ANALYSIS OF ALGORITHMS

- Asymptotic Analysis
- Worst, Average and Best Cases
- Analysis of loops

ASYMPTOTIC ANALYSIS

- Problem description:
- **Given two algorithms for a task, how do we find out which one is better?**
  - 1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
  - 2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.
- Solution:
  - With **asympthotic analysis** we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the **input size**.
  - The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require time taken by implemented algorithms to be compared.
  - The machine dependent constants can always be ignored after certain values of input size.

WORST, AVERAGE, BEST CASES

Asymptotic analysis we analyze an algorithm in :

- **Worst Case** **(Usually Done)**
  - We calculate upper bound on running time of an algorithm.
  - We must know the case that causes maximum number of operations to be executed.

- **Average Case** (Sometimes done)
  - 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.
  - We must know (or predict) distribution of cases.
- **Best Case** (Bogus)
  - We calculate lower bound on running time of an algorithm.
  - We must know the case that causes minimum number of operations to be executed.

Most of the times we calculate worst case scenario.

EXAMPLE: WORST, AVERAGE, BEST CASES

```
// linearly search x in arr[]
int searchLinear(int arr[], int n, int x)
{
    for (int i = 0; i < n; ++i) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Worst Case : The worst case happens when the element to be searched (x in the above code) is not present in the array. The worst case complexity is **O(n).**

Average Case: Lets assume that all cases are uniformly distributed (including the case of x not being present in array). We sum all the cases and divide the sum by (n+1). The average case complexity is **O(n).**

Best Case: The best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be **O(1)**

ANALYSIS OF LOOPS

- O(1)

```
for (int i = 0; i < 3; ++i) {
    // some O(1) expressions
}
```

loop runs a constant number of times

- O(n)

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

loop variables is **incremented / decremented** by a **constant amount.**

- O(n^c)

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

count the number of times the innermost statement is executed. (On2)

- O(logn)

```
for (int i = 0; i < n; ++i) {
    n = n/2;
    // some O(1) expressions
}
```

loop variables is divided/multiplied by a constant amount.

- O(nlogn)

```
for(int i = 0; i < n; ++i) {
    for(int j = i; j <n; j /= 2)
    {
        // some O(1) operation
    }
}
```

For *n* times, a loop variables is divided/multiplied by a constant amount.

- O(loglogN)

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

the loop variables is exponentially increased/reduced (sqrt/pow) by a constant amount.

LOOPS ANALYSIS EXAMPLES

Example 1

```
x = n
while ( x > 0 ) {
    x = x - 1
}
```

O(n)

Example 2

```
x = n
while ( x > 0 ) {
    x = x / 2
}
```

O(logn)

Example 3

```
x = n
while ( x > 0 ) {
   y = n
   while ( y > 0 ) {
       y = y - 1
   }
   x = x - 1
}
```

O(n2)

Example 4

```
x = n
while ( x > 0 ) {
   y = x
   while ( y > 0 ) {
       y = y - 1
   }
   x = x - 1
}
```

O(n2)

Example 5

```
x = n
while ( x > 0 ) {
   y = n
   while ( y > 0 ) {
       y = y / 2
   }
   x = x - 1
}
```

O(nlogn)

LOOPS ANALYSIS EXAMPLES

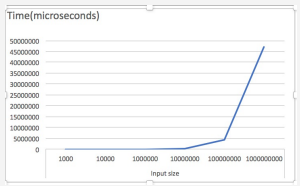
Example 6

```
1  x = n
2  while ( x > 0 ) {
3     y = x
4     while ( y > 0 ) {
5        y = y / 2
6     }
7     x = x - 1
8  }
```

O(nlogn)

Example 7

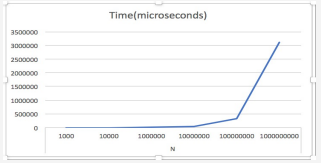
```
1  x = n
2  while ( x > 0 ) {
3     y = n
4     while ( y > 0 ) {
5        y = y - 1
6     }
7     x = x / 2
8  }
```



O(nlogn)

Example 8

```
1  x = n
2  while ( x > 0 ) {
3     y = x
4     while ( y > 0 ) {
5        y = y - 1
6     }
7     x = x / 2
8  }
```



O(nlogn)/ O(n)?

Example 9

```
1  x = n
2  while ( x > 0 ) {
3     y = n
4     while ( y > 0 ) {
5        y = y / 2
6     }
7     x = x / 2
```

O(log2n)

Example 10

```
1  x = n
2  while ( x > 0 ) {
3     y = x
4     while ( y > 0 ) {
5        y = y / 2
6     }
7     x = x / 2
8  }
```

O(log2n)

When x is n, the inner loop executes n times

When x is n/2, the inner loop executes n/2 times

When x is n/4, the inner loop executes n/4 times

so on till x is 1, the inner loop then executes just 1 time.

So the total number of executions of the inner loop is the n + n/2 + n/4 + ... + 1

HOW TO COMBINE TIME COMPLEXITIES OF CONSECUTIVE LOOPS?

- When there are consecutive loops, we calculate time complexity as **sum** of time complexities of individual loops.
- If there are nested loops, we **multiply** the time complexities of individual loops.
- Example:

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

//Time complexity of above code is O(m) + O(n) which is O(m+n)
//If m == n, the time complexity becomes O(2n) which is O(n).
```

HOW TO CALCULATE TIME COMPLEXITY WHEN THERE ARE MANY IF, ELSE STATEMENTS INSIDE LOOPS?

- Using asymptotic analysis to analyze an algorithm, we can distinguish between the following cases :
  1) Worst Case **<- The most useful**
  2) Average Case
  3) Best Case
- Therefore we need to **consider worst case.**
- We evaluate the situation when values in **if-else conditions cause maximum number of statements to be executed.**
- When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

HOW TO CALCULATE TIME COMPLEXITY OF RECURSIVE FUNCTIONS?

- Time complexity of a recursive function can be written as a mathematical recurrence relation.
- To calculate time complexity, we must know how to solve recurrences.
- Making a long story short :
  - **1) Substitution Method**: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.
  - **2) Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.
  - **3) Master Method** is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type : $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$