COMPLEXITY ANALYSIS

CSE221

Course:CSE221 (Algorithms)

Course Teacher: Md. Al-Amin Hossain (Lecturer)

Section: P

Depertment: CSE(43 Batch)

Group Members:

01. Md. Ashaf Uddaula (**161-15-7473**)

02. Alamin Hossain (**161-15-7483**)

03. Md. Khasrur Rahman (**161-15-7214**)

04. Md. Eram Talukder(**161-15-7485**)

05. Ijaz Ahmed Utsa (**161-15-7180**)

GOING TO TELL ABOUT……….

- Motivations for Complexity Analysis.
- Machine independence.
- Best, Average, and Worst case complexities.
- Simple Complexity Analysis Rules.
- Simple Complexity Analysis Examples.
- Asymptotic Notations.
- Determining complexity of code structures.

MOTIVATIONS FOR COMPLEXITY ANALYSIS

- There are often many different *algorithms* which can be used to solve the same problem. Thus, it makes sense to develop techniques that allow us to:
  - compare different algorithms with respect to their “efficiency”
  - choose the most efficient algorithm for the problem

- The ***efficiency*** of any algorithmic solution to a problem is a measure of the:
  - **Time efficiency**: the time it takes to execute.
  - **Space efficiency**: the space (primary or secondary memory) it uses.

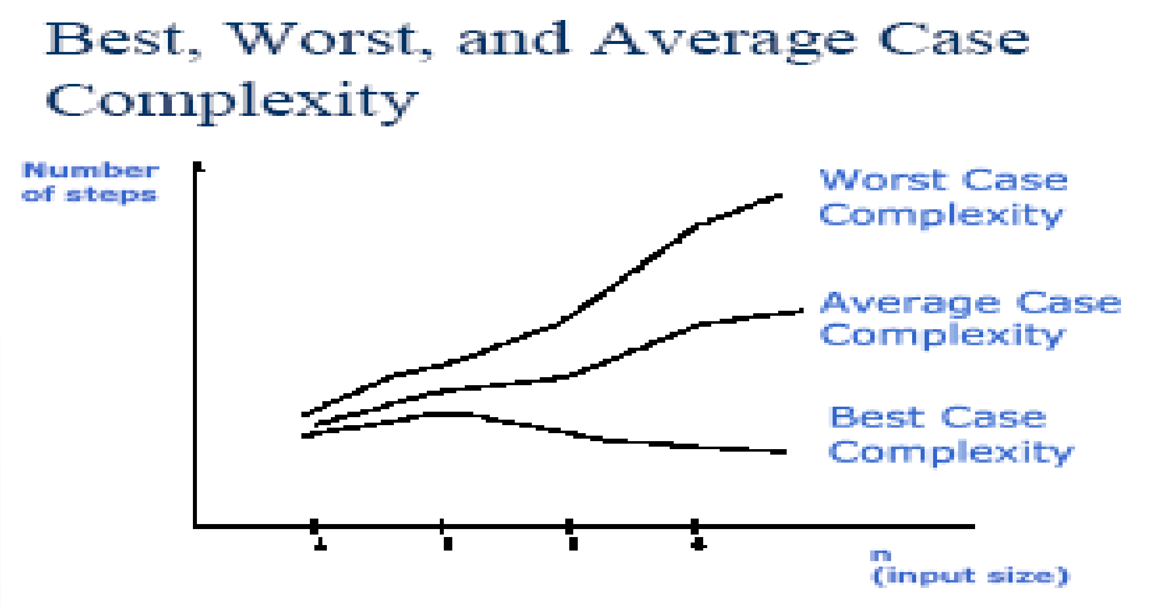
- We will focus on an algorithm’s efficiency with respect to time.

MACHINE INDEPENDENCE

- The evaluation of efficiency should be as machine independent as possible.
- It is not useful to measure how fast the algorithm runs as this depends on which particular computer, OS, programming language, compiler, and kind of inputs are used in testing
- Instead,
  - we count the number of *basic operations* the algorithm performs.
  - we calculate how this number depends on the size of the input.
- A basic operation is an operation which takes a constant amount of time to execute.
- Hence, <u>the efficiency of an algorithm is the number of basic operations it performs</u>. This number is a function of the input size **n**.

BEST, AVERAGE, AND WORST CASE COMPLEXITIES

- We are usually interested in the **worst case** complexity: what are the most operations that might be performed for a given problem size. We will not discuss the other cases -- **best** and **average case**
- Best case depends on the input
- Average case is difficult to compute
- So we usually focus on worst case analysis
  - Easier to compute
  - Usually close to the actual running time
  - Crucial to real-time systems (e.g. air-traffic control)



BEST, AVERAGE, AND WORST CASE COMPLEXITIES

- Example: Linear Search Complexity
- Best Case : Item found at the beginning: ***One comparison***
- Worst Case : Item found at the end: ***n comparisons***
- Average Case :Item may be found at index 0, or 1, or 2, . . . or n - 1

  -Average number of comparisons is: (1 + 2 + . . . + n) / n = **(*n*+1) / 2**

- Worst and Average complexities of common sorting algorithms

Method	Worst Case	Average Case
Selection sort	n^2	n^2
Inserstion sort	n^2	n^2
Merge sort	n log n	n log n
Quick sort	n^2	n log n

SIMPLE COMPLEXITY ANALYSIS: LOOPS

- We start by considering how to count operations in **for**-loops.
  - We use integer division throughout.

- First of all, we should know the number of iterations of the loop; say it is **x**.
  - Then the loop condition is executed **x + 1** times.
  - Each of the statements in the loop body is executed **x** times.
  - The loop-index update statement is executed **x** times.

SIMPLE COMPLEXITY ANALYSIS: LOOPS (WITH <)

- In the following for-loop:

```
for (int i = k; i < n; i = i + m){
        statement1;
        statement2;
 }
```

The number of iterations is: **(n – k) / m**

- The initialization statement, **i = k**, is executed **one** time.
- The condition, **i < n**, is executed **(n – k) / m + 1** times.
- The update statement, **i = i + m**, is executed **(n – k) / m** times.
- Each of **statement1** and **statement2** is executed **(n – k) / m** times.

SIMPLE COMPLEXITY ANALYSIS : LOOPS (WITH <=)

- In the following for-loop:

```
for (int i = k; i <= n; i = i + m){
        statement1;
        statement2;
 }
```

- The number of iterations is: (**n – k) / m + 1**
- The initialization statement, **i = k**, is executed **one** time.
- The condition, **i <= n**, is executed (**n – k) / m + 2** times.
- The update statement, **i = i + m**, is executed (**n – k) / m + 1** times.
- Each of **statement1** and **statement2** is executed (**n – k) / m + 1** times.

SIMPLE COMPLEXITY ANALYSIS: LOOP EXAMPLE

- Find the exact number of basic operations in the following program fragment:

```
double x, y;
x = 2.5 ; y = 3.0;
for(int i = 0; i < n; i++){
    a[i] = x * y;
    x = 2.5 * x;
    y = y + a[i];
}
```

- There are 2 assignments outside the loop => 2 operations.
- The **for** loop actually comprises
- an assignment (i = 0) => 1 operation
- a test (i < n) => n + 1 operations
- an increment (i++) => 2 n operations
- the loop body that has three ***assignments***, two ***multiplications***, and an ***addition*** => 6 n operations
  Thus the total number of basic operations is 6 * n + 2 * n + (n + 1) + 3
  = **9n + 4**

SIMPLE COMPLEXITY ANALYSIS: LOOPS WITH LOGARITHMIC ITERATIONS

- In the following for-loop: (with <)

```
for (int i = k; i < n; i = i * m){
        statement1;
        statement2;
 }
```

-The number of iterations is: $\lceil$**(Log$_m$ (n / k))**$\rceil$

- In the following for-loop: (with <=)

```
for (int i = k; i <= n; i = i * m){
        statement1;
        statement2;
 }
```

-The number of iterations is: $\lfloor$ **(Log$_m$ (n / k) + 1)** $\rfloor$

ASYMPTOTIC NOTATIONS

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

DETERMINING COMPLEXITY OF CODE STRUCTURES

Loops:

Complexity is determined by the number of iterations in the loop times the complexity of the body of the loop.

Examples:

```
for (int i = 0; i < n; i++)
      sum = sum - i;
```

$O(n)$

```
for (int i = 0; i < n * n; i++)
      sum = sum + i;
```

$O(n^2)$

```
int i=1;
while (i < n) {
      sum = sum + i;
      i = i*2
}
```

$O(\log n)$

```
for(int i = 0; i < 100000; i++)
      sum = sum + i;
```

$O(1)$

DETERMINING COMPLEXITY OF CODE STRUCTURES

Nested independent loops:

Complexity of inner loop * complexity of outer loop.

Examples:

```
int sum = 0;
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        sum += i * j ;
```

$O(n^2)$

```
int i = 1, j;
while(i <= n) {
    j = 1;
    while(j <= n){
        statements of constant complexity
        j = j*2;
    }
    i = i+1;
}
```

$O(n \log n)$

DETERMINING COMPLEXITY OF CODE STRUCTURES

Nested dependent loops: Examples:

```
int sum = 0;
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= i; j++)
        sum += i * j ;
```

Number of repetitions of the inner loop is: 1 + 2 + 3 + . . . + n = n(n+2)/2
Hence the segment is O(n2)

```
int sum = 0;
for(int i = 1; i <= n; i++)
    for(int j = i; j < 0; j++)
        sum += i * j ;
```

Number of repetitions of the inner loop is: 0
The outer loop iterates n times.
Hence the segment is O(n)

```
int n = 100;
// . . .
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        sum += i * j ;
```

An important question to consider in complexity analysis is whether the problem size is a variable or a constant.

DETERMINING COMPLEXITY OF CODE STRUCTURES

If Statement:

O(max(O(condition1), O(condition2), . . ,

O(branch1), O(branch2), . . ., O(branchN))

```
char key;
........
if(key == '+')   {                      O(1)
   for(int i = 0; i < n; i++)           O(n^2)
     for(int j = 0; j < n; j++)
        C[i][j] = A[i][j] + B[i][j];
}

else if(key == 'x')                     O(1)
   C = matrixMult(A, B, n);             O(n^3)

else                                    O(1)
   System.out.println("Error! Enter '+' or 'x'!");
```

Overall complexity $O(n^3)$

DETERMINING COMPLEXITY OF CODE STRUCTURES

O(if-else) = Max[O(Condition), O(if), O(else)]

```
int[] integers = new int[100];
// n is the problem size, n <= 100
........
if(hasPrimes(integers, n) == true)
      integers[0] = 20;               → O(1)
else
      integers[0] = -20;              → O(1)


public boolean hasPrimes(int[] x, int n) {
      for(int i = 0; i < n; i++)
                ..........
                ..........          O(n)
}
```

$O(\text{if-else}) = O(\text{Condition}) = \boldsymbol{O(n)}$

DETERMINING COMPLEXITY OF CODE STRUCTURES

Switch: Take the complexity of the most expensive case including the default case

```
char key;
int[] x = new int[100];
int[][] y = new int[100][100];
........
 // n is the problem size (n <= 100)
switch(key)  {
     case 'a':
        for(int i = 0; i < n; i++)
          sum += x[i];
        break;
     case 'b':
        for(int i = 0; i < n; j++)
          for(int j = 0; j < n; j++)
               sum += y[i][j];
        break;
  }
```

o(n)

o(n^2)

Overall Complexity: o(n^2)

THE END

Success comes from Experience

&

Experience comes from Bad Experience