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
   b. Identical Sub-problems

   Divide & Conquer
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)

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

3. Recursive Call

for each i, $Y_i = f(X_i)$

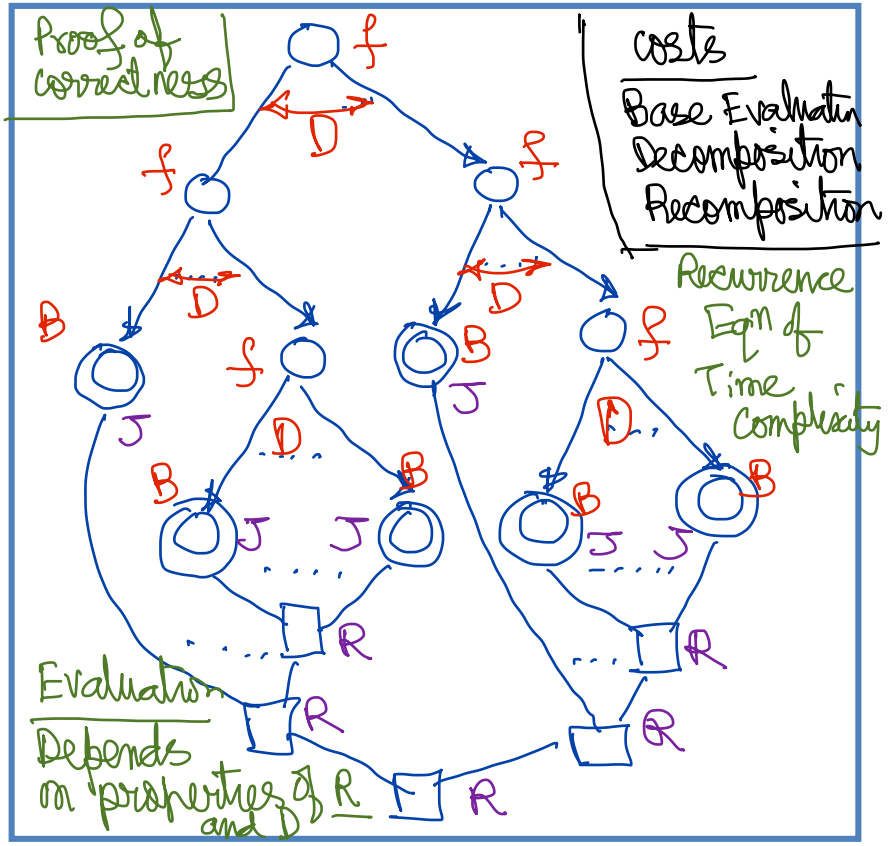
4. Recomposition R(Y)

$J = R(Y_1, Y_2, \ldots, Y_k)$

5. Returning the Result

Return (J)

}



Basics of Divide & Conquer Method

1. Mechanism of Decomposition and the CHOICE POINTS

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

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 = {s1, s2, ..., sn}
  if(|L| = 0) return (false)
  if(|L| = 1)
      if (s1 = x) return (true)
      else return (false)
  split L into non-empty L1, L2
  if(Search-U(L1, x))
      return (true)
  OR
  else return (Search-U(L2, x))
}
```

B

D

R

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

Choice: Any split is equal in complexity

→ choose 1, n-1

Solution:

An iterative loop

```
for i = 1 to n do
  { if (x = si) 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. SearchUL(L, s) ✓✓
   / where we have
   $L = \{l_1, l_2, \ldots, l_n\}$
   $s = \{s_1, s_2, \ldots, s_m\}$
   Both unordered and we have to return the elements of s that are in L.
   $\{L \cap S\}$ → how do we solve this problem

Searching Ordered Lists

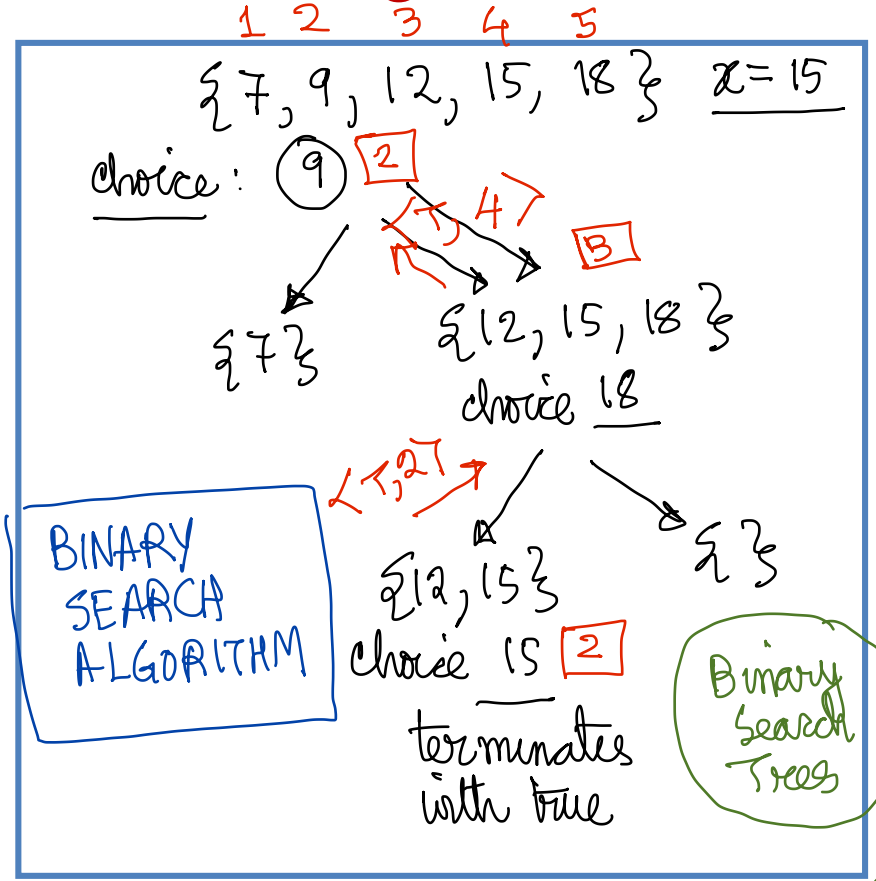
```
search_O(L, x)
{ L = {s_1, s_2, ..., s_n}.   [s_i ≤ s_j for j > i]
  B: If |L| = ∅ return (False, ∅)
  D: choose (s_i)
     if (s_i = x) return (true, i)
     else
       if (s_i < x)
         { L' = {s_{i+1}, ..., s_n}
           ⟨y, z⟩ = search_O(L', x)
           if (y) return (true, z+i)
           else return (false, ∅)
         }
       else { L' = {s_1, ..., s_{i-1}}
              ⟨y, z⟩ = search_O(L', x)
              if (y) return (⟨true, z⟩)
              else return (false, ∅)
            }
     }
}
```

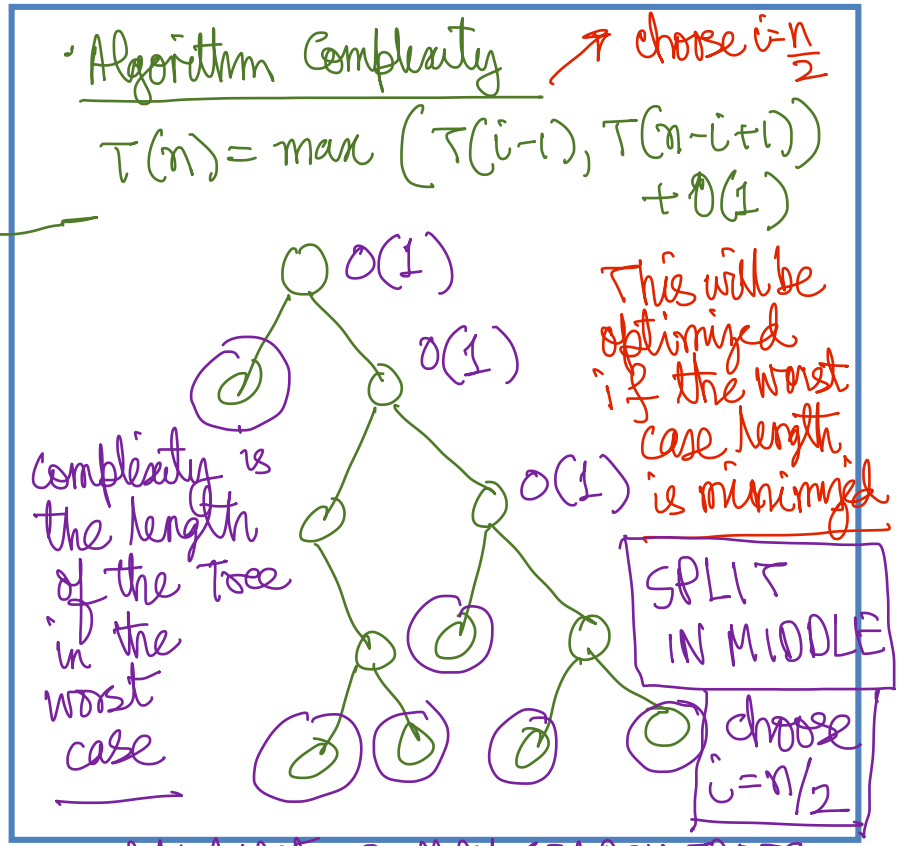
D: choice of i

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

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

$= O(\log n)$

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

Other options

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

$= T(\frac{2}{3}n) + O(1)$

$= O(\log n)$

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

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

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

VARIATIONS

1. Search-0 (L, S)
   - ordered
   - Ordered or Unordered
2. Dynamic search
   - L → interleaved queries
     - (insert into L, find 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.

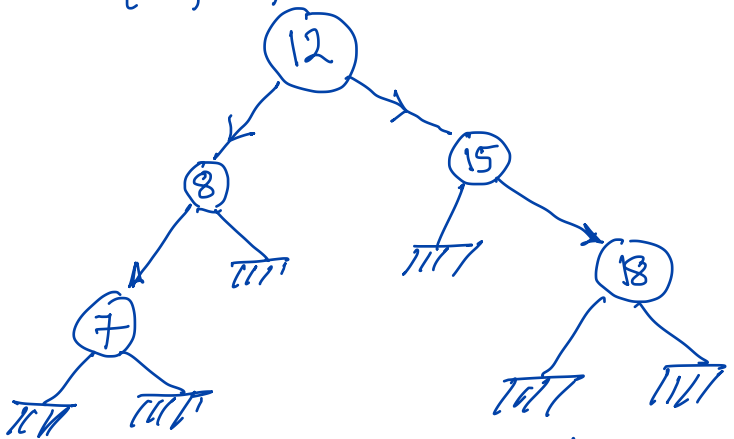
L and S may be ordered or unordered

L is ordered and S is a single element → BINARY SEARCH

Another Data Structure called HEAP

Recursion Tree in ordered (L) based search → Binary search Tree (BST) structure

{7, 8, 12, 15, 18}

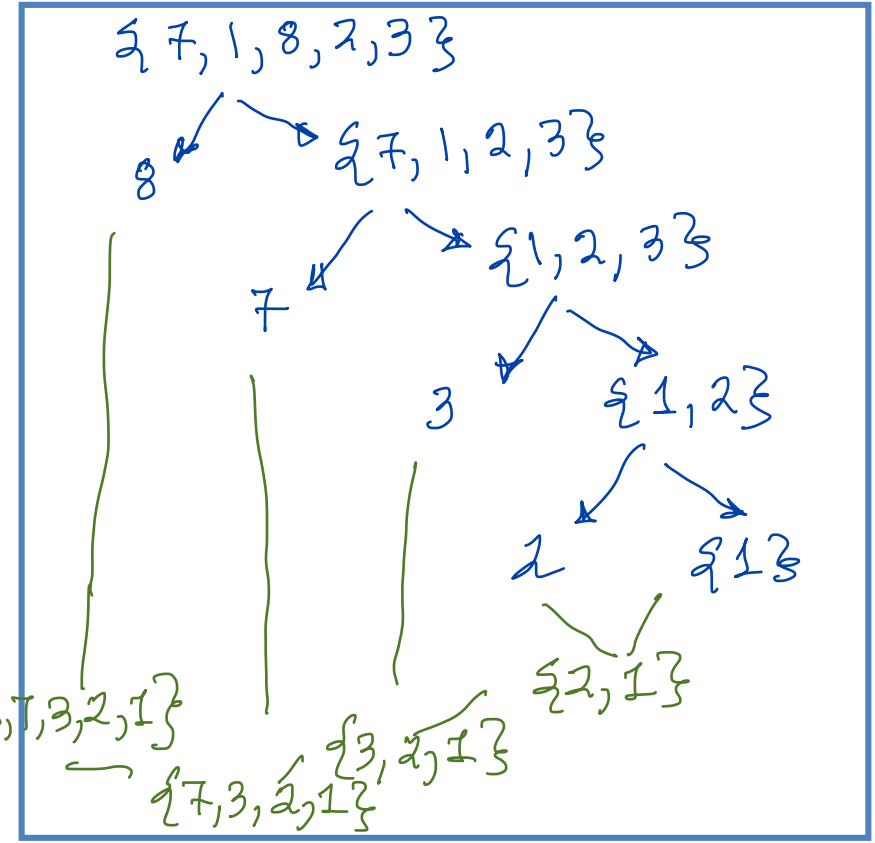


Height Balanced BST that minimizes the length of the longest path is optimal in the worst case.

Sorting by Max Removal

```
Sort1(L)
{ if |L| ≤ 1  return (L)
  x = Max(L)                          D
  L1 = L − {x}   / Remove x from L /
  M1 = Sort1(L1)
  M = {x} || M1                       C
        ↑ concat puts x in front of M1
  return (M)
}
```

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



Sorting by Max Removal: Data Structure

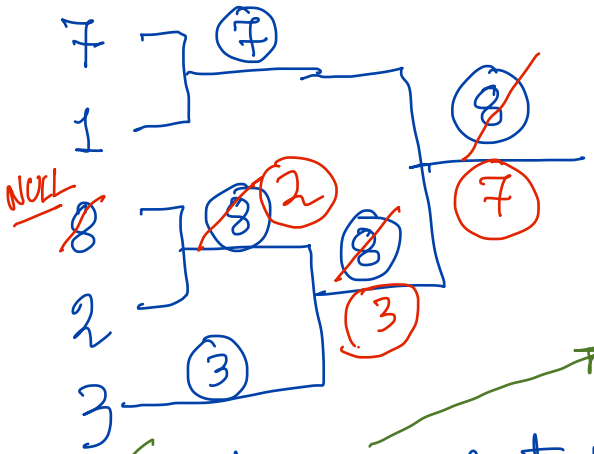
Max 1 - Max 2

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

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| ≤ 1 return (L)
    H = max1+HEAP (L)
        (first max)
    M={} which creates HEAP H)
    for i = 1 to |L| do
    { xi = rem_max (H)
      M = M || xi (enqueue
    }
}
```

$O(n \log n)$

Insertion Sort

```
Sort 2 (L)
{ Let L = {x1, x2, ..., xn}
  B  If |L| ≤ 1 return (L)
  D  choose xi from L                    O(1)
     L1 = L - {xi}
     M1 = sort 2 (L1)                    O(n) ?
  R  M = Insert (M1, xi)
       ↳ inserts xi in its proper place in M1 so that M is sorted
  return (M)
}
```

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

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

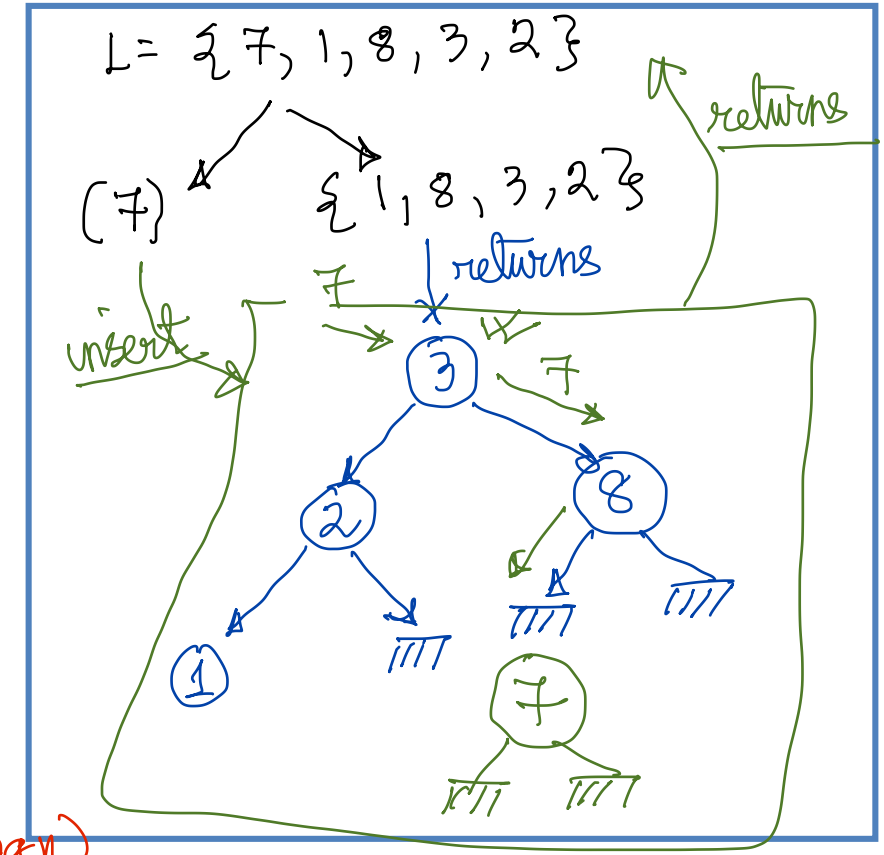
Insertion Sort: Data Structure

How are we going to store the ~~ele~~ 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 $O(n)$ ✓

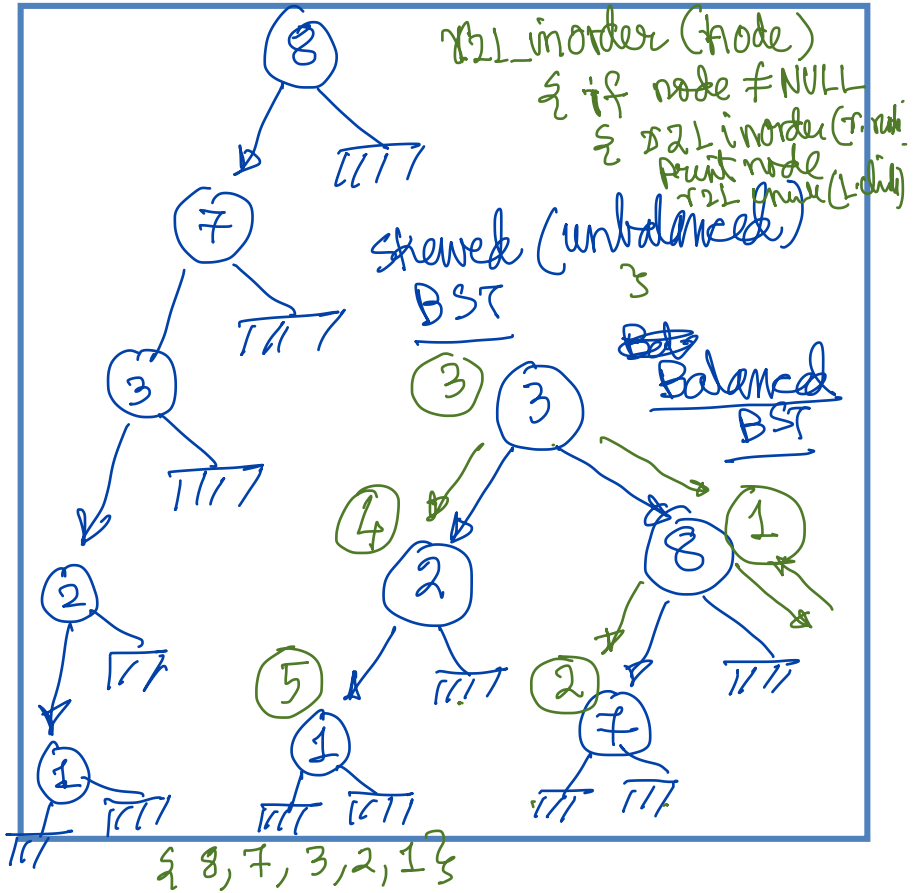
$T(n) = T(n-1) + O(n) = O(n \log n)$



Insertion Sort: Finalization

Sort_M2_final (L)

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



Sorting by Divide & Conquer Method

```
Sort (L)
{ if(|L| ≤ 1) return (L)
  Decomposition (choice points)
    split L into non-empty L1, L2
    Appropriately choose an element
  Recursion
    M1 = Sort(L1)
    M2 = Sort(L2)
  Recomposition
    Mechanism to combine M1 & M2 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 $O(n)$

$L_1 = \{x_i\}$, $L_2 = L - L_1$

Recomposition :- Insert x_i in M_2

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

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

MergeSort

```
Mergesort (L)
{ if(|L| ≤ 1) return (L)
  split L into L1, L2 which are
      non-empty
  M1 = Mergesort (L1)
  M2 = Mergesort (L2)
  M  = Merge (L1, L2)
  return (M)
}
```

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

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

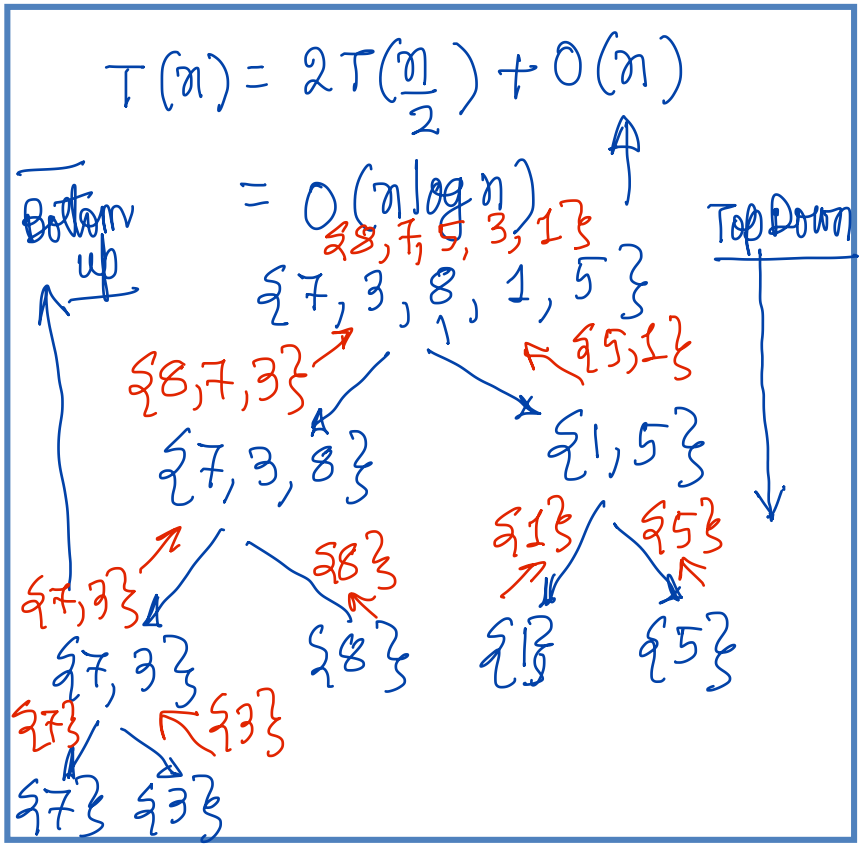
choose $k = 2$ → optimal performance

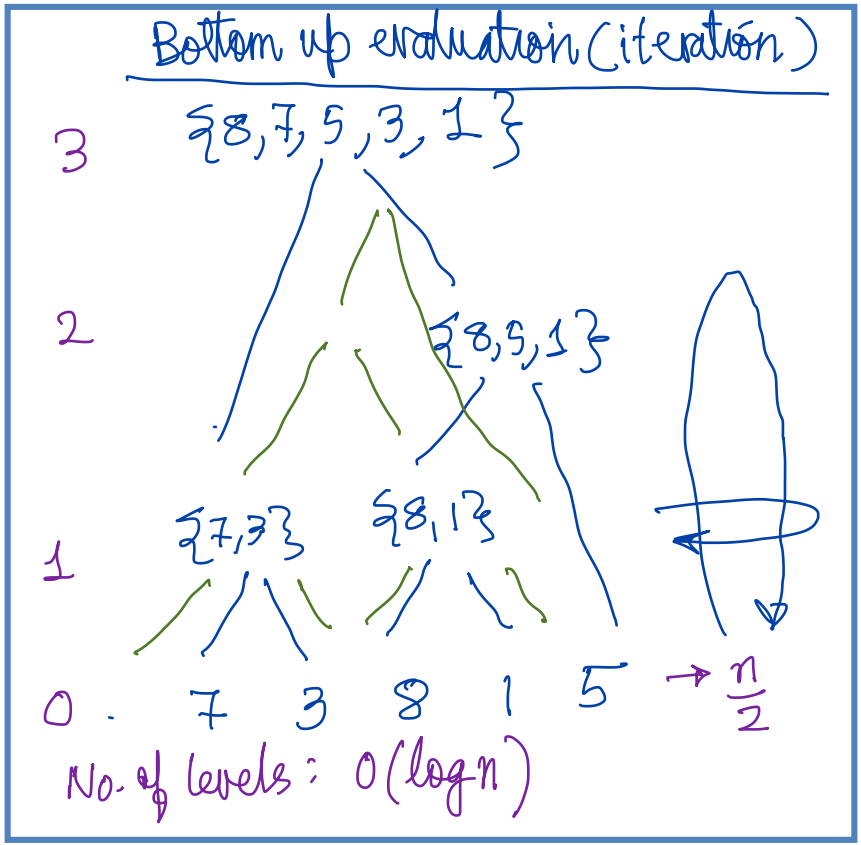
```
Merge (L1, L2)
{ if (L1 = NULL) return (L2)
  if (L2 = NULL) return (L1)
  let L1 = {x1, x2, ..., xn}
      L2 = {y1, y2, ..., ym}
  if (x1 ≤ y1)
  L = {x1} || Merge (L1 - {x1}, L2)
        ↑ concat
  else
  L = {y1} || Merge (L1, L2 - {y1})
  return (L)
}
```

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

$$= O(n)$$

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

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

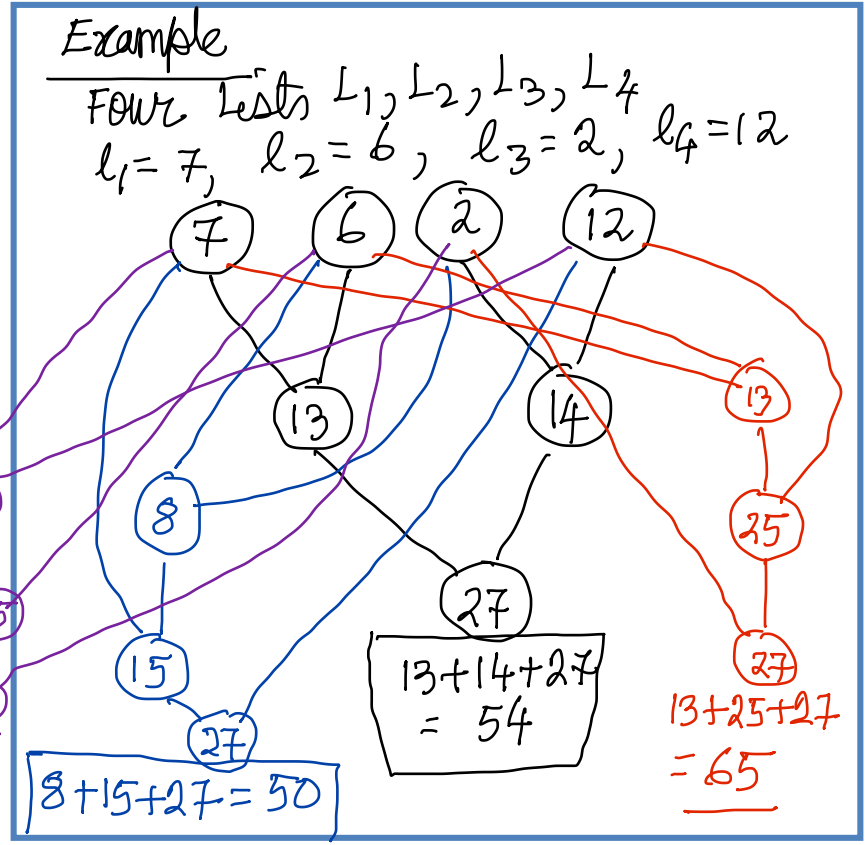
for each pair (L_i, L_j)

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

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

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

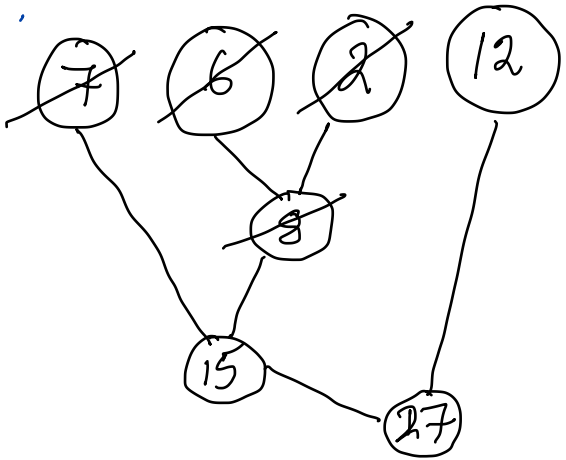
Best Solution / option of choosing L_i & L_j



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

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

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

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

$\vdots$

$L_n = \{$ $\}$

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

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

If we use a simple Max Data Structure

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

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

$= O(n \log n)$

each op $O(\log n)$

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 = {x1, x2, ..., xn}
  y = choose (L)
      /choose an element from L/
  L1 = {z | z ≤ y}
  L2 = {z | z > y}
  M1 = Quicksort (L1)
  M2 = Quicksort (L2)
  M = M2 || M1 (concatenation)
  return (M)
}
```

(D) (R)

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

In the worst case $k = 1$

$\hookrightarrow O(n^2)$

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

$\hookrightarrow$ classical Algo.

Case 2: Randomly choose y from L.

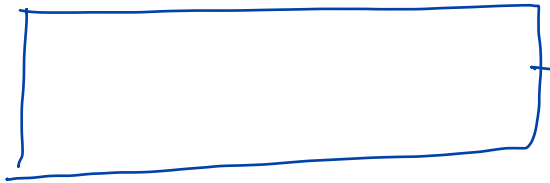
$\hookrightarrow$ Average case: $O(n \log n)$

QuickSort: Analysis

→ Worst Case : $O(n^2)$

↳ Median Finding : $O(n \log n)$ 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'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 — 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 $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, 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 $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,
   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

$x = 101001$ n-bit

$y = 101010$ n-bit

~~101001.0~~

1010010.1 ←

~~10100100.0~~

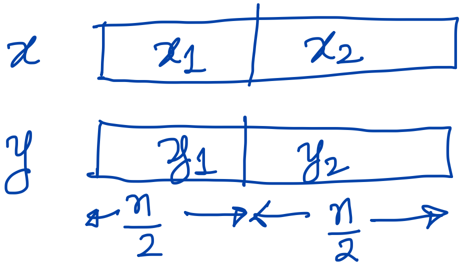
101001000.1

~~1010010000.0~~

10100100000.1

11010111010 Result

$O(n^2)$



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

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

$x \cdot y = 2^n \underbrace{x_1 y_1}_{1} + 2^{n/2}(\underbrace{x_1 y_2}_{2} + \overset{3}{x_2 y_1}) + \underbrace{x_2 y_2}_{4}$

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

$O(n^2)$

$A = x_1 y_2 + x_2 y_1$

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

$xy = 2^n \underbrace{x_1 y_1}_{1} + 2^{n/2} \cdot A + \underbrace{x_2 y_2}_{2}$

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

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

Strassen's Algorithm for Matrix Multiplication

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

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

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

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

$$= O(n^3)$$

$$P_1 = a(f-h), \quad P_2 = (a+b)h$$

$$P_3 = (c+d)e, \quad P_4 = d(g-e)$$

$$P_5 = (a+d)(e+h) \quad P_6 = (b-d)(g+h)$$

$$P_7 = (a-c)(e+f)$$

$$\left[\begin{array}{c|c} P_5+P_4-P_2+P_6 & P_1+P_2 \\ \hline P_3+P_4 & P_1+P_5-P_3-P_7 \end{array}\right]$$

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

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

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 = {⟨x1,y1⟩, ⟨x2,y2⟩, ..., ⟨xn,yn⟩}
  split S into 2 disjoint non-empty subsets S1 & S2        O(1)
  R1 = closestpair (S1)                                    O(n)
  R2 = closestpair (S2)
  Let R = min (R1, R2)
  R3 = combine (S1, S2, R)                                 f(n) = O(n)
  return (R3)
}
```

equal

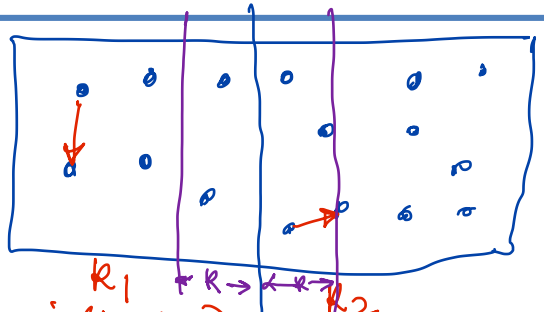
$$T(n) = T(n_1) + T(n_2) + f(n)$$
$$= 2T(n/2) + f(n)$$

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 2R strip on the boundary.

$I = \text{strip}(S_1, R) \cup \text{strip}(S_2, R)$

Sort (I) by the y-axis

$I = \{q_1, q_2, \ldots, q_r\}$

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

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

NO

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

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

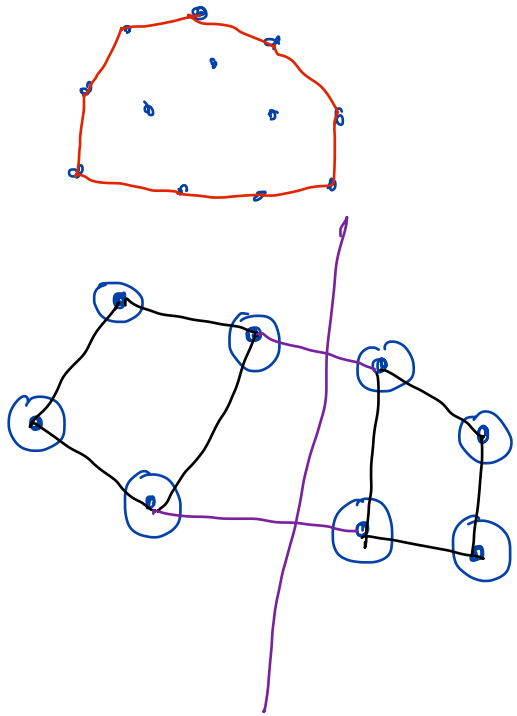
case 1: $f(n) = O(n \log n)$

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 S1, S2                         O(n)
    H1 = convexHull(S1)
    H2 = convexHull(S2)
    H3 = combine(H1, H2)                        → O(n)
    return(H3)
}
```

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

$$O(n \log n)$$

Summary

Balancing the split

↳ Decomposition → $f_1(n)$
Recursion → a, b
Recomposition → $f_2(n)$

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

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?