M2 : Fundamental Algorithms

Searching techniques have two major components:

Search Key
Search Space

Types of Searches:

Linear Search is a searching technique in which the search space is reduced by one element after every operation.

Time complexity: O(N)
Space complexity: O(1)

Binary Search technique is used to search an element in a sorted array by repeatedly dividing the search interval into half based on a middle element and a condition.

Sample code to search a key 'k' in a sorted array a[N]

```
int l=0, m;  //low = l, mid = m
int h=N-1;  //high = h
while(l<=h){  //as long as we have a non-empty subarray
    m=(l+h)/2;
    if(arr[m]==a[k]) return m;
    else if(a[m]<k) l=m+1;
    else h=m-1;
}
```

Time complexity: O(log2N)
Space complexity: O(1)

Note: Binary search requires an ordered search space & not necessarily a sorted search space

Common Observations

If the key is not present in an array then at the end high=low+1 such that:
∴ arr[high] < key < arr[low]

Eg. arr = [1, 10, 12, 21, 30], key=22

Then at the end of the loop, arr[low]=30 and arr[high]=21

Applications:

Q. At what index will you insert an element in a sorted array so that it remains sorted?

=> We will insert it at index = low

First and Last Occurrence

We have been given a sorted array containing repeated elements and a key 'k'. We have to find the first & last occurrence of the key.

Approach:

We can use a simple "For" loop to find the first and the last occurrence of the key.
Time complexity: O(N)
Space complexity: O(1)
Binary Search: We can write two separate binary search codes to find the first and the last occurrences of the key.
First occurrence: When mid is equal to the key and the element before mid is not equal to the key.

```
if(arr[m]==k){
   if(m==0 or arr[m-1]!=k)return m;
}
```

Secondary occurrence: When mid is equal to the key and the element after mid is not equal to the key.

```
if(arr[m]==k){
   if(m==N-1or arr[m+1]!=k)return m;
}
```

Time complexity: O(log2N)
Space complexity: O(1)

Note: Occurrence of element 'k' in the array is = last occurrence - first occurrence + 1

Search in Sorted Rotated Array

We have been given a sorted rotated array containing 'N' distinct elements and a key 'k'. We have to check if the key lies in the array or not.

Input: Arr[7] = {5, 6, 7, 1, 2, 3, 4}, k = 3
Output: 5

Approach:

We know that the sorted array is rotated, therefore there exist two individual subarrays that are sorted in ascending order.
Eg. Arr = [5, 6, 7, 1, 2, 3, 4]

We can use binary search to find the pivot. Property of the pivot element:
- It is the largest element of the array
- It is the only element for which a[i] > a[i+1]

We can compare a[mid] with a[N-1] to find out the part in which "mid" is lying in. Let the pivot index be j, then there will be two sorted subarrays - from 0 to j and from j+1 to N-1.
We can compare the key with a[N-1] to know in which subarray it lies. And then we can directly search the key in that subarray.
Time complexity: O(log2N)
Space complexity: O(1)

Note: The above technique may not work for arrays containing non-distinct elements as we may not be able to identify the part in which our "mid" is lying in. This will create a hindrance in finding the pivot by using Binary Search.
Eg. Arr = [3, 3, 3, 5, 9, 1, 2, 3, 3, 3, 3]
Here part2 >= Arr[N-1] but part1 :≠ Arr[N-1]

Peak Element

We have been given an unsorted array Arr[N] and we have to find a peak element in the array.

Input: Arr[7] = {10, 20, 15, 2, 23, 90, 67}
Output: 20 (or 90)

Note: An element Arr[i] is said to be a peak if:

Arr[i-1] =< Arr[i] <= Arr[i+1] (i<=1 and i<N-1)

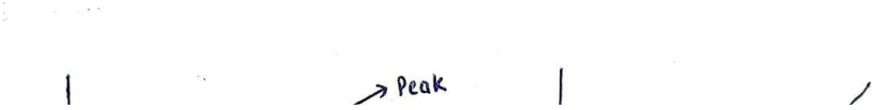
The first and the last elements can qualify as a peak element only if:

Arr[0]>=Arr[1]

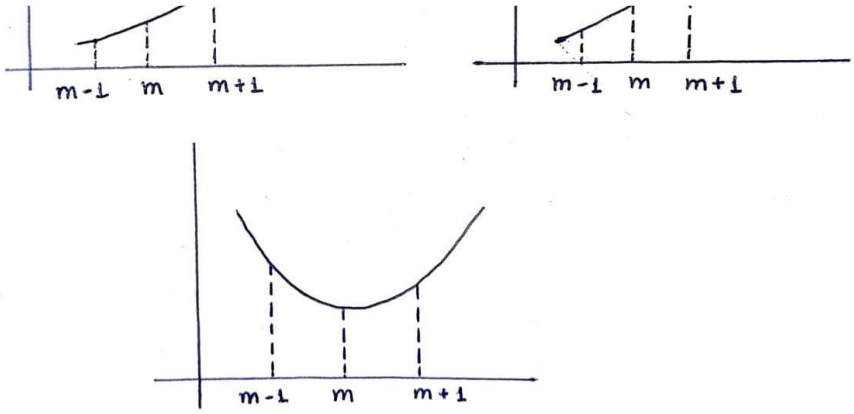
Arr[N-1]>=Arr[N-2]

Approach:

If we apply binary search on the array then our mid will be the peak when -
Arr[mid]>=Arr[mid-1] and Arr[mid]>=Arr[mid+1]
But what if the above condition is false? Then where should we move next, to the left or to the right?
The answer is to the direction where finding a peak is certain i.e. towards the greater adjacent element, and if both are greater then we can move randomly in any direction. It can be easily understood from the illustration given below.



M2 : Fundamental Algorithms



Time complexity: O(log2N)
Space complexity: O(1)

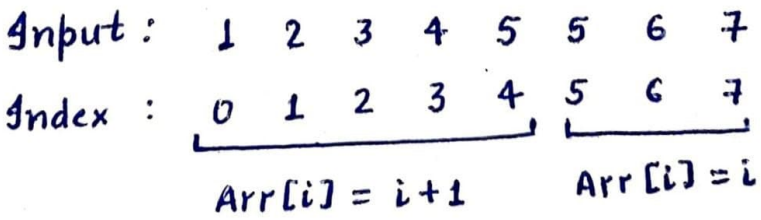
Repeated Element

We have been given a sorted integer array Arr[N] containing elements from 1 to N-1 with one element occurring twice in the array. Find out that element, given that N>=2.

Input: Arr[8] = {1, 2, 3, 4, 5, 5, 6, 7}
Output: 5

Approach:

On observing the elements before and after the repeated element we will find that they have distinct identification characteristics.



From the above illustration, it is clear that our key will be the last element of the subarray defined by Arr[i]=i+1. Thus, we can use binary search to find the repeated element such that Arr[mid] = Arr[mid+1].
Time complexity: O(log2N)
Space complexity: O(1)

Single Element

We have been given an unsorted array Arr[N] where all elements occur in pairs (together at index i & i+1) except one element. Find out that element.

Input: Arr[9] = {4, 4, 1, 1, 3, 7, 7, 6, 6}
Output: 3

Approach:

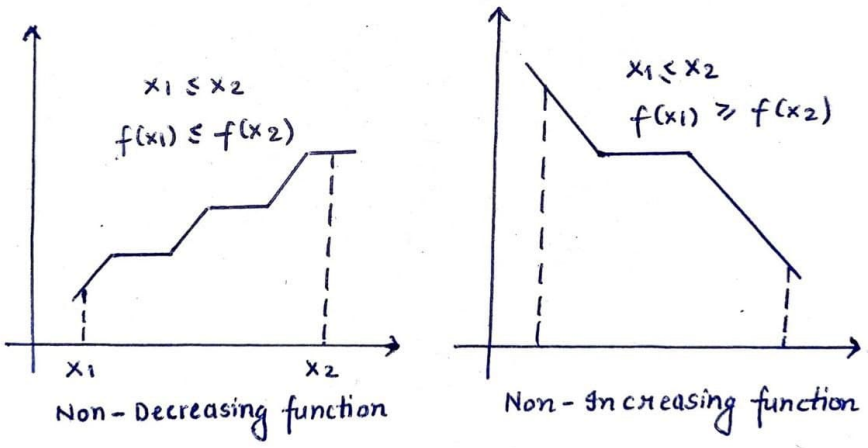
On observing the pairs before and after the single element we find that both the parts of the array have a distinct identification characteristic.

In the first subarray, the first and the second occurrences of the element has an even and an odd index respectively while it is the reverse case for the second part of the array. The answer that is the single element will act as a pivot between the two subarrays. Therefore, we can apply binary search on the above rule to find the answer.
Arr[mid] is the answer if Arr[mid]!=Arr[mid-1] and Arr[mid]!=Arr[mid]+1.
Time complexity: O(log2N)
Space complexity: O(1)

Note: Take special care of the boundary conditions to ensure that the array indices are in their legal range.

Monotonic Functions

A function is said to be monotonic only when it is either non-decreasing or non-increasing in nature.



In the upcoming lectures, we will learn how to figure out these monotonic functions hidden in problems and to apply binary search on the answer. The steps used for the following are:

Figure out a monotonic function
Find a range of answers
Apply Binary Search

Square Root

In this lecture, we will learn how to calculate the square root of a number 'N' with the help of monotonicity and binary search.

Property: √N = x such that x^2=N or x^2<=N<=(x+1)^2

Since we know that

f(x) = x^2 is monotonically increasing from [0,∞]
1<=√N<=N
Hence we can apply binary search on the answer that is low = 1 and high = N to find the square root of 'N'.

K-th Smallest in Array-1

We have been given an unsorted array Arr[N] and we have to find the kth smallest element.

Input: Arr[8] = {40, 10, 10, 30, 40, 20, 50, 70, 50}, k=6

Output: 40

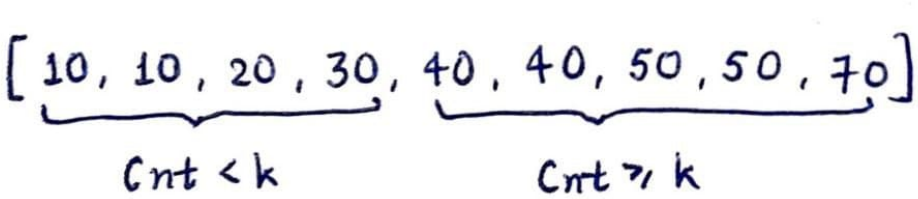
Approach:

We can sort the array and return Arr[k-1] as the answer.
Input: Arr = [40, 10, 10, 30, 40, 20, 50, 70, 50], k=6

M2 : Fundamental Algorithms

Input: Arr = [40, 10, 10, 30, 40, 20, 50, 70, 50], k=6
temp = [40, 10, 10, 30, 40, 20, 50, 70, 50]
Sorted array temp = [10, 10, 20, 30, 40, 40, 50, 50, 70]

Time complexity: O(NlogN)
Space complexity: O(N)
Hint: Think of some property related to the answer.



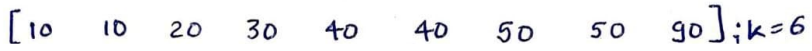
Arr[i] will be the required answer only when CntArr[i]>=k and Cnt'Arr[i]<k.
Where, CntArr[i] = Count of elements less than or equal to Arr[i]
Cnt'Arr[i] = Count of elements exactly less than Arr[i]
Thus we can use brute force to calculate the above two values for each array element to find the answer.
Time complexity: O(N^2)
Space complexity: O(1)

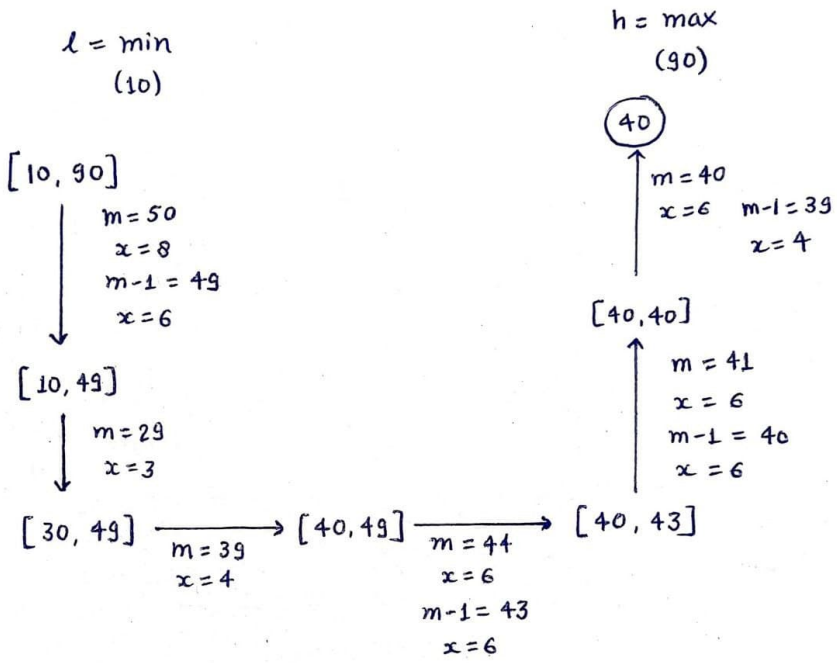
K-th Smallest in Array-2

In this lecture, we will continue with the previous problem and learn how to use the concept of monotonicity to apply binary search on the answer.

We have been given an unsorted array Arr[N] and we have to find the kth smallest element.
Input: Arr[9] = {40, 10, 10, 30, 40, 20, 50, 90, 50}, k=6
Output: 40
Approach:

As discussed in the previous lecture, f(x) is a monotonically increasing function where x = Arr[i] and f(x) = Count of elements less than or equal to Arr[i].
The answer can vary in the range of minimum and the maximum element of the array. Therefore we can run a binary search on the answer - [min, max] and shift the mid according to the monotonic rule f(x).
If Cntmid<k, it means that it is not the answer and we can shift our low to mid+1.
If Cntmid>=k, then it may be the answer, therefore we check the value of count for (mid-1). If we get Cnt>=k then we shift high=mid-1, otherwise if Cnt<k, then mid is the answer.
Time complexity: O(Nlog(max-min))
Space complexity: O(1)





K-th Smallest in Matrix

We have been given a 2D matrix of dimension NxN where each row is sorted. We have to find the kth smallest element in the matrix.

Input: Arr = [{1 3 5}, {1 2 9}, {4 5 6}], k=6
Output: 5 ∵ [1 1 2 3 4 5 5 6 9]

Approach:

Copy the original array to a temporary array temp[N2]. Sort it and return Arr[k-1].
Time complexity: O(N2logN)
Space complexity: O(N^2)
We can also try the approach followed in the previous lecture. We can run a binary search on the answer and find the count of elements less than or equal to the mid.
Time complexity: O(N2log(max-min))
Space complexity: O(1)

But we can achieve this time complexity even when the rows are unsorted. Can we do better than this?
We can use the above fact to count Cnt using binary search in each row.
Cntmid = Cntrow=1 + Cntrow2 + ... + Cntrow = N
Time complexity: O(Nlog(N)log(max-min))
Space complexity: O(1)

Note: You can also use the upper_bound & lower_bound functions for the third approach. They are internally implemented using binary search only.

Maximize K

We have been given an unsorted array Arr[N] containing only positive elements and an integer 'x' such that x>=0. We have to find the maximum possible 'k' such that none of the subarrays of size 'k' has a sum>x.

Approach:

Brute Force: We can calculate the sum of all the subarrays of size 1 to N by using the sliding window technique and find the maximum value of 'k' following the given criteria.
Time complexity: O(N^2)
Space complexity: O(1)
Binary Search on Answer: Since we know that 'k' can vary from 1 to N and beyond a certain value of 'k', the subarrays will have sum>x (since the array contains only positive elements). We can find this pivot with the help of binary search and the given condition. Here, low = 1 and high = N.
Time complexity: O(NlogN)
Space complexity: O(1)

Note:
- Sliding window technique can be used to effectively calculate the sum of subarrays.
- Handle array indices with care

Place the Cows

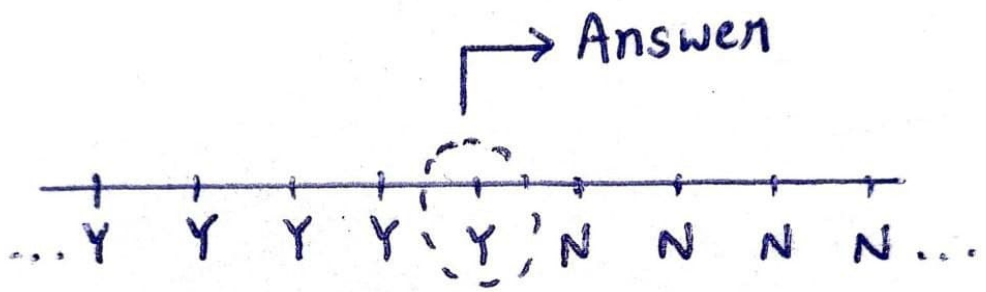
A farmer has 'N' stalls located at some points on a number line x0, x1, x2,.., xN. Each stall can contain at max 1 cow. Given that N>=2, C>=2 and C<N, place 'C' cows such that the minimum distance between any two adjacent cows is maximum possible.

Approach:

Brute Force - Use recursion to figure out all the NCC combinations and find the maximum possible value of the minimum distance between any two adjacent cows.
Binary Search on Answer: Hint: Focus on the minimum adjacent distance(d) between any two cows in a specific configuration.

If it is possible to put 'C' cows at 'N' stalls for d=4 then, it is also possible for d<=3.

Similarly, if it is not possible to put 'C' cows at 'N' stalls for d=9, then it is also not possible for d>=10.



Therefore, we have a monotonic function on which we can apply binary search to find the correct answer.

Where low = 1, the minimum possible distance between any two adjacent cows/stalls
And high = max-min, the maximum possible distance between any two cows

We can implement a function to check whether it is possible to put C cows at N stalls for a given 'd'.

Note: We should allocate the cows in a greedy fashion such that the first stall is always occupied.

Time complexity: O(Nlog(max-min))
Space complexity: O(1)

Note: In case you are facing difficulty in finding the low and high, then you can also consider 0 and INT_MAX respectively.

Allocate the Books

We have been given 'N' books containing P[ ] = {P0, P1, P2,.., PN} pages respectively. We have to allocate 'N' books to 'M' students such that:

Each student gets at least one book
All the books should be allotted
Allotment must be contiguous

We have to allocate them in such a manner that the maximum number of pages allocated to any student is minimum. Given that no partial allocation of books is allowed.

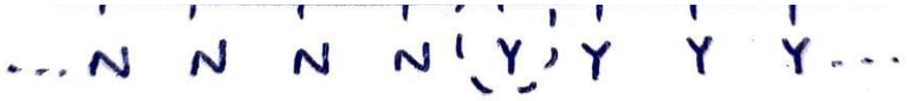
Input: N = 4, M = 2, P[4] = {12, 24, 67, 90}
Output: 113
Explanation: Consider all valid combinations:

Case 1: S1 = 12 | S2 = (24+67+90) = 191
Maximum = 191
Case 2: S1 = (12+24) = 36 | S2 = (67+90) = 157
Maximum = 157
Case 3: S1 = (12+24+67) = 113 | S2 = 90
Maximum = 113

Answer = Minimum(Maximum in Case 1, Case 2, Case 3) = 113

Approach: Hint: Focus on the upper bound on the maximum number of pages(d) allocated to any student in a specific configuration.
If it is possible to allocate 'N' books to 'M' students such that the upper bound on the maximum number of pages(d) is 120 then, it is also possible for d>=120.

Similarly, if it is not possible to allocate 'N' books to 'M' students for d=95 then, it is also not possible for d<95.



Therefore, we have a monotonic function on which we can apply binary search to find the answer.

Where low = maximum number of pages a book contains
And high = total sum of pages of all the books

We can implement a function to check whether it is possible to allocate 'N' books to 'M' students or not, considering the provided constraints.

Note: We should allocate the books in a greedy fashion such that a student gets as many pages as possible, below the upper bound.

Time complexity: O(Nlog((i=0)Σ(i=N-1)(P[i] - max)))
Space complexity: O(1)

Note: In the problem statement, we have not been given any relation between N & M. Therefore we have to account for cases where N<M since the allocation is not possible in such a case

Smallest Good Base

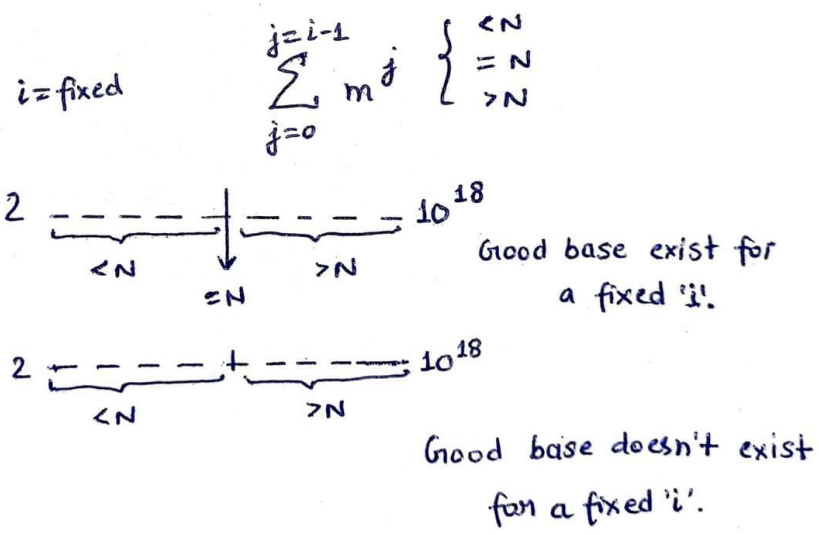
We have been given a number 'N' and we have to find its smallest good base 'k' such that k>=2 and N ∈ [3, 10^18].

A base is said to be a good base iff all the digits in that base representation are 1.

Input: 4681
Output: 8 ∵ 4681 = (11111)8

Approach:

A number 'N' can have multiple good bases.
Eg. 13 = (111)3 and 13 = (11)12
1 & N-1 will always be a good base of any number N.
N = (11)N-1 = (111111....N times)1
If we look at the representation of a good base 'm' then it will be expressed as:
N = (111111....i times)m = 1+m+m2+...+mi-1 (i terms)
From the above representation, it is clear that if 'N' has two good bases 'k1' and 'k2' with 'i1' and 'i2' terms respectively. Then for k1<k2, i1>i2. Thus, there is monotonicity on the base if we fix the number of terms 'i'.



Hence we can apply binary search on the number of terms 'i' to find if there exists a good base that has 'i' terms in the representation of 'N' where 2 < = i < 63.
(10^18) = Base 2 will have around 63 terms
In order to find the smallest good base, we should vary 'i' from 63 to 2 since the smallest good base will have the greatest number of terms.

Time complexity: O(63*log2N*63)
Space complexity: O(1)

Note: While writing the code you may encounter "Integer overflow" at multiple places. Therefore, use long long datatype and handle summation and multiplication operations carefully.

M2 : Fundamental Algorithms

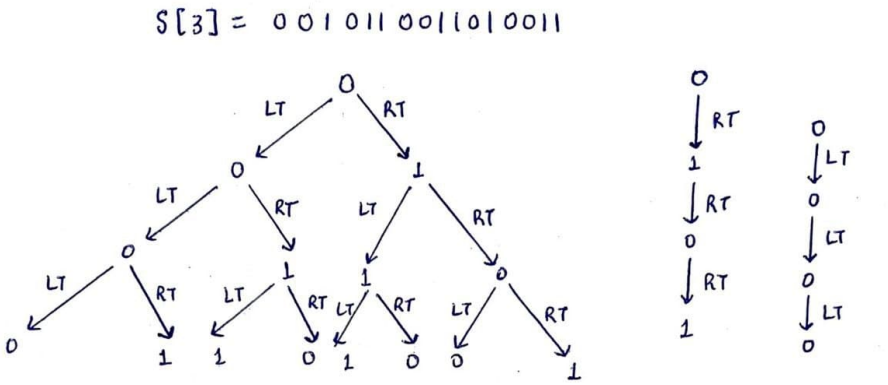
Family of Strings

Consider a family of boolean strings defined by the rule: S(0) = 0 and S(i) = S(i-1).0.S(i-1)'. We have to find the kth character of ith member i.e. S(i).

Input: i=3, k=2
Output: 0
S(3) = 0010110011010011

Approach:

What will be the middle character of a string S(i)? The answer is 0 and it is obvious from the definition of the family of string.
What will be the length of the ith string? It is 2^(i+1)-1.
S[0] = 0 : 1 (2^(0+1)-1)
S[1] = 001 : 3 (2^(1+1)-1)
S[2] = 0010110 : 7 (2^(2+1)-1)
S[3] = 0010110011010011 : 15 (2^(3+1)-1)
If we try to represent any random string in the form of a tree with the centre of the string as the root of the tree, then we will find that -
The character changes when we move towards the right (RT)
It remains the same when we move towards the left (LT)



Thus we can apply binary search on the string length and move the mid depending on whether mid<k, mid=k or mid>k.
Time complexity: O(2^(N+1)-1) = O(N)
Space complexity: O(1)