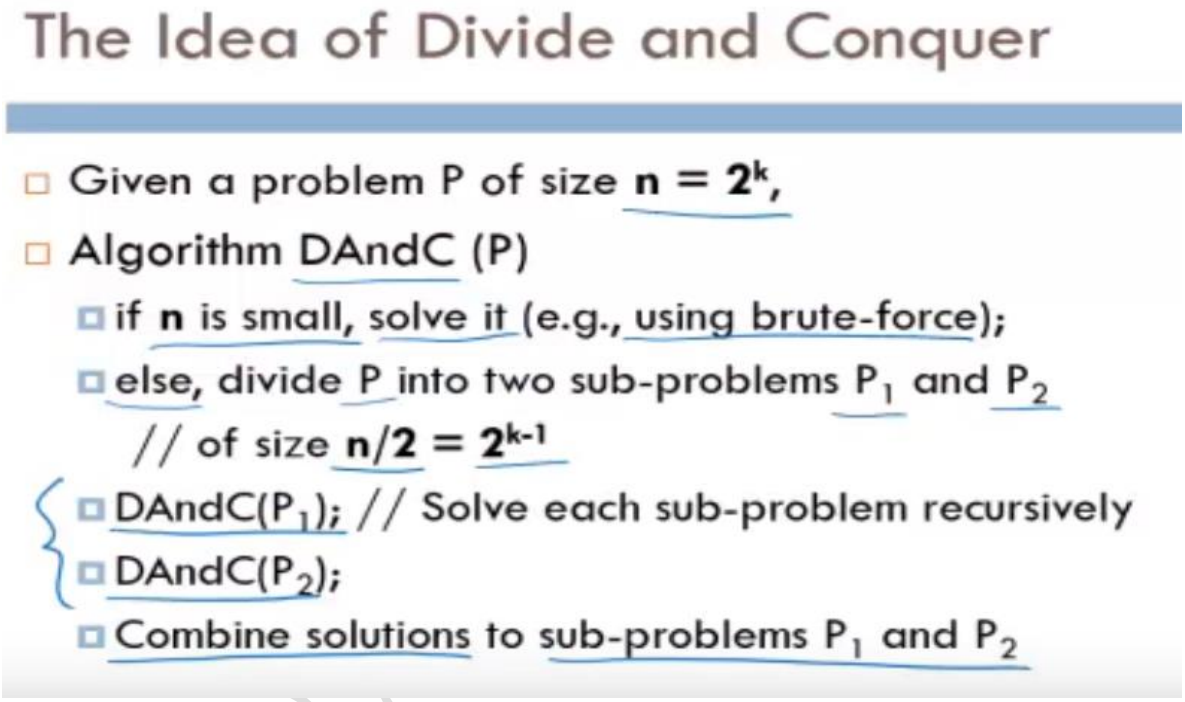
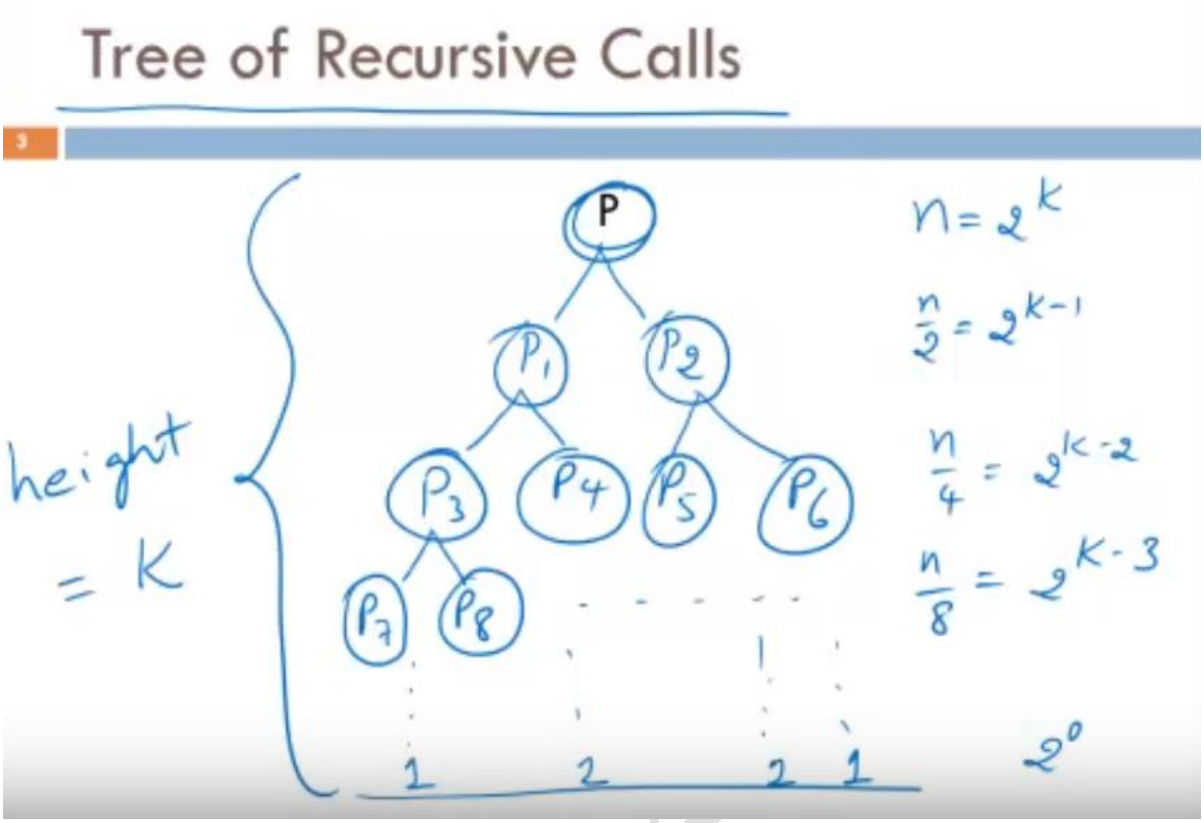
Lecture 8-Divide & Conquer

Many algorithms are recursive in nature to solve a given problem recursively dealing with sub-problems.

In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.





In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

Max-Min Problem

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

Analysis

The number of comparisons in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1, where **y** is greater than or equal to **x**.

Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y]

Algorithm: Max - Min(x, y)

```
if x – y ≤ 1 then
  return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
  (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
  (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
R  eturn (max(max1, max2), min(min1, min2))
```

Analysis

Let ***T(n)*** be the number of comparisons made by $Max - Min(x, y)$, where the number of elements $n = y - x + 1$.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

Let us assume that ***n*** is in the form of power of **2**. Hence, $\mathbf{n = 2^k}$ where **k** is height of the recursion tree.

So,

$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\frac{n}{4}) + 2\right) + 2.... = \frac{3n}{2} - 2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

Binary Search

Problem Statement

Binary search can be performed on a sorted array. In this approach, the index of an element **x** is determined if the element belongs to the list of elements. If the array is unsorted, linear search is used to determine the position.

Solution

In this algorithm, we want to find whether element **x** belongs to a set of numbers stored in an array ***numbers[]***. Where ***l*** and ***r*** represent the left and right index of a sub-array in which searching operation should be performed.

```
Algorithm: Binary-Search(numbers[], x, l, r)
if l = r then
  return l
else
  m := ⌊(l + r) / 2⌋
  if x ≤ numbers[m]  then
    return Binary-Search(numbers[], x, l, m)
  else
    return Binary-Search(numbers[], x, m+1, r)
```

Analysis

Linear search runs in ***O(n)*** time. Whereas binary search produces the result in ***O(log n)*** time

Let **T(n)** be the number of comparisons in worst-case in an array of **n**elements.

Hence,

$$T(n) = \begin{cases} 0 & if\, n = 1 \\ T(\frac{n}{2}) + 1 & otherwise \end{cases}$$

Using this recurrence relation $T(n) = log\, n$.

Therefore, binary search uses $O(log\, n)$ time.

h.w

A Sample Problem: Finding the Non-Duplicate.You are given a sorted array of numbers where every value except one appears exactly twice; the remaining value appears only once. Design an efficient algorithm for finding which value appears only once. Here are some example inputs to the problem: 1 1 2 2 3 4 4 5 5 6 6 7 7 8 8 10 10 17 17 18 18 19 19 21 21 23 1 3 3 5 5 7 7 8 8 9 9 10 10

Clearly, this problem can be solved in O(n) time, where n is the number of elements in the array, by just scanning across the elements of the array one at a time and looking for one that isn't paired with the next or previous array element. But can we do better? Since we know that the array is sorted, we might want to think about approaching this problem using some sort of binary-search-like algorithm. We can't use an exact copy of binary search to solve this problem, though, because we don't know what value we're looking for. The key insight needed to solve this problem is the following: suppose you look at the pairs of elements at positions (0, 1), (2, 3), (4, 5), etc. All pairs that appear before the singleton element must consist of two copies of the same value. If we look at the pair containing the singleton value, then the first and second element of the pair will be different, with the singleton element at the first position. Importantly, if we look at any pair after the singleton element, the values in that pair will be different, since the pair containing the singleton will have "stolen" an element from the pair that appears after it, shifting everything down by one position. Let's see how to formalize this into an algorithm. We can pick a pair of elements close to the middle of the the array and check whether the two elements there are equal or different. If they're equal, we can discard that pair and all pairs that come before it, since the singleton element must come after it. If they're different, we can discard the second element of that pair and everything that comes after it, since we know that the second element is part of a pair and that the singleton is either the first element of the pair or comes before it. Finally, when we get down to one element left, we know that element will be the singleton. We'd expect this to run in time O(log n), since we're discarding about half the elements on each iteration. Let's begin by formalizing the algorithm:

Algorithm: Let A be our array and let its length be n = 2k + 1 (since it consists of k pairs and one singleton).

If n = 1, return A[0].

If n > 1, compare A[2⌊k / 2⌋] and A[2⌊k / 2 + 1 ⌋] (using zero-indexing).

If the elements are equal, recursively apply this algorithm to the subarray beginning at position 2⌊k / 2 + 2 and extending to ⌋ the end. Otherwise, recursively apply this algorithm to the subarray starting at the beginning of the array and extending to 2⌊k / 2 , ⌋ inclusive.

Now that we have a formal version of the algorithm, we need to prove that the algorithm works correctly. In order to show correctness, we need to show that we can determine in which half of the array the singleton element appears simply by looking at the elements at positions 2⌊k / 2 and 2 ⌋ ⌊k / 2 + 1. ⌋ Intuitively:

1. If the singleton appears after positions 2⌊k / 2 ⌋ and 2⌊k / 2 ⌋ + 1, then all the elements before the singleton would be paired up correctly. Therefore, A[2⌊k / 2⌋] = A[2⌊k / 2 + 1 ⌋], so we can discard the first half of the array.

2. If the singleton appears at or before position 2⌊k / 2 , ⌋ then all the elements after the singleton would be mismatched. Therefore A[2⌊k / 2⌋] ≠ A[2⌊k / 2 + 1 ⌋], so we can discard the second half of the array (but not A[2⌊k / 2⌋], since it might be the singleton).

Runtime: Our algorithm's runtime is given by the following recurrence in terms of k: T(0) = Θ(1) T(k) ≤ T(⌊k / 2) + Θ(1)

⌋ By the Master Theorem, this solves to T(k) = O(log k). Since n = 2k + 1, this means that the runtime as a function of n is O(log n).