Lecture 1

The Big Picture
When to use what and how much does it cost?

We all make decisions, such as buying a computer or a car, etc. And, to make a wise decision, we ask the same question over and over again.
“For what do I need this and how much am I willing to pay?”

“Is a program good enough **as long as it works**?”

I. Perspective: Performance is important! A program that is terribly slow is almost as useless as a wrong one.

II. The goal is to make sure a program works and also it is effective. However, paying too much attention to performance is also not desirable.

III. **The first priority** as an application programmer is to make your code **work and clear** to understand.

IV. Focusing too much on performance can lead you to create complicated code that is difficult to understand.

V. Then, the next question is “Why do we care about performance in the first place?”

VI. One of the common mistakes made by application programmers is to **ignore performance**. Where is the root of this ignorance? One major reason is the lack of basic understanding of data structures. In this course, we will try to overcome this by looking at data structures and the fundamentals of “Big O” algorithmic complexity analysis.

What is the Cost? (Big O Notation)

How fast my program runs with 100 data to process?

How about if I double the size of the data?

What if I double up again?

The relationship between the problem size n and running time, T(n).

How do we measure the cost? We can definitely measure time that each program takes to run and compare them. But is it really useful?

Automobiles are divided into several categories: subcompacts, compacts, midsize, full-size, etc. These categories provide a quick idea about what size car you're talking about, without needing to mention actual dimensions.

Similarly, we want to define the cost without depending on the implementation details.

Example: Find a card
I am looking for a card, 5 heart, from a deck of cards that are shuffled.
The deck has 5 cards.

The deck now has 10 cards. (Double up!)

The deck has 20 cards. (Double up again!)

In the search process, we need to compare the value of a card from a deck each time (a step). Initially, this will take 5 steps to finish. The total computational time to search n cards, $T(n) = c * n$, where c is time taken by comparing the value of a card from a deck.

This will of course take different time on different computers. $T(n) = c1 * n$ and $T(n) = c2 * n$ respectively. But, we can conclude that T(n) grows as input size, n, increases on both computers.

This process of abstraction to determine the running time ***in relation to the input size*** is what we call, Big O Notation, in computer science.

Classifications (Most common ones)

O(1) : Constant Time
An algorithm does NOT depend on the input size.

O(log n) : Logarithmic or Proportional to log(n)
An algorithm gets *slightly* slower as n grows.

O(n) : Linear or Proportional to n
The running time grows as much as n grows. (When n doubles, so does the running time.)

O(n log n) : Linear-logarithmic or linearithmic time
Simply means a product of a linear term and a logarithmic term. In many cases, linearithmic time is the result of performing O(log n) operation n times or performing O(n) operation log n times.

O(n^2) : Quadratic
Whenever n doubles, the running time increases fourfold. Practical for use only on relatively small problems. *Most of the efforts we will make are related to NOT to fall into the Quadratic time case.*

The ones above are categorized to be of polynomial time (tractable and feasible).

The ones below are categorized to be of super-polynomial time (non-tractable).

O(2^n) : Exponential
Now, n becomes the exponent. The running time increases even more.
Comparison with Quadratic:
n^2 (Quadratic): 2^2 = 4, 4^2 = 16, 8^2 = 64, 16^2 = 256
2^n (Exponential): 2^2 = 4, 2^4 = 16, 2^8 = 256, 2^16 = _____

O(n!) : Factorial

By definition, n! = 1*2*3…(n-1)*n. The running time increases even more than exponential.

Comparison with Exponential.

2^n(Exponential): 2^2 = 4, 2^4 = 16, 2^8 = 256, 2^16 =_____

n!(Factorial): 2! = 2, 4! = 24, 8! = 40,320, 16! = _____

Some Examples

1. int sum; sum = 0;

2. Search a card in a deck of cards that is shuffled (Assuming that you are checking each card one by one till you find the card you are looking for.)

3. The Guess-a-Number Game (Let's play a game!)

4. Sort baseball players, assuming that you are near-sighted, which means you can only see a player who is right beside you.

Graphs of Big O relationships between time and number of items.

Categories of Data Structures and Algorithms

Very generally, we can categorize data structures and general algorithms as follows.

1. General-purpose data structures: arrays, linked lists, trees and hash tables
2. Specialized data structures: stacks, queues, priority queues
3. Sorting and searching: bubble sort, insertion sort, selection sort, quicksort, mergesort, heapsort, linear search and binary search
4. Graphs

I. General-Purpose Data Structures

Assuming that you use Facebook, we all know that Facebook has a lot of data about yourself. They have your personal information, list of friends, status updates and pictures, etc. To deal with this kind of real-world user-accessible data, you may need general-purpose data structures.

Which of these general-purpose data structures is appropriate for a given problem?

1. Arrays
    - The amount of data is reasonably small
    - The amount of data is predictable in advance

2. Linked Lists
    - The amount of data is comparatively small
    - The amount of data cannot be predicted in advance
    - When data will frequently be inserted and deleted

3. Binary Search Trees
    - Generally, a binary search tree is the first structure to consider when arrays and linked lists prove too slow.
    - It provides fast insertion, searching, and deletion.

- When you are sure that the input data will arrive in random order (Ordered data as an input can reduce its performance, no better than a linked list.)

4. Hash Tables
    - Generally, the fastest, especially for search, insertion, and deletion
    - Not sensitive to the order in which data is inserted
    - Requires additional memory
    - Performance is affected by initial capacity and load factor

Relationship of general-purpose data structures

Comparing General-Purpose Data Structures

Data Structure	Search	Insertion	Deletion	Traversal
Array				
Ordered array				
Linked list				
Ordered linked list				
Binary tree (average)				
Binary tree (worst case)				
Balanced tree (avg & worst)				
Hash table				

II. Special-Purpose Data Structures

The special-purpose data structures are usually used by a computer program to aid in carrying out some algorithms, rather than dealing with user-accessible data.
These are Abstract Data Types. The ADTs can be seen as conceptual aids. Their functionality could be obtained using the underlying structures (such as an array) directly but the reduced or limited interface they offer simplifies many problems.

1. Stack: used when a program needs to access only the last data item inserted. (LIFO)

2. Queue: used when a program needs to access only the first item inserted (FIFO)

3. Priority Queue
    - Used when the only access desired to the data item with the highest priority
    - Can be implemented using an array, a linked list or a heap. When speed is important, a heap is a better choice.

Data Structure	Insertion	Deletion	Comment
Stack (array or linked list)			Deletes most recently inserted item
Queue (array or linked list)			Deletes least recently inserted item
Priority Queue (Ordered array)			Deletes highest-priority item
Priority Queue (heap)			Deletes highest-priority item

III. Sorting and searching (Algorithms)

As we deal with data, it is obvious that we usually need to sort them or search for an item among them. Understanding common sorting and searching algorithms is incredibly valuable!

IV. Graphs

Quick Tip!
A simple and general rule to select a data structure for your applications or algorithms:

- Analyze the problem to determine the basic operations that must be supported.
- Quantify the resource constraints for each operation.
- Select a data structure (or more) that best meets these requirements.

Also, remember that any data structure requires:

- Space for each element or data item to be stored.
- Time to perform basic operations.
- Effort to program

From lecture 2, we will explore JAVA Collections and other classes and look closely at data structures implemented or used by them and their costs and benefits.