Lec 1

Topics
→ Time complexity
→ STL : Vectors, Pair, Set, Map

Time complexity

- time taken to execute an algorithm according to the size of input.

- why is it required?
  → Our logic/algo shouldn't take months/years to be executed. It should be done in some time limit.

- we would express it as a function of input, f(n).

- A normal computer has it processing speed in GHz i.e. approx 10^9 instructions/sec.

- for calculating time we would calculate no. of instructions i.e. f(n) would reperesent no of instructions as function of n.

  f(n) = no. of instruction

Ex: ★

```
int count = 0                        // 1        , n is input variable
for ( int i = 0 ; i < n ; ++i )      // 2*(n+1) + 1
    { count++; }                     // n
```

$f(n) = 3n + 2 + 1 = 3n + 3$

[You needn't worry about exact calculation rather you should worry about predominant term i.e. here it's 3*n

even forget coefficient of predominant term n is the main thing]

```
* int count = 0                          // 1
  for (int i=0; i<n; ++i)                // 2*(n+1) + 1
  {  for(int j=0; j<n; ++j)              // n*(2*(n+1)+1)
       {  count += j; }                  // n*n
  }
```

$f(n) = 3n^2 + 4n + 3$ ($3n^2$: predominant term)

* We talk about time complexity keeping in mind n is very large like of order of $10^5, 10^9$ etc. Hence for $f(n) = 3n^2 + 4n + 3$, $4n+3$ is insignificant as compared to $3*n^2$ Moreover 3 is also insignificant if n is very large. This is called Asymptotic complexity.

Big-O notation

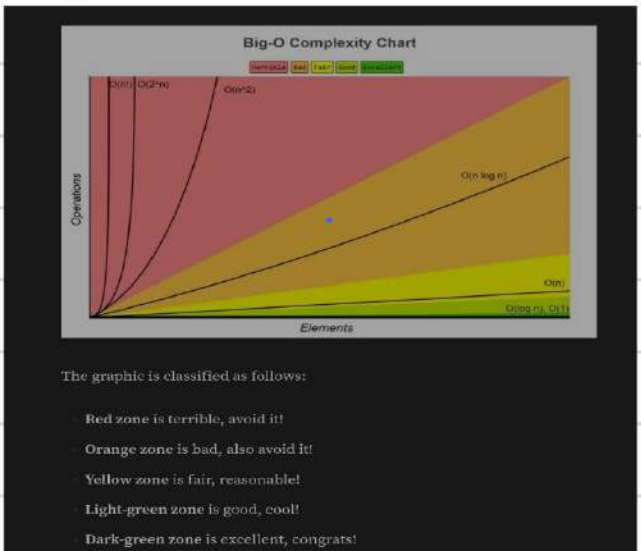
- used to classify algo based on growth rate of operations as input size grows
  * It's actually an upper bound

So, for $f(n) = 3n + 3$ we say it runs in $O(n)$ time complexity [$O(n)$ read as order of n]

Similary $f(n) = 3n^2 + 4n + 3$ we say it $O(n^2)$

$f(n) = 7$ we say it $O(1)$

- Most common classes
  - $O(1)$ → constant time
  - $O(\log n)$ → logarithmic
  - $O(n)$ → Linear
  - $O(n^2)$ → Quadratic
  - $O(2^n)$ → exponential
  - $O(n!)$ → factorial



Def^n: Given two increasing fns $f(n)$ & $g(n)$ we say $f(n)$ is $O(g(n))$ "read as order of $g(n)$" if there exists constants c and $n_0 > 0$ s.t.

$$f(n) < c \cdot g(n) \qquad \forall\, n \geq n_0$$

E.g:- $f(n) = 3n+3$ $\quad g(n) = n$ $\quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 3$

$3n+3 < c \cdot n$ ✓ $\quad \forall\, n \geq 1$

$n_0 = 1$ $\quad 3n+3 < 7 \cdot n$ ✓

$c = 7 \quad n_0 = 1.$

$f(n) = O(g(n)) = O(n)$

E.g:- $f(n) = 3n+3$ $\quad g(n) = n^2$

$3n+3 < c \cdot n^2 \qquad \forall\, n \geq n_0$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$n_0 = 1$ $\quad 6 < c \cdot 1$

$3n+3 < 7n^2 \qquad \forall\, n \geq 1$

$f(n) = O(g(n)) = O(n^2)$

E.g:- $f(n) = 4n^2$ $\quad g(n) = n$

$\exists$ c & n_0 s.t.

$4n^2 < c \cdot n \qquad \forall\, n \geq n_0$

$4n < c \qquad \forall n \geq n_0$

(contradiction).

$4n^2 = f(n) \neq O(g(n)) = O(n)$

Consider we are given an array of length N of integers and we want to find out if there exist a subset with sum = k?

Idea: Check all subsets of array one by one if its sum is k.

There are 2^n subsets and in finding sum of each subset it will take max O(n) time. So this would run in $O(n.2^n)$ time complexity. $n = 10$ 2^{10}

Consider n can be max 10^9. So we are intersted in finding worst time complexity. So no. of instructions would be of order $10^9 . 2^{10^9}$. Consider each processing speed 10^9/s

Time required $\approx 10^9 . 2^{10^9} \times 10^{-9} = 2^{10^9}$ s

$\approx 2^{1000,000,000}$ s

Age of universe $\approx 2^{60}$ s

STL (Standard template Library of c++)

Containers: place to hold data alongwith some special functionality.

- Use correct header in order to use its container like #include <vector>. Also write "using namespace std;"

★ **Vector** : It's just an array with added functionalities

Declaration:

```
vector <int> v;
vector <int> v (10);      // vector with 10 elements
vector <int> v (10, -1);  // vector with 10 elements each = -1.
X vector <int> v [10];    // array of 10 vector<int>'s
vector <char> v(10, -1);
vector < vector<int>> s;
```

Not the same

→ push_back(); adds element at the end increasing size by 1.

```
vector <int> v;
for (int i=0; i<n; ++i)
    v.push_back (i);   // gives you vector having elements 0, 1, 2 ..., n-1
```

v. pop_back();

→ size(); gives you current size of vector / no. of elements.

v.size(); ← it's unsigned integer, when you want to use it atleast typecast into (int)

→ empty(); gives true if vector is empty

v.empty()

→ clear(); remove all elements of vector

v.clear()

→ resize(); it makes it contain the require no. of elements.

v.resize(25)

Some intialisations:

```
vector <int> v2 = v1; // v1 is a vector
vector <int> v3 (v1);
```

→ erase();

→ insert();

sort(v.begin(), v.end()) → O(nlogn)

sort(arr, arr+n);

★ **Pairs** : keeping two elements together.

The first element of the pair is reffered to as first & second element of the pair is reffered as second.

p.first → p.second →

Syntax:- pair < data type 1, data type 2 > p;

eg:- pair < ll, string > p = { 10, "hello" }

pairs can used along with vectors & other STL based datastructures.

eg:- vector < pair < ll, ll > > v;

v.push_back ({10, 20});

Note:- while sorting a vector of pairs, the vector is sorted according to the first element.

eg:- v = [{5,2}, {1,6}, {4,12}, {3,8}]

vector<pair<ll,ll>> v(4)

sort(v.begin(), v.end());

gives:- [{1,6}, {3,8}, {4,12}, {5,2}]

Sets

Two types of sets in STL — set - based on balanced binary tree - O(log n)

unordered_set - based on hashing - O(1).

set — maintains elements in sorted order, stores unique elements

unordered set - stores unique elements, but does not maintain element in sorted order.

Multiset - stores elements in sorted order, but contain multiple instances of an element.

MAP

map<ll, ll> m;

generalized array that consists of key - value - pairs.

m[10] = 1

MAP — unordered_map - based on hashing - O(1)

map - based on balanced binary tree - O(log n)

MAP stores key value pairs.

eg:- map < string, ll > m;

m["hello"] = 4

map < ll, ll > m;

m[2] = 4.

map < data type, key, data type of value > m;

Binary search

prefix[i] = v[0] + v[1] + ... v[i]

prefix[i-1] = v[0] + v[1] + ... v[i-2]

prefix[i-1] + v[i] = prefix[i]

prefix[0] = v[0];

for(i=1; i<n; i++)

prefix[i] = prefix[i-1] + v[i];

prefix[b] - prefix[a-1]

Prefix sum