TUTORIAL

Complexity of Programs

Chapter

In Computer Science, we develop programs to solve different problems. Before writing programs, we go for an informal description of the idea to solve a problem called algorithms. So, an algorithm is a step by step procedure to solve a problem. Now there may be multiple algorithms to solve the same problem due to different techniques used to solve the problem. Now the main question is which algorithm needs to be implemented actually. As all algorithms may not take the same amount of resources (time and space) to solve the problem it is better to analyze them and find out the best algorithm. So that once we implement a solution on computer it must be the best solution. The algorithms are analyzed for time and space taken by them to solve a particular instance of the problem. Then this time and space values are generalized for the problem set.

If we have two algorithms with different complexities, obviously we are behind the faster one. So let us take a scenario to differentiate between two algorithms with different complexities: -

We have two algorithms each with O(n) and O(Log[base 2]n) complexities respectively to solve a problem. Following table will show the rough estimates of times for large inputs to these algorithms with the assumption that one comparison is done in 1ns on some hypothetical computer: -

Input Size	Algorithm 1 O(n)	Algorithm 2 O(Log_2n)
$10^1 = 10$	10^1 ns = 10 ns	4 ns
$10^2 = 100$	10^2 ns = 100 ns	7 ns
$10^3 = 1000$	10^3 ns = 1 micro sec	10 ns
$10^6 = 1000000$	10^6 ns = 1 millisecond	20 ns
$10^9 = 1000000000$	10^9 ns = 1 sec	30 ns
$10^{12} = 1000000000000$	10^{12} ns = 16.66 minutes	40 ns
$10^{15} = 1000000000000000$	10^{15} ns = 277 Hours = 11.57 days	50 ns
$10^{18} = 1000000000000000000$	10^{18} ns = 1653 weeks = 380 Months = 31.70 Years	60 ns

From above table it can be seen that when the input tends to large values, the Algorithm2 still takes time in nanoseconds to solve it, while the Algorithm1 is taking several years to solve the same problem. So, fast algorithms are always demanded in market. Now a day's only solving the problem is not a good thing, we must ensure that we solve the problem efficiently.

So, we will analyze the algorithms for two complexities: -

Time complexity

It denotes the time taken by an algorithm to solve a problem. Time taken by an algorithm will depend on many factors in computers like single or multi processor machine, read/write speed of memory, 32-bit architecture or 64-bit architecture & which input is given to algorithm etc. So, basically we need a close approximation of time which is hardware independent. In computer science, we follow asymptotic analysis for this purpose, which provides an approximation of complexity in terms of different inputs. So it is widely accepted measure of complexity of algorithms.

The following program will run the four statements and always take same amount of time i.e. constant amount of time: -

C

```c
#include<stdio.h>
int main()
{
  int a=10, b=15;
  int c = a + b;
  printf("%d",c);
}

```

Java

```java
import java.util.Scanner;
// Other imports go here
// Do NOT change the class name
class Main{
  public static void main(String[] args)
  {
    int a=10, b=15;
```

```
    int c = a + b;
    System.out.println(c);
  }
}
```

```C++
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int main(){
  int a=10, b=15;
  int c = a + b;
  cout<<c<<endl;
}

```

Space complexity

It denotes the amount of memory taken by an algorithm to solve a problem. It can also be measured by asymptotic analysis in terms of memory space required.

For example: Find the greatest common divisor (GCD) of two integers, m and n. The algorithm for GCD may be defined as follows:

1. While m is greater than zero:
2. If n is greater than m, swap m and n.
3. Subtract n from m.
4. n is the GCD

The space-complexity of the above algorithm is a constant. It just requires space for three integers m, n and t. So, the space complexity is O(1).

In both the cases, we will use asymptotic analysis to measure the complexity of algorithms and based on this analysis we can figure out the better algorithm among all possible ones.

Asymptotic analysis

In this, we will analyze the algorithm complexity, with respect to the input provided to the algorithm. In asymptotic analysis, we always talk about large input sizes. It might be possible that those large inputs are never given to your software, but we analyze the algorithms for these large inputs for generality.

We analyze the algorithms for three cases of inputs: -

a. **Best Case**: In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location.

b. **Average Case**: In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

c. **Worst Case**: In the worst case analysis, we calculate upper bound on running time of an algorithm. It is the case, which causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search() functions compares it with all the elements of array one by one.

For example, in following array if we want to find the element 9, then it found at the first position, so it becomes a best case for us,

9, 7, 5, 4, 8, 2, 54, 23, 45, 1

Similarly, if we search for 1, then we needs to compare with all the elements way to the last element, it becomes the worst case for us. In general, best case does not provide us much information, as guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run. Average case analysis is rarely performed as it is not easy to calculate it. Worst case analysis is mainly conducted in all cases, as guaranteeing the maximum time taken by the algorithm is useful information.

In Asymptotic analysis, Big-O notation is used for Worst case complexity of an algorithm; The Big-Omega notation is used for Best Case complexity of an algorithm & the Big-Theta notation is used for Average case complexity of an algorithm. Following table shows the various asymptotic notations in order of their time taken: -

Time Taken	Asymptotic Notation	Name
Small	O(1)	Constant Time
	O(Log Log n)	Double Logarithmic
	O(Log n)	Logarithmic Time
	O(sqrt(n))	Root
	O(n)	Linear Time
	O(n Log n) = O(Log n!)	Linearithmic Time / Log-linear Time
	O(n^2)	Quadratic Time
	O(n^3)	Cubic Time
	O(n^k) k is some constant	Polynomial Time
	O(2^n)	Exponential Time
	O(n!)	Factorial Time
Large	O(n^n)	N to the N

Analyzing the algorithms: We analyze the programs with asymptotic notations. Our programs, contains following kind of constructs in general: -

a. Simple statements like assignment and other ALU operations.

b. Selective statements: like if-else and switch case statements.

c. Definition and Calling statements: like variable declaration, function calling and returning values.

d. Loops: like for, while, do-while loops etc.

In the first three cases, the statements will take constant amount of time in respect of the inputs, as input does not affect the time taken by these statements. But input mostly affects the running of a loop in program. One more aspect in programming other than iterative programs is recursive programs. The depth of recursion will also depend on the input provided. So basically we have to analyze the iteration or recursions of a program to analyze its time complexity.

Iterative Programs

In case of loops, we can analyze the running time of loops on different inputs provided. For example, the following loop will run always 10 times irrespective of the value of input: -

C

```c
#include<stdio.h>
int main()
{
  int i;
  for(i=0; i<10; i++)
    printf("\nCodeQuotient - Get better at Programming.");
}

```

Java

```java
import java.util.Scanner;
// Other imports go here
// Do NOT change the class name
class Main{
  public static void main(String[] args)
  {
    int i;
    for(i=0; i<10; i++)
      System.out.print("\nCodeQuotient - Get better at Programming.");
  }
}
```

C++

```cpp
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int main(){
        for(int i=0; i<10; i++)
        cout<<"\nCodeQuotient - Get better at Programming.";
   return 0;
}
```

The loop is running a constant amount of time i.e. 10 times, each time you run the program. So this type of program complexity is called constant time complexity denoted by O(1).

Following program will run depending on input: -

```
#include<stdio.h>
int main()
{
                int i,number;
                scanf("%d", &number);
                for(i=0; i<number; i++)
                                printf("\nCodeQuotient - Get better at
Programming.");
}
```

Now the loop is running equal to the number provided as input. So now the time taken by the program is proportional to the input provided by the user. This complexity is denoted by O(n) complexity also known as Linear time complexity.

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have O(n^2) time complexity:

```
#include<stdio.h>
int main()
{
                int i, j, number;
                scanf("%d", &number);
                for(i=0; i<number; i++)
                                for(j=0; j<number; j++)
                                                printf("\nCodeQuotient - Get
better at Programming.");
}
```

This program will print the statement quadratic to the input provided i.e. if input is 3 then it will print 3*3=9 times. This type of complexity is denoted by O(n^2) also known as quadratic time complexity. If we increase the number of nested loops the quadratic term will also increase as shown below: -

```
#include<stdio.h>
int main()
{
                int i, j, k, number;
                scanf("%d", &number);
                for(i=0; i<number; i++)
                                for(j=0; j<number; j++)
        for(k=0; k<number; k++)

printf("\nCodeQuotient - Get better at Programming.");
}
```

This program will print the statement quadratic to the input provided i.e. if input is 3 then it will print 3*3*3=27 times. This type of complexity is denoted by O(n^3) also

known as quadratic time complexity. In general quadratic complexity is denoted by O(n^k) where k is a constant.

Similarly we can calculate the time complexity of different programs for example,

```
#include<stdio.h>
int main()
{
                int i, j, k, number;
                scanf("%d", &number);
                for(i=1; (i^2)<number; i++)         // (i^2)<number is same as
sqrt(i)<= n
                                printf("\nCodeQuotient - Get better at
Programming.");
}
```

We can put some values for input and then generalize it. Following table shows the values of input and number of times loop will execute: -

Input by the user	Times the loop condition evaluates to TRUE
3	1 as 1^2 = 1 but 2^2=4 which is > 3
4	2 as 2^2 = 4
9	3 as 3^2 = 9
16	4 as 4^2 = 16
25	5 as 5^2 =25
100	10 as 10^2=100

This program will print the message equal to square root of number times. So the time complexity of this program is O(sqrt(n)).

Nested loops will always not increment the complexity in quadratic manner, if they execute only constant amount of time, then the complexity is also constant for them, for example, the following program has linear complexity, even it has two nested loops: -

```
#include<stdio.h>
int main()
{
                int i, j, number;
                scanf("%d", &number);
                for(i=0; i<number; i++)
                                for(j=0; j<10; j++)
                                                printf("\nCodeQuotient - Get
better at Programming.");
}
```

We can put some values for input and then generalize it. Following table shows the values of input and number of times loop will execute: -

Input by the user	Times the loop condition evaluates to TRUE
3	3*10 = 30
4	4*10 = 40
5	5*10 = 50
6	6*10 = 60
10	10*10 = 100
100	100*10 = 1000
1000	1000*10 = 10000

This program will print the message equal to the input multiplied by 10 always. So the time complexity of this program is O(n) not O(n^2).

Sometimes even the single loop will cause the quadratic time complexity as shown in below program: -

```
#include<stdio.h>
int main()
{
                int i, number;
                scanf("%d", &number);
                for(i=0; i < (number^2); i++)
                            printf("\nCodeQuotient - Get better at Programming.");
}
```

The following program has logarithmic complexity:

```
#include<stdio.h>
int main()
{
                int i, number;
                scanf("%d", &number);
                for(i=1; i<number; i=i*2)
                                printf("Welcome to programming.");
}
```

Above program will increment the loop counter by multiplication factor, after each iteration loop counter is doubled, hence the loop will run only log of input times. It is denoted by O(Log[Base2]n) times. Similarly, the below program runs O(Log[Base3]n) times as the loop counter is multiplied by 3.

```
#include<stdio.h>
int main()
{
                int i, number;
                scanf("%d", &number);
                for(i=1; i<number; i=i*3)
                                printf("\nCodeQuotient - Get better at
Programming.");
}
```

So the complexity of iterative programs will depend on the loop iterations. If a program has more than one segment of code than we have to calculate the

complexity of each segment and then the complexity of whole program is largest of these complexities as Big-O notation always finds the upper bound of complexity. Consider the following program: -

```
#include<stdio.h>
int main()
{
                int i, j, number;
                scanf("%d", &number);
                for(i=0; i<number; i++)
                            printf("\nCodeQuotient - Get better at Programming.");
                for(i=0; i<number; i++)
                                for(j=0; j<number; j++)
                                                printf("\nCodeQuotient - Get
better at Programming.");
}
```

First loop will run O(n) times, but the second set of loops will run O(n2) times, so the whole program has a complexity of O(n^2). So similarly we can calculate the complexity of programs.

Complexity of Recursive programs

In case of recursive program, we have to write the recursive equation of the program and then solve the recurrence to find the complexity. Most of famous problems are solved with recursion like binary search, merge sort etc. We can solve recurrences with methods like back substitution, recursion tree or masters theorem.

In, Recursive programs we divide the problem in some smaller instance and then call the same solution recursively till we hit the base condition which is trivially solved. For example, to find the factorial of 3, we will call factorial of 2 and then multiply the result returned with 3 to return the answer. Similarly, to find the factorial of 2 we call factorial of 1 and multiply the result returned with 2 to return the answer. Factorial of 1 is solved trivially as it returns the answer as 1. so all these answers stack in the recursion and finally solves the problem. Following is the implementation of factorial problem recursively: -

```c
#include <stdio.h>
int fact_recursive(int num)
{
  int result;
  if (num == 0)
    return 1;     // fact() return 1 if argument is 0
  else
  {
    result = num * fact_recursive(num-1);      // Call recursively
with lesser number.
    return result;
  }
}
int main()
{
```

```
  int number, fact1;
  number = 5;    // Number whose factorial required
  fact1 = fact_recursive(number);        // Call the Recursive version
  printf("Number=%d\n", number);
  printf("Recursive_Factorial=%d\n", fact1);
}

```

Java

```
import java.util.Scanner;
// Other imports go here
// Do NOT change the class name
class Main{
  static int fact_recursive(int num)
  {
    int result;
    if (num == 0)
      return 1;     // fact() return 1 if argument is 0
    else
    {
      result = num * fact_recursive(num-1);      // Call recursively with lesser number.
      return result;
    }
  }
  public static void main(String[] args)
  {
    int number, fact1;
    number = 5;    // Number whose factorial required
    fact1 = fact_recursive(number);        // Call the Recursive version
    System.out.println("Number=" + number);
    System.out.println("Recursive_Factorial=" + fact1);
  }
}
```

C++

```
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int fact_recursive(int num)
{
  int result;
  if (num == 0)
    return 1;     // fact() return 1 if argument is 0
  else
  {
    result = num * fact_recursive(num-1);      // Call recursively with lesser number.
    return result;
  }
}
int main() {
```

```
17    int number, fact1;
18    number = 5;     // Number whose factorial required
19    fact1 = fact_recursive(number);        // Call the Recursive version
20    cout<<"Number="<<number<<endl;
21    cout<<"Recursive_Factorial="<<fact1<<endl;
22  }
```

In the *fact_recursive()* function, the function is called recursively with smaller numbers to reach at 1. so we can say the problem is reduced 1 number at a time and at each call we compare with base case. if true then recursion ends, otherwise new recursive call will be made. So in recurrence terms it can be written for complexity as below: -

```
T(n) = T(n-1) + 1
```

Where T(n) is the time to solve the problem for input n, we make a comparison so 1 is added in each call and we call the same function with (n-1) as argument.

Following are the recurrence Relations for Binary Search and Merge Sort Algorithms: -

```
Binary Search : T(n) = T(n/2) + 1 = O(Log n)
Merge Sort : T(n) = 2.T(n/2) + n = O(n Log n)
```

The next table represents comparison of typical running time of algorithm of different order

Input Size	**Logarithmic : Log_2N**	**Linear: N**	**Quadratic: N^2**	**N * Log_2N**	**Cubic: N^3**	**Exponential: 2^N**
5	3	5	25	15	125	32
10	4	10	100	40	10^3	10^3
100	7	100	10^4	700	10^6	10^{33}
1000	10	1000	10^6	10^4	10^9	10^{300}

The following chart shows different complexities with respect to their rate of growth: -