Complexity Analysis (Part I)

• Motivations for Complexity Analysis.

• Example of Basic Operations

• Average, Best, and Worst Cases.

• Simple Complexity Analysis Examples.
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:
  o compare different algorithms with respect to their “efficiency”
  o choose the most efficient algorithm for the problem

• The *efficiency* of any algorithmic solution to a problem is a measure of the:
  o *Time efficiency*: the time it takes to execute.
  o *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,
  o we count the number of basic operations the algorithm performs.
  o 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, the efficiency of an algorithm is the number of basic operations it performs. This number is a function of the input size $n$. 
Example of Basic Operations:

- Arithmetic operations: *, /, %, +, -
- Assignment statements of simple data types.
- Reading of primitive types
- Writing of a primitive types
- Simple conditional tests: if (x < 12) ...
- Method call (Note: the execution time of the method itself may depend on the value of parameter and it may not be constant)
- A method's return statement
- Memory Access
- We consider an operation such as ++ , += , and *= as consisting of two basic operations.
- **Note**: To simplify complexity analysis we will not consider memory access (fetch or store) operations.
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: \( \frac{1 + 2 + \ldots + n}{n} = \frac{n+1}{2} \)

- Worst and Average complexities of common sorting algorithms

<table>
<thead>
<tr>
<th>Method</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Merge sort</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
</tr>
<tr>
<td>Quick sort</td>
<td>(n^2)</td>
<td>(n \log n)</td>
</tr>
</tbody>
</table>
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:

```java
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:

```java
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 \textbf{statement1} and \textbf{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:

```java
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

\[6 \times n + 2 \times n + (n + 1) + 3 = 9n + 4\]
Simple Complexity Analysis: Examples

• Suppose \( n \) is a multiple of 2. Determine the number of basic operations performed by the method `myMethod()`:

```java
static int myMethod(int n){
    int sum = 0;
    for(int i = 1; i < n; i = i * 2)
        sum = sum + i + helper(i);
    return sum;
}
```

```java
static int helper(int n){
    int sum = 0;
    for(int i = 1; i <= n; i++)
        sum = sum + i;
    return sum;
}
```

• Solution: The number of iterations of the loop:
  
  \[
  \text{for}(\text{i} = 1; \text{i} < n; \text{i} = \text{i} \times 2)
  \sum = \sum + \text{i} + \text{helper}(i);
  \]

  is \( \log_2 n \) (A Proof will be given later)

  Hence the number of basic operations is:

  \[
  1 + 1 + (1 + \log_2 n) + \log_2 n[2 + 4 + 1 + 1 + (n + 1) + n[2 + 2] + 1] + 1
  = 3 + \log_2 n + \log_2 n[10 + 5n] + 1
  = 5n \log_2 n + 11 \log_2 n + 4
  \]
Simple Complexity Analysis: Loops With Logarithmic Iterations

• In the following for-loop: (with <)

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

  – The number of iterations is: \( \lceil \log_m \left( \frac{n}{k} \right) \rceil \)

• In the following for-loop: (with \( \leq \))

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

  – The number of iterations is: \( \lfloor \log_m \left( \frac{n}{k} \right) + 1 \rfloor \)