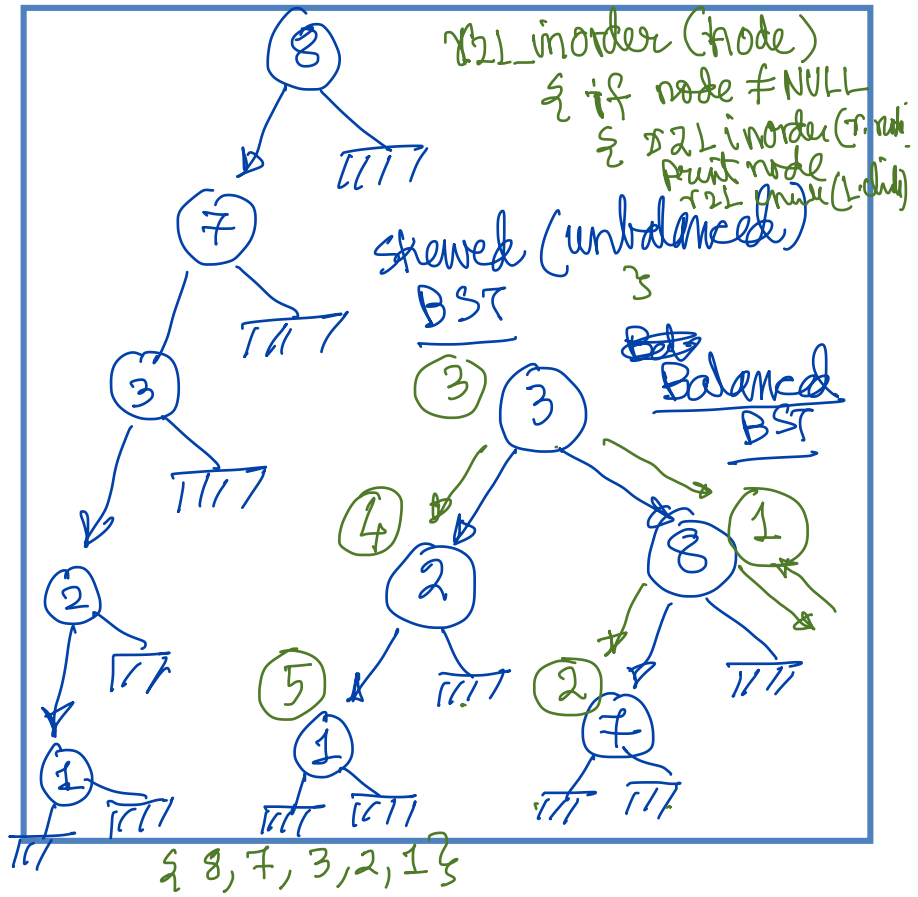
↳ insert (can be written recursively)

→ Traversal (Recursive)

{ [pre-order
 { [~~in~~ in order [
 { [post order

left to right or
right to left

2. Balanced BST $\rightarrow$ height $O(\log n)$



Sorting by Divide & Conquer Method

sort (L)

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

Decomposition (choice points)

split L into non-empty L_1, L_2
Appropriately choose an element

Recursion

$M_1 = \text{sort}(L_1)$
 $M_2 = \text{sort}(L_2)$

Recomposition

Mechanism to combine M_1 & M_2 to get M
return (M)

Method 1 :-

Decomposition :- Remove Max $O(n)$

$L_1 = \text{max}(L)$

$L_2 = L_1 - \text{max}(L)$ $O(1)$

Recomposition :- concat. (M_1, M_2) $O(1)$

Method 2 :-

Decomposition :- Remove an element

$L = \{x_i\}$, $L_2 = L - L_1$ $O(n)$

Recomposition :- Insert x_i in M_2

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

Data Structures { $\rightarrow$ HEAP $\rightarrow$ Balanced BST } $O(n \log n)$

MergeSort

Mergesort (L)

{ if ($|L| \leq 1$) return (L)
split L into L_1, L_2 which are
non-empty

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

$M_2 = \text{Mergesort}(L_2)$

$M = \text{Merge}(L_1, L_2)$

return (M)

$$T(n) = T(n/k) + T(n - \frac{n}{k}) + O(n^{\frac{1}{k}})$$

choose $k=2 \rightarrow O(n \log n)$
optimal performance

Merge (L_1, L_2)

{ if ($L_1 = \text{NULL}$) return (L_2)

if ($L_2 = \text{NULL}$) return (L_1)

let $L_1 = \{x_1, x_2, \dots, x_n\}$

$L_2 = \{y_1, y_2, \dots, y_m\}$

if ($x_1 > y_1$)

$L = \{x_1\} \parallel \text{Merge}(L_1 - \{x_1\}, L_2)$
↑ concat

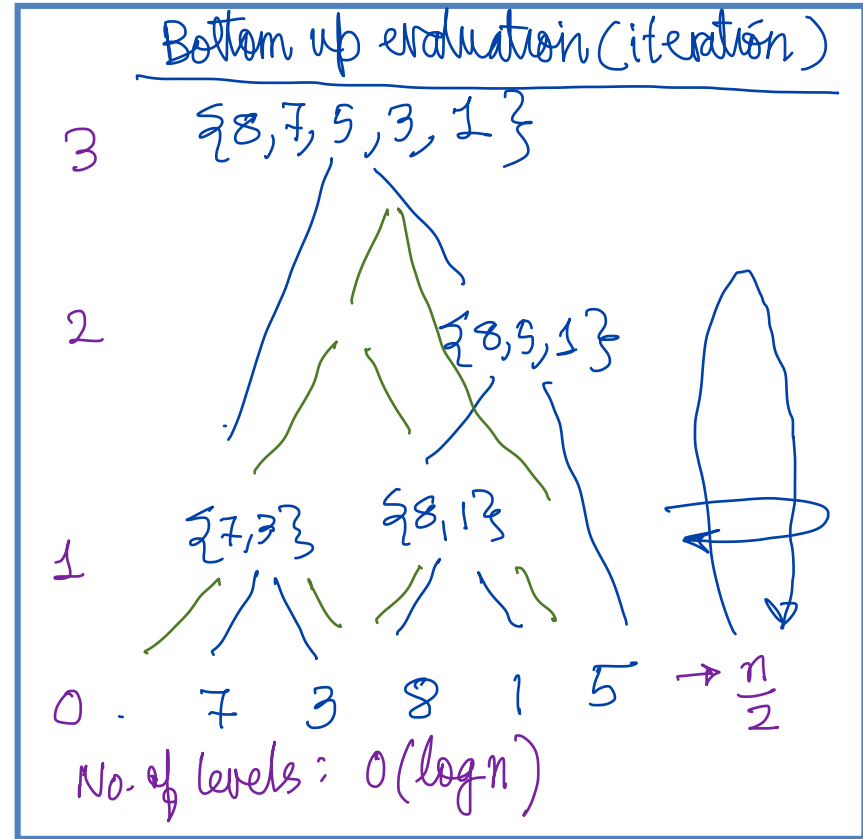
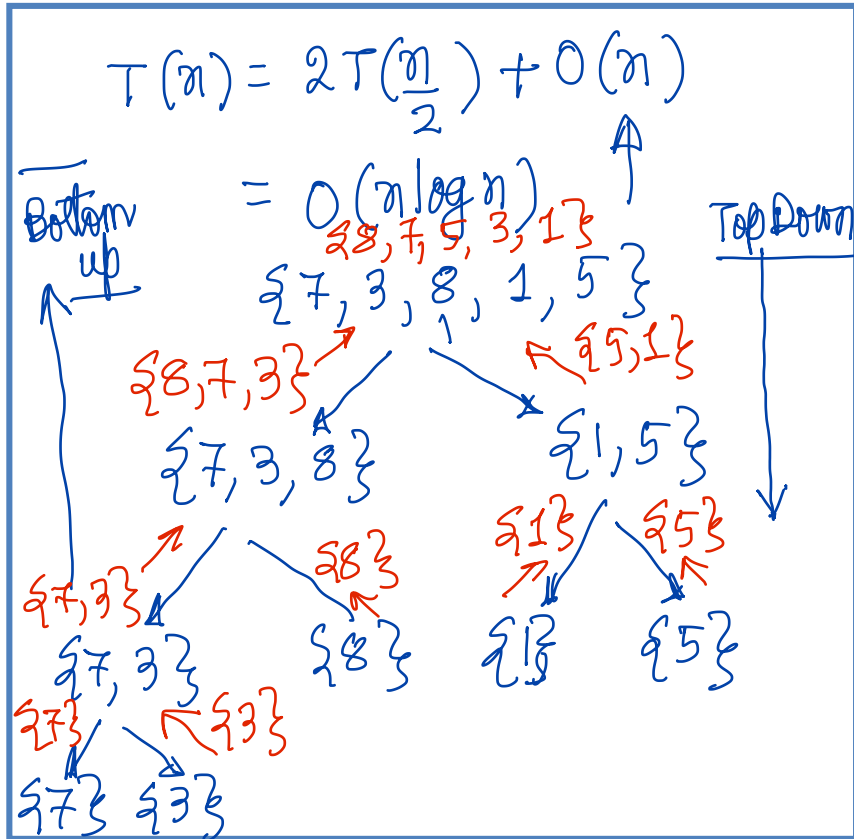
else

$L = \{y_1\} \parallel \text{Merge}(L_1, L_2 - \{y_1\})$

return (L)

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

MergeSort: Analysis



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

Optimal Merge Sequence: Problem

Merge Sequence (L)

$\{ L = \{ L_1, L_2, \dots, L_n \}$
 each L_i has $|L_i| = l_i$ elements
 which are already sorted

GENERAL RECURSIVE METHOD

for each pair (L_i, L_j)

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

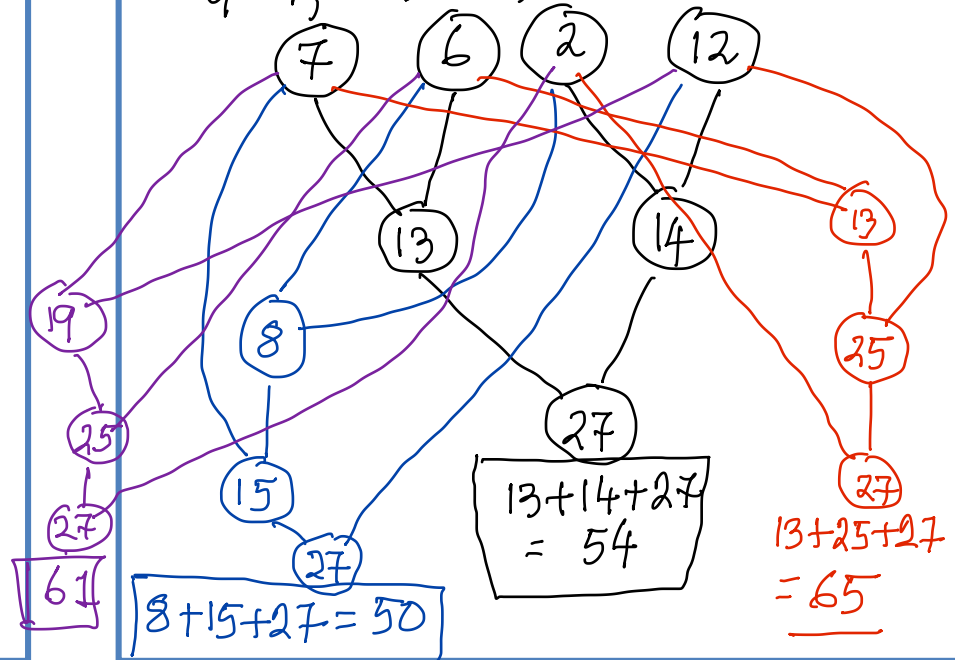
$L_R = L - \{L_i\} - \{L_j\} + L_{ij}$

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

Best Solution / option of
 choosing L_i & L_j

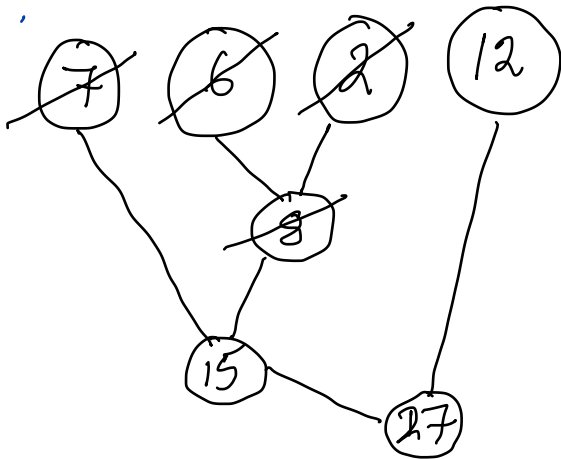
Example

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

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

$L_1 = \{x_1, x_2, \dots, x_{l_1}\}$
 $L_2 = \{y_1, y_2, \dots, y_{l_2}\}$
 $L_3 = \{z_1, z_2, \dots, z_{l_3}\}$
 $\vdots$
 $L_n = \{ \quad \quad \quad \}$

1. Find the largest element from all the elements at the head of each list
2. Remove it and put it in 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 $\rightarrow$ REMOVE-MAX
 $\rightarrow$ INSERT.

$$\begin{aligned}
 T(n) &= T(n-1) \\
 &\quad + O(\log n) \\
 &= O(n \log n)
 \end{aligned}$$

each op
 $O(\log n)$

Optimal Merge Sequence: Finalization

1. Greedy Algorithm $\rightarrow$ 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 ($|L| \leq 1$) return (L)

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

① $y = \text{choose}(L)$
 /choose an element from L /

$L_1 = \{x \mid x \leq y\}$

$L_2 = \{x \mid x > y\}$

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

$M_2 = \text{Quicksort}(L_2)$

② $M = M_2 \parallel M_1$ (concatenation)

 return (M)

}

$$T(n) = T(k) + T(n-k) + \underline{O(n)}$$

In the worst case $k=1$
 $\rightarrow O(n^2)$

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 $y = \text{MEDIAN}(L)$

Question: How to find Median
in $O(n)$ time

$\rightarrow$ classical Algo.

Case 2: Randomly choose y from L .

$\rightarrow$ Average case: $O(n \log n)$

QuickSort: Analysis

→ Worst Case : $O(n^2)$

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

→ Average Case : $O(n \log n)$

$T(n) =$

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'l space
2. Insertion Sort	<u>BST</u>
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 $\rightarrow$ Greedy Algorithm
5. Alternative Data Structures
6. Special Sorting KNUTH'S BOOK

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and

Let $T(n)$ be defined on nonnegative integers by the recurrence

$T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

$T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
and if, for some constant $c < 1$ and all sufficiently large n ,
we have $a \cdot f(n/b) \leq c f(n)$, then $T(n) = \Theta(f(n))$.

Multiplication of Two n-bit Numbers

$$x = 101001 \quad n\text{-bit}$$

$$y = 101010 \quad n\text{-bit}$$

$$\begin{array}{r}
 \cancel{101001.0} \leftarrow \\
 1010010.1 \leftarrow \\
 \cancel{10100100.0} \\
 101001000.1 \\
 \cancel{1010010000.0} \\
 10100100000.1
 \end{array}$$

$O(n^2)$

$$11010111010 \text{ Result}$$

$$x \quad \boxed{x_1 \quad x_2}$$

$$y \quad \boxed{y_1 \quad y_2}$$

$\xleftarrow{\frac{n}{2}} \quad \xrightarrow{\frac{n}{2}}$

$$x = x_1 * 2^{n/2} + x_2$$

$$y = y_1 * 2^{n/2} + y_2$$

$$x \cdot y = 2^n \underbrace{x_1 y_1}_1 + 2^{n/2} (\underbrace{x_1 y_2}_2 + \underbrace{x_2 y_1}_3) + \underbrace{x_2 y_2}_4$$

$$T(n) = 4T(n/2) + O(n)$$

$O(n^2)$

$$A = x_1 y_2 + x_2 y_1$$

$$= (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2$$

$$xy = 2^n \underbrace{x_1 y_1}_1 + 2^{n/2} A + \underbrace{x_2 y_2}_2$$

$$T(n) = 3T(n/2) + O(n)$$

$$= O(n^{\log_2 3}) = O(n^{1.59})$$

Strassen's Algorithm for Matrix Multiplication

Let $A = n \times n$ matrix, $B : n \times n$

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array} \right] \times \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array} \right]$$

$$= \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array} \right]$$

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

$$= O(n^3)$$

$$p_1 = a(f-h), \quad p_2 = (a+b)h$$

$$p_3 = (c+d)e, \quad p_4 = d(g-e)$$

$$p_5 = (a+d)(e+h) \quad p_6 = (b-d)(g+h)$$

$$p_7 = (a-c)(e+f)$$

$$\left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array} \right]$$

$$T(n) = \boxed{7} T\left(\frac{n}{\boxed{2}}\right) + O(n^2)$$

$$O(n^{\log_2 7}) = O(n^{2.81})$$

Closest Pair of Points

Given a set of n points in a 2-D plane, find the closest pair of points. [General version is in some d -D]

straightforward Method: $O(n^2)$

closestpair(S)

Let $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

split S into 2 disjoint non-empty subsets S_1 & S_2

$R_1 = \text{closestpair}(S_1)$

$R_2 = \text{closestpair}(S_2)$

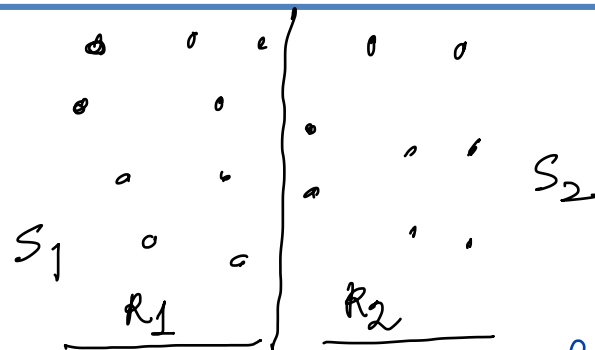
Let $R = \min(R_1, R_2)$

$R_3 = \text{combine}(S_1, S_2, R)$

return(R_3)

$\frac{O(1)}{O(n)}$

$\frac{f(n)}{O(n)}$



$$T(n) = T(n_1) + T(n_2) + f(n)$$

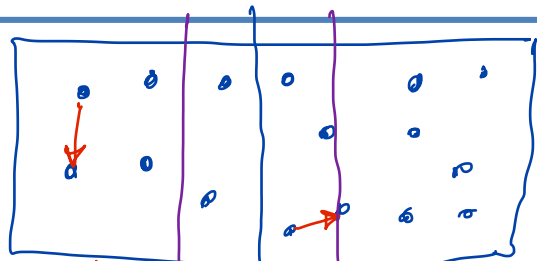
$$= 2T(n/2) + f(n) \leftarrow$$

1. Divide into 2 equal parts
2. Divide randomly [check for av. case]

Median along x -axis

sorting $O(n \log n)$
Median Finding $O(n)$

Closest Pair of Points: Strip Combine



$$R = \min(R_1, R_2)$$

Examine only those points along this $2K$ strip on the boundary.

$$Z = \text{strip}(S_1, K) \cup \text{strip}(S_2, K)$$

Sort (Z) by the y -axis

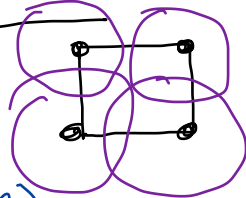
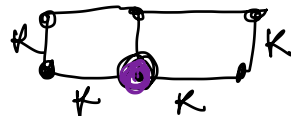
$$Z = \{q_1, q_2, \dots, q_n\}$$

For each q_i we need to check which points are there within K distance and if so find the min

Does this comparison require every point in Z to be compared with many or $O(n)$ other points in Z ?

NO

Theorem: For each q_i we need to check at most 7 points



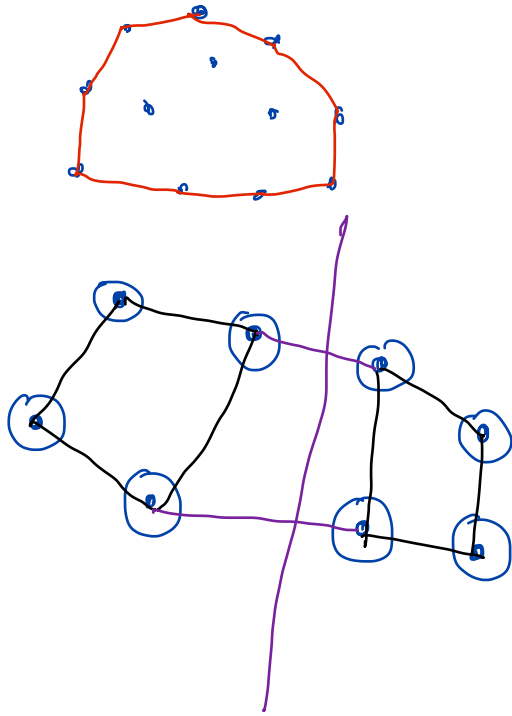
$$T(n) = 2T(n/2) + f(n)$$

Case 1: $f(n) = O(n \log n)$

Case 2: Median finding in $O(n)$
 x -axis & y -axis sorting is done once globally: $O(n) \rightarrow f(n) = O(n)$
 $= O(n \log n)$

Higher or d -dimensions $O(n \log^{d-1} n)$

Finding the Convex Hull



convexHull(S) median

{ split S into S_1, S_2 ($O(n)$)

$H_1 = \text{convexHull}(S_1)$

$H_2 = \text{convexHull}(S_2)$

$H_3 = \text{combine}(H_1, H_2)$

return(H_3) $\rightarrow O(n)$

}

$T(n) = 2T(n/2) + O(n)$

$O(n \log n)$

Summary

Balancing the split

↳ Decomposition $\rightarrow f_1(n)$

Recursion $\rightarrow a, b$

Recomposition $\rightarrow f_2(n)$

$$T(n) = a T(n/b) + f_1(n) + f_2(n)$$

optimizing the ratio of a/b
and the f_1, f_2 or
 $\max(f_1(n), f_2(n))$

Classical Problems

MEDIAN FINDING in $O(n)$
TIME

↳ which we shall discuss
in a subsequent class.

Thank you

Any Questions?