# Week Lecture Notes

## O-notation Examples

In-class activity: O-notation quiz

**Rule of thumb**: The highest order term in an expression tells you it’s tightest O-notation expression. For example:

* n^2 – 5n + 100 is O(n^2)
* 1 + 20n^2 + 4n^3 + 20n is O(n^3)
* n(n + 4) + 60 is O(n^2)
* n + n log 2n + 5 is O(n log n)

**Fact**: Degree k polynomials are O(n^k), e.g.

* n^2 + n + 100 is O(n^2)
* n^3 – n^2 + 100n is O(n^3)

We will **not** be covering big omega or big theta notation.

|  |  |  |  |
| --- | --- | --- | --- |
| O-notation | Name | Example | Example Algorithm |
| O(1) | Constant | 1, 50, 483, … | **Finding the max** of a sorted array |
| O(log n) | Logarithmic | 4 + log 2n | **Binary search** on an array of n items in sorted order |
| O(n) | Linear | 10n+4 | **Linear search** on an array of n items in any order |
| O(n log n) | n-log-n | 2n log (n + 4) | Worst-case performance of **mergesort** to sort n items |
| O(n^2) | Quadratic | 3n^2 + 3n - 5 | Worst-case performance of **insertion sort** to sort n items |
| O(n^3) | Cubic | n^3 + n + n log n - 1 | **Multiplying two n-by-n matrices** in the standard way |
| O(2^n) | Exponential | 2^n, 3 \* 2^(n+5), … | **Printing all subsets** of {1, 2, …, n}. |

## ADTs: Abstract Data Types

An **abstract data type (ADT)** is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies **what** each operation does, but **not** how it does. In other words, an ADT carefully specifies the interface between it and the rest of the program it’s operating in.

**Example**. In C++, ADTs are typically represented as abstract base classes, such as **Wordlist\_base** from assignment 1. The base defines **how** the methods do, but does not **not** say how to implement those methods.

**Example**. A good intuitive example of an ADT is an electrical outlet on a wall. It has 3 precisely defined holes that let you can insert any plug into. It “promises” to provide electricity through the plug, but it says nothing about how the electricity is generated. The electricity could come from solar energy, a gas engine, geo-thermal energy, or even from a hamster running in a wheel.

ADTs also help with:

* **Encapsulation**: low-level details specific to the ADT are hidden, “encapsulated”, inside the ADT. In C++, we can declare such details private, to prevent code outside the class from seeing/changing these details.  
  **Example**. C++ strings typically store a pointer to an underlying array of characters, plus a length and capacity variable. These variables are private in the string class, as we don’t want programmers changing them. A regular programmer who uses strings doesn’t need to worry about these details, or even know that they exist. Compared to C-strings, C++ strings are much easier to use since you can usually ignore the implementation details.
* **Modularity**: Since ADTs have a clearly defined interface, they can be treated as independent modules. We built and test ADTs on their own separate from the rest of the program, and then “plug” them in when they are ready. Making programs modular is an important part of good software engineering.  
  **Example**. There are multiple ways that C++ strings could be implemented. For example, a string might use the rope data structure, which works well for large strings. If you are currently using one string implementation, then you “plug in” in a new implementation without needing to change any of the code that calls the string.

## Stacks, Queues, Deques

The **stack** ADT.

An array/vector implementation of a stack.

A linked list implementation of a stack.

The **queue** ADT.

An array/vector implementation of a queue.

A linked list implementation of a queue.

The **deque** ADT.

An array/vector implementation of a deque.

A linked list implementation of a deque.

**Adaptors**: implementing stacks and queues using deques.