



COM410 Programming in Practice

B1.2 Counting Operations



Analysis of Algorithms

- Difficult to compute the actual time requirement of an algorithm
- Instead, find a **function of the problem size** that behaves like the algorithm's actual time requirement:

As the time requirement increases by some factor, the value of the function increases by the same factor and vice versa (the value of the function is said to be directly proportional to the time requirement)



This is known as a **growth-rate function** because it measures how an algorithm's time requirement grows as the problem size grows

Counting Basic Operations

- An algorithm's **basic operation** is the most significant contributor to its total time requirement

// Algorithm A

```
long sum = 0;
for (long i=1; i<=n; i++) {
    sum = sum + i;
}
```

// Algorithm C

```
long sum = 0;
sum = n * (n + 1) / 2;
```

// Algorithm B

```
long sum = 0;
for (long i=1; i<= n; i++) {
    for (long j=1; j<=i; j++) {
        sum = sum + 1;
    }
}
```


- The most frequent operation is not necessarily the basic operation (e.g. assignment operations are often most frequent but rarely basic)

Counting Basic Operations

- Ignoring operations that are not basic (e.g. initialization of variables, operations that control loops, etc.) will not affect any final conclusion about an algorithm's speed

```
// Algorithm A  
long sum = 0;  
for (long i = 1; i <= n; i++) {  
    sum = sum + i;  
}
```

basic
operation



growth-rate
function



- Algorithm A requires $(2n + 2) + 2n + (n + 1) = 5n + 3$ operations
- Algorithm A requires time that increases linearly with n ($5n + 3$) so we can conclude that:
Algorithm A requires time that is directly proportional to n

Counting Basic Operations

// Algorithm B

```
long sum = 0;
for (long i=1; i<= n; i++) {
    for (long j=1; j<=i; j++) {
        sum = sum + 1;
    }
}
```

- Computes the sum as:
 $0 + (1) + (1 + 1) + .. + (1 + 1 + .. + 1)$
- This works out as $n(n + 1) / 2$ additions, which is equivalent to $(n^2 + n) / 2$ basic operations

- Algorithm B requires time proportional to n^2

growth-rate function

Counting Basic Operations

```
// Algorithm C
```

```
long sum = 0;
```

```
sum = n * (n + 1) / 2;
```

- Computes the sum as algebraic identity involving 3 operations independent of n
- Algorithm C requires time that is **constant: 3**

growth-rate function

Estimating Performance

- Consider two algorithms that require $10n$ operations and n^3 operations, respectively. Which of these algorithms is the slowest to execute in terms of its operations?

n	$10n$	n^3
-----	-------	-------

Best, Worst, and Average Cases

- For some algorithms, execution time depends only on size of data set
- The time requirement for other algorithms may depend on the size and nature of the dataset... e.g. finding a specified integer value in an array of integers:

If the algorithm finds the value in the first array element, it makes only one comparison – this is the **best case**

If the algorithm finds the value after searching every array element – this is the **worst case**

- Best case and worst case rarely occur, so we consider the **average case**
- More difficult to estimate and not the average of best case and worst case!

Scenario

Measure algorithm performance

- Create a new class called **SumAnalysis** in a new file called *SumAnalysis.java* within your **Analysis** project.
- In the **main()** method of the new class, measure and record the execution time in nanoseconds for each the **SumIntegers** class methods **sumA()**, **sumB()** and **sumC()** for parameter values 1, 10, 100, 1000, 10000, 100000 and 1000000
- Print the results of the execution time measurements in a table and observe the rate of increase for each.
- Estimate the growth rate function for each of **sumA()**, **sumB()** and **sumC()**.