15-09-22

* Time and Space Complexity:

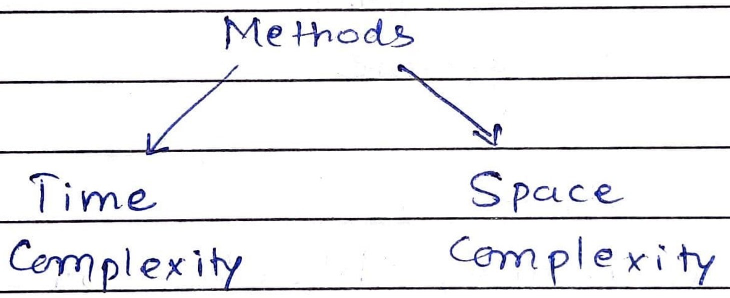
to compare algorithms

→ The method/~~algorithm~~ must be:

① Independent of the machine and its configuration, on which the algorithm is running on.

② Shows a direct correlation with the number of inputs.

③ Can distinguish two algorithms clearly without ambiguity.



⇒ Time Complexity: (express/measures)

→ The time complexity quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

→ Time to run an algorithm

↓

function of the length of input

$\boxed{T = f(n)}$

T ⇒ Time taken
n ⇒ Input
f → function

→ Ex: Find whether a pair (X, Y) exists in an array, A of N elements whose sum is Z.

→ Solution: Check every pair possible & compare with Z.

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

bool findPair(int a[], int n, int z) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <n; ++j) {
            if (i != j && a[i] + a[j] == z)
                return true;
        }
    }
    return false;
}

int main() {
    int a[] = { 1, -2, 1, 0, 5 };
    int ~~00~~, z = 0;
    int n = sizeof(a)/ sizeof(a[0]);
    cout<< findPair(a, n, z);
    return 0;
}
```

Output:
```
false
```

Number of lines of code depends on 'Z' (in this case)

→ During analysis of algorithm, mostly the worst-case scenario is considered.

→ In the worst case, (in this example)

① N*c operations are required for input

② The outer loop runs N times

③ For each i, the inner loop run N times

∴ Total execution time = $\cancel{N^{*}c} + N^{*}N^{*}c + \cancel{c}$

$= N^{*}N$

Big-O Notation for worst case $= O(N^2)$

↓

Time complexity

→ Order of growth is how the time of execution depends on the length of input.

→ Ex: Calculate time complexity of below algorithm.

```
count = 0;
for (int i = N; i > 0; i /= 2)
     for (int j = 0; j < i; ++j)
          count++;
```

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

Page No.

Date $\frac{2N}{2^n}$ $1 - \frac{2^n}{2^n}$

1st loop: $N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots = N \sum_{i=0}^{n} \frac{1}{2^n}$

$= N/2^n - 1$

2nd loop: $N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots$

$a = N, \ r = \frac{1}{2}$

$S = \frac{a(1-r^n)}{1-r} = \frac{N(1-(1/2)^n)}{1-1/2} = \frac{2N(2^n - 1)}{2^n}$

$= 2^{1-n} \cdot N(2^n - 1)$

$= 2N - 2^{1-n} \cdot N$

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

$=$

In ~~fut~~ future

⇒ Space Complexity:

→ The space complexity quantifies the amount of space taken by an algorithm to run as a function of the length of the input.

→ Ex: Fequvency of array elements

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

```
void countFreq(int arr[], int n) {
    unordered_map <int, int> freq;
    for (int i = 0; i < n; ++i)
        ~~frequency~~
        freq[arr[i]]++;
    for (auto x : freq)
        cout << x.first << " " << x.second << endl;
}

int main() {
    int arr[] = {10, 20, 20, 10, 10, 20, 5, 20};
    int n = sizeof(arr)/sizeof(arr[0]);
    countFreq(arr, n);
    return 0;
}
```

Output:

```
5   1
10  3
20  4
```

→ Two arrays of length N & variable i

∴ Total space used

$= N^*c + N^*c + 1^*c$

$= 2N^*c + \boxed{c}$ ← unit space taken

$= O(N)$ {Space Complexity}

Auxiliary Space

Space Complexity	Auxiliary Space
quantifies	quantifies
↓	↓
total space used by algo.	extra space used in algo. (apart from given input)

For above example,

Auxiliary space = space used by freq[]

↓

as freq[] is not part of input

∴ Total auxiliary space

$= N*c + c$

$= O(N)$ {Auxiliary space}