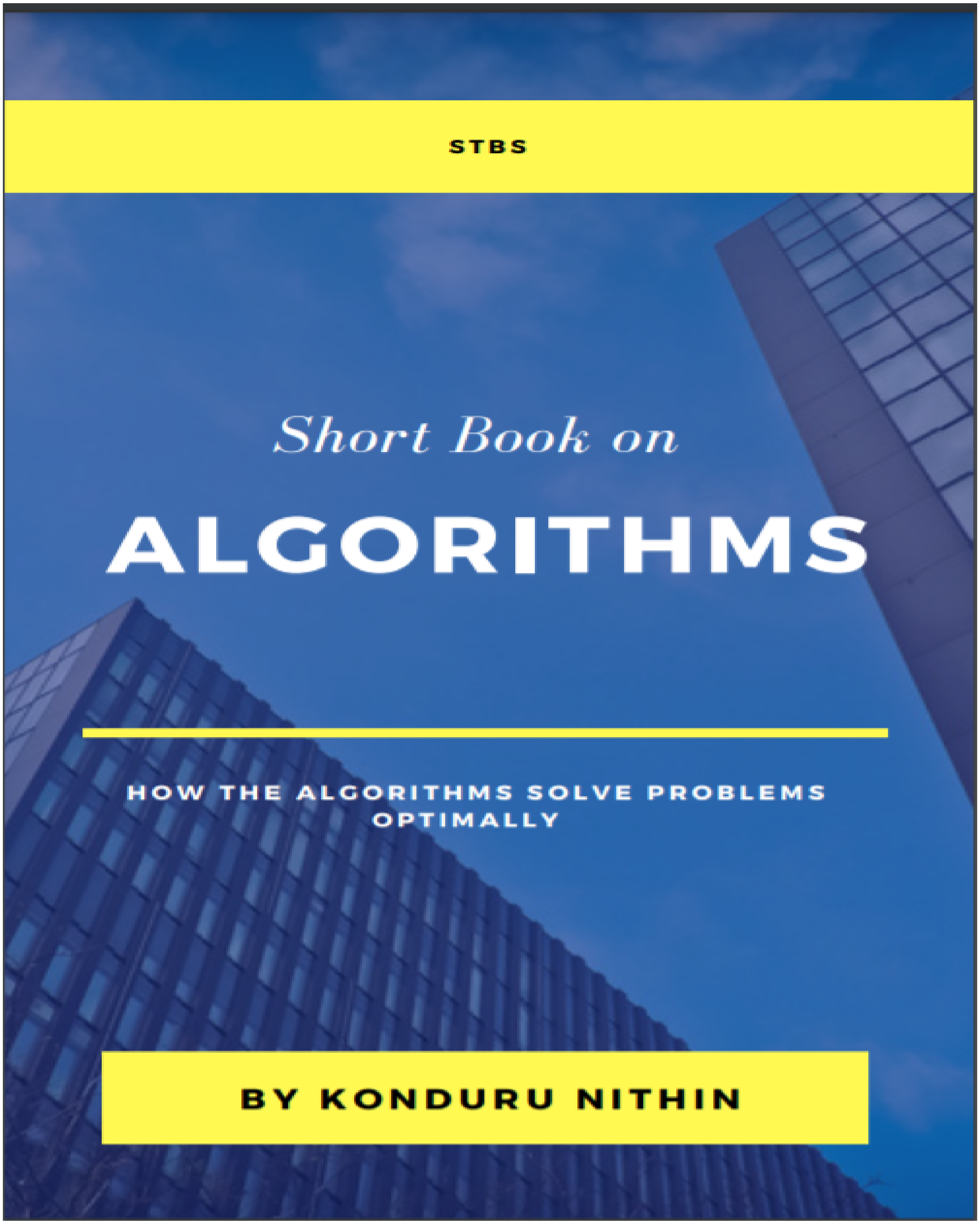


STBS

Short Book on ALGORITHMS

HOW THE ALGORITHMS SOLVE PROBLEMS OPTIMALLY

BY KONDURU NITHIN

Short Book on Algorithms

TOPICS:

- **SORTING TECHNIQUES**
- **BIT MANIPULATION**
- **TREES**
- **STRINGS**

1. Sorting Techniques

A Sorting Algorithm rearranges the members of an array or list based on a comparison operator on the elements. In the relevant data structure, the comparison operator is employed to determine the new order of elements.

Types of sorting methods

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Counting Sort
- Bucket Sort
- Heap Sort
- Quick Sort

There are many more sorting techniques also….but in this book we will focus on above mentioned as they are the most widely used techniques used by Competitive Programmers.

1.1.Selection sort

The selection sort method sorts an array by continually choosing the smallest member in the unsorted segment and placing it first.We will maintain 2 subarrays 1 is sorted and another one is

Unsorted in the given array.The least element from the unsorted subarray is chosen and transferred to the sorted subarray in each iteration of selection sort.

We will understand this with one example,

array = [100, 55, 15, 0]

- We have to find the minimum element from index 0 to index 4 and place it at beginning position by swapping.

[100, 55, 15, **0**] -------> [**0**, 55, 15, 100]

- Now find the minimum element from index 1 to index 4 and place at beginning by swapping.

[0, 55, **15**, 100] -------> [0, **15**, 55, 100]

- Now do the same from index 2 to 4…...and then do the same from index 3 to 4.

Now the array has been sorted,

Pseudocode as follows

```
Selection sort(array,n)
        Int min_ind;
        for i=0 to n-2
          min_ind=i
          for j=i+1 to n-1
             if (arr[j] < arr[min_idx])
                  min_idx = j
          swap(arr[min_idx], arr[i])
```

Time complexity of this selection sort is in all cases ,that is best case,average case,worst case is **O(n^2).**(n is the size of the input array).Space complexity of selection sort is **O(1)** as we use constant amount space.

1.2. Bubble Sort

Bubble sort is a sorting algorithm in which two adjacent elements are compared and swapped until they are no longer in the desired order.

We've been provided an input array that should be sorted in ascending order. We begin with the first element at the i=0 index, and if the element at i+1 is bigger than the first, we swap the items at index I and i+1. If none of the above applies, no switching will take place. The value of " I " is now increased, and the previous two stages are repeated until the array is exhausted. The last index will be ignored because it has already been sorted.The array's greatest element will now be at the final index.We'll now set i=0 once again and repeat the procedures above, which will finally place second biggest in second last place in the array. The array's last two indexes are now sorted.

Pseudocode as follows

```
Bubble sort(array,n)
      for i=0 to n-i-1
        for j=0 to n-i-2
          If( array[j]>array[j+1])
              swap( array[j] and array[j+1])
```

Let us take one example and try to understand with that,

array = [100, 55, 15, 0]

i=0 → [55,15,0,100]

i=1 → [15,0,55,100]

i=2 → [0,15,55,100]

i=3 → [0,15,55,100]

Time complexity of bubble sort in all best case, worst case, Average case is **O(n^2).**Here n is the size of the input array. Space complexity of the bubble sort is **O(1)** as no auxiliary space required.

1.3.Insertion sort

Insertion sort is a straightforward sorting algorithm.First we divide an array into two halves, one sorted and the other unsorted part and then we take an element from the unsorted part and place it in the sorted part in the correct manner.

In this algorithm,

- If it is the first element, do nothing.
- Pick the next element,and compare it with the all elements in the sorted part
- Shift all the values to sorted part if the value which is greater than value to be sorted
- Pick up the next element and do the same

Pseudocode as follows,

```
InsertionSort(array,size n)
    int i, key, j;
    for i=1 to n-1
        key = array[i];
        j = i - 1;
        while  j >= 0 and array[j] > key
            arr[j + 1] = arr[j];
            j = j - 1;
        arr[j + 1] = key
```

We will try to understand this with one example,

Array = [8,7,3,2]

after ,

i=0 → [**8**,7,3,2]   key=8

i=1 → [**7**,8,3,2]   key=7

i=2 → [**3**,7,8,2]   key=3

i=3 → [**2**,3,7,8]   key=2

Time complexity of insertion sort in best case is **O(n)** and in average,worst case is **O(n^2).** Space complexity of the insertion sort is **O(1)** as no auxiliary space required.

1.4.Merge sort

Merge sort is a sorting method that uses the divide and conquer method. It is one of the most regarded algorithms, with a worst-case time complexity of O(n log n). Merge sort splits the array into equal parts before combining them in a sorted order.

To identify an ideal solution to an issue, the **divide-and-conquer** model is frequently applied. Its core concept is to break down a given issue into two or more comparable but smaller subproblems, solve each one separately, and then combine the solutions to solve the original problem

How does it work ?

Merge sort divides the list into equal halves until it can't be divided any further. If there is just one entry in the list, it is sorted by definition. Merge sort then joins the smaller sorted lists together, keeping the resultant list sorted as well.

In this algorithm,

- Return if there is just one entry in the list that has already been sorted.
- Recursively split the list into two half until it can no longer be divided
- Now In sorted order, combine the smaller lists into a new list.
- Resultant array is the sorted array
- For any given array use the following methods to sort the array

Pseudocode as follows,

```
mergesort( array a )

    if ( n == 1 )
        return a

    left array = a[0] ... a[n/2]
    right array = a[n/2+1] ... a[n]

    l1 = mergesort( l1 )
    l2 = mergesort( l2 )

    Return  merge(left array,right array )



merge( array a,array b )

    array c
    while ( a and b are not empty )
      if ( a[0] > b[0] )
        add b[0] to the end of c
        remove b[0] from b
      else
        add a[0] to the end of c
      remove a[0] from a

  while ( a is not empty )
     add a[0] to the end of c
     remove a[0] from a

  while ( b is not empty )
     add b[0] to the end of c
     remove b[0] from b

  return array c
```

This is the merge sort algorithm that is for dividing and merging.

Time complexity of merge sort in all three cases is **O(n log n)** and Space complexity of the merge sort is **O(n)** as it uses arrays in the algorithm.

1.5.Counting sort

Counting sort is a sorting method based on the keys that connect certain ranges of data.Sorting is accomplished by counting items with separate key values, similar to hashing. Following that, it does certain mathematical operations to determine the index position of each object in the output sequence. When the range of items to be sorted is not bigger than the number of objects to be sorted, counting sort is effective.

In this algorithm,

- Create a count array to keep track of how many distinct objects there are.
- Make a change to the count array such that each element at each index records the total of previous counts.
- Each object's location in the output sequence is indicated by the modified count array.
- Decrease the count in the count array after placing it in the output array.

Pseudo code as follows,

```
CountingSort(array A)

 // n is the length of the array
 // k is the range of the key values
  Initialize array c[n] all values to zero

 //Storing Count of each element
  for j = 0 to n
      c[A[j]]++

 // change c[i] by adding the previous index values.
   for i = 1 to k
      c[i] = c[i] + c[i-1]

 //Now create output array from C[i] let's say B
 for j = n-1 to 0
      B[ c[A[j]]-1 ] = A[j]
      c[A[j]] = c[A[j]] - 1
```

Let us understand this with one example,

Input array → [4,3,3,2,1]

Creating an count array and adding the count of the input array values

B=[0(0 index),1(1 index),1(2 index),2(3 index),1(4 index)]

Add the values with the previous index values.

B=[0(0 index),1(1 index),2(2 index),4(3 index),5(4 index)]

Next by following the above mentioned points the new output sorted array,

sorted array=[1,2,3,3,4]

Actually we put the things in their proper places and reduce the count by one.

Time complexity of counting sort is **O(n+k)** and space complexity of counting sort is **O(n+k)**
Here n is the number of values and k is the range of the key values.

1.6.Bucket sort

Bucket Sort is a sorting technique that separates an unsorted array of data into many buckets. After that, each bucket is sorted using one of the appropriate sorting algorithms or by recursively using the same bucket method. The sorted buckets are then concatenated to create a final sorted array.

In this algorithm,

- Create empty buckets.
- Insert the input array values to the (value%no.of.buckets) numbered bucket.
- Sort the individual buckets using any sorting technique.
- Concatenate all sorted buckets at last.

Pseudo code as follows,

```
function BucketSort(array a)
    Create the buckets of n empty lists
     H is the number of buckets
     N is the length of the array
    for i = 1 to  N
        inserting a[i] into buckets[(a[i]%H)]
    for i = 1 to H
        sort(buckets[i])
    Output array=concatenation of buckets[1] to buckets[k]
    return Output array
```

For example,

Array=[3,2,1,2]

Consider 4 buckets,

1 → 1 (1%4=1st bucket)

2 → 2, 2 (2%4=2nd bucket)

3 → 3 (3%4=3rd bucket)

4

Now sort those bucket values ,after sorting it will be,

1 → 1

2 → 2, 2

3 → 3

4

Now concatenate each bucket value elements

[1,2,2,3] → This is the required sorted array

Worst time complexity of bucket sort is **O(N^2)** but average time complexity is **O(N+k)**.Space complexity is also **O(N+k)** as it requires memory for creating buckets.Here N is the length of the array and k is the number of buckets.

1.7.Heap sort

Using heaps we can sort the elements.The maximum element in max-heaps will always be at the root,This attribute of heap is used by Heap Sort to sort the array.In the similar way min-heap also. Heap sorting is a comparison-based sorting approach that uses the Binary Heap data structure to order items.

In this algorithm,

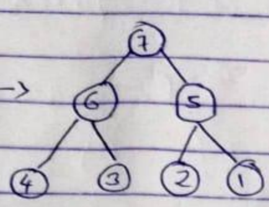
- Create a maximum heap of items in Array at first.
- The root element is the maximum element in the array and after doing this swap it with the last element.
- heapify the max heap, except the last member, which is already in its proper place, and then shorten the heap by one element.
- Next repeat the above step until all numbers are in the correct position.

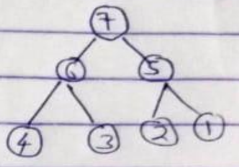
Pseudo code as follows,

```
Function heapify(array A, length, i )
{
    maximum = i
    left_child = 2i + 1
    right_child = 2i + 2

    if (left_child <= length) and (A[i] < A[left_child])
        maximum = left_child
    else
        maximum = i

    if (right_child <= length) and (A[maximum]  > A[right_child])
        maximum = right_child

    if (maximum != i)
        swap(A[i], A[maximum])
        heapify(A, length, maximum)
}
```

```
Function heapsort(array A)
{
  length = length(A)
   for i = n/2 to 1
     heapify(A, length ,i)

   for i = n to 2
     exchange A[1] with A[i]
     A.heapsize--
     heapify(A, i, 0)
}
```

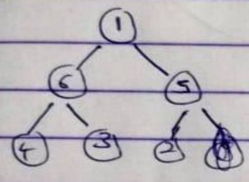
We will understand this with the below example, Array = [7, 6, 5, 4, 3, 2, 1]

Heap →


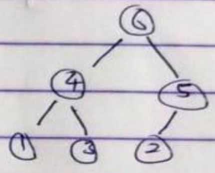
Heapify →


max-heap

delete the root and swap it with last element.

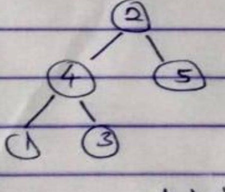


Heapify →

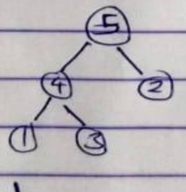


Array = [6, 4, 5, 1, 3, 2, | 7]

delete the root and swap it with last element

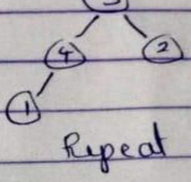


$\xrightarrow{\text{Heapify}}$

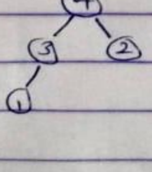


Array = [5, 4, 2, 1, 3, | 6, 7]

delete the root and swap it with last element.

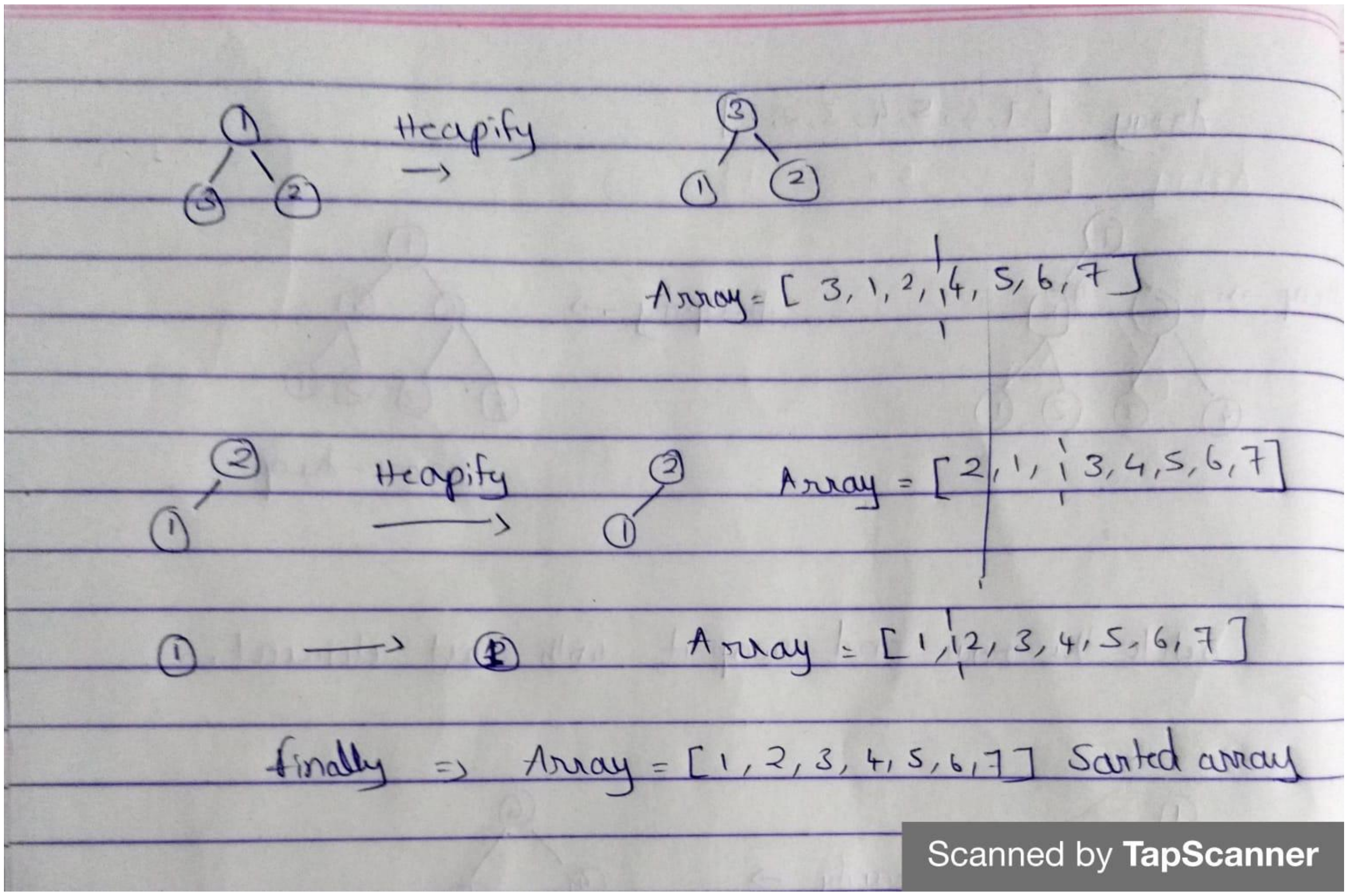


$\xrightarrow{\text{Heapify}}$



Array = [4, 3, 2, 1, | 5, 6, 7]

Repeat same step.


Heapify
Array = [3, 1, 2, 4, 5, 6, 7]
Heapify
Array = [2, 1, 3, 4, 5, 6, 7]
Array = [1, 2, 3, 4, 5, 6, 7]
finally => Array = [1, 2, 3, 4, 5, 6, 7] Sorted array
Scanned by TapScanner

Heap sort time complexity in best case,average case and in worst case **O(n logn)**.Space complexity of Heapsort is **O(1)** as Heap sort is a space-saving method since it works in-place. Only inside the same array are elements altered during each recursion.

1.8.Quick sort

Quicksort is a popular sorting algorithm that uses n log n comparisons to sort an array of n elements in the average situation. It is a more efficient and quicker sorting method. This algorithm employs a divide-and-conquer strategy. Breaking down algorithms into subproblems, solving the subproblems, and then merging the findings to solve the main issue is known as divide and Conquer.

Choose a pivot element first in Divide. After that, divide or reorganise the array into two sub-arrays, with each element in the left sub-array being less than or equal to the pivot element, and each element in the right sub-array being greater.QuickSort comes in a variety of flavours, each of which selects pivot in a unique way.

- Always choose the first element as the pivot.
- Always use the final element as the pivot.
- As the pivot, choose 1 element at random.
- As the pivot, use the median.

In this algorithm,

- Choose the element as pivot.
- Partition the array based on the pivot.
- Recursively apply quick sort on the left partition.
- Recursively apply quick sort on the right partition.

Pseudo code as follows,

```
function quickSort(arr, beg, end)
  if (beg < end)
    pivotIndex = partition(arr,beg, end)
    quickSort(arr, beg, pivotIndex)
    quickSort(arr, pivotIndex + 1, end)

function partition(arr, beg, end)
  set end as pivotIndex
  pIndex = beg - 1
  for i = beg to end-1
  if arr[i] < pivot
    swap arr[i] and arr[pIndex]
    pIndex++
  swap pivot and arr[pIndex+1]

return pIndex + 1
```

Best time complexity of quick sort is **O(nlogn)** but worst case time complexity is **O(n^2).**Space complexity is **O(n).**

Practice the questions on this link based on above concepts**,**

https://codeforces.com/problemset?tags=sortings,800-1200

2. Bit Manipulation

Internally, all data in computer programmes is stored as bits, or integers 0 and 1. The bit representation of integers will be discussed now, along with examples of how to perform bit operations. It turns out that bit manipulation has a lot of applications in algorithm Development.

2.1.Representation of Bits

An n-bit integer is stored in a computer as an n-bit binary number. The C++ type int, for example, is a 32-bit type, meaning that every int integer is made up of 32 bits.

Like for example, 34 is **00000000000000000000000000100010**

To convert a bit representation to a number we do the following,

Bits are of form a_n**........**$a_2 a_1$ **--->** a_n***** 2^n **+** a_{n-1}***** 2^{n-1} **+.....+**a_0***** 2^0

34 can be written from bit representation,

34 = 1 * 2^5 **+ 0 *** 2^4 **+ 0 *** 2^3 **+ 0 *** 2^2 **+ 1 *** 2^1 **+ 0 *** 2^0

A number's bit representation is either signed or unsigned. The most common method is to employ a signed representation, which allows both negative and positive integers to be Represented. A signed variable of n bits can contain any integer between -2^{n-1} and $2^{n-1} - 1$.

The sign of the number ,that is 0 for positive number and 1 for negative number, is represented by the first bit in a signed representation, while the remaining n-1 bits carry the magnitude of the number.The opposite number of a number is computed using two's complement, which entails inverting all of the bits in the number and then raising it by one.

For example,

5 = 00000101 ----> 11111010 ----> 11111010 + 00000001 = 11111011

Only nonnegative integers can be utilised in an unsigned form, although the top bound for the values are bigger. Any integer between 0 and 2^{n-1} can be stored in an unsigned variable with n Bits.

A signed number −x equals an unsigned number 2^{n}-x and the number will overflow if it is bigger than the upper bound of the bit representation.In a signed representation, the next number after 2^{n-1} −1 is $-\ 2^{n-1}$, and in an unsigned representation, the next number after 2^{n}-1 is 0.

2.2 Bit Operations

Bit Operations include **and,or,xor** and **not .**

Or Operation

The x | y or operation generates a number with one bit in situations where at least one of x and y have one bit.

For example, 2 = 10
                      3 = 11

Now,
      2 | 3 = 10 | 11 = 11

So **or** of 2 and 3 is **3**

And Operation

The x & y operation results in a number with one bit in situations where both x and y have one bit.

For example, 18 = 10010
                      17 = 10001

Now ,

18 & 17 = 10010 & 10001 = 10000 **(16).**

So **and** of 18 , 17 is **16**

Xor Operation

The xor operator x and y yields a number with one bit in each of the locations where precisely one of x and y contains one bit**.**

For example, 18 = 10010
17 = 10001

Now,
18 ^ 17 = 10010 | 10001 = 00011 **(3)**

So **Xor** of 18 and 17 is **3**.

Not Operation

The not operation ~x returns an integer in which all of x's bits are inverted.

Let's take a number **2** whose bit representation is **10**

Now by doing **Not** of **2**,

~ 2 = ~ 10 = 01 = **1**

So **Not(~)** of **2** is **1.**

Shifting Bits

The left bit shift x<<k adds k zero bits to the number, whereas the right bit shift x >> k subtracts the integer's final k bits.

For example, 18 = 00000000000000000000000000010010

If we shift it by 2 then, 18 << 2 = 00000000000000000000000001001000 =**72,** It is same as multiplying with powers of 2

Or if we shift the bits of 18 right by 2 then 18>>2 = 00000000000000000000000000000100 = **4** It is the same as dividing with the powers of 2.

Applications are,

We can use numbers of the pattern 1 << k to access single bits of numbers since they have one bit in position k and all other bits are zero.

If we have a number N and we want bit representation of it then we can do it by the following Code,

```
for(int i=31;i>=0;i--){

        if(N&(1<<i)==1){
            printf("1")
         }
         else{
            printf("0")

         }
}
```

The formula x & (x-1) puts the final one bit of x to zero, while the expression x & -x sets all except the last one bit to zero. After the final bit, the formula x | (x-1) inverts all the bits. It's also worth noting that a positive number x is a power of two when x & (x-1) = 0.

2.3 Sets Representation

For every subset of a set {0...1,2,...,n-1}may be expressed as an n-bit integer, with the one bits indicating which items are in the subset. Because each element takes only one bit of memory and set operations may be written as bit operations, this is an economical approach to describe sets.

Like for example, subset {1,3}

Then corresponding bit representation is **00000000000000000000000000001010**

$2^3 + 2^1$ = 10 This is the number for the corresponding bit representation.

If we given a number and we want know the set representation of that number then we can use the following code to do that,

```
for (int i=0; i<32;i++){
    if(number &(1<<i)==1){
        printf("%d",i);
    }
}
```

Set operations

Set operations and corresponding bit operations are as follows,

	Set representation	Bit representation
Intersection	$a \cap b$	a & b
Union	a U b	a | b
Complement	$\overline{a}$	~a
Difference	a \ b	a & (~b)

2.4. Optimization of Bits

Bit operations may be used to improve several algorithms. These improvements can modify the algorithm's time complexity, but they can have a significant influence on the code's actual execution time. In this part, we'll look at several examples of such scenarios.

Hamming distance :

The Hamming distance between two strings of similar length is the number of points where the corresponding symbols are different in information theory.

For example, if we have two strings of equal length **10011** and **11001** then

Hamming distance between them is (10011 and 11001) = 2.

as they differ in 2nd and 3rd positions.

If we have a set of strings [10011,11000,11001] then if we want to find the minimum hamming distance in between them,

(10011,11000)=3
(10011,11001)=2
(11000,11001)=1

Minimum hamming distance is 1.

If we write normal code for this then it will be ,

```
int hamming_distance(string a,string b){
    hdist=0;
    for(int h=0;h<k;h++){
        if(a[h]!=b[h]){
            hdist++;

        }

    }
    return hdist;


}
```

For this function it will take time complexity of **O(n^2*k).** Here n is the length of each string list and k is the length of each string.But we can do these using bit operations as well,If k is small, however, we may save time by storing bit strings as integers and computing Hamming distances with bit operations.

Look at the code below,

```
int hamming_dist(int a ,int b){

    return __builtin_popcount(a^b);

}
```

__builtin_popcount(number) function is used to calculate the number of ones in the number in g++.

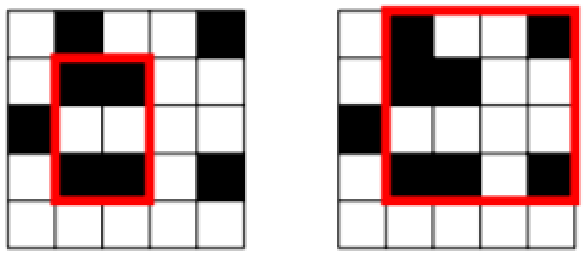
Here by doing the xor we will get to know at what places bits differ and this function returns as the number of ones in the number.

According to some experiments, they created a collection of 10000 random bit strings of length 30 to compare the implementations. The search took 13.5 seconds using the first method, but just 0.5 seconds after bit optimization. As a result, the bit optimised code was over 30 times quicker than the original.

Thus bit optimization optimizes our code complexity.

2.5.Counting subgrids

Calculate the number of subgrids with all black corners given a nxn grid with each square either black (1) or white (0). For instance, consider the grid.



It takes total time complexity of O(n^3) as we need to go to each O(n^2) pairs and then calculate the number of columns that include a black square in both rows in O(n) time for each pair (a,b). Color[y][x] specifies the colour in row y and column x, respectively, in the following code.

```
int count=0;
for(int i=0;i<n;i++){
   if(color[a][i]==1 && color[b][i]==1){

         count++;

   }

}
```

To make this algorithm more efficient, we partition the grid into blocks of columns with N consecutive columns in each block. The colours of the squares are then recorded as a list of N-bit numbers for each row. Using bit operations, we can now process N columns at once. Color[y][k] represents a block of N colours as bits in the following code.

```
int count=0;
for(int i=0;i<=n/N;i++){
    count += __builtin_popcount(color[a][i]& color[b][i]);
}
```

It will take complexity of O(n^2*(n/N)).

According to research a 2500x2500 random grid was created and compared the original and bit optimised implementations. The bit optimised version took only 3.1 seconds with N = 32 (int numbers) and 1.7 seconds with N = 64, compared to 29.6 seconds for the original code.

2.6.Bit masks

Bit masks means we are given a set and now we need to store the subset of that set in a less space,that is in an efficient manner.

Let's say S={1,2,6,7} is the given set

Now subsets could be S_0={1,2}

S_1={1,6}

……………..

Like this subsets will be up to 2^n.

We can store these subsets in vectors and sets but can we do better?? The answer is Yes.

Let's maintain a bool array which tells us that if i^{th} element is present then bit is 1 or else it is 0.

For example,

subset={1,4}

Array = [1,0,0,1,0] ---> this represents exactly what the subset is .

But can we do better than this? Again the answer is YES.

We know numbers can be represented in binary form,

5= 0….000101
6= 0….000110
.
..
…..
……..

Every number is represented in the binary form.

In the similar way let's take subset s={1,3,6}

If an element is present in the subset, put 1 at that position in the bit representation.

Now (0..0100101) = 37 is the formed number using that bit representation.

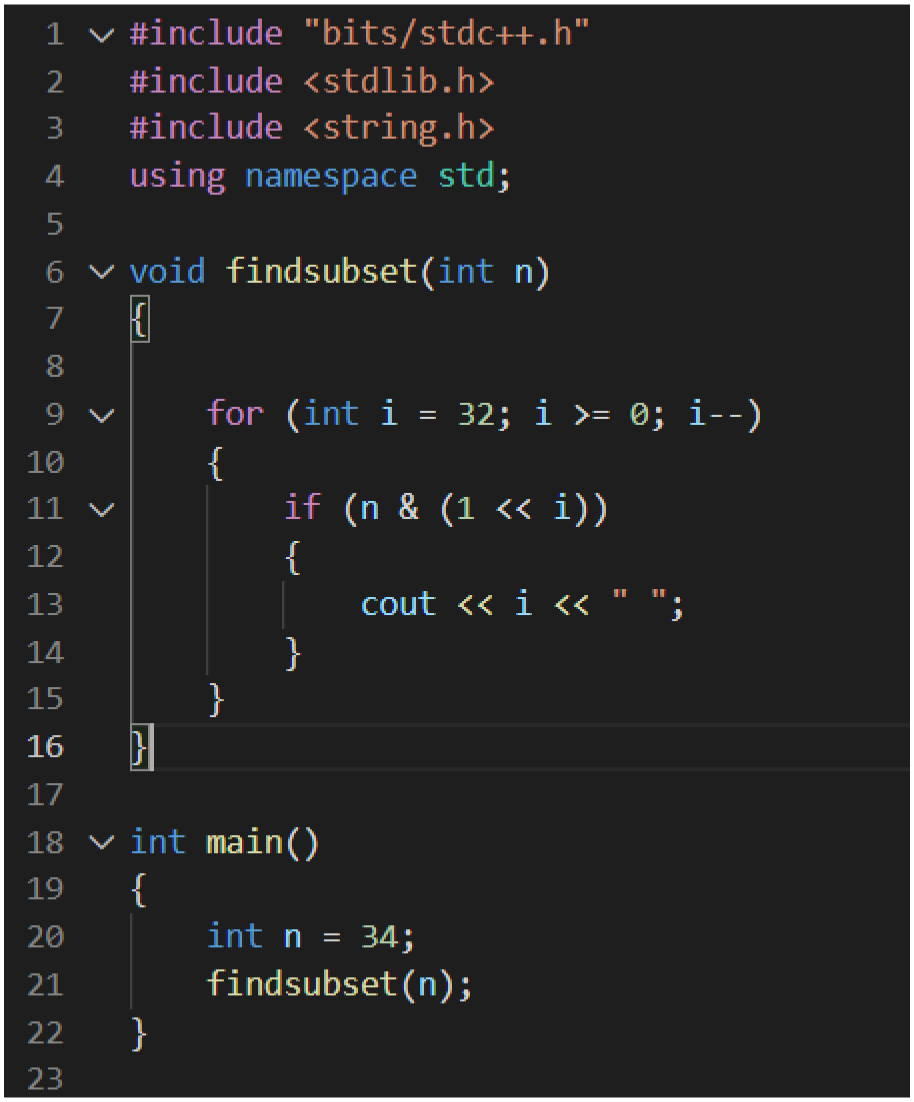
This 37 represents the subset s={1,3,6}.

Now we will implement some functions using bit masks.

(i) find subset

We are given a number N and we are asked to find the corresponding subset of that number.So what we need to do is to find the bits in the bit representation which are 1.Those are the included numbers of the set.

Now we look at the code for it,

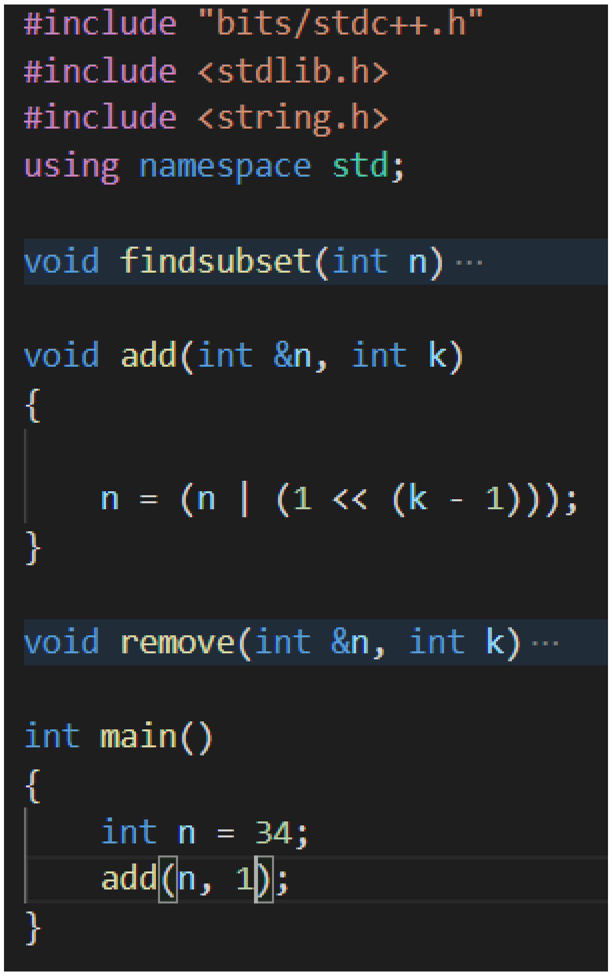


```cpp
#include "bits/stdc++.h"
#include <stdlib.h>
#include <string.h>
using namespace std;

void findsubset(int n)
{

    for (int i = 32; i >= 0; i--)
    {
        if (n & (1 << i))
        {
            cout << i << " ";
        }
    }
}

int main()
{
    int n = 34;
    findsubset(n);
}

```

For the above code it prints the result as **5 1 = (0...100010) = 34**

(ii) Add

If we want to include a number to the subset then we can just do the **or** operation in that particular bit's position which we want to add by 1.



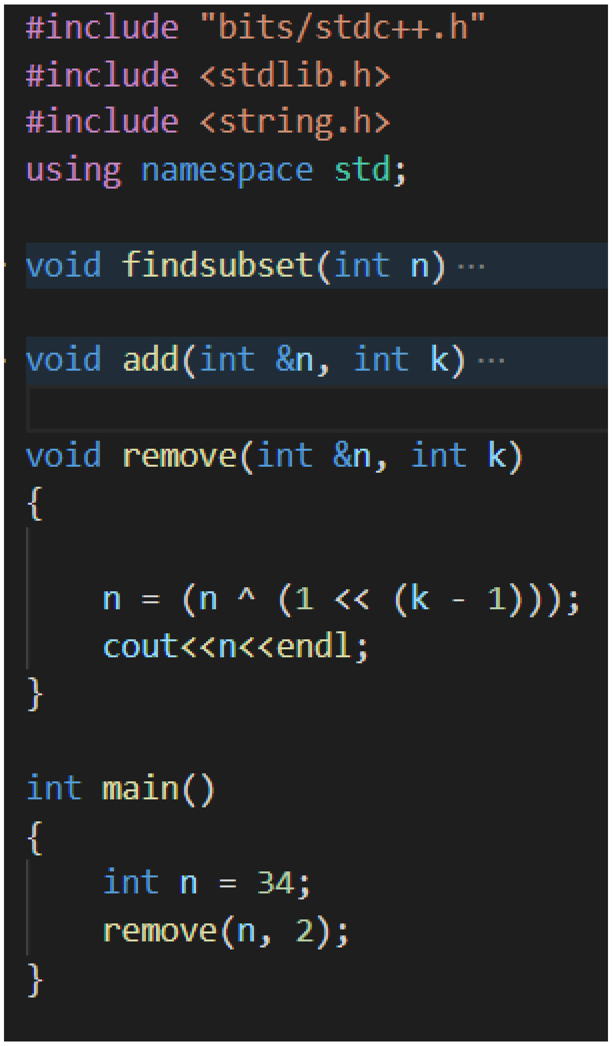
```
#include "bits/stdc++.h"
#include <stdlib.h>
#include <string.h>
using namespace std;

void findsubset(int n) ···

void add(int &n, int k)
{

    n = (n | (1 << (k - 1)));
}

void remove(int &n, int k) ···

int main()
{
    int n = 34;
    add(n, 1);
}
```

Result by the above code is 35 as we are adding the number 1 to the subset of 34 as a result it becomes 35.

0….100010 | 0….000001 = 0….100011 = 35

(iii) Remove

If we want to remove a number from the subset then we can just do the **xor** operation in that particular bit's position which we want to remove by 1.



```
#include "bits/stdc++.h"
#include <stdlib.h>
#include <string.h>
using namespace std;

void findsubset(int n) ···

void add(int &n, int k) ···

void remove(int &n, int k)
{

    n = (n ^ (1 << (k - 1)));
    cout<<n<<endl;
}

int main()
{
    int n = 34;
    remove(n, 2);
}
```

The above code outputs n as 32 because, we are changing the bit at position 2 to 0.So the resultant number we will be 32.

0….100010 ^ 0….000010 = 0….100000 = 32

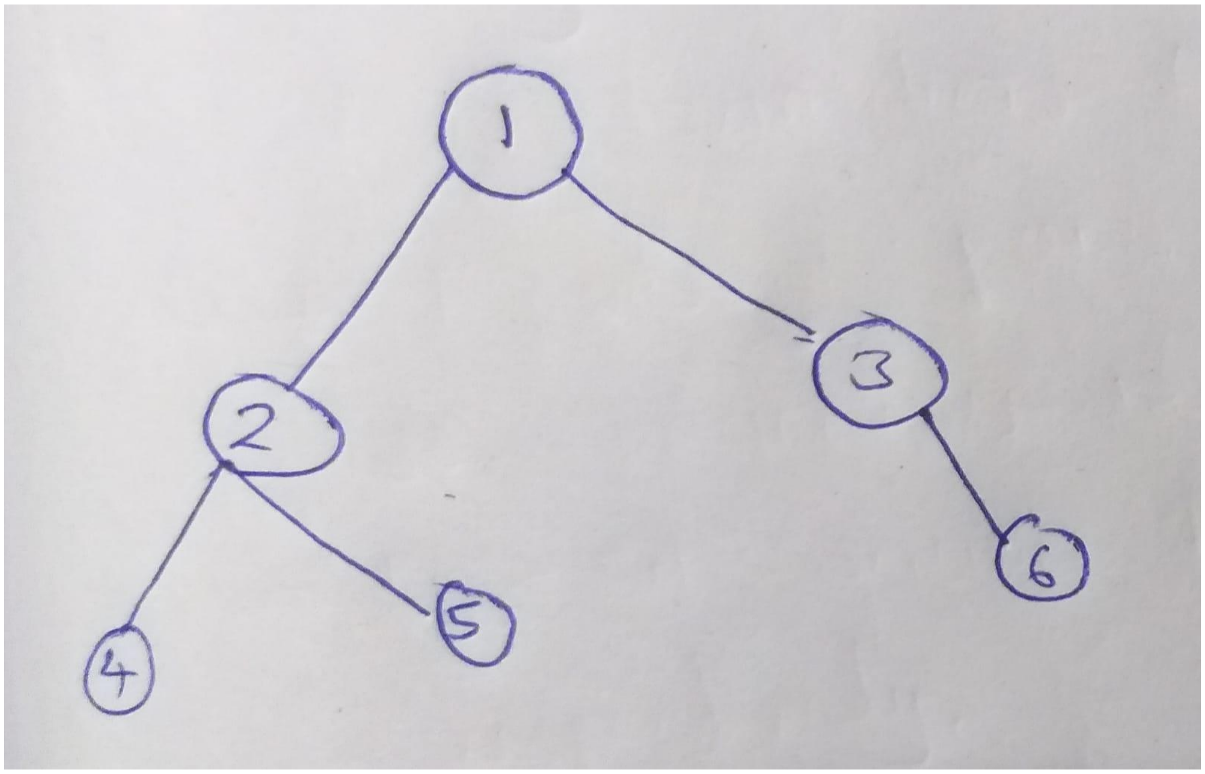
You can practice the questions on the link below on the above concept,

https://codeforces.com/problemset?tags=bitmasks,800-1200

3. Trees

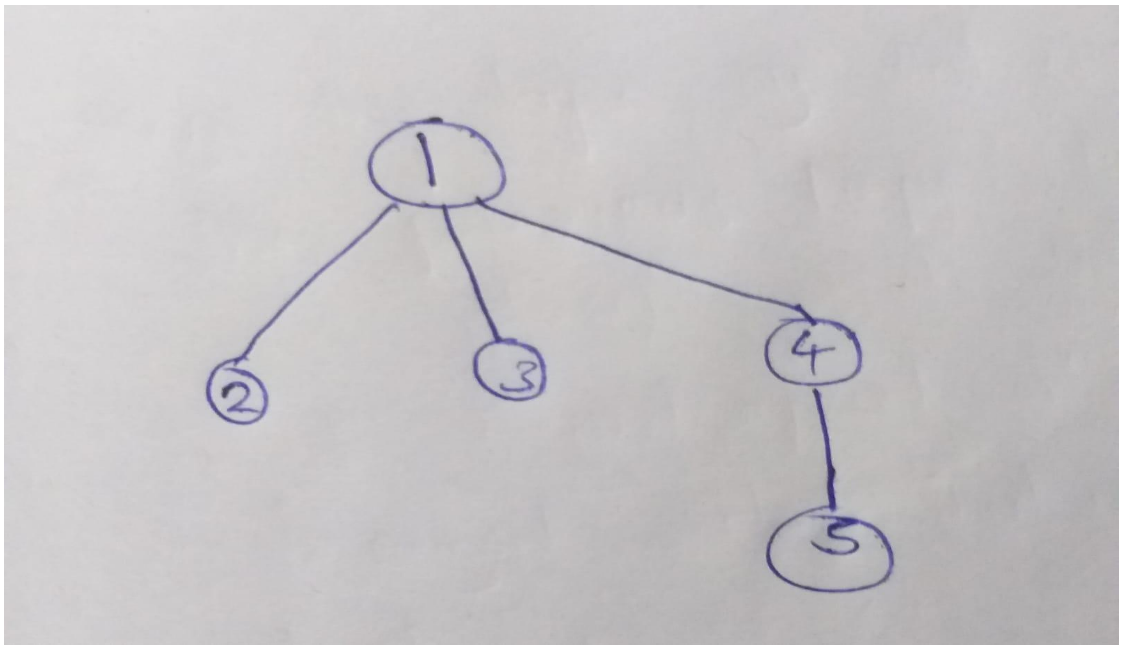
A tree is an acyclic graph with n nodes and n 1 edges that is connected.
Any edge that is removed from a tree divides it into two components, while any edge that is added to a tree generates a cycle. Furthermore, between any two nodes of a tree, there is always a unique path.

The following tree consists of 6 nodes and 5 edges:



The nodes of a tree of degree 1 are the leaves.

For example look at the following tree,



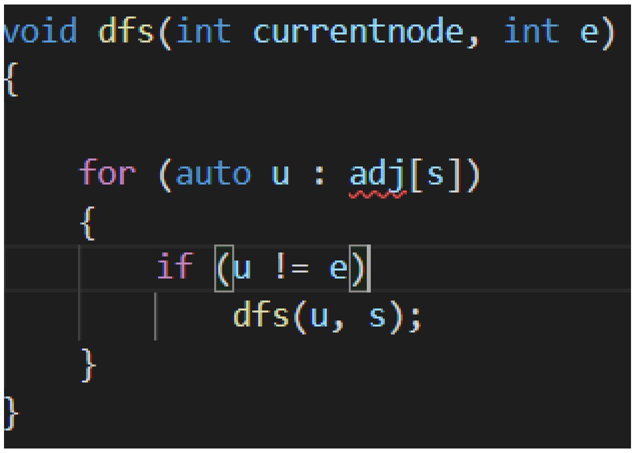
Leaves of the above tree are 2,3 and 5 and above tree is also called as rooted tree ,The root of the tree is designated by one of the nodes, and all other nodes are put beneath the root.

A rooted tree's structure is recursive: each node acts as the root of a subtree that comprises the node itself as well as all nodes in its children's subtrees.

3.1.Tree Traversal

To traverse the nodes of a tree, general graph traversal techniques can be utilised. Because there are no cycles in the tree and it is not feasible to approach a node from numerous routes, traversal of a tree is easier to construct than traversal of a generic graph.

Starting a depth-first search at an arbitrary node is a common approach to traverse a tree.See the code below.


```
void dfs(int currentnode, int e)
{

    for (auto u : adj[s])
    {
        if (u != e)
            dfs(u, s);
    }
}
```

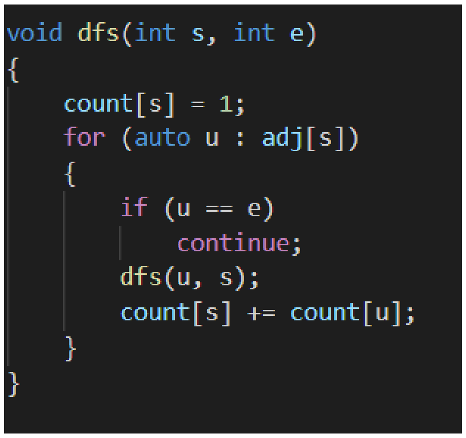
Two parameters are passed to the function: the current node s and the prior node e. The e option ensures that the search only proceeds to nodes that have not been visited previously.

The function call starts at node x ---> dfs(x,0)

3.2.Dynamic programming

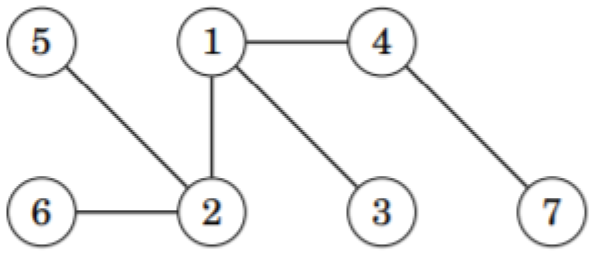
During a tree traverse, dynamic programming can be utilised to calculate certain information. We may, for example, calculate the number of nodes in a rooted tree's subtree or the length of the longest path from the node to a leaf in O(n) time using dynamic programming.

Let us construct a value count[s] for each node s: the number of nodes in its subtree. Because the subtree comprises the node and all nodes in its children's subtrees, Look at the following code to determine the number of nodes recursively.


```
void dfs(int s, int e)
{
    count[s] = 1;
    for (auto u : adj[s])
    {
        if (u == e)
            continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

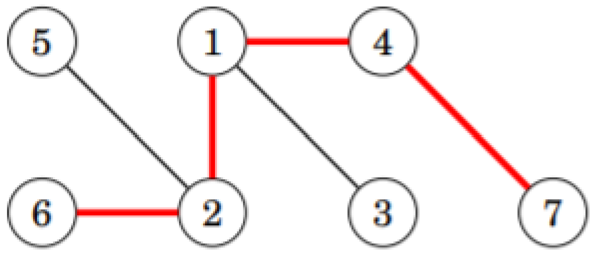
3.3. Maximum length of a path between two nodes

Look at the following diagram,



Maximum length of a path between two nodes is also called as “Diameter”

So now diameter for the above figure is,



This tree has a diameter of 4, which corresponds to the following path,those red lines.

It's worth noting that there could be many maximum-length pathways. We could substitute node 6 with node 5 in the above path to get a path with a length of 4 and if we have another node to 2 then we can substitute with that also.

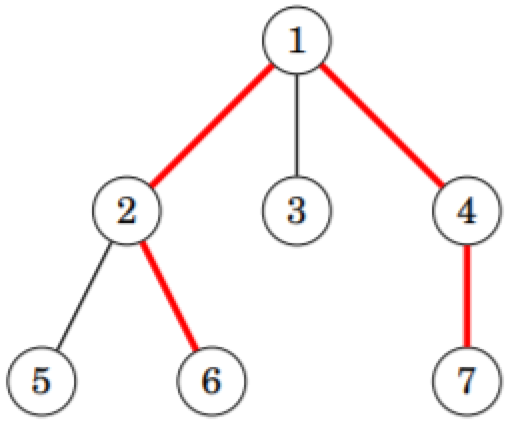
Next, we'll look at two O(n) time techniques for estimating a tree's diameter. The first algorithm employs dynamic programming, whereas the second employs two depth-first searches.

3.4. Algo 1

Many tree problems can be approached in a general fashion by first rooting the tree arbitrarily. After that, we can try to solve the problem for each subtree separately. This is the basis of our initial algorithm for estimating the diameter.

A key point to remember is that every path in a rooted tree has a highest point: the path's highest node.As a result, we can calculate the length of the longest path whose highest point is the node for each node. The diameter of the tree corresponds to one of the pathways.

Node 1 is the highest point on the path that corresponds to the diameter in the following tree:



We calculate x two values for each node:

- to_leaf(x) : the maximum distance between x and any leaf
- Max_length(x): the maximum length of a path whose highest point is x

Here I will explain with an example, in the above tree, to_leaf(1) = 2, because there is a path 1 → 2 → 6, also there is a path between 1→ 4→ 7 and 1→ 2→ 5. In all these paths maximum length is 2 only.

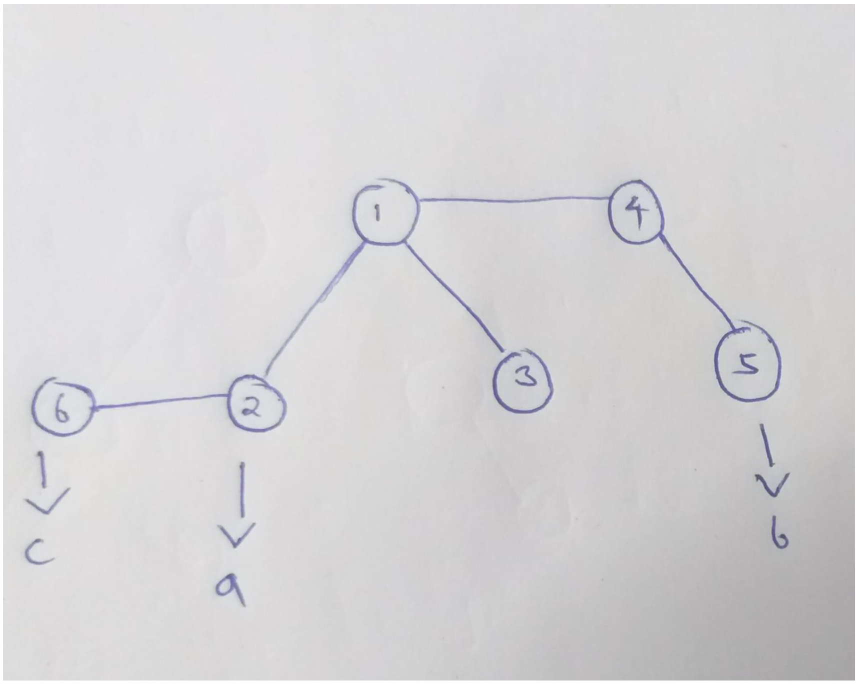
Max_length(1) = 4, because there is a path 6 → 2 → 1 → 4 → 7 and also 5 → 2 → 1 → 4 → 7. In both the cases, Max_length(1)=4 which equals the diameter.

The above mentioned values for all nodes can be calculated in O(n) time using **dynamic programming**. To compute to_leaf(x), we first go through x's children, selecting a child c with the highest to_leaf(c), and adding one to that value. Then, to calculate Max_length(x), we pick two separate childnodes, a and b, whose sum to_leaf(a)+to_leaf(b) is the greatest, and add two to that sum.

Algo 2

Two depth first searches are another effective approach to calculate a tree's diameter. First, we take an arbitrary node in the tree and search for the node b that is the furthest away from a. Then we identify the node c that is the furthest away from b. The distance between b and c is the tree's diameter.

For example look at the below diagram, The a,b,c values could be,



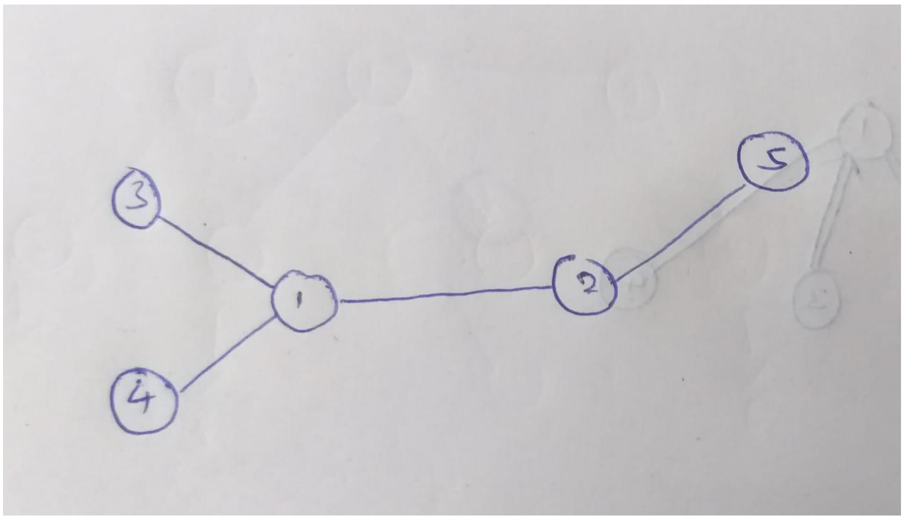
See from the figure if we select a node as “a” then farthest node from that node is “b” and farthest node from b is “c”.

So the longest path is from b to c with a diameter of 4.

3.5. All longest paths

The next task is to determine the maximum length of a path that starts at each node in the tree. Because the largest of the lengths equals the diameter of the tree, this can be considered as an extension of the tree diameter problem.

This problem can be solved in **O(n)** time.

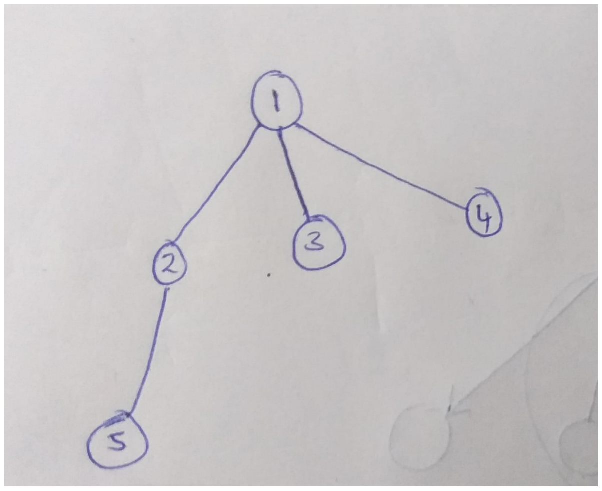


Here In the above tree, Max_length(4) = 3, because of the path that is 4 → 1 → 2 → 5.

The Complete table of the values shown below,

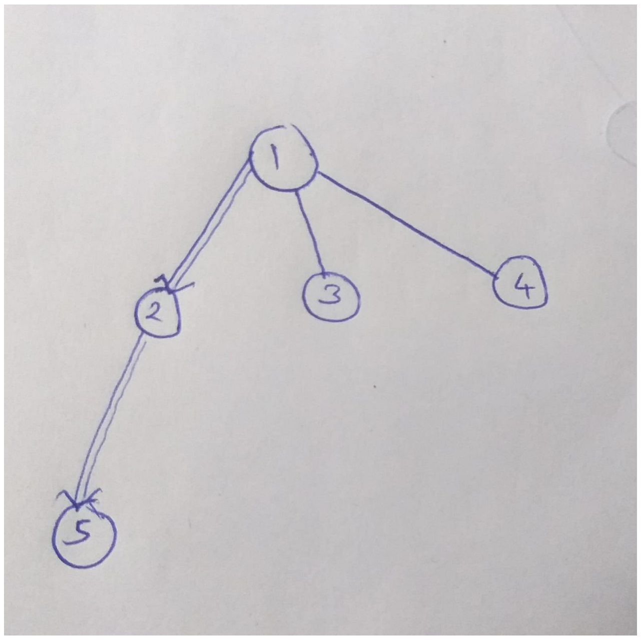
node x	1	2	3	4	5
Max_length(x)	2	2	3	3	3

In this problem, rooting the tree arbitrarily is also a viable beginning point for solving the problem.



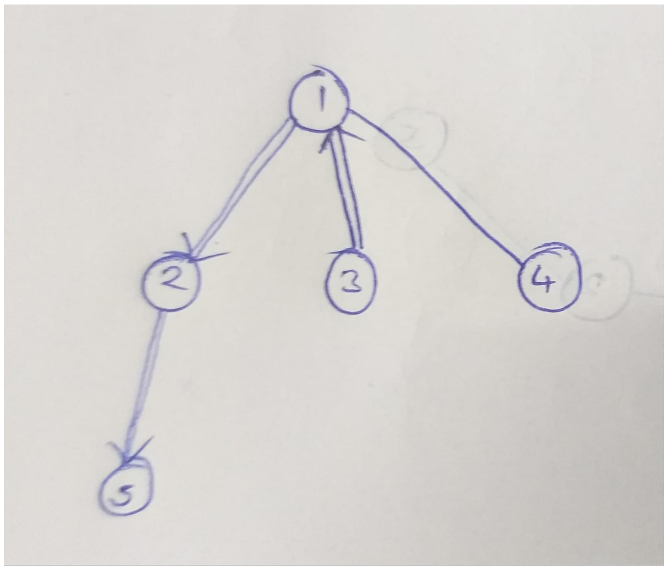
The first component of the problem is to determine the maximum length of a path that passes through a child of node x for each node x.

The longest path from node 1 through its child 2 is, for example.



Because we can utilise dynamic programming, as we did earlier, this component is simple to solve in **O(n)** time. The second component of the problem is to determine the maximum length of a path through each node x's parent p.

The longest path from node 3 to its parent 1 is, for example.



By storing two maximum lengths for each node x, we can solve the second half in **O(n)** time.

- Max_length1(x)
- Max_length2(x)

Max_length1(x) is the maximum length of a path from x

Max_length2(x) the longest path from x in a different direction than the first path

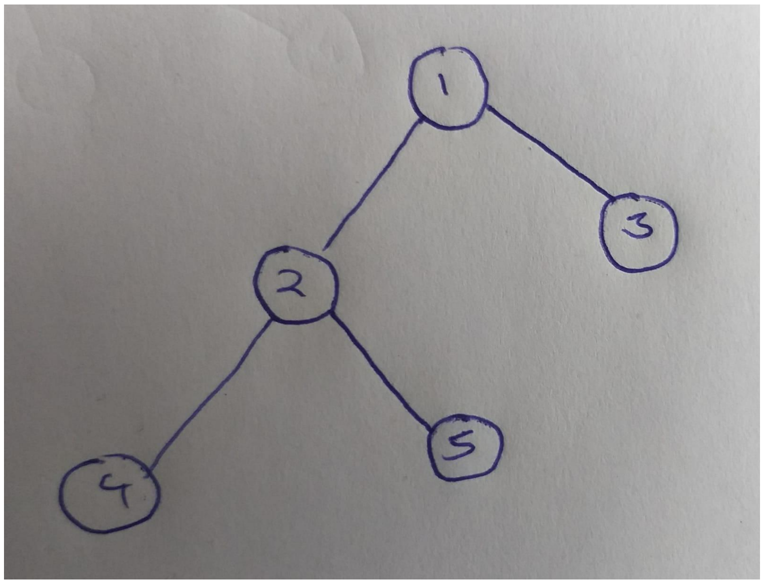
Here , In the above graph, Max_length1(1) = 2 because of the path 1 → 2 → 5, and maxLength2 (1) = 1 because of the path 1 → 3.

Finally, if the path corresponding to Max_length1(p) passes via x, we conclude that the maximum length is Max_length2(p)+1, and otherwise, Max_length1 (p)+1.

3.6.Binary trees

A binary tree is a rooted tree with left and right subtrees for each node. It's possible that a node's subtree is empty. In a binary tree, each node has zero, one, or two children.

Example for binary tree is,



A binary tree's nodes have three natural orderings, each of which corresponds to three alternative ways to recursively traverse the tree.

- **Preorder :** Process the root first, then the left subtree, and finally the right subtree.

Pseudo code as follows,

```
function preorder(p : pointer to a tree node)
  if p != nullptr
     Visit the node pointed to by p
     preorder(p->left)
     preorder(p->right)
```

For the above binary tree Preorder traversal is 1 2 4 5 3

- **Inorder :** The left subtree is visited first, followed by the root, and finally the right subtree in this traversal strategy.

Pseudo code as follows,

```
Function inorder(p : pointer to a tree node)
 if p != nullptr
     inorder(p->left)
     Visit the node pointed to by p
     inorder(p->right)
```

For the above binary tree Inorder traversal is 4 2 5 1 3

- **Postorder :** The left subtree is traversed first, followed by the right subtree, and ultimately the root node.

Pseudo code as follows,

```
Function postorder(p : pointer to a tree node)
  if p != nullptr
    postorder(p >left)
    postorder(p->right)
    Visit the node pointed to by p
```

For the above binary tree Postorder traversal is 4 5 2 3 1

Practice problems on the below link based on above concepts,

https://codeforces.com/problemset?tags=trees,800-1200

4. Strings

This chapter discusses string processing algorithms that are both efficient and effective. Many string problems can be solved in O(n^2) time, but finding methods that work in O(n) or O(nlogn) time is a difficulty.

The pattern matching problem, for example, is a fundamental string processing problem: given a string of length n and a pattern of length m, our objective is to discover all occurrences of the pattern in the string. The pattern ABC, for example, appears twice in the string ABABCBABC.

A brute force approach that checks all spots where the pattern may occur in the string can readily solve the pattern matching problem in O(nm) time. However, we will see in this chapter that there are more efficient algorithms that only take O(n+ m) time.

4.1.String Terminology

Throughout the chapter, we'll assume that strings employ zero-based indexing. As a result, characters s[0],s[1],...,s[n1] make up a string s of length n. An alphabet is a collection of letters that can occur in strings.For example, the alphabet {A,B,...,Z} consists of the capital letters of English.

In a string, a substring is a group of consecutive characters. A substring of s that starts at location a and ends at position b is denoted by the notation s[a...b]. There are n(n + 1)/2 substrings in a string of length n. A, B, C, D, AB, BC, CD, ABC, BCD, and ABCD are all substrings of ABCD.

Subsequence is a group of characters in a string that are not always in the same order. There are 2^n-1 subsequences in a string of length n. A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD, and ABCD are examples of ABCD subsequences.

A prefix is a substring that begins at the beginning of a string , whereas a suffix is a substring that ends at the end of a string. ABCD, for example, has the prefixes A, AB, ABC, and ABCD, as well as the suffixes D, CD, BCD, and ABCD.

A rotation can be created by moving each character of a string from the beginning to the finish one by one (or vice versa). ABCD's rotations, for example, are ABCD, BCDA, CDAB, and DABC.
A period is a string prefix that can be used to build the string by repeating the period. The last repeat could be incomplete, with merely a period prefix. The shortest period of ABCABCA, is ABC.

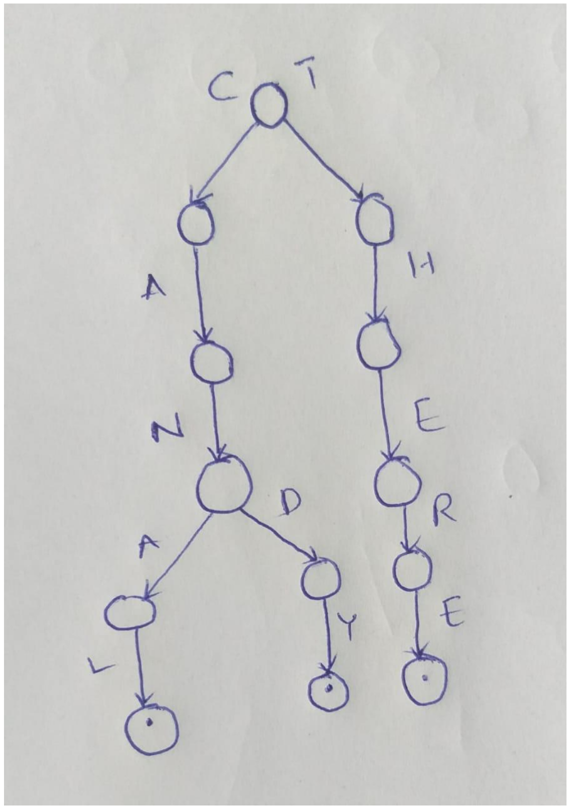
A border is a string that functions as both a prefix and a suffix. ABACABA's borders, for example, are A, ABA, and ABACABA.

The lexicographical order (which corresponds to the alphabetical order) is used to compare strings .It means that x < y if either x != y and x is a prefix of y.

4.2.Trie Structure

A trie is a rooted tree that keeps track of a string set. Each string in the set is saved as a sequence of characters beginning at the root. If two strings have the same prefix, they will have the same tree chain.

Look at the following trie,



The set {CANAL,CANDY,THE,THERE} corresponds to this trie. A node with the character "." indicates that a string in the set ends at that node. Because a string can be a prefix of another string, such a character is required.

Because we can trace the chain that starts at the root node, we can verify whether a trie includes a string of length n in **O(n)** time. By first following the chain and then adding additional nodes to the trie if required, we may add a string of length n to the trie in **O(n)** time.

We can discover the longest prefix of a given string that belongs to the set by using a trie. We can also determine the number of strings that belong to the set and have a specific string as a prefix by storing additional information in each node.

A trie can be stored in an array ,

int trie[N][A];

N is the maximum number of nodes and A is the alphabet's size. The nodes of a trie are numbered 0,1,2,... so that the number of the root is 0, and trie[s][c] is the next node in the chain when we move from node s using character c.

4.3 String hashing

String hashing is a technique for quickly determining whether two strings are equivalent. The objective behind string hashing is to compare string hash values rather than individual Characters.

Calculating hash values

A string's hash value is a number derived from the characters in the string. When two strings are identical, their hash values are likewise same, allowing you to compare strings based on their hash values.

Polynomial hashing is a common approach to do string hashing, which means that the hash value of a string s of length n equals,

$(s[0]A^{n-1} + s[1]A^{n-2} + \cdots + s[n-1]A^{0})$ mod B,

where s[0], s[1],..., s[n-1], are the codes for the characters in s, and A and B are pre-determined Constants.

For example,if A=2 and B=96 then

NITHIN →	N	I	T	H	I	N
	78	73	84	72	73	78

$(78.2^{5} + 73.2^{4} + 84.2^{3} + 72.\ 2^{2} + 73.2^{1} + 78.2^{0})$ mod 96 = 48

Preprocessing

After an O(n) time preprocessing, we can determine the hash value of each substring of a string s in O(1) time using polynomial hashing. The goal is to create an array h with the hash value of the prefix s[0...k] stored in h[k]. The array values may be computed in a recursive manner as follows:

h[0] = s[0]

h[k] = (h[k −1]A +s[k]) mod B

Furthermore, we create an array p, where p[k] Equals $A^k \mod B$.

p[0] = 1

p[k] = (p[k −1]A) mod B.

It takes O(n) time to build these arrays. After that, using the formula, the hash value of any substring s[a...b] may be determined in O(1) time.

(h[b]−h[a−1]p[b − a+1]) mod B

assuming that a is greater than zero ,The hash value is just h[b] if a = 0.\

Using hash values

Using hash values, we can compare strings quickly. The aim is to compare the hash values of the strings rather than the individual characters. If the hash values are the same, the strings are almost surely the same; if the hash values are different, the strings are almost certainly different.

We can typically make a brute force approach more efficient by using hashing. Consider the pattern matching problem: given a string s and a pattern p, discover the places in s where p appears. A brute force method analyses the strings character by character, going through all possible spots where p may appear. The complexity of such an algorithm in terms of time is O(n^2).

Because the process compares substrings of strings, we may make the brute force algorithm more efficient by applying hashing. Because only hash values of substrings are compared when hashing is used, each comparison takes just O(1) time. As a result, a method with time complexity O(n) is generated, which is the best time complexity for this task.

It is also feasible to obtain the lexicographic order of two strings in logarithmic time by combining hashing and binary search. This may be done by utilising binary search to calculate the length of the common prefix of the strings.We can simply check the following character after the prefix to establish the order of the strings once we know the length of the common prefix.

Collisions and parameters

A collision, which occurs when two strings have distinct contents but same hash values, is an obvious issue when comparing hash values. In this scenario, a hash value-based method concludes that the strings are equivalent, but they are not, and the process may provide *inaccurate results*.Because the number of distinct strings is greater than the number of various hash values, collisions are always possible. However, if the constants A and B are carefully selected, the likelihood of a collision is low. Choosing random constants near 10^9 is a common method.

For example,

A= 9123456899
B= 8009933113

But we may get a doubt that is it enough to have 10^9 hash values ?

Consider the following three instances in which hashing may be useful:

- A comparison is made between the strings x and y. If all hash values are equally likely, the likelihood of a collision is 1/B.
- Strings $y_1, y_2, \ldots, y_n$ are compared to string x.The probability of one or more collisions is,

$$1 - (1 - 1/B)^n$$

- The strings $x_1, x_2, \ldots, x_n$ are all compared to one another.The probability of one or more collisions is,

$$1 - \frac{B.(B-1).....(B-n+1)}{B^n}$$

When n = 10^6 and B fluctuates, the following table displays the collision probability.

B	Case 1	Case 2	Case 3
10^3	0.00100	1.00000	1.00000
10^6	0.00001	0.63212	1.00000
10^9	0.00000	0.00100	1.00000
10^{12}	0.00000	0.00000	0.39346
10^{15}	0.00000	0.00000	0.00050
10^{18}	0.00000	0.00000	0.00001

When B $\approx 10^9$ The likelihood of a collision in case 1 is small, as shown in the table.In case 2, a collision is possible but the probability is quite small. In case 3, however, the situation is significantly different: at B $\approx 10^9$, a collision will nearly definitely occur.

The birthday paradox describes the phenomena in Case 3: if there are n persons in a room, the likelihood that two of them share the same birthdate is high, even though n is small. When all hash values are compared to each other in hashing, the likelihood that some two hash values are equal is high.By creating numerous hash values with varied parameters, we may reduce the likelihood of a collision. It's rare that all of the hash values will collide at the same moment. Two hash values with parameter B $\approx 10^9$, for example, equate to one hash value with parameter B $\approx 10^{18}$, making the chance of a collision extremely low.

4.4. Pattern Searching

Our job is to find the pattern in the string given a pattern and a string. We'll start with the Naive algorithm, which is a fundamental algorithm.

For example Let's say we have string of length M = “abcde” and pattern of length n =”bcd”

We will keep on iterating through Given string along with pattern simultaneously and then check for pattern matching.

Like for the above example,

```
a  b  c  d  e  → a  b  c  d  e  → a  b  c  d  e  → a  b  c  d  e
i                 i                 i                       j

b  c  d         → b  c  d         → b  c  d         → b  c  d
j                 j                     j                   j
```

In this way our basic algorithm works but drawback is it repeatedly checks for the pattern,

In the worst case if we have string like this, **aaaaaaab** of length m and pattern **aab** of length n

Then every time i value increases and decreases due to dismatching at the end,

Like if we see here,

```
a a a a a a a a b → a a a a a a a a b → a a a a a a a a b → a a a a a a a a b
i                     i                      i                  i

a  a  b             → a  a  b            → a  a  b            → a  a  b
j                        j                    j                    j
```

In this way its keep on search for the pattern without knowing what the string is in this case it takes time complexity of **O(m*n) .**

Can we now reduce the time it takes to find the pattern??

Now we will look at the working of KMP algorithm(Knuth-Morris-Pratt).

If we have a string **ababcababd** of length M**.**

And a pattern of **ababd** of length N.

```
String :  a b  a  b  c   a b a b d
indexes: 0 1  2  3  4   5 6 7 8 9
```

```
Pattern:  a   b   a   b  d
```

indexes: 1   2   3   4 5

In the pattern string only 2 alphabets those are **a** and **b** are repeating again,So what we will do is

We will put second occurrence of a to be 1 and second occurrence of b to be 2.

- We will compare first if i==j,then we will move j aswell as i,

i									
a	b	a	b	c	a	b	a	b	d
0	1	2	3	4	5	6	7	8	9

j				
1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

- Do it again,

	i								
a	b	a	b	c	a	b	a	b	d
0	1	2	3	4	5	6	7	8	9

	j			
1	2	3	4	5
a	b	a	b	d

0	0	1	2	0

In the same way if we do it again and again after some time we will have the situation,

				i					
a	b	a	b	c	a	b	a	b	d
0	1	2	3	4	5	6	7	8	9

				j
1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

Now we have unmatching with i and j so now we move the index j to the left most 0.

					i				
a	b	a	b	c	a	b	a	b	d
0	1	2	3	4	5	6	7	8	9

j				

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

Now the pattern i==j then we will increment i and j and gain i==j again we will increment i and j. So in the sameway,we will end up with,

									i
a	b	a	b	c	a	b	a	b	d
0	1	2	3	4	5	6	7	8	9

				j
1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

So , as we reached the end of pattern in pattern string so there pattern matching in the string.

Our earlier basic algorithm takes time complexity of **O(M*N)**

Now in this algorithm we have to parse the main string once so M and for preparing the table N.

So overall time complexity becomes **O(M+N)**.

Note that we are backtracking only Pattern not the original string.

Practice problems on the below link based on the above concepts,

https://codeforces.com/problemset?tags=strings,800-1200

---THE END--