# Lecture Notes 7 – Introduction to algorithm analysis
## CSS 501 – Data Structures and Object-Oriented Programming – Professor Clark F. Olson

### Reading for this lecture: Carrano, Chapter 10

## Introduction to algorithm analysis
This lecture we are going to start discussing the efficiency of algorithms. This is an important topic that we will return to every time that we talk about a new algorithm. Understanding the **efficiency** of an algorithm **is important**. The **speed** and/or responsiveness of a wide-variety of applications **depend on the efficiency of the algorithm** used in the application. Efficient algorithms are **much more important than coding tricks and optimization**. Examples of applications that rely on efficient algorithms include computer games, productivity software (such as word processing and spreadsheets), systems you encounter everyday (such as ATMs and grocery checkout systems), critical applications (life support, air-traffic control), and Mars rovers.

This lecture will be largely **theoretical**. The techniques that we will talk about form the basis for analysis that we will perform over and over again in this course and that you will use in future courses. They will allow you to **compare algorithms** for some problem and **determine which is more efficient** (primarily in terms of the **computation time**, but similar techniques can be applied also to the amount of **memory required**). This lecture will concentrate on simple algorithms that you have probably seen before, but we will apply the techniques to more advanced algorithms in subsequent lectures.

For the most part, we have (so far) considered the **human cost of algorithms** (readability, modularity, etc.) This impacts how easy it is to generate, modify, test, and support programs, rather than speed or responsiveness for the user. Of course, **efficiency is also important**. You should always consider the efficiency when selecting a particular algorithm to solve a problem.

We will discuss the **efficiency of algorithms, rather than programs**. Two different implementations of the same algorithm usually vary little in speed. Clever tricks may gain you a little speed, but the big picture (that is, the efficiency of the algorithm) is far more important. We examine the analysis of algorithms abstractly for another reason. If we **compare two programs** to determine which is faster, the result depends on **which program was coded better**, which **computer was used to test** the programs, and **what data was used** for the test. Looking at the overall efficiency of an algorithm overcomes these problems.

Reasons to analyze the efficiency of an algorithm:

- Analysis helps **choose which solution to use** to solve a particular problem.
- **Experimentation tells us about one test case**, while **analysis can tell us about all test cases** (**performance guarantees**).
- **Performance can be predicted before programming.** If you wait until you've coded a large project and then discover that it runs slowly, much time has been wasted.
- If you can **tell which parts of your solution execute quickly** and which parts execute slowly, then you know which to work on to improve the overall solution.

## Growth rates
We want to **analyze the running time without knowing the details of the input**. This is done by measuring the running time as a function of the **size of the problem** (for example, the value of an integer input, the number of elements in an array, or the number of nodes in a linked list). The size is usually called *n*, although any variable can be used to denote this size.

Let's say we have **3 algorithms** to solve some problem. One takes time proportional to **100n**, one takes time proportional to **10n^2**, and one takes time proportional to **2^n**. These are the growth rates for the algorithms in terms of the size of the problem n. Let's examine how long these algorithms take for various problem sizes, assuming that **times are all in milliseconds**.

| | Alg. 1 | Alg. 2 | Alg. 3 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| n = 5 | 500 | 250 | 32 |
| n = 10 | 1000 | 1000 | 1024 |
| n = 20 | 2000 | 4000 | 1,048,576 |
| n = 50 | 5000 | 25,000 | 1.1259 x 10^15 (over 36,000 years) |
| n = 1000 | 100,000 | 10,000,000 | > 10^300 |

Algorithm 3 is the fastest for a very small problem, but all of the algorithms are done in under a second. For a slightly larger problem, they are all about the same. As the size of the problem gets bigger, algorithm 3 becomes completely unreasonable and algorithm 1 looks better and better. Usually, it is **the performance of algorithms for large problems that we are concerned about**, since **small problem can be solved quickly using most algorithms**. The **rate of growth** is the primary **measurement we will use to analyze efficiency**. Clearly, the rate of growth is the largest for algorithm 3 and lowest for algorithm 1. Algorithm 1 is the best to use if you don't know the size of the problem in advance. We will examine growth rates using Big-O notation.

## Big-O notation

Informally, if the **time required** by an algorithm **is proportional to** some function **f(n),** then the algorithm's **running time** is said to be **O(f(n)),** where f(n) is the growth-rate function. This is called Big-O notation due to the O (short for order) at the beginning of the notation. Since the running time of **algorithm 2** is proportional to n^2, its running time is **O(n^2).** O(n) is usually spoken as "big-oh of n" or simply "oh of n".

**Formal definition:**
**A function (such as an algorithm's growth rate) f(n) is O(g(n)), if there are constants *k* and $n_0$ such that f(n) <= k*g(n) for all n >= $n_0$.**

**Example 1:**
**Algorithm 1** required 100n milliseconds to run. In this case, **f(n) = 100n** (we will ignore whatever units are used in the measurement). Is the running time O(n)? Yes. If we choose k to be 100, then f(n) <= 100n for all n > 0. (Is the running time of the algorithm O(n^2)? O(log n)?)

**Example 2:**
Let's say an algorithm requires time 20n^2 + 100. Is this O(n)? No. 20n^2 + 100 grows faster than n. No matter what constant you use, there is an n for which 20n^2 + 100 is bigger than k*n. Is this O(n^2)? Yes. If we choose k to be 21 then 20n^2+100 is always less than 21n^2 for n > 10.

Why did I choose those constants? I wanted to prove that 20n^2 + 100 <= kn^2. One methodology is to choose a reasonable value of k (we can see it needs to be greater than 20 here) and solve for n (if possible). With k = 21, we get 100 <= n^2 and this is true for n>=10.

What Big-O notation indicates is that, for a sufficiently large problem ($n > n_0$), the growth rate is proportional to (or less than) the g(n) in O(g(n)).

Note that, according to the definition **f(n) = 1 is O(n^25),** since it is always less than or equal to n^25 for n >= 1. This doesn't tell us much. For this reason, we **always choose the function g(n) to be as small as we can** and still be true.

**Can a function be O(1)?** Yes, if the size of the function can be bounded, no matter what n is. An example is f(n) = 10 - n. No matter how large n is, this function can never get larger than 10. Therefore, we can choose k = 10 and $n_0$ = 0.
With these values f(n) >= k * 1 for all $n > n_0$.

**Rules for simplifying Big-O notation:**

- **Ignore lower order terms**: **f(x) = x^2 + 1000x + 1000000 is O(x^2)**. Note also that it doesn't matter what variable you use. It is customary to use n, but any variable can be used.
- **Ignore multiplicative constants**: f(n) = 42n is O(n).
- **O(f(n)) + O(g(n)) = O(f(n) + g(n)):** If your function has two parts and the first has complexity f(n) = 10n and the second has complexity g(n) = n^2. Then the total complexity is O(10n + n^2), which is O(n^2).
- **O(f(n)) * O(g(n)) = O(f(n) * g(n)):** If your function repeats the inner loop x times and the inner loop takes time x + 5, then the total complexity is O(x * (x + 5)) = O(x^2).

Examples:

| | | |
|---|---|---|
| 8n + 10n^3 + 35 | is: | O(n^3) |
| ¼ n^2 + 25n log n | is: | O(n^2) |
| 5n + 2^n / 5 | is: | O(2^n) |

For algorithms, **the complexity of the algorithm refers to the rate of growth**, so it common to say that the complexity of an algorithm is O(something). The smallest possible complexity for algorithms is **O(1),** since at least some constant amount of time must be spent no matter what. **The hierarchy of complexities** is:

| | |
|---|---|
| O(1): | Essentially constant. The complexity can be bounded, no matter how big n is. Example: adding 10 to an integer, accessing one array element. |
| O(log n): | Logarithmic growth. This complexity grows very slowly. The base doesn't matter. Example: binary search of a sorted list. |
| O(n): | Linear growth. Example: traversing a linked list with n nodes, or sequential search of an array. |
| O(n log n): | Slightly worse than linear. Usually occurs when you break a problem down into two problems of half the size that must both be solved. Example: Merge sort. |
| O(n^2): | Quadratic growth. Grows with the square of the problem size. Often two nested loops. Example: Insertion sort. |
| O(n^3): | Cubic growth. Worse than quadratic. Often three nested loops. Example: Matrix multiplication (straightforward approach). |
| O(2^n): | Exponential growth. Usually too slow for large problems. Example: The towers of Hanoi. |

O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n)

Note that these are **not the only possible complexities**, but they are the most common.

Before we start looking at particular algorithms, I want to note that, while **Big-O complexity** is very important, it **does not tell the whole story**. As we can see from the above rules, **Big-O hides the "constant of proportionality"** and lower order terms So, algorithms with the same complexity can vary by an arbitrary constant factor (the lower order terms are usually less important for large problems). If some algorithm has an extremely large constant multiplier, it may still be impractical, even if the complexity is low.

If a function f(n) is O(g(n)), this says that f(n) grows no faster than g(n) after hiding constants for large enough values of n. It doesn't say that f(n) grows as fast as g(n). There is a different notation f(n) = Ω(g(n)) (pronounced Omega of g of n) that says f(n) grows at least as fast as g(n) for large enough values of n (and after hiding constants). If f(n) is both O(g(n)) and Ω(g(n)), then it is said to be Θ(g(n)) (theta of g of n). Also f(n) = o(g(n)) (little o) says that the growth rate is strictly less than g(n).

## Analysis of search algorithms

Now, we are going to analyze the complexity of two search algorithms: sequential search and binary search. We will determine the efficiency of the algorithms in terms of the number of items in the list that is being searched. When the running time depends on the data that is given to the algorithm, there are several ways we can analyze the running time of the algorithm.

- Best case: The smallest number of comparisons that can be achieved with any input. This can be interesting, but it is not very useful, in general.
- Worst case: The largest number of comparisons that can be achieved with any input. This can be very important if we want to guarantee that the algorithm will finish within some time.
- Average case: The average number of comparisons that are performed. This is usually the most important case to analyze, but it is usually the most difficult to determine.

If the algorithms performance does not depend on the data (summing an array, outputting a linked list), then all of these will be the same. However, the performance of many algorithms depends on the data given to the algorithm. One example is searching a list. If the item is at the beginning of the list, we find it right away using sequential search. If the item isn't in the list, then we have to examine the entire list.

Let's consider the following sequential search code. This is an iterative version of the algorithm we saw previously.

```
// Find the position of an item in a list. Returns –1 if the item is not in the list.
template <typename Comparable>
int sequentialSearch(const vector<Comparable> &aList, const Comparable &item) {
        for (int i = 0; i < aList.size(); i++)
                if (item == aList[i]) return i;
        return -1;
}
```

We can measure the complexity of the search in terms of the number of comparisons made, since there is a constant amount of other work performed per comparison. We will assume that the comparisons can be performed in O(1) time, although this isn't necessarily true, in general. In the best case, sequential search finds the item on the first try. In the worst case, sequential search examines each item in the list and, thus, n iterations and comparisons are performed. Therefore, the complexity is O(n). The average case is the position of **item** in the list (assuming that it is in the list). If **item** is in the list, let's assume that each position in the list is equally likely. That is: P[i] = 1/n. Assuming **item** is in the list, the average number of comparisons is found by summing the probability of being at each position with the number of comparisons required for each position:

$$C(n) = \sum_{i=1}^{n} P[i] \cdot i = \sum_{i=1}^{n} \frac{1}{n} \cdot i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{(n+1)}{2} = \text{O(n).}$$

If **item** isn't in the list, then the number of comparisons is always n, which is also O(n).

Say the probability of an item being in the list is p, then, overall, the average number of iterations of the loop is:

$$p\frac{(n+1)}{2} + (1-p)n = n - \frac{p(n-1)}{2}$$

This is also O(n). Note that the complexity doesn't depend on whether the list is sorted or not.

If the list is sorted, we can use binary search, as we saw last lecture.

```
// Binary search using recursion.
// Precondition: the list must be sorted into ascending order.
template <typename Comparable>
int binarySearch(const vector<Comparable> &list, const Comparable &item, int first, int last) {
        if (first > last) return –1;
        int mid = (first + last) / 2;
        if (item == list[mid]) return mid;
        if (item < list[mid]) return binarySearch(list, item, first, mid - 1);
        else return binarySearch(list, item, mid + 1, last);
}
```

Example: aList = 2 3 5 7 11 13 17 item = 5

In general, how long does the function take? Each iteration of the while loop requires a constant amount of time - O(1) assuming that the comparison can be performed in constant time. This means we can measure the complexity of the function by counting the number of iterations through the loop. In the best case, we find it on the first try. The average case is a little tricky for this function and we won't compute it in this class, but we can compute the worst case. We can define the number of iterations necessary in the worst case using a recurrence relation:

$F(1) = 1$
$F(n) \leq 1 + F(n/2) \leq 1 + 1 + F(n/4) \leq k + F(n/2^k)$

The process ends when $n/2^k$ is less than or equal to 1 and we can use the base case F(1):

$F(n) \leq 1 + \log_2 n = O(\log n)$

Usually, when you can break the problem in half and then find the solution by solving only one half of the original problem, the complexity of the algorithm will be O(log n). If you have to solve both halves of the problem, this is not true, as we will see when we discuss sorting.

It turns out that when n is large, the average case is nearly the same as the worst case, although it is slightly lower. How much better is binary search than sequential search? For a 1 million word dictionary, sequential search does 500,000.5 comparisons on average (if the word is in the dictionary). For the same dictionary, binary search does 21 comparisons *in the worst case*. Note, however, that you must have a sorted list to use binary search.

Interestingly, the average case performance of binary search can be improved using **interpolation search**. The idea is that you don't start looking in the middle of the list. You predict where the item should be using interpolation. For example, Hank Aaron would be expected to be near the start of the phone book. It turns out that the worst-case performance of this algorithm is O(n), but the average case is O(log log n). However, because of the computation time to perform the interpolation, it is usually not as fast as binary search! The best time to use this algorithm would be when the list of items to search is larger than can fit in memory. Since accessing the disk is expensive (compared to working in memory), it is ideal to reduce the number of times the disk needs to be accessed.

## Practice problems (optional):

Carrano, Chapter 10:

#1, #3, #4, #5

Which of the following functions are $O(n^2)$? (Remember that Big-O notation provides an upper bound only.)

a. $15n - 1$
b. $n^3 / 10$
c. $6 n^2 + 12 n + 8$
d. $n \log n - 10$
e. $n^4 / ((n - 2) \cdot (n - 3))$
f. $2^n / n^2$

For the typical algorithms that you use to perform calculations by hand, determine the running time in Big-O notation to:

a. Add two N-digit integers
b. Multiply two N-digit integers
c. Divide two N-digit integers