Chapter 3 Exercises

Exercise 3-1

Big O notation and order of growth

1. `cube(n) + square(n)`, `100000 * cube(n) + square(n)`, and `cube(n) + 100000 * square(n)` are all of the order **O(cube(n))**
2. `(square(n) + n) * (n + 1)` is of the order **O(cube(n))**
3. If *f* is **O(g)**, and `a` and `b` are both constants, *af + b* will also be **O(g)**.
4. If *f1* and *f2* are both in **O(g)**, `f1 + f2` will also be in **O(g)**.
5. If *f1* is in **O(g)** and *f2* is in **O(h)**, `f1 + f2` will be in the order **O(abs(g) + abs(h))**. (Not quite sure about this one, wikipedia seems to suggest that `f1 + f2` will be in **O(i)**, where *i* is a convex cone)
6. If *f1* is in **O(g)**, and *f2* is in **O(h)**, `f1 * f2` will be in **O(gh)**.

Exercise 3-2

Sorting algorithms

1. A comparison sort is a sorting algorithm that only uses a single operation to compare elements, e.g. comparing every pair of items with `<=`. The lowest worst-case order of growth for a comparison sort is *n *.log(n)*, which is demonstrated by:
   - Merge sort
   - Heapsort
   - Binary tree sort
   - etc. The lowest wort-case order of growth for *any* sorting algorithm is *n + r*, (bucket sort), where *r* is the range of numbers that is being sorted. This only applies to integer keys, however. (This is also ignoring some special cases like spaghetti sort, which is **O(n)**, provided you have *n* parallel processors.)
2. Bubble sort has its average *and* best order of growth equal to *square(n)*. However, for a list that's already partly sorted, it can be of **O(n)**. Its worst-case performance is much worse than many other sorting algorithms, which have a worst-case performance of **O(n.log(n))**.

3. Radix sort has the best case order of growth of *O(k * n)*, where k is the maximum number of digits of the keys being sorted. Radix sort requires integer or floating point keys that have *positional notation*, i.e. each position in the key identifies a different order of magnitude.

4. A stable sort maintains the original order of items that have equal keys, e.g. when sorting `ad bz ab ce` by the first letter, giving `ad ab bz ce`. With an unstable sorting algorithm, a second sort based on a different key could destroy the ordering of the first sort. E.g. when sorting by last name, then by first name, an unstable sort could destroy the last name ordering in the first name sort.

5. The worst named sorting algorithm is bogosort/stupid sort, where the array is randomly shuffled, then checked to see if it is in order. It has an average order of growth of *O(n * n!)*, and in the worst case, tends toward infinity.

6. The default sorting algorithm in Python is *timsort*, a hybrid of insertion and merge sort that takes advantages of pre-existing ordering in the data. It has a best case performance of *O(n)*, and a worst case of *O(nlog(n))*. In **C**, the standard library includes quicksort, mergesort and heapsort. timsort and mergesort are stable. quicksort can be, but not in its most efficient form. heapsort is unstable.

7. Python uses a comparison sort to allow for comparisons of many different data-types. Non-comparison sorts such as radix can be faster, but are restricted to sorting numbers.