Analysis and Design of Algorithms

Analysis of Algorithms

Table of Contents

Analysis of Algorithms

- **Analysis of Algorithms** is the determination of the amount of **time**, **storage** and/or other **resources** necessary to execute them.
- Analyzing algorithms is called **Asymptotic Analysis**
- **Asymptotic Analysis** evaluate the performance of an algorithm

Time complexity

Time complexity

- **time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm

- We can have three cases to analyze an algorithm:
  1) Worst Case
  2) Average Case
  3) Best Case

Asymptotic Notations

1) **Big O Notation: is** an Asymptotic Notation for the worst case.

2) **Ω Notation** (omega notation): is an Asymptotic Notation for the best case.

3) **Θ Notation** (theta notation) : **is** an Asymptotic Notation for the worst case and the best case.

Big O Notation

Big O Notation

1) O(1)

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function. For example swap() function has O(1) time complexity.

```
def swap(s1, s2):
    return s2, s1
```

Big O Notation

2) O(n)

- Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example the following loop statements have O(n) time complexity.

Big O Notation

2) O(n)

- Another Example:

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

Big O Notation

3) $O(n^c)$

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

Big O Notation

❑ Another Example

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
        // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <=n; j += c) {
        // some O(1) expressions
}
```

Big O Notation

4) O(Logn)

- Another Example

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

Big O Notation

5) O(LogLogn)

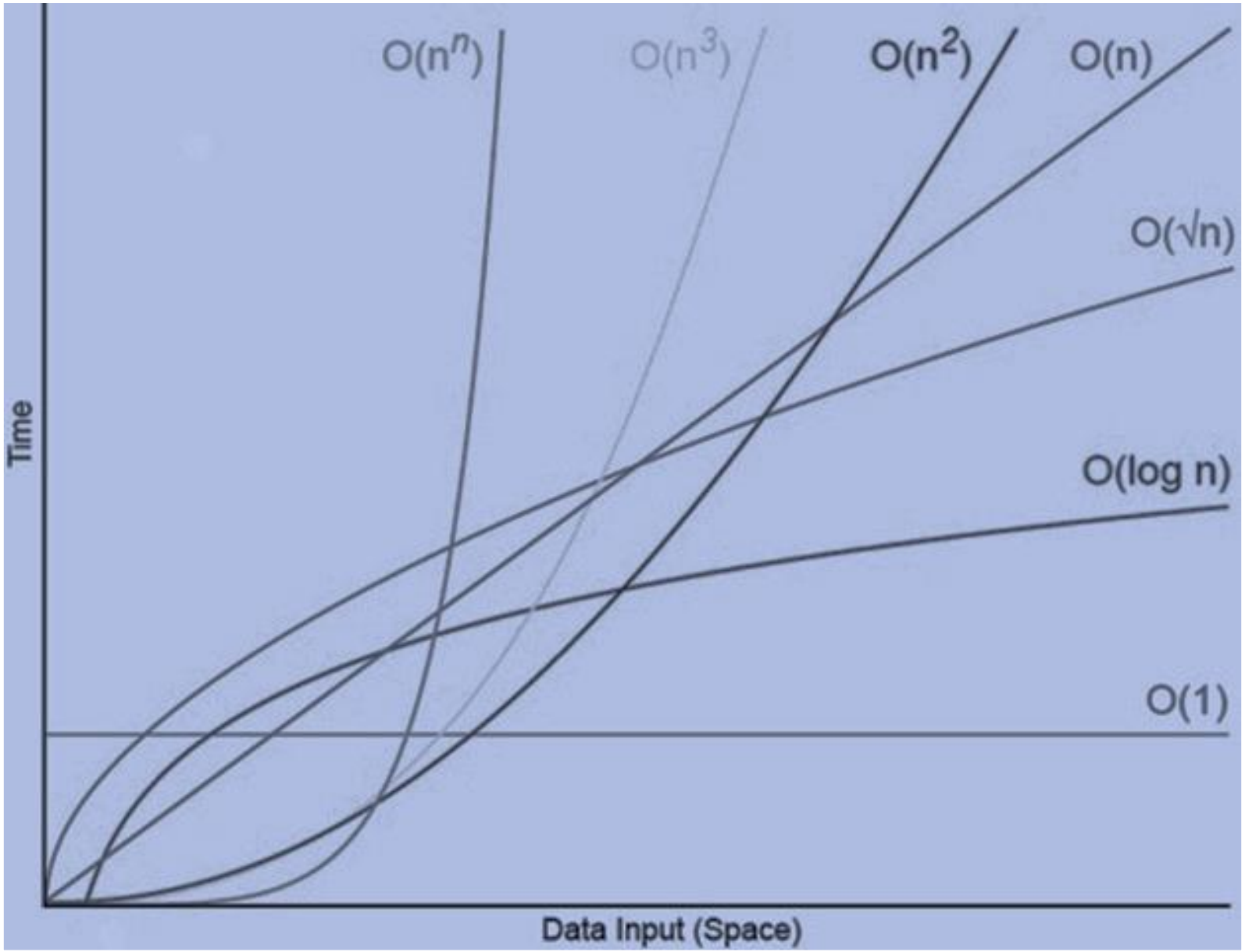
- Another Example

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

Growth Orders

n	O(1)	O(log(n))	O(n)	O(nlog(n))	$O(N^2)$	$O(2^n)$	O(n!)
1	1	0	1	1	1	2	1
8	1	3	8	24	64	256	$40xx10^3$
30	1	5	30	150	900	$10x10^9$	$210x10^{32}$
500	1	9	500	4500	$25x10^4$	$3x10^{150}$	$1x10^{1134}$
1000	1	10	1000	$10x10^3$	$1x10^6$	$1x10^{301}$	$4x10^{2567}$
$16x10^3$	1	14	$16x10^3$	$224x10^3$	$256x10^6$	-	-
$1x10^5$	1	17	$1x10^5$	$17x10^5$	$10x10^9$	-	-

Growth Orders



Growth Orders

Length of Input (N)	Worst Accepted Algorithm
≤10	$O(N!), O(N^6)$
≤15	$O(2^N * N^2)$
≤20	$O(2^N * N)$
≤100	$O(N^4)$
≤400	$O(N^3)$
≤2K	$O(N^2 * \log N)$
≤10K	$O(N^2)$
≤1M	$O(N * \log N)$
≤100M	$O(N), O(\log N), O(1)$

Problems

- Find the complexity of the below program:

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Problems

- **Solution:** Time Complexity O(n). Even though the inner loop is bounded by n, but due to break statement it is executing only once.

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        // Inner loop executes only one
        // time due to break statement.
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Problems

- Find the complexity of the below program:

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

Problems

- **Solution:**

Time

$O(n^2 \log n)$

```
void function(int n)
{
    int count = 0;

    // outer loop executes n/2 times
    for (int i=n/2; i<=n; i++)

        // middle loop executes  n/2 times
        for (int j=1; j+n/2<=n; j = j++)

            // inner loop executes logn times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Problems

- Find the complexity of the below program:

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

Problems

- **Solution:**

  Time

  $O(n \log^2 n)$

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)

        // Executes O(Log n) times
        for (int j=1; j<=n; j = 2 * j)

            // Executes O(Log n) times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Problems

- Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=0; i<n; i++)
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

Problems

- Solution:

  Time $O(n^5)$

```
void function(int n)
{
    int count = 0;

    // executes n times
    for (int i=0; i<n; i++)

        // executes O(n*n) times.
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                // executes j times = O(n*n) times
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

More examples

- Every time we run the program we need to estimate how long a program will run since we are going to have different input values so the running time will vary.
- The worst case running time represents the maximum running time possible for all input values.
- We call the worst case timing big Oh written O(n).The n represents the worst case execution time units.

Complexities

- Simple programming statement
- Example:

```
k++;
```

- Complexity : O(1)
- Simple programming statements are considered 1 time unit.

Complexities

- Linear ***for*** loops
- Example:

```
k = 0;
for(i = 0; i < n; i++)
   k++;
```

- Complexity : O(n)
- ***for*** loops are considered n time units because they will repeat a programming statement n times.
- The term linear means the for loop increments or decrements by 1

Complexities

- Non Linear loops
- Example:

```
k=0;
for(i=n; i>0; i=i/2)
    k++;
```

- Complexity : O(log n)
- For every iteration of the loop counter i will divide by 2.
- If i starts is at 16 then then successive i's would be 16, 8, 4, 2, 1. The final value of k would be 4. Non linear loops are logarithmic.
- The timing here is definitely $\log_2 n$ because $2^4 = 16$. Can also works for multiplication.

Complexities

- Nested ***for*** loops
- Example:

```
k=0;
for(i=0; i<n; i++)
   for(j=0; j<n; j++)
     k++;
```

- Complexity : $O(n^2)$
- Nested for loops are considered n^2 time units because they represent a loop executing inside another loop.

Complexities

- Sequential ***for*** loops
- Example:

```
k=0;
for(i=0; i<n; i++)
   k++;
k=0;
for(j=0; j<n; j++)
   k++;
```

- Complexity : O(n)
- Sequential for loops are not related and loop independently of each other.

Complexities

- Loops with non-linear inner loops
- Example:

```
k=0;
for(i=0;i<n;i++)
   for(j=i; j>0; j=j/2)
      k++;
```

- Complexity : O(n log n)

- The outer loop is O(n) since it increments linear.
- The inner loop is O(n log n) and is non-linear because decrements by dividing by 2.
- The final worst case timing is: O(n) * O(log n) = O(n log n)

Complexities

- Inner loop incrementer initialized to outer loop incrementer
- Example:

```
k=0;
for(i=0;i<n;i++)
   for(j=i;j<n;j++)
      k++;
```

- Complexity : $O(n^2)$
- In this situation we calculate the worst case timing using both loops.
- For every i loop and for start of the inner loop j will be n-1 , n-2, n-3.

Complexities

- Power Loops
- Example:

```
k=0;
for(i=1;i<=n;i=i*2)
   for(j=1;j<=i;j++)
      k++;
```

- Complexity : $O(2^n)$
- To calculate worst case timing we need to combine the results of both loops.
- For every iteration of the loop counter i will multiply by 2.
- The values for j will be 1, 2, 4, 8, 16 and k will be the sum of these numbers 31 which is $2^n - 1$.

Complexities

- If-else constructs
- Example:

```
/* O(n) */
if (x == 5)
{
     k=0;
     for(i=0; i<n; i++)
       k++;
 }
/* O(n2) */
else
{
    k=0;
    for(i=0;i<n;i++)
        for(j=i; j>0; j=j/2)
            k++;
}
```

- Complexity : $O(n^2)$

Complexities

- Recursive
- Example:

```
int f(int n)
{
      if(n == 0)
            return 0;
       else
            return f(n-1) + n
 }
```

- Complexity : O(n)

THANKS FOR YOUR TIME