Date :
Page :

2021

1(a)

Data structure:
Data structures are specialized formats for organizing and storing data so that data can be used efficiently.

Algorithm :
Algorithms are step by step procedure or flow of operation for solving problems or accomplishing tasks.

Together, an algorithm and data structure make up a program. Efficient algorithms ~~often~~ often depend on the choice of appropriate data structure. The choice of data structure can significantly impact the efficiency of an algorithm, and the choice of ~~data structure can significantly~~ algorithm can influence the choice of data structure. So, the combination of an appropriate data structure and algorithm is crucial for

CamScanner

creating efficient and effective computer programs.

1 (b)

```
int power (a, n) {
    if( n==1) return a ;
    else if (n%2==0) return power(a,n/2)*
                            power(a, n/2);

    else return a*power (a, n/2) * power (a, n/2);
}
```

This algorithm has time complexity

$O(\log n)$

because, at each step, the problem size is divided by 2.

Date :
Page :

1 (c)

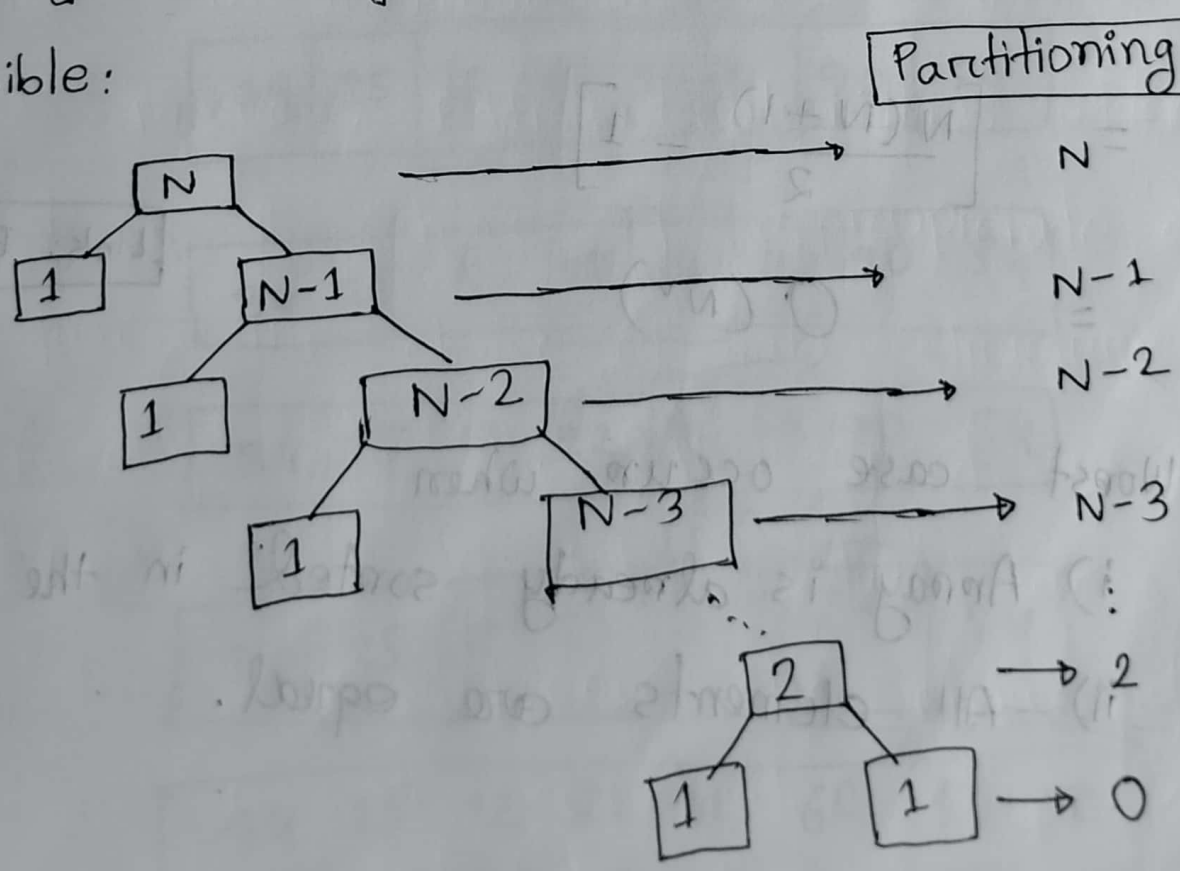
$O(n)$.

1(d)

$O(n^2)$

3(a)

Lets assume that we choose a pivot element in such a way that it gives the most unbalanced partitions possible:



All the numbers in the box denote the size of the arrays or the subarrays.

Let consider an input array size N. The first partition call takes N times to perform the partition step on the input array. Each partition step is invoked recursively from the ~~following~~ previous one.

Therefore, the time complexity of the Quicksort algorithm in worst case is:

$$[N + (N-1) + (N-2) + (N-3) + \cdots 2]$$

$$= \left[\frac{N(N+1)}{2} - 1\right]$$

$$= O(N^2)$$

Link: Baeldung.com

Worst case occur when

i) Array is already sorted in the same order

ii) All elements are equal.

Sorting

Bubble sort :

2019 – 4(a)

Sorting the array using bubble sort algo :

64	34	25	12	22	11	90	54

step 1:

34	64	25	12	22	11	90	54
34	25	64	12	22	11	90	54
34	25	12	64	22	11	90	54
34	25	12	22	64	11	90	54
34	25	12	22	11	64	90	54
34	25	12	22	11	64	90	54
34	25	12	22	11	64	54	90

Date :
Page :

Step: 2 :

25 34 12 22 11 64 54 90

25 12 34 22 11 64 54 90

25 12 22 34 11 64 54 90

25 12 22 11 34 64 54 90

25 12 22 11 34 64 54 90

25 12 22 11 34 54 64 90

Step: 3

12 25 22 11 34 54 64 90

12 22 25 11 34 54 64 90

12 22 11 25 34 54 64 90

12 22 11 25 34 54 64 90

12 22 11 25 34 54 64 90

CamScanner

Step: 4:

12 22 11 25 34 54 64 90

12 11 22 25 34 54 64 90

12 11 22 25 34 54 64 90

12 11 22 25 34 54 64 90

Step: 5:

11 12 22 25 34 54 64 90

11 12 22 25 34 54 64 90

11 12 22 25 34 54 64 90

Step: 6:

11 12 22 25 34 54 64 90

11 12 22 25 34 54 64 90

Step 7:

11 12 22 25 34 54 64 90

2019 | 4 (b)

Insertion Sort : (algorithm) :

```
void insertionSort( int A[ ], int n)
{
    for ( i = 1 ;   i<n   ; i++)
    {
        temp = A [i] ;
        j = i - 1 ;
        while ( j ≥ 0 && A [i] > temp)
        {
            A [j+1] = A [j] ;
            j -- ;
        }
        A [j+1] = temp ;
}
```

Worst case : $O(n^2)$

Best " : $O(n)$

Date :
Page :

Counting sort :

⇒ Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values.

Algorithm :

2021 | 3(b)

Step: 1

	0	1	2	3	4	5	6	7	8
inputArray	2	5	3	0	2	3	0	3	6

max: 6

Step 2:

	0	1	2	3	4	5	6
count	0	0	0	0	0	0	0

Step 3:

	0	1	2	3	4	5	6
count ~~prefixSum~~	2	0	2	3	0	1	1

Step 4:

Calculate prefix sum of the count array.

	0	1	2	3	4	5	6
count	2	2	4	7	7	8	9

Step 5: i=8

	0	1	2	3	4	5	6	7	8
input	2	5	3	0	2	3	0	3	6

6

	0	1	2	3	4	5	6
count	2	2	4	7	7	8	9

9-1=8

	0	1	2	3	4	5	6	7	8
output									6

Step 6 : i = 7

	0	1	2	3	4	5	6	7	8
input	2	5	3	0	2	3	0	3	6

0	1	2	3	4	5	6
2	2	4	7	7	8	8

7-1=6

0	1	2	3	4	5	6	7	8
						3		6

And this process repeat upto when i = 0.
Finally we found the sorted array given as follows:

0	1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5	6

~~Quick Sort~~

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

Merge Sort

```
Algorithm Merge (l, mid, h)
{
    i = l ; j = mid+1; k = 1 ;

    while (i<=mid && j<=h)
    {
        if (A [i] < A [j])
            C [k++] = A[i++];
        else
            C [k++] = B [j++];
    }
    for ( ; i<=mid ; i++)
        C [k++] = A[i];

    for ( ; j<=h ; j++)
        C [k++] = B[j]
```

Date :
Page :

Algorithm MergeSort(l, h)

```
{
    if( l < h)
    {
        mid = (l+h) /2 ;
        MergeSort (l, mid);
        MergeSort (mid+1, h);
        Merge (l, mid, h);
    }
}
```

~~Time Cplx~~

Pros:

1. Large size list
2. Linked list
3. External Sorting
4. Stable → 3 8 5 8 10 7

↓

3 5 7 8 8 10

মেধাবী
সেলিম টেপঘর
Mobile : 01740-808669

Date :
Page :

cons :

1. Extra space (not inplace sort)

2. No small problem

3. Recursive

space : $O(n + \log n)$

n → external array

$\log n$ → stack.

Time complexity :

merge sort is a sorting algorithm that uses the divide and conquer approach. It keeps on dividing the array into two parts until they cannot be divided further and then they are sorted and merged back together into sorted arrays.

CamScanner

Date :
Page :

Here,

$T(k)$ = time taken to sort k elements.

$M(k)$ = time taken to merge k elements.

So, it can be written,

$$T(N) = 2 * T(N/2) + M(N)$$

$$= 2 * T(N/2) + \text{costant} * N$$

These N/2 elements are further divided into two halves, So,

$$T(N) = 2 * [2 * T(N/4) + \text{costant} * N/2] + \text{constant} * N$$

$$= 4 * T(N/4) + 2 * N * \text{constant}$$

$$= 2^k * T(N/2^k) + k * N * \text{constant}$$

It can be divided maximum until there is one element left. So, then $N/2^k = 1$.

$$\Rightarrow k = \log_2 N$$

মেধাবী
সেলিম টেপহর
Mobile : 01740-808660

$$T(N) = N * (T(1) + N * \log_2 N * \text{constant}$$

$$= N + N * \log_2 N$$

Time : $O(N * \log_2 N)$.

Quick sort:

```
Partition (l, h)
{
    pivot = A[l];

    i = l ; j = h ;

    while (i < j)
    {
        do
        {
            i++;
        } while ( A[i] ≤ pivot) ;

        do
        {
            j-- ;
        } while ( A[j] > pivot)

        if (i < j)
            swap ( A[i], A[j]);
```

```
        return j;

}


QuickSort (l, h)
{
    if (l < h)
    {
        j = partition (l, h);
        QuickSort( l, j);
        QuickSort(j+1, h);
    }
}
```

1101

1011

Date :
Page :

2021 / 4 (a)

In Fibonacci series calculation using recursion, the naive approach often results in redundant function calls, which can lead to inefficiency, especially for larger values of n. To reduce the number of function calls and improve efficiency we can use techniques like dynamic programming

- Create an array to store the results of previously computed Fibonacci numbers.
- Before a recursive call, check if the results for the current Fibonacci number has already been computed and stored in the array. If so, return the stored result.

Here c++ implementation:

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

int save[100005];
```

CamScanner

```
int     fibo( int n){
        if(n==0)    return 0;
        if(n==1)    return 1;

        if(save[n] != -1) return save[n];

        return  save[n]= fib(n-1) + fib(n-2);
}

int main(){
    memset (save,-1, sizeof(save));
    cin>>n ;
    fib(n);
}
```

Date :
Page :

2021 / 5(a)

In BIT, finding the parent node serves a crucial purpose in ~~various~~ quering the tree efficiently. When quering for a cumulative value up to a certain index (like prefix sum) we need to traverse up the tree from a given node to its parent and its ancestors to calculate the cumulative value.

To obtain the parent of a node with index i :

1. Convert i to binary
2. Flip the rightmost bit containing 1 to 0.

example: $i = i - (i \& -i)$

$i = 3\ (11) \rightarrow$ parent $= 2\ (10)$

5(b)

Here the update function of BIT :

```
void update (int idx , int val){
    while ( idx <= n){
        BIT [idx] += val;
        idx += (idx & -idx);
    }
}
```

The idea of BIT is based on the fact that all positive integers can be represented as the sum of powers of 2. Every node of the BIT stores the sum of n elements where n is a power of 2.

example: $19 = 16 + 2 + 1$ (power of 2 representation)

The number of set bits in the binary representation of a number n is

$O(\log n)$. Hence, update required $O(\log n)$ traversal.

5 (c)

For, $n = 16$

A			T
5	1	0001	
2	2	0010	
9	3	0011	
-3	4	0100	
5	5	0101	
20	6	0110	
10	7	0111	
-7	8	1000	
2	9	1001	
3	10	1010	
-4	11	1011	
0	12	1100	
-2	13	1101	
15	14	1110	
5	15	1111	
	16	10000	

T
5
7
9
13
5
25
10
41
2
5
-4
1
-2
13
5

For any update at a particular index in the given array, we need to update all the nodes that contain the element of that index in the corresponding binary indexed tree. incrementing the index as follows:

$i = i + (i \,\&\, 2\text{'s complement}(i))$;

6 (a)

(iv) Segment tree / Binary index tree

Date :
Page :

2021/ ~~8(a)~~ 8(b)

There are several ways to implement a priority queue, including using an array, linked list, heap or binary search tree.

Binary heap is generally used to implement priority queue efficiently. Here the comparison with insert (), remove(), peek() operations :

Time complexity	insert	remove	peek
Heap	O(logn)	O(logn)	O(1)
Array	O(1)	O(n)	O(n)
Linked list	O(n)	O(1)	O(1)

CamScanner

8 (c)

The function mnp() implement the Robin-Karp Algorithm.

Robin Karp algorithm matches / find, all the occurances of a pattern from a text. using a hash value calculated from the pattern. Print the index of all occurances.

Date :
Page :

2020

1(a)

1. Understand the problem
2. Decide on: computational means
   exact vs approximate solution
   Algorithm design Techniques
3. Design an algorithm
4. Prove correctness
5. Analyze the algorithm
6. Code the algorithm

CamScanner

Date :
Page :

2020 / 1 (b)

Here, given $A = 100n^2$; $B = 2^n$

Start from $n = 1$;

	$A = 100n^2$	$B = 2^n$
$n = 1$	100	2
$n = 2$	400	4
$n = 4$	1600	16
$n = 8$	6400	256
$n = 16$	25600	65536

Now, $l = 8$ $r = 16$;

using binary search

$m = 12$ $A = 14400$ $B = 4096$

$m = 14$ $A = 19600$ $B = 16384$

$m = 15$ $A = 22500$ $B = 32768$

Ans: $n = 15$

CamScanner

Date :
Page :

1 (c)

To determine the size of each white connected component in the final image after black pixels have been drawn, we can use a graph based approach BFS to traverse the image pixels and identify connected components.

algorithm:

1. Initialize an empty list to store the sizes of white connected components.

2. Traverse the image.

3. BFS traversal: For each white pixel encountered, perform BFS traversal starting from that pixel to explore all connected pixels. While traversing mark visited.

4. Count pixels while traversing and store it into list when

CamScanner

once traversal complete for a connected component.

5. Repeat this process untill all pixels in the image have been visited.

2(a)

The cumulative sum means "how much so far". The definition of the cumulitive sum is the sum of a given sequence that is increasing or getting bigger with more additions

Input	10	15	20	25	30
Output	10	25	45	70	100

Date :
Page :

2 (b)

ways of finding cumulitive sum :

1) Simple iteration

If we have a small array or a single query, simple iteration is efficient. It require $O(n)$ time.

2) Pre fix sum

If we have multiple cumulitive sum queries on different ranges, pre fix sum array is efficient. In prefix sum way we need no updates of the main array.

Time complexity:

query : $O(1)$

Prefix array : $O(n)$

3) Binary Index tree:

If we need to handle frequent updates and query operations simultaneously, BIT

is used.

Time complexity:

Update and Query: $O(\log n)$

Prepare BIT : $O(n \log n)$.

3(a)

Time complexity of merge sort:

$T(k)$ = time taken to sort k elements

$M(k)$ = time taken to merge k elements

It can be written as:

$$T(N) = 2 * T(N/2) + M(N)$$

$$= 2 * T(N/2) + \text{constant} * N$$

These N/2 element further divided into two halves,

$$T(N) = 2 * [2 * T(N/4) + \text{constant} * N/2] + \text{cons} * N$$

$$= 4 * T(N/4) + 2 * N * \text{constant}$$

....

$$= 2^k * T(N/2^k) + k * N * \text{constant}$$

It can be divided maximum until there is one element left. So,

$$N/2^k = 1$$

$$\Rightarrow N = 2^k$$

$$\Rightarrow \log_2 N = k$$

$$\therefore T(N) = N * 1 + \log_2 N * N * \text{constant}$$

$$= N + N * \log_2 N$$

$\therefore$ Time complexity $= O(N \log_2 N)$.

Date :
Page :

3(b)

Counting sort algorithm have $O(N+n)$ time complexity.

Algorithm:

1. Find range of given array (A).
2. Initialize an array according to range with all elements 0. (B)
3. Store count of each element at their respective index in array (B).
4. Store commulative sum of elements in (B)
5. Place elements in (C).
6. Decrease count of element in (B).

CamScanner

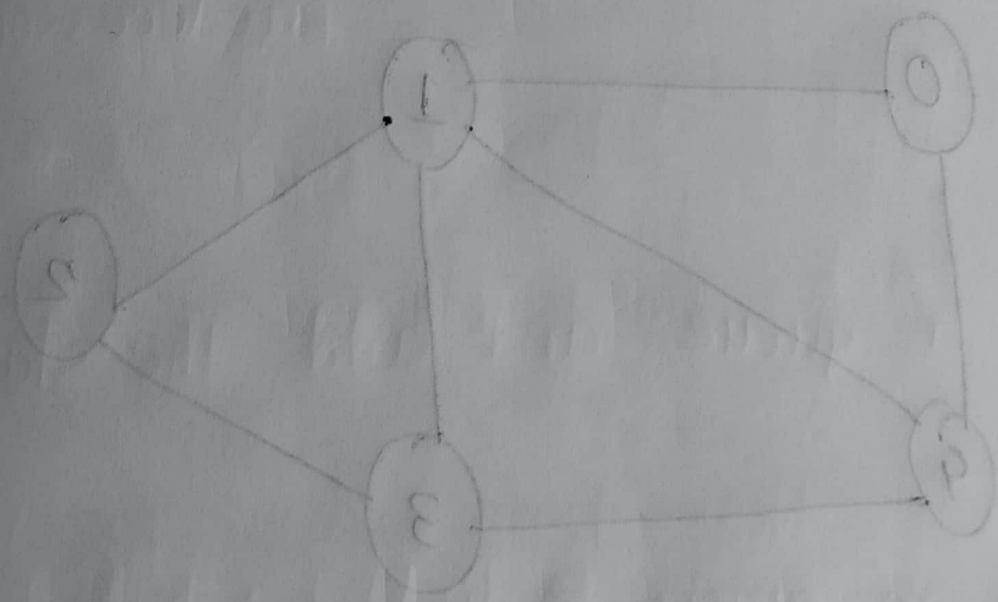
Date
Page

3(c)

- Time complexity of 1st function:

  $O(n)$

- 2nd function: $O(n^2)$

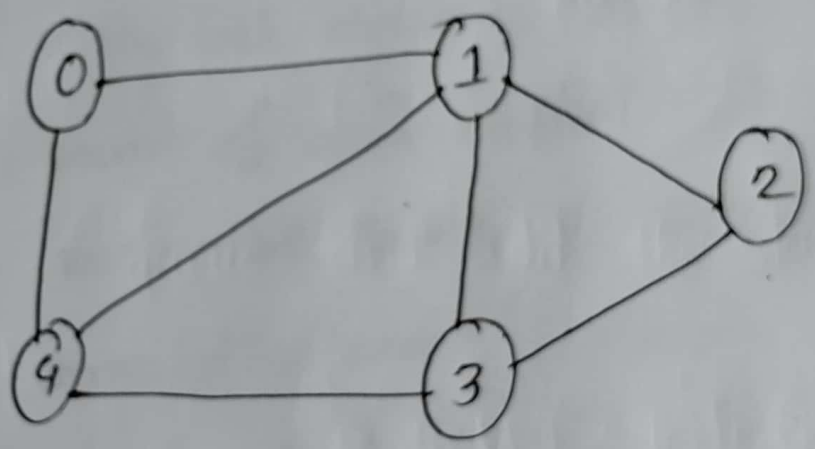


2020 4(a)

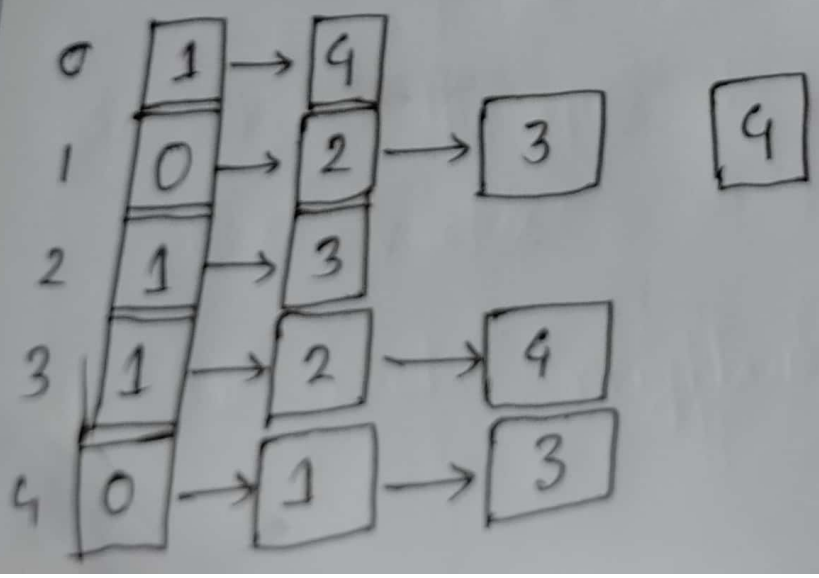
We use vectors in STL to implement graph using adjacency list representation.

The idea is to represent a graph as an array of vectors such that every vector represents the adjacency list of a vertex.

example:



adjacency list representation of the graph:



// function of add ~~edge~~ edge for undirected graph:

```
void addEdge ( vector<int> adj[], int u , int v)
{
        adj [u]. push_back (v);
        adj [v]. push_back (u);
}

void main(){
  // Graph declaration:
   vector <int> adj [5] ;

   addEdge (adj, 0, 1);
   addEdge (adj, 0,4) ;
   addEdge (adj, 1,2) ;
   addEgde (adj, 1,3);
   addEdge (adj, 1,4);
   addEdge (adj, 2,3) ;
   addEdge (adj,3,4) ;

}
```

Date :
Page :

4 (b)

An un-weighted and undirected graph is given, we can use BFS to find the minimum distance between two nodes. Here we use modified version of BFS. We need two array

dist [0, 1 ... V-1] → keep track distance

pred [0, 1, ... V-1] → keep track immediate predecessor.

Algorithm:

1. Create a queue and add the starting vertex to it
2. Create an array dist[]. Initialize with 0. for starting index, others with infinitive.
3. While the que is not empty, dequeue the next vertex.
4. For each neighbor of the dequed vertex

that ~~was~~ has not visited, set its distance to the distance of dequeued vertex plus 1 and add it to the queue.

5. Repeat steps 3-4 until the queue is empty.

6. The distances array now contains the shortest path distances from the starting vertex to all other vertices.

CamScanner

2020 / 5 (a)

step 1: Set status = 1 (ready status) for each node in Q.

Step 2: Push the starting node A on the stack and set its status = 2 (waiting state).

step 3: Repeat steps 4 and 5 until STACK is empty.

step 4: Pop the top node N. Process it and set its status = 3 (Processed state).

step 5: Push on the stack all the neighbors of N that are in the ready state (whose status = 1) and set their status = 2.

CamScanner

[End of Loop]

Step 6 : Exit.

Time complexity : $O(V+E)$

where, V is the number of vertex and

E is the " " Edges.

6 (a)

	0	1	2	3	4	5	6	7	8	9
3	T	F	F	T	F	F	F	F	F	F
4	T	F	F	T	T	F	F	T	F	F
5	T	F	F	T	T	T	F	T	T	T
8	T	F	F	T	T	T	F	T	T	T
12	T	F	F	T	T	T	F	T	T	T
34	T	F	F	T	T	T	F	T	T	T

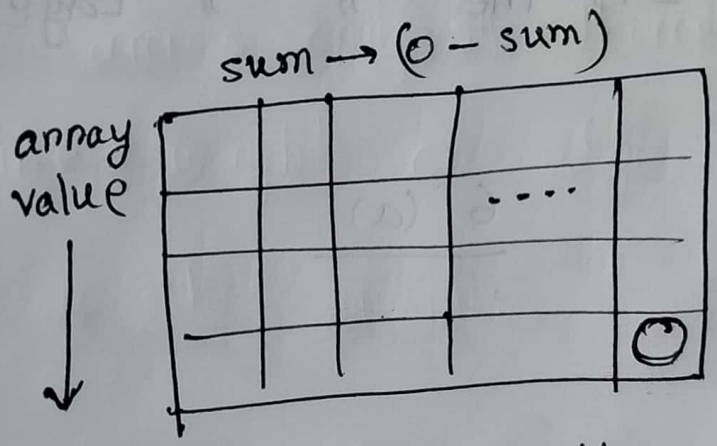
Yes, there exist a subset $(5+4)=9$

6 (b)

Two algorithm:

i) Subset sum using DP

Use 2D - array to store the value of each step.



Start from begining of this 2D table. If we able to making this sum any array value mark this cell. Otherwise backtrack (sum – array[i]value) if we found mark true, then mark it as true. At last cell if it is true sum is possible otherwise no.

ii) meet in the middle :

- Divide the given array into two halves. Calculate all possible sumset sum. Then sort the second part. Then start from the first part and search (sum - first[i]) value in the second portion. If found sum is possible.

6 (e)

Sumset sum DP :

$$O(n * sum)$$

meet in the middle :

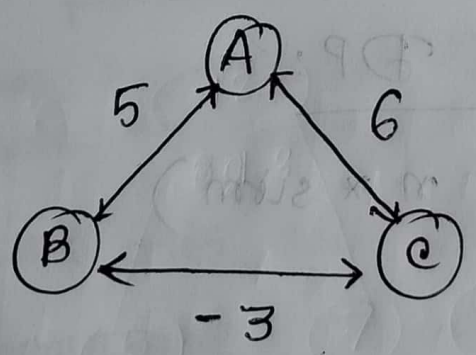
$$O(2^n \log 2^n)$$

Date :
Page :

7(b)

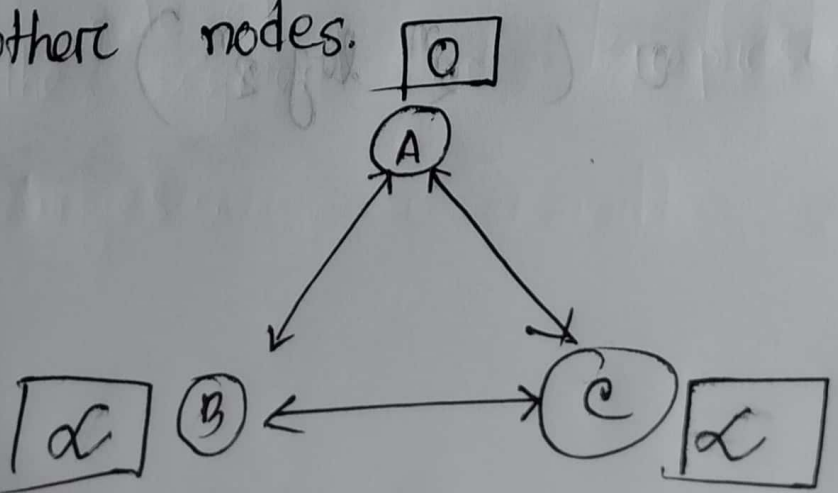
Dijkstra follows a Greedy Approach, once a node is marked as visited it cannot be reconsidered even if there is another path with less cost or distance. This issue aries only if there exists a negative weight or edge in the graph.

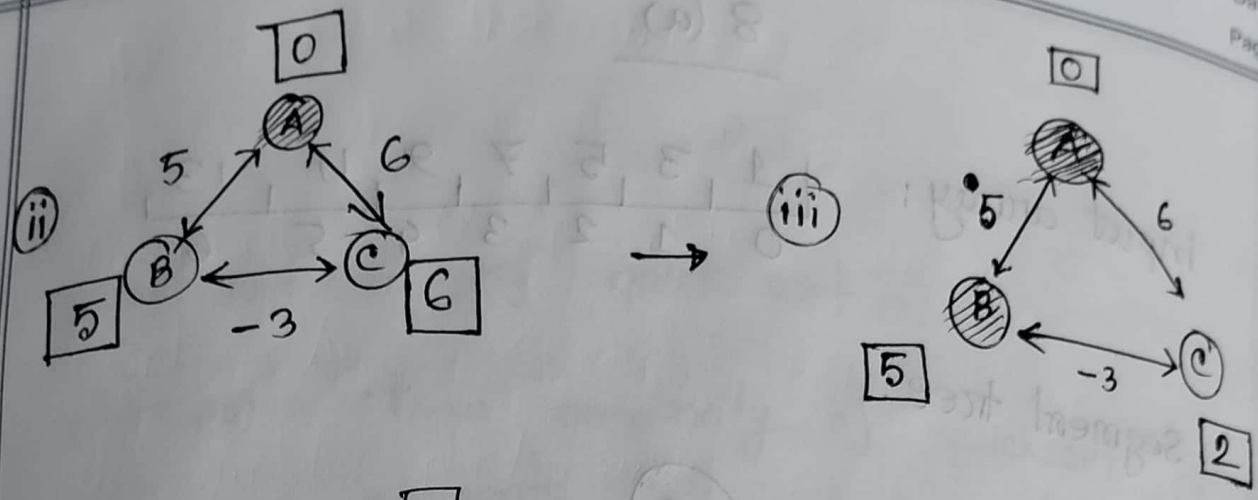
Consider this graph:

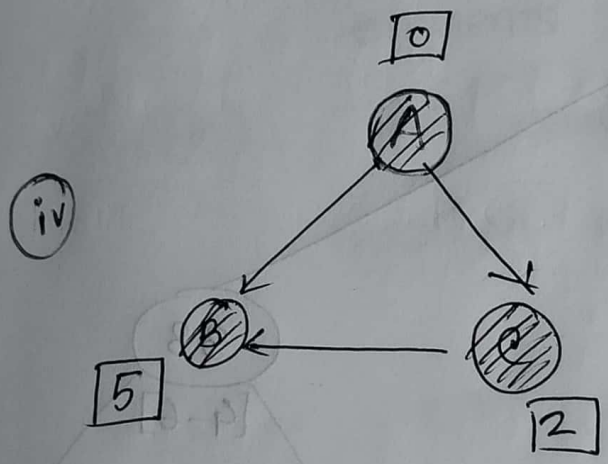


we want to shortest path from A to other nodes.

(i)







We can see distance between A → B is 5. But If we follow A → C → B this path is 3 length long, so negative weight will affect our goal is to determined the shortest path.

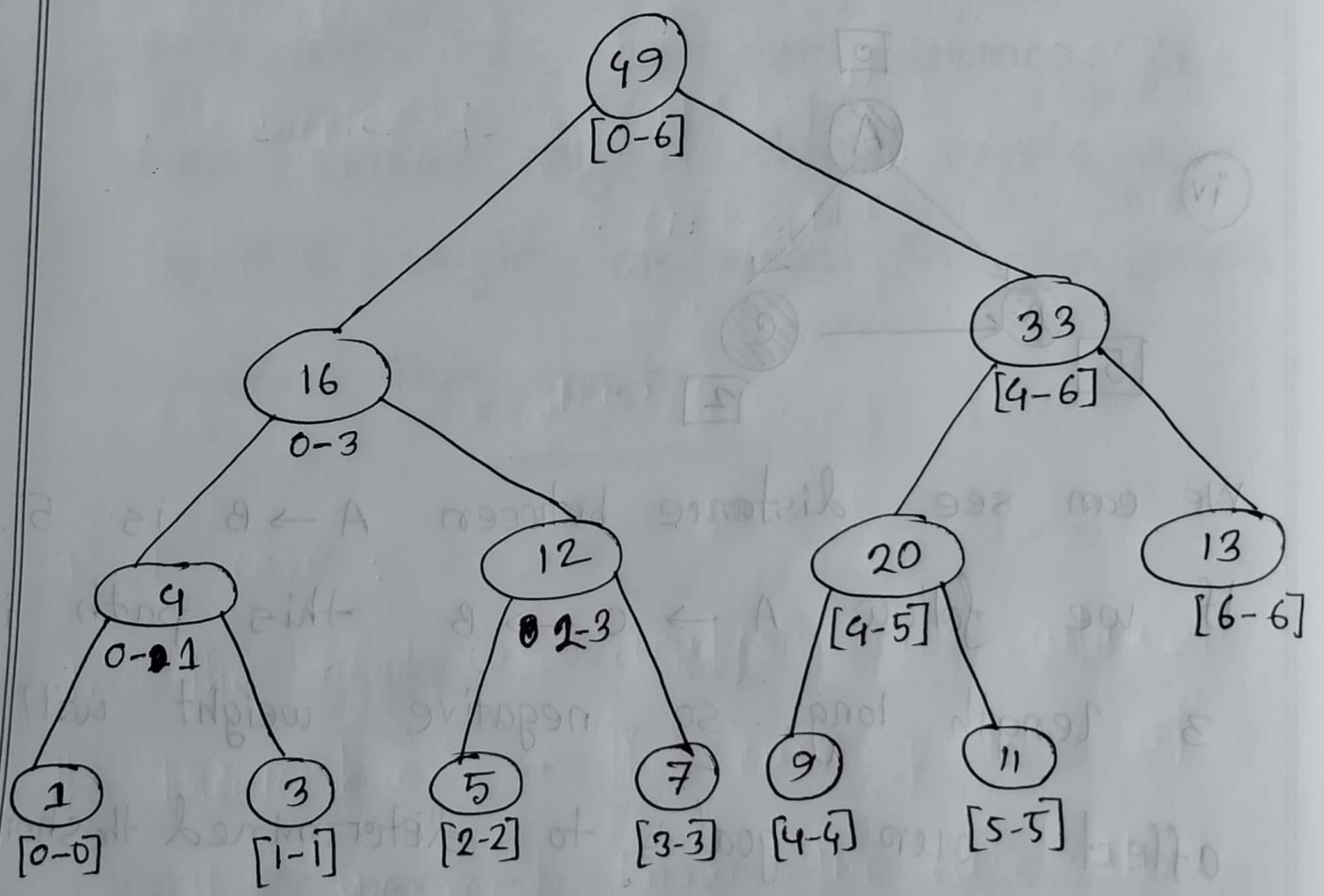
Date :
Page :

8 (a)

Input array:

1	3	5	7	9	11	13
0	1	2	3	4	5	6

Segment tree:



Here, the node of segment tree represents sum of ranges.

2019

1(b)

Time complexity of quick sort is:

$T(K)$: Time complexity of quicksort of K elements

$P(K)$: Time complexity ~~of~~ for finding the position of pivot among K elements.

$$\therefore T(N) = 2 * T(N/2) + N$$

Now, $T(N/2)$ is also $2 * T(N/4) + N/2$, so,

$$T(N) = 2 * [2 * T(N/4) + N/2] + N$$

$$= 4 * T(N/4) + 2 * N$$

So, we can say that,

$$T(N) = 2^K * T(N/2^K) + K * N$$

Here, $\frac{N}{2^K} = 1$

$\Rightarrow 2^k = N$

$\therefore k = \log_2 N$

$\therefore T(N) = N * T(1) + \log_2 N * N$

$\therefore$ Complexity $= O(N * \log_2 N)$

$= O(N \log N)$

1(c)

Best case :

In the best-case analysis, we calculated the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.

Date :
Page :

Worst case :

In the worst case analysis, we calculated the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. Most of the time, we do worst-case analyses to analyze algorithm.

Average case :

In average case analysis, we take all possible inputs and calculated the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. Average case analysis is not easy to do in most practical cases and it is rarely done.

CamScanner

Date :
Page :

1 (d)

Space complexity refers to the total amount of memory space used by an algorithm/program including the space of input value for execution. Calculate the space occupied by variables in an algorithm/program to determine space complexity.

Space complexity = Auxillury space + space use by input values

Here, auxillury space means simply extra or temporary space. Note that,

2(a)

KMP / Rabin-Karp Algorithm use করতে হবে।

CamScanner

Date :
Page :

. lps

	0	1	2	3 ← index
P	A	A	B	A
	0	1	0	1

At first, create lps array.
count = 0.

step 1:

i

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A

j

	0	1	2	3
lps	A	A	B	A
	0	1	0	1

count = 0

T[i] == P[j]

i++ ; j++

step2:

i

0	1	2	3	4	5
A	A	B	A	A	C

j

0	1	2	3
A	A	B	A
0	1	0	1

count = 0

T[i] == P[j]

i++ ; j++

step3:

i

0	1	2	3	4	5
A	A	B	A	A	C

j

A	A	B	A
0	1	0	1

cont = 0

T[i] == P[j]

i++ ; j++

CamScanner

Date :
Page :

step4:

	0	1	2	3	4	
	A	A	B	A	A	...

	0	1	2	3
	A	A	B	A
	0	1	0	1

T[i] == P[j] ; j == Pattern_size

count = 1 i++ ; j = 0

step 5:

	0	1	2	3	4	5	6
	A	A	B	A	A		C

	0	1	2	3
	A	A	B	A
	0	1	0	1

i++ ; j++

count = 1

Step6:

	0	1	2	3	4	5	6
	A	A	B	A	A	A	C

	0	1	2	3
	A	A	B	A

i++ ; j++

count = 1;

Date :
Page :

step 6:

0	1	2	3	4	5 (i)	
A	A	B	A	A	C	...

0	1 (j)	2	3
A	A	B	A
0	1	0	1

$T[i] \neq P[j]$

$j = 0$

step 7:

4	5 (i)	6	
... A	C	A	...

0	1 (j)	2	3
A	A	B	A
0	1	0	1

$T[i] \neq P[j]$

$i++;$

step 8:

4	5	6 (i)	7	
... A	C	A	A	...

0	1 (j)	2	3
A	A	B	A
0	1	0	1

$T[i] = P[j]$

$i++; \quad j++$

and so on. i increase, j increase and match if found, At last we found one more pattern.

$\therefore$ count = 2.

3(a)

Greedy:

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. Greedy algorithm works in a top-down approach.

Problems that exhibit the following properties are suitable for greedy algorithms:

1. Greedy choice property:

If an optimal solution to the problem can be found by choosing the best choice at each step without

Date :
Page :

reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure:

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

CamScanner

Date :
Page :

4 (b)

Insertion sort algorithm :

1. Start with the second element of the array.

2. Compare the current element with the one before it.

3. If the current element is smaller, swap it with the element before it.

4. Continue this process until either the current element is greater than or equal to the element before it, or until it reaches the begining of the array.

5. Move to the next element and repeat steps 2-4 until the entire array

CamScanner

is sorted.

4 (c)

In the concept of algorithm design, optimization is considered in time and space mostly. Here, some general steps for optimizing algorithms is as follows:

1. Understand the problem:
The first step to optimize an algorithm is to understand the problem, you are trying to solve. What are the expected use cases and edge cases? By clarifying this aspects, you can narrow down the scope of the problem and avoid unnecessary complexity or over-engineering.

2. Choose the right data structure:
Data structure are the building blocks of algorithm and affect how store, access, and manipulate

data. In a specific problem right data structure also affects time and space of an algorithm.

3. Analyze the complexity:
It is usually expressed in terms of big O notation, which captures the worst-case scenario of your algorithm. By analyzing the complexity of your algorithm, you can identify the bottlenecks and sources of inefficiency, and compared different solutions.

4. Apply optimization techniques:
Reduce naive approach and replace with better one. Such as divide an conquer, dynamic programming, backtracking etc.

5. Test and debug:
It is essential for ensuring the correctness and reliability of your program.

CamScanner