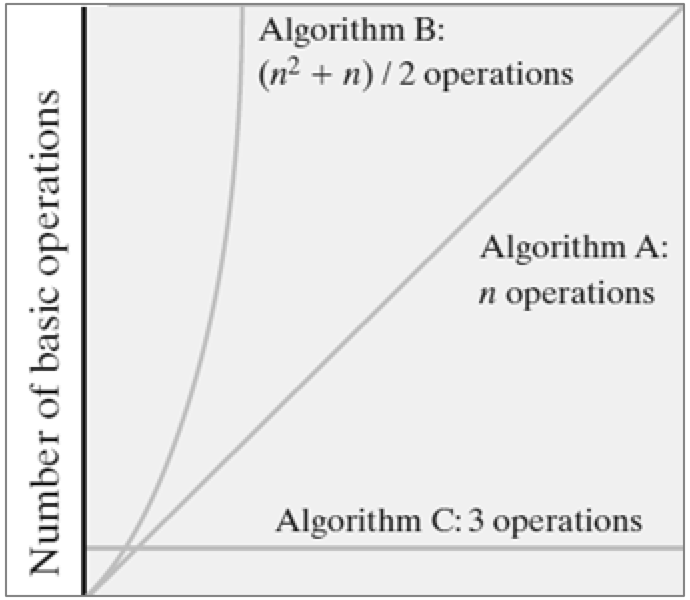
COM410 Programming in Practice

B1.3 The Big O Notation

Big O Notation

- Computer scientists use a *notation* to represent an algorithm's complexity



- Algorithm A has a time requirement proportional to ***n***
- Algorithm B has a time requirement proportional to ***n*²**
- Algorithm C has a **constant** time requirement

- We would instead say that Algorithm A is **O(*n*),** Algorithm B is **O(*n*²)**, and Algorithm C is **O(1)**
- This is known as **Big O Notation** (read "Big Oh of *n*" or "order of *n*")

Big O Notation

- A means of expressing the runtime of an application
  - in terms of how quickly it grows,
  - relative to the input,
  - as the input get arbitrarily large

Constant time

```
public static void printFirstItem(int[] items) {
    System.out.println(items[0]);
}
```

- No matter what size the input array is, the execution time will always be the same
  - O(1) time

Linear time

```
public static void printAllItems(int[] items) {
    for (int item : items) {
        System.out.println(item);
    }
}
```

- Execution time is directly proportional to the size of the input array
  - O(n) time

Quadratic time

```
public static void printAllorderedPairs(int[] items) {
    for (int first : items) {
        for (int second : items) {
            System.out.println(first + “, “ + second);
        }
    }
}
```

- Execution time is directly proportional to the square of the size of the input array
  - $O(n^2)$ time

Drop the constants

```
public static void printAllItemsTwice(int[] items) {
    for (int item : items) {
        System.out.println(item);
    }
    for (int item : items) {
        System.out.println(item);
    }
}
```

- Execution time is still directly proportional size of the input array
  - O(n) time even though there are 2n operations

Use the most significant term

```
public static void printPairsThenItems(int[] items) {
    for (int first : items) {
        for (int second : items) {
            System.out.println(first + ", " + second);
        }
    }
    for (int item : items) {
        System.out.println(item);
    }
}
```

- Execution time is still directly proportional to the square of the size of the input array
  - $O(n^2)$ time even though there are $n^2 + n$ operations

Efficiency of Implementations of Bag ADT

Operation	Fixed-size Array	Linked Chain
`add(newEntry)`	O(1)	O(1)
`remove()`	O(1)	O(1)
`remove(anEntry)`	O(1), O(n), O(n)	O(1), O(n), O(n)
`clear()`	O(n)	O(n)
`getFrequencyOf()`	O(n)	O(n)
`contains()`	O(1), O(n), O(n)	O(1), O(n), O(n)
`toString()`	O(n)	O(n)
`getCurrentSize()`	O(1)	O(1)

Challenge

Prove the time complexity of the `LinkedList` add and remove operations

- Add a new class called `ListComplexityTest` to the **Analysis** project
- The `main()` method of `ListComplexityTest` should create a new `LinkedList` of `Integer` objects and generate N random integers < N*10, storing each in the `LinkedList` as it is generated.
- For each integer generated, measure the time taken in nanoseconds to store the element in the `LinkedList` and obtain the total time taken to add the elements.
- Now generate a further N random integers in the same range and attempt to remove an element of that value from the `LinkedList`, again calculating the total time to remove.
- Repeat for values of N of 10, 100, 1000, 10000 and 100000 elements and report for each value of N, the total time to add, the average time to add, the total time to remove and the average time to remove.