Lecture 02

Complexity Analysis of Algorithms

Md. Tanvir Rahman
Lecturer, Dept. of ICT
MBSTU

Outlines

- 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"

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.

BEST, AVERAGE & 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 & 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 complexities of may be found at index 0, or 1, or 2, . . . or n – 1
- Some Examples

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

LOOPS

- We start by considering how to count operations in **for**-loops.
- 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 <)

```
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** times
- 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 <=)

```
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** times
- The condition, **i < n**, is executed (**n – k) / m + 1 + 1** 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

- 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

```
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];
}
```

- 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**

Asymptotic Notation

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

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.

```
# n is variable
# c is increment
for i in range(1,n,c):
    #some O(1) expressions
    print(i)
```

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

```
# n is variable
# c is increment
for i in range(1,n,c):
    #some O(1) expressions
    for j in range(1,n,c):
        #some O(1) expressions
        print(i,j)
```

$O(n^2)$

Big O Notation

3) $O(n^2)$

- 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)

- Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
# n is variable
# c is constant
i=2
while i<=n:
    print(i)
    i=i*c
```

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

- How to combine time complexities of consecutive loops?

```
# n,k is variable

for i in range(n):
    print(i)
for j in range(m):
    print(j)
```

- Time complexity of above code is O(n) + O(m) which is O(n+m)

Problem

Find the complexity of the program?

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

Problem

Even though the inner loop is bounded by n, but due to break statement it executing only once.

So, Time Complexity : O(n)

```
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;
        }
    }
}
```

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)