Data Structure

- organising data into memory for efficient processing along with operations which can be performed on that data (eg. insert, search, delete)

to acheive :

1) Efficiency
   - can be measured in two terms
     i) time - required to required
     ii) space - required inside memory

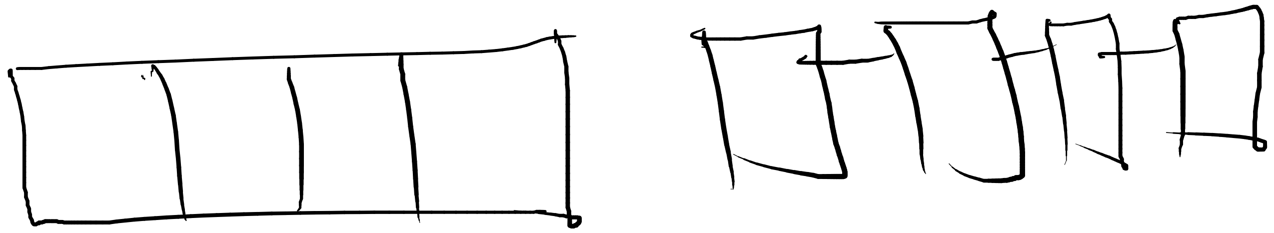
2) Abstraction
   - Abstract Data types

3) Reusability
   - reused to solve some algorithm
   - reused to implement another data structure

Types of Data structrure

Linear Data structure

- data is orgnised one after another



- data is accessed sequentially or linearly

- Basic data structures
  1) Array
  2) structure / class
  3) stack
  4) Queue
  5) Linked list

(Hash Table)

Non Linear Data structure

- data is orgnised in multiple levels (heirarchy)

- data is accessed non linearly / non sequentially

- Advanced data structures
  1) Tree — (Heap)
  2) Graph

Algorithm

program - set of instructions to machine (CPU)

Algorithm - set of instructions to human (developer/programmer)

- step by step solution of given problem statement

e.g. find sum of array elements

1) create sum & initialise to 0
2) traverse array from 0 to N-1 index
3) add every element into sum
4) return / print sum variable

- programming language independent
- like blue prints / templates
- algorithm - template
- program - implementation

Algorithm analysis / Efficiency measurement / Complexities

- finding time and space requirement of an algorithm
    1. Time - time required to execute the algorithm (ns, us, ms, s)
    2. Space - space required to excute the algorithm inside memory (bytes, kb, mb,..

1. Exact analysis

- finding exact space and time of the algorithm
- it depends on some exeternal factors
- time is dependent on type of machine(cpu), no of processes running at that time
- space is dependent on type of machine(architecture), data types

2. Approximate analysis

- finding approximate time and space of the algorithm
- mathematical approach is used to find time and space complexity of the algorithm and it is known as "Asymptotic analysis"
- it also tells about behavior of the algorithm when input is changed or sequence of input is changed
- behaviour of algorithm can be observed into three cases
    1. Best case
    2. Average case
    3. Worst case

to denote time and space complexity we use Big-O notation

Time Complexity

- count the number of iterations for the loop which is used inside the algorithm
- timp required is directly proportional to the iterations of the loop

1. print 1D array on console

```
void print1DArray(int arr[], int N){
        for(int i = 0 ; i < N ; i++)
                sysout(arr[i]);
}
```

loop iterations = n

Time $\propto$ iterations

Time $\propto$ n

Time complexity = $T(n) = O(n)$

2. print 2D array on console

```
void print1DArray(int arr[], int N){
        for(int i = 0 ; i < N ; i++)
                for(int j = 0 ; j < N, j++)
                        sysout(arr[i][j]);
}
```

iterations (outer loop) = n

iterations (inner loop) = n

Total iterations = $n * n = n^2$

Time $\propto n^2$

Time complexity = $T(n) = O(n^2)$

3. add two numbers

```
int addTwoNumbers(int n1, int n2)
{
        return n1 + n2;
}
```

- time required is constant because it will not varry according to the values of variable.
- constant time requirement can be denoted as

$$\boxed{T(n) = O(1)}$$

4. print table of given number

```
void printTable(int num){
        for(int i = 1 ; i <= 10 ; i++)
                sysout(num * i);
}
```

- loop is going to iterate constant number of times always
- constant time requirement

$$\boxed{T(n) = O(1)}$$

5. print binary of decimal number

```
void printBinary(int num){
    while(num > 0){
        sysout(num % 2);
        num = num / 2
    }
}
```

num	num>0	rem
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$$\begin{array}{r|l} 2 & 9 \\ \hline & 4 \\ \hline & 2 \\ \hline & 1 \end{array}$$

1
0
0
1

$(9)_{10} = (1001)_2$

$n = 9, 4, 2, 1, 0$

$n = n, n/2, n/4, n/8 \ldots$

$n = n/2^0, n/2^1, n/2^2, \ldots\ldots n/2^{itr}$

$\therefore$ last time body of loop will be executed for $n = 1$

$n/2^{itr} = 1$

$2^{itr} = n$

$\log 2^{itr} = \log n$

$itr \log 2 = \log n$

$itr = \dfrac{\log n}{\log 2}$

$Time \propto itr$

$Time \propto \dfrac{\log n}{\log 2}$

$Time \propto \log n$

$$\boxed{T(n) = O(\log n)}$$

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), ... O(2^n), ...

modification : '+' or '-' : time complexity will be in terms of n
modification: '*' or '/' : time complexity will be in terms of log n

for(int i=0; i<n; i++) → O(n)
for(int i=n; i>0; i--) → O(n)
for(int i=1; i<=10; i++) → O(1)
for(int i=0; i<n; i+=20) → O(n)
for(int i=n; i>0; i/=2) → O(log n) $i = \overbrace{9,4,2,1}^{4},0$
for(int i=1; i<n; i*=2) → O(log n) $i = \underbrace{1,2,4,8}_{4},16$

for(int i=0; i<n; i++)
  for(int j=0; j<n; j++) } → O(n²)

for(int i=0; i<n; i++); – n
+ = 2n Time ∝ 2n T(n) = O(n)
for(int j=0; j<n; j++); – n

for(int i=0; i<n; i++) → n
  for(int j=n; j>0; j/=2) → log n
Time ∝ n log n
T(n) = O(n log n)

Space Complexity

- finding approximate space required to execute an algorithm

Total space = input space + Auxillary space

input space (space of actual input (data))

Auxillary space (space requires to process actual input)

find sum of array elements

```
int findSum(int arr[], int size){
    int sum = 0 ;
    for(int i = 0 ; i < size ; i++)
        sum+=arr[i];
    return sum;
}
```

Input variable = arr
Processing variable = i, sum, size
Input space = n units
Auxillary space = 3 units
Total space = n+3
space $\propto$ n+3

space complexity = $S(n) = O(n)$

Auxillary space complexity
Processing variable = i, sum, size
Auxillary space = 3 units

$\boxed{AS(n) = O(1)}$

Searching Algorithms

- finding some key(data to be searched) into collection(set) of values
  1. linear search (data is random)
  2. binary search (data is sorted)

1. Linear Search

//1. decide key to be searched
//2. traverse array from 0 to N-1 index
//3. compare key with every element of array
//4. if key is found, stop and return index of the index
//5. if key is not found, then return -1

2. Binary Search

//1. decide key top be searched
//2. divide array into two parts
//3. compare key with middle element of the array
//4. if key is matching with middle element,
    //stop and return index of middle element
//5. if key is less than middle element, then search key into left side
//6. if key is greater than middle element, then search key into right side
//7. repeat step 2 to 6 till key is not found or array is valid

Searching Algorithms Analysis

- for searching and sorting algorithms, we count number of comparisions
- time is directly proportinal to number of comparision

Linear Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(n)
Worst case :	key is found in last few comparitions	:	O(n)
	key is not found		

Binary Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(log n)
Worst case :	key is found in last few comparitions	:	O(log n)
	key is not found		

Recursion

- **function calling itself**
- **we can use recursion if**
    1. **we know the process/formula in term of itself**
    2. **we know the terminating condition**

$n! = n * (n-1)!$

$1! = 1$

```
int fact(int n)
{
    if(n==1)
        return 1;
    return n * fact(n-1);
}
```

1

```
int fact(5)
{
    if(5==1)
        X
    return 5 * fact(4)
}
```

5 * 24

2

```
int fact(4)
{
    if(4==1)
        X
    return 4 * fact(3)
}
```

4 * 6

3

```
int fact(3)
{
    if(3==1)
        X
    return 3 * fact(2)
}
```

3 * 2

4

```
int fact(2)
{
    if(2==1)
        X
    return 2 * fact(1)
}
```

2 * 1

5

```
int fact(1)
{
    if(1==1)
}
```

1 2 3 4 5

Algorithm Implementation Approches

Any algorithm can be implemented using two approches

1. Iterative approach
- loops are used

```
int fact(int n)
{
    int fact=1;
    for(i=1; i<=n; i++)
        fact *= i;
    return fact;
}
```

Time ∝ no. of iteration

$T(n) = O(n)$

2. Recursive approach
- recursion is used

```
int fact(int n)
{
    if(n==1)
        return 1;
    return n * fact(n-1);
}
```

Time ∝ no. of recursive calls

$T(n) = O(n)$

Sorting Algorithms

- arrangement of data in either ascending or descending order of their values

- Basic sorting algorithms
    1. Selection sort
    2. Bubble sort
    3. Insertion sort

- Advanced sorting algorithms
    4. Merge sort
    5. Quick sort
    6. Heap sort

Selection sort

//1. select one position of the array (start from index 0)
//2. compare selected position element with all other elements
//3. if selected position element is greater than other element
    // then swap both
//4. repeat above steps untill array is sorted - (N-1 times)

Bubble sort

//1. compare all pairs of consecutive elements of the array
//2. if left element is greater than right element
    // then swap both
//3. repeat above two steps untill array is sorted -- (N-1 times)

$j < N-1$

i	j start	j end
1	0	4
2	0	4
3	0	4
4	0	4
5	0	4

$j < N-i$

i	j start	j end
1	0	4
2	0	3
3	0	2
4	0	1
5	0	0

Sorting Algorithms Analysis

Array length = n

n = 6

pass1 → 5 (n-1)

pass2 → 4 (n-2)

pass3 → 3 (n-3)

pass4 → 2 ⋮

pass5 → 1 1

Total comps = 1+2+3+4+5 = 15

$= 1+2+3+ \ldots + (n-2) + (n-1)$ (n)

$= \frac{n(n-1)}{2}$

comps $= \frac{n^2 - n}{2}$

Time $\propto \frac{n^2-n}{2}$

Time $\propto n^2 - n$

$$\boxed{T(n) = O(n^2)}$$

n	n^2
1	1
10	100
100	10000
1000	1000000

- mathematical polynomial
- degree of polynomial
  ↳ highest power
- while writing time complexity consider only degree term because it is highest growing term.

Basic Sorting Algorithms Analysis and Comparisions

		Worst case	Avg case	Best case
selection sort	–	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	–	$O(n^2)$	$O(n^2)$	$O(n)$
insertion sort	–	$O(n^2)$	$O(n^2)$	$O(n)$