Tech interview

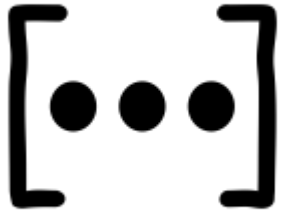
Practice

Dmitry Ryabokon, github.com/dryabokon

Typical assignments

- Arrays/Lists/Stacks
- Queue +
- Hashing
- Heap
- Sorting
- BST
- Dynamic programming
- Recursion +
- Bits Manipulation
- Math

Arrays/Lists/Stacks



Arrays/Lists/Stacks

- **List vs Set**
  - Set: unsorted, unique
  - List: ordered(?), allow dups

- **Array vs List(=Vector)**
  - List (+): Dynamic size, quick insert/delete
  - List (-): quick random access

Arrays/Lists/Stacks

- Array**List**
  - resizable array.
  - size is increased dynamically.
  - elements can be accessed directly, since ArrayList is essentially an array.
  - multiple threads can work on ArrayList at the same time

- Linked**List**
  - implemented as a double linked list
  - better performance on add/remove
  - worse performance on access

- **Vector** = synchronized Array**List**
  - Single thread can work on at the same time

Arrays/Lists/Stacks

- **Hash Map vs Hash Set**
  - Set: unique

- **Hash map vs Hash Table**
  - Hash Table: only one thread can access, does not allow NULL
  - Hash Map : many threads can access, allows NULL

Arrays/Lists/Stacks

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Arrays/Lists/Stacks

- Implement algorithm to check if string has all unique characters. What if you cannot use additional data structures
- Given two strings, write a method to decide if one is a permutation of the other.
- Given a string, write a function to check if it is a permutation of a palindrome

Strings

recur_02 Count triplets with sum smaller than a given value:
sort_06 Convert array into Zig-Zag fashion
search_03 Pythagorean Triplet in an array
Find the smallest positive integer value that cannot be represented as sum of any subset of a given array
Smallest subarray with sum greater than a given value
Stock Buy Sell to Maximize Profit
All Possible Palindromic Partitions

Generate all possible sorted arrays from alternate elements of two given sorted arrays
Reverse an array without affecting special characters
Length of the largest subarray with contiguous elements

Queue

Length of longest valid substring

```
# Given a string S consisting of opening and closing parenthesis '(' and ')' f
# find length of the longest valid parenthesis substring.
# ---------------------------------------------------------------------------------------------------
def find_valid_substring(A):

    stack =[-1]
    longest_length,res=0,None

    for i,a in enumerate(A):
        print(stack)
        if a=='(':
            stack.append(i)
        else:
            stack.pop()
            if len(stack)>0:
                valid_length = i - stack[-1]
                if valid_length>longest_length:
                    longest_length = valid_length
                    res = A[i - valid_length + 1:i + 1]
            else:
                stack.append(i)

    return longest_length,res
```

Jumping numbers

```
# Find jumping numbers <= X.  Example: 7, 8987 and 4343456 are Jumping numbers but 796 and 89098 are not.
# A number is called as a Jumping Number if all adjacent digits in it differ by 1
# ---------------------------------------------------------------------------------------------------
def find_jumping_num2(x):

   res = []
   queue = list('123456789')

   while len(queue)>0:
      value = int(queue.pop())
      if value>x:continue
      res.append(value)
      last_digit = value%10
      if last_digit >= 1:
         queue.append(value*10+last_digit-1)

      if last_digit <= 8:
         queue.append(value*10+last_digit+1)

   res.sort()

   return res
```

Is valid string

```
A = list('()(((()))()(')
# ----------------------------------------------------------------------------------------------------
def is_valid_substring2(A):
   def is_pair(a,b):return (a=='(' and b==')')or(a=='[' and b==']')or(a=='{' and b=='}')
   queue = [A[0]]
   for a in A[1:]:
      if len(queue)==0:
         queue.append(a)
         continue

      if is_pair(queue[-1],a):
         queue.pop()
      else:
         queue.append(a)

   return len(queue)==0
```

Hashing

Hashing

- Largest subarray with 0 sum
- Swapping pairs make sum equal
- Count distinct elements in every window
- Array Pair Sum Divisibility Problem
- Longest consecutive subsequence
- Array Subset of another array
- Find all pairs with a given sum
- Find first repeated character
- Zero Sum Subarrays
- Minimum indexed character
- Check if two arrays are equal or not
- Uncommon characters
- Smallest window containing all the characters of another string
- First element to occur k times
- Check if frequencies can be equal

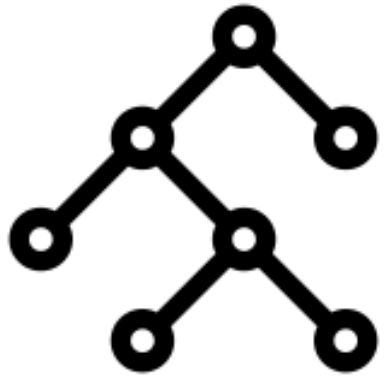
Two Sum

```
dct = {}

for each in A:
    if target-each not in dct:
        dct[target-each] = each
    else:
        if each==target//2:
            return[each, each]

for each in A:
    if each in dct and each!=target//2:
        return [each, dct[each]]
```

[3,3,2,4]

[3,2,4]

Heap



Heap

```
def heapify(A, i):
    N = len(A)
    left, right   = (i+1)*2-1, (i+1)*2
    argmax = i

    if left  < N and A[left]  > A[argmax]: argmax = left
    if right < N and A[right] > A[argmax]: argmax = right

    if argmax!= i:
        A[i],A[argmax] = A[argmax],A[i]
        heapify(A, argmax)
    return
# ---------------------------------------------------------
def build_heap(A):
    i=len(A)//2
    while i>=0:
        heapify(A, i)
        i-=1

    return
```

37871209 -> 97871203

Quick Select k-th best with heap

```
def best_kth_element(A, k):

    H = A.copy()
    heap.build_heap(H)

    res=H[0]
    for i in range(0,k-1):
        H[0] = H[-1]
        del(H[-1])
        heap.heapify(H, 0)
        res = H[0]

    return res
```

[3, 7, 8, 7, 1, 2, 0]

[8, 7, 3, 7, 1, 2, 0]

[7, 7, 3, 0, 1, 2]

[7, 2, 3, 0, 1]

[3, 2, 1, 0]

[3, 7, 8, 7, 1, 2, 0]

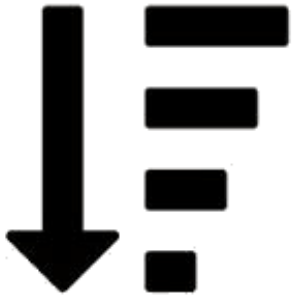
[8, 7, 3, 7, 1, 2, 0]
[0, 7, 3, 7, 1, 2]
[7, 7, 3, 0, 1, 2]

[7, 7, 3, 0, 1, 2]
[2, 7, 3, 0, 1]
[7, 2, 3, 0, 1]

[7, 2, 3, 0, 1]
[1, 2, 3, 0]
[3, 2, 1, 0]

[3, 2, 1, 0]
[0, 2, 1]
[2, 0, 1]

Sorting



Sorting

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Sorting

- **Bubble Sort:** swap elements
- **Selection Sort:** Linear search for a smallest element => move it to the front.
- **Merge Sort:** split/sort/sort/merge
- **Quick Sort:** partition/sort/sort
- **Heap search**
- **Binary search:**
- **Bucket sort:**

The quick sort is internal sorting method where the data is sorted in main memory.

The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.

Sorting: k-th best element

- **Alg0**: sort and select
- NlogN

- **Min Heap – HeapSelect (link)**
- N + K*Log N

- **Max-Heap**
- K + (N-K)*Log K

- **Quick select**
- Average: N

Sorting

Binary Search
Bubble Sort
Insertion Sort
Merge Sort
Heap Sort (Binary Heap)
Quick Sort
Find Kth Smallest/Largest Element In Unsorted Array
Search an element in a sorted and rotated array
Interpolation Search
Given a sorted array and a number x, find the pair in array whose sum is closest to x

Partition

```
def partition(A,left,right):
    p = left - 1
    pivot = A[right]
    for i in range(left, right):
        if pivot < A[i]:
            p+= 1
            A[p], A[i] = A[i], A[p]
    p += 1
    A[p], A[right] = A[right], A[p]
    return p
```

[3, 7, 8, 7, 1, 2, 5]
[7, 3, 8, 7, 1, 2, 5]
[7, 8, 3, 7, 1, 2, 5]
[7, 8, 7, 3, 1, 2, 5]
[7, 8, 7, 5, 1, 2, 3]

best case
[3, 7, 8, 7, 1, 2, 9]
[9, 7, 8, 7, 1, 2, 3]

worst case
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]
[3, 7, 8, 7, 1, 2, 0]

Quick Select k-th best

```
def quick_select(A, k):

    if k>len(A) or k<=0:return []
    if len(A)==1 and k==1:return A[0]

    p = partition(A, 0, len(A)-1) #A[:p] > A[p+1:]

    if p == k - 1:
        return A[p]
    elif p > k - 1: return quick_select(A[:p], k)
    else:           return quick_select(A[p + 1:], k - p - 1)
```

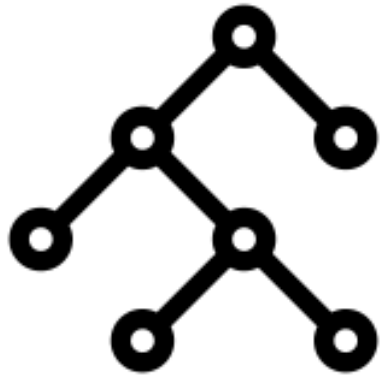
Quick Sort

```
def quick_sort(A,low,high):

    if low < high:
        p = partition(A, low, high)
        quick_sort(A, low, p - 1)
        quick_sort(A, p + 1, high)

    return
```

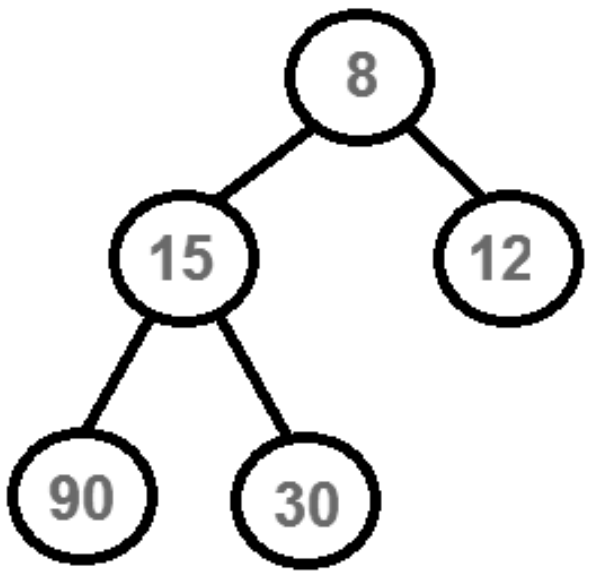
Merge Sort

```
def merge_sort(A):

    if len(A) < 2:
        return

    mid = len(A) // 2
    L = A[:mid].copy()
    R = A[mid:].copy()

    merge_sort(L)
    merge_sort(R)

    i,j,k = 0,0,0

    while i < len(L) and j < len(R):
        if L[i] > R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        k += 1

    while i < len(L):
        A[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        A[k] = R[j]
        j += 1
        k += 1
```

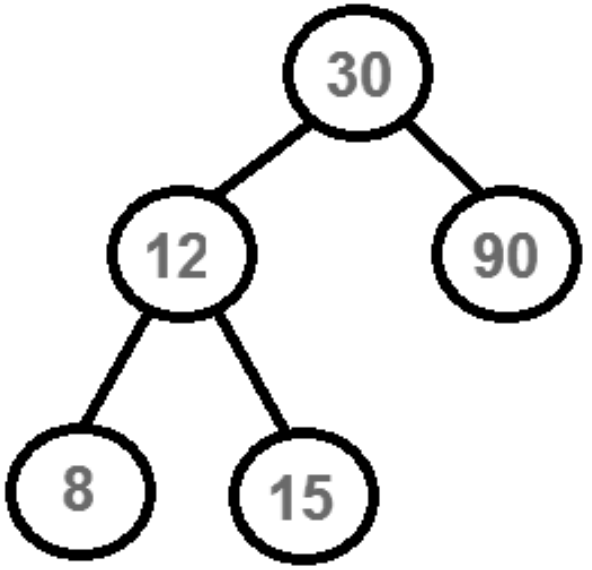
BST



BST vs Heap



Min Heap



Binary Search Tree

BST vs Heap

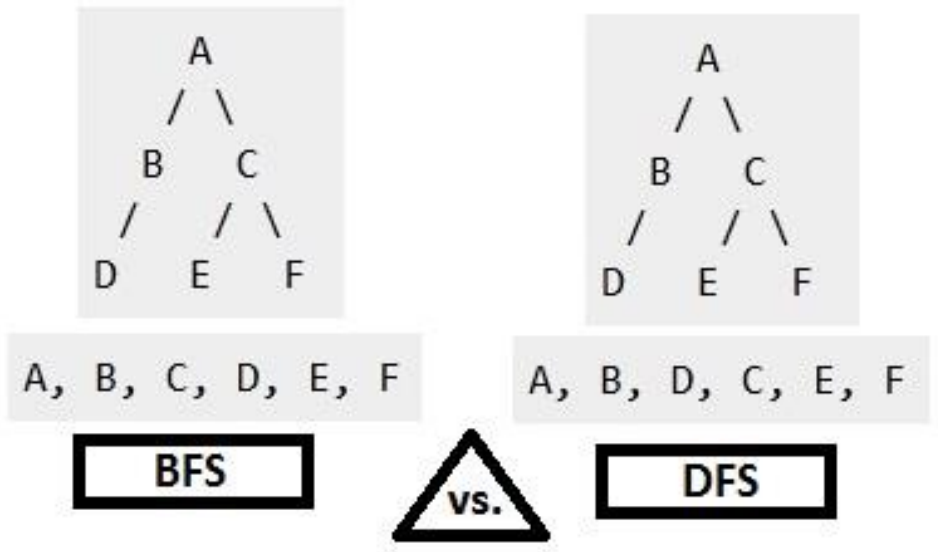
- **Binary Search Tree**
- search for any elements is O(logN)

- **Heap**
- search Min/Max: O(1)

- https://stackoverflow.com/questions/6147242/heap-vs-binary-search-tree-bst
- If you only care about find Min/Max (e.g. priority-related), go with heap.
- If you want everything sorted, go with BST

BFS vs DFS

	Breadth First Search	Depth First Search
Basic	Vertex-based algorithm	Edge-based algorithm
Data structure	Queue	Stack
Structure of the constructed tree	Wide and short	Narrow and long



BST

1. Find Minimum Depth of a Binary Tree
2. Maximum Path Sum in a Binary Tree
3. Check if a given array can represent Preorder Traversal of Binary Search Tree
4. Check whether a binary tree is a full binary tree or not
5. Bottom View Binary Tree
6. Print Nodes in Top View of Binary Tree
7. Remove nodes on root to leaf paths of length < K
8. Lowest Common Ancestor in a Binary Search Tree
9. Check if a binary tree is subtree of another binary tree
10. Reverse alternate levels of a perfect binary tree
11. Red-Black tree

BST

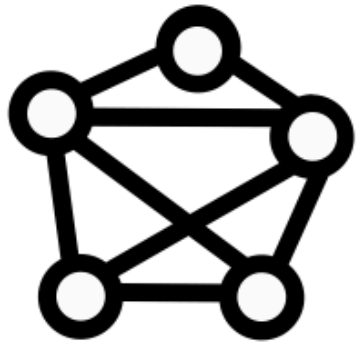
- Check for BST
- Lowest Common Ancestor in a BST
- Write Code to Determine if Two Trees are Identical or Not
- Height of Binary Tree
- Check if given Binary Tree is Height Balanced or Not
- Serialize and Deserialize a Binary Tree

- Print Left View of Binary Tree
- Print Bottom View of Binary Tree
- Print a Binary Tree in Vertical Order
- Level order traversal in spiral form
- Connect Nodes at Same Level
- Convert a given Binary Tree to Doubly Linked List
- Given a binary tree, check whether it is a mirror of itself
- Maximum Path Sum
- Diameter of a Binary Tree
- Number of leaf nodes

Algorithms: graph

1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. Shortest Path from source to all vertices **Dijkstra**
4. Shortest Path from every vertex to every other vertex **Floyd Warshall**
5. To detect cycle in a Graph **Union Find**
6. Minimum Spanning tree **Prim**
7. Minimum Spanning tree **Kruskal**
8. Topological Sort
9. Boggle (Find all possible words in a board of characters)
10. Bridges in a Graph

Dynamic programming



Dynamic programming

Longest Common Subsequence
Longest Increasing Subsequence
Edit Distance
Ways to Cover a Distance
Longest Path In Matrix
Optimal Strategy for a Game

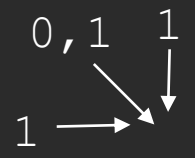
0-1 Knapsack Problem
Boolean Parenthesization Problem
Flood fill Algorithm
Number of paths
Combination Sum – Part 2
Special Keyboard
Water Overflow
Josephus problem

Minimum Partition
Subset Sum Problem

Math

1. Modular Exponentiation
2. Modular multiplicative inverse
3. Primality Test | Set 2 (Fermat Method)
4. Euler's Totient Function
5. Sieve of Eratosthenes
6. Convex Hull
7. Basic and Extended Euclidean algorithms
8. Segmented Sieve
9. Chinese remainder theorem
10. Lucas Theorem

Levenshtain




```python
def levenshtain_distance2(A1,A2):

    D = numpy.zeros((len(A1),len(A2)),dtype=numpy.int)
    D[:, 0] = numpy.arange(0, D.shape[0], 1)
    D[0, :] = numpy.arange(0, D.shape[1], 1)

    for i in range(1,len(A1)):
        for j in range(1,len(A2)):
            cost = 1-1*(A1[i]==A2[j])
            D[i,j] = min(D[i-1][j-1]+cost,1+D[i-1][j],1+D[i][j-1])


    print(D)
    return D[-1,-1]
```

```
      c a t
  [[0 1 2 3]
c [1 0 1 2]
i [2 1 1 2]
t [3 2 2 1]]

      c a t h y
  [[0 1 2 3 4 5]
c [1 0 1 2 3 4]
a [2 1 0 1 2 3]
t [3 2 1 0 1 2]]

      c o n
  [[0 1 2 3]
c [1 0 1 2]
a [2 1 1 2]
m [3 2 2 2]
o [4 3 2 3]
n [5 4 3 2]]

C +A +M ON
```

Staircase

```
def count_number_of_ways2(N,steps=[1,2]):

    D = numpy.zeros(N,dtype=numpy.int)
    for step in steps:
        D[step-1]=1

    for n in range(N):
        for step in steps:
            if n-step>=0:
                if D[n-step]>0:
                    D[n]+=D[n-step]


    print(D)
    return D[-1]
```

Recursion

Recursion

- Staircase
- Fibonacci
- All permutations
- All subsets
- Has Sub-array
- All Subarrays

Staircase

```
def get_number_of_ways2(N,steps=[1,2]):

   res = []
   for step in steps:
      if step==N:
         res.append([step])

      if N-step>=0:
         res_temp = get_number_of_ways2(N - step, steps)
         for each in res_temp:
            res.append([step] + each)

   return res
```

Subarray given sum

```
def subarray_given_sum(A,S):

   res = []
   for i,a in enumerate(A):
      if int(a)==S:
         res.append([a])
      elif int(a)<S:
         A_temp = numpy.delete(A,i)
         temp_res = subarray_given_sum(A_temp,S-int(a))
         for each in temp_res:
            res.append([int(a)]+each)

   return res
```

Triplet given sum

```
def triplets_given_S_unique2(A,S):

   res = []
   for i,a in enumerate(A):
      A_temp = numpy.delete(A, i)
      temp_res = hash_01_two_elements_summup.two_sum_unique(A_temp, S-a)
      for each in temp_res:
         if each[0]>=a and each[1]>=a and each[0]>=each[1]:
            res.append((a,each[0],each[1]))

   res = [r for r in set(res)]
   return res
```

Bits manipulation

Bits

1. Maximum Subarray XOR
2. Magic Number
3. Sum of bit differences among all pairs
4. Swap All Odds And Even Bits
5. Find the element that appears once
6. Binary representation of a given number
7. Count total set bits in all numbers from 1 to n
8. Rotate bits of a number
9. Count number of bits to be flipped to convert A to B
10. Find Next Sparse Number

- Find first set bit
- Rightmost different bit
- Check whether K-th bit is set or not
- Toggle bits given range
- Set kth bit
- Power of 2
- Bit Difference
- Rotate Bits
- Swap all odd and even bits
- Count total set bits
- Longest Consecutive 1's
- Sparse Number
- Alone in a couple
- Maximum subset XOR

Math

TODO

- **Recursion vs Dynamic,**
- **greedy algo for partition case**
- binary-indexed-tree-or-fenwick-tree
- count-inversions-array-set-3-using-bit

- Connect Nodes at Same Level + link
- Count BST nodes that lie in a given range
- Implement LRU Cache
- Interleaved Strings
- Check if a Binary Tree contains duplicate subtrees of size 2 or more
- Find largest word in dictionary by deleting some characters of given string
- Modular Exponentiation (Power in Modular Arithmetic)
- Maximum Index
- How to write an efficient method to calculate x raise to the power n?
- A sorted array is rotated at some unknown point, how to efficiently search an element in it.