# CMPS 10
# Introduction to Computer Science
# Lecture Notes

## Chapter 3: Attributes of Algorithms

- Correctness
- Clarity
- Elegance
- Efficiency

The issue of Correctness is really a mathematical one: an algorithm should produce correct results on *all* possible instances of the problem it purports to solve. To prove correctness then, it is not sufficient to trace execution on any finite number of instances. Rigorous proofs of correctness are really outside the scope of this class, and we will be satisfied with intuitive explanations of an algorithm's operation, along with an occasional trace.

Clarity means readability and ease of understanding. One way to achieve this is by choosing good descriptive variable names, and by choosing logical constructs which clarify meaning. For instance, in line 10 of Sequential Search we could have had

10.) if $i > n$

or

10.) if found = false

instead of the much better

10.) if not found

Each of these conditional operations are logically equivalent in the context of Sequential Search, but only the last alternative really reads like natural English. The goal should be to write code which is so clear that no comments or explanations are necessary.

Elegance is related to clarity, and is sometimes in conflict with it. Elegance means essentially simplicity and brevity, accomplishing the task with as few lines of code as possible. Consider the following algorithm which gets a positive integer *n* from the user, then prints out the sum of all the integers from 1 to *n*.

1.) get $n$
2.) sum = 0
3.) $i = 1$
4.) while $i \leq n$
5.) sum = sum + $i$
6.) $i = i + 1$
7.) print sum
8.) stop

This is a very simple and elegant algorithm for adding up a range of integers, but it's surprising when one first learns there is a much more elegant and succinct solution to the problem.

1.) get $n$
2.) print $n(n+1)/2$
3.) stop

At first glance, it may not be clear that these two algorithms do the same thing. To prove that they do, we must prove the formula

$$1+2+3+\cdots\cdots+(n-2)+(n-1)+n \;=\; \frac{n(n+1)}{2}$$

which was discovered by the mathematician C. F. Gauss at an early age. Let $S$ denote the left hand side of the formula. Then

$$S \;=\; 1+2+3+\cdots\cdots+(n-2)+(n-1)+n\,,$$

reversing the order of the terms, we get

$$S \;=\; n+(n-1)+(n-2)+\cdots\cdots+3+2+1\,,$$

then adding the last two equations yields

$$S+S \;=\; (n+1)+(n+1)+(n+1)+\cdots\cdots+(n+1)+(n+1)+(n+1)\,.$$

Since the right hand side has exactly $n$ terms, this is equivalent to

$$2S \;=\; n(n+1)\,,$$

whence upon dividing both sides by 2, we get

$$S \;=\; \frac{n(n+1)}{2}\,,$$

which proves the formula.

**Exercise** Write an algorithm in pseudo-code which given a positive integer $n$ as input, finds the sum of the first $n$ odd positive integers, then prints out that sum. Do this first (a) by using a while loop, then (b) write a more elegant version by deducing a simple and elegant formula for the sum. Do the same thing for the first $n$ even positive integers.

Efficiency is the term used to describe an algorithm's use of resources. The two resources we are most concerned with are space (i.e. memory), and time. Space efficiency can be judged by the amount of information the algorithm stores in order to do it's job, i.e. how many and what kind of variables are used. Some memory must be used to store the input data itself. If an algorithm uses just a few more variables to process the data, it is considered space efficient. If it uses as much space or more than that need to store the input data, it is considered space inefficient.

The rest of this chapter is concerned with classifying algorithms as to their time efficiency. To do this we must have some way of measuring the run time of an algorithm. One way to determine run time would be to write a program in some computer language to implement the algorithm, run the program on some computer with some input data, and observe the amount of time consumed. There are two serious problems with this approach.

1. Are we measuring the speed of the algorithm, or the speed of the of the particular computer, or perhaps the speed of the particular computer language? In fact we are measuring a mixture of all these things.

2. We should expect to get different results for different sets of input data representing different instances of the problem. How should these results be combined to give a meaningful measure of an algorithm's efficiency?

To deal with (1) we seek a measure of run time which is independent of any particular computing machine or computer language. We measure run time not by counting seconds, but by counting the **number of instructions executed**. However, not all instructions should be counted equally.

**Example** Each of the following instructions are fundamentally different.

```
if a < b
    a = b
else
    print c
```

To deal with (2) we need a measure of run time that tells us something about all possible inputs of a given size. We consider three such measures.

**Worst Case** is the maximum time taken over all inputs of a given size
**Best Case** is the minimum time taken over all inputs of a given size
**Average Case** is the average time consumed over all inputs of a given size

Recall the Sequential Search algorithm from chapter 2:

Input: $n \geq 1$ (the number of numbers in the list), $a_1, \cdots, a_n$ (the list of numbers itself), and target (the number to search for.)
Output: The first index *i* such that target = $a_i$, or 0 if no such index exists.

Sequential Search

```
1.) Get n, a_1, ..., a_n, target from the user
2.) i = 1
3.) found = false
4.) while i ≤ n and not found
5.)     if a_i = target
6.)         found = true
```

7.) else
8.) $i = i + 1$
9.) if **not** found
10.) $i = 0$
11.) print $i$
12.) stop

To measure the run time of Sequential Search, we will take our unit of work to be the comparison of *target* to a number $a_i$ in the list. Thus we count the number of times line 5 is executed. The other instructions will be considered peripheral tasks and will not be counted. Why should we do this? Notice that steps 7 and 12 don't really do anything. Also steps 1, 2, 3, 9, 10, and 11 are executed just once or (in the case of 10) at most once, hence their contribution to the total cost is negligible. Steps 4, 6, and 8 are executed (approximately) the same number of times as step 5. It might make sense therefore, to count the number of target comparisons, then multiply by 4, or some other factor depending on the relative cost of steps 4, 5, 6, and 8. Ultimately it does not matter what factor we use. *What matters most is not the actual cost of the operations performed, but the way that number scales up with n – the size of the input*. (More on this point later.) Thus we will count the number of target comparisons (step5) performed in best, average, and worst cases, on lists of length *n*.

The best case clearly occurs when the target is the first element in the list, in which case only 1 target comparison is performed. The worst case occurs when either the target is not in the list, or when the target is the last element in the list. In this case *n* comparisons are performed. To analyze the average case, we assume, for the sake of definiteness, that the target is both in the list, and is equally likely to found at any position in the list. Thus the average case breaks into *n* (equally likely) subcases. If $\text{target} = a_1$, then 1 comparison is performed; if $\text{target} = a_2$, then 2 comparisons are performed; if $\text{target} = a_3$, then 3 comparisons are performed; ... ; and if $\text{target} = a_n$, then *n* comparisons are performed. Thus the average number of comparisons (under our simplifying assumptions) is, applying Gauss' formula:

$$\frac{1+2+3+\cdots+n}{n} = \frac{\left(\frac{n(n+1)}{2}\right)}{n} = \frac{n+1}{2} = \frac{1}{2}n + \frac{1}{2}$$

Summarizing, we find that:

| | # Target Comparisons |
|---|---|
| Best Case: | 1 |
| Average Case: | $n$ |
| Worst Case | $\frac{1}{2}n + \frac{1}{2}$ |

**Exercise** Find the average number of comparisons performed by Sequential Search on lists of length *n* when the possibility that *target* is not in the list is allowed. Assume that *target* is equally likely to be in the list as not, and when it is in the list, it is equally likely to be in any position in the list. (**Answer:** $\frac{3n+1}{4}$).

## Sorting

The problem of sorting a list of numbers is much studied in Computer Science, and there are many algorithms that solve this problem. We will study several sorting algorithms in order to gain further experience in the use of pseudo-code and algorithm design techniques, and also to illustrate the notion *time efficiency*. In what follows we will say that a list of numbers is 'sorted' if it's elements are arranged in increasing order. Throughout we use $n$ to denote the length of the list.

*Selection Sort* is a simple algorithm that works by dividing the list into two sections: a sorted section on the right, and an unsorted section on the left. Initially the sorted section is empty and the unsorted section constitutes the entire list. A marker defines the boundary between the two sections. Initially this boundary marker is placed to the right of the rightmost element in the list. We repeatedly locate the maximum element in the unsorted section, exchange it with the rightmost element in the unsorted section, then move the marker one step to the left, thus enlarging the sorted section by one element and shrinking the unsorted section. We halt when the unsorted section contains just 1 element, and the sorted section contains the $n-1$ largest elements in the list, arranged in increasing order. At this point the full list is sorted.

**Example** $n=6$. We illustrate on the following list: 7 9 5 0 4 3. We use the character '*' to stand for the boundary marker between the two sections. We encode this marker by keeping track of the index $R$ of the rightmost element in the unsorted section.

| | $R$ |
|---|---|
| 7 9 5 0 4 3 * | 6 |
| 7 3 5 0 4 * 9 | 5 |
| 4 3 5 0 * 7 9 | 4 |
| 4 3 0 * 5 7 9 | 3 |
| 0 3 * 4 5 7 9 | 2 |
| 0 * 3 4 5 7 9 | 1 |

The list is now sorted, as claimed. Notice that in this example, there is no change between the next-to-last line and the last line, other than to move the marker to the left. This is because in the next-to-last line, the rightmost element and largest element in the unsorted section are one and the same, namely 3, so that element is swapped with itself.

**Example** $n=8$. Consider the list: 2 5 1 3 7 8 4 6

| | $R$ |
|---|---|
| 2 5 1 3 7 8 4 6 * | 8 |
| 2 5 1 3 7 6 4 * 8 | 7 |
| 2 5 1 3 4 6 * 7 8 | 6 |
| 2 5 1 3 4 * 6 7 8 | 5 |
| 2 4 1 3 * 5 6 7 8 | 4 |
| 2 3 1 * 4 5 6 7 8 | 3 |
| 2 1 * 3 4 5 6 7 8 | 2 |
| 1 * 2 3 4 5 6 7 8 | 1 |

In the following pseudo-code for Selection Sort, for the sake of brevity, we omit the get and print statements that read input and write output.

Input: $n \geq 1$ (the number of numbers in the list), $a_1, \cdots, a_n$ (the list of numbers to be sorted).
Output: The modified list arranged in increasing order.

SelectionSort
1.) $R = n$
2.) while $R \geq 2$
3.) find the index *i* of the maximum element in the unsorted section
4.) swap $a_i$ with $a_R$
5.) $R = R - 1$
6.) stop

The algorithm appears very simple indeed when expressed in this way, however it cannot be considered complete since it contains two operations which are not truly primitive, namely steps (3) and (4). Step (3) can be refined by inserting the pseudo-code for the algorithm FindLargest, applied to the unsorted sublist: $a_1, \cdots, a_R$.

3.1) $i = 1$
3.2) for $j = 2$ to $R$
3.3) if $a_j > a_i$
3.4) $i = j$

To refine step (4) we must introduce a temporary variable to facilitate the swap.

4.1) $\text{temp} = a_i$
4.2) $a_i = a_R$
4.3) $a_R = \text{temp}$

This process of algorithm design in which we begin with high level non-primitive operations, then successively refine them down to known operations is called *stepwise refinement*. The opposite approach, where we build up more and more abstract operations starting with only known primitives, is called *bottom-up design*.

**Exercise** Rewrite the pseudo-code for Selection Sort with the above refinements inserted, re-numbering the steps as needed.

**Exercise** Carefully trace the pseudo-code for Selection Sort on the previous examples and verify that it operates as expected.

To analyze the run-time of Selection Sort, we take as our basic operation the comparison of two array elements, which occurs on line 3.3 above. We should therefore count the number of array comparisons in best, worst, and average cases. Notice however that the number of comparisons performed depends only on the list length *n*, and not on the particular arrangement of list elements. Thus for this algorithm, best, worst, and average cases are all the same. This situation is not typical in Algorithm Analysis, and is an indication of the simplicity of Selection Sort.

Observe that line 3.3 is inside two loops: an outer while loop controlled by $R$; and an inner for loop controlled by $j$. Note also that 3.3 is executed exactly once on each iteration of the inner for loop.

| Outer loop | | Inner loop | | Comparisons |
|---|---|---|---|---|
| $R = n$ | $\Rightarrow$ | $2 \le j \le n$ | $\Rightarrow$ | $\#\text{comp} = n-1$ |
| $R = n-1$ | $\Rightarrow$ | $2 \le j \le n-1$ | $\Rightarrow$ | $\#\text{comp} = n-2$ |
| $R = n-2$ | $\Rightarrow$ | $2 \le j \le n-2$ | $\Rightarrow$ | $\#\text{comp} = n-3$ |
| $\vdots$ | | $\vdots$ | | $\vdots$ |
| $\vdots$ | | $\vdots$ | | $\vdots$ |
| $R = 3$ | $\Rightarrow$ | $2 \le j \le 3$ | $\Rightarrow$ | $\#\text{comp} = 2$ |
| $R = 2$ | $\Rightarrow$ | $2 \le j \le 2$ | $\Rightarrow$ | $\#\text{comp} = 1$ |

Thus the total number of comparisons (in best, worst, and average cases) is:

$$1 + 2 + 3 + \cdots\cdots + (n-2) + (n-1) \;=\; \frac{n(n-1)}{2} \;=\; \frac{1}{2}n^2 - \frac{1}{2}n$$

*Bubble Sort* is a very similar algorithm which also works by dividing the list into a sorted section on the right, and an unsorted section on the left. This algorithm repeatedly steps through the unsorted section, swapping any pair of consecutive elements that it finds to be out of order. On each sweep of the unsorted section, the largest element 'bubbles up' to the rightmost position in the unsorted section. The marker is then moved one element to the left, thus enlarging the sorted section by one.

**Example** $n = 6$. We illustrate on a list from a preceding example: 7 9 5 0 4 3. Again we use '*' to stand for the boundary marker between the two sections, and keep track of the index $R$ of the rightmost element in the unsorted section. We show the list after every swap.

| | $R$ |
|---|---|
| 7 9 5 0 4 3 * | 6 |
| 7 5 9 0 4 3 * | |
| 7 5 0 9 4 3 * | |
| 7 5 0 4 9 3 * | |
| 7 5 0 4 3 9 * | |
| 7 5 0 4 3 * 9 | 5 |
| 5 7 0 4 3 * 9 | |
| 5 0 7 4 3 * 9 | |
| 5 0 4 7 3 * 9 | |
| 5 0 4 3 7 * 9 | |
| 5 0 4 3 * 7 9 | 4 |
| 0 5 4 3 * 7 9 | |
| 0 4 5 3 * 7 9 | |
| 0 4 3 5 * 7 9 | |
| 0 4 3 * 5 7 9 | 3 |
| 0 3 4 * 5 7 9 | |
| 0 3 * 4 5 7 9 | 2 |
| 0 * 3 4 5 7 9 | 1 |

The list is now sorted. Observe in this example that first the largest element 9 'bubbles' up to the rightmost position, then the second largest element 7 does the same thing. Thus Bubble Sort accomplishes the same thing that Selection Sort does on each iteration, namely it brings the largest element in the unsorted section up to the rightmost position in that section. We can see though that Bubble Sort does more swapping in the process.

Input: $n \geq 1$ (the number of numbers in the list), $a_1, \cdots, a_n$ (the list of numbers to be sorted).
Output: The modified list arranged in increasing order.

BubbleSort
1.) $R = n$
2.) while $R \geq 2$
3.) $j = 2$
4.) while $j \leq R$
5.) if $a_j < a_{j-1}$
6.) swap $a_j \leftrightarrow a_{j-1}$
7.) $j = j + 1$
8.) $R = R - 1$
9.) stop

As before (6) should be refined to

6.1) $\text{temp} = a_j$
6.2) $a_j = a_{j-1}$
6.3) $a_{j-1} = \text{temp}$

From now on we will assume that the computing agent is provided with this pseudo-code, and so the swapping operation is now considered primitive. The runtime analysis of Bubble Sort is very similar (in fact identical) to that of Selection Sort. Again we count comparisons between list elements, which takes place on line (5).

| Outer loop | | Inner loop | | Comparisons |
|---|---|---|---|---|
| $R = n$ | $\Rightarrow$ | $2 \leq j \leq n$ | $\Rightarrow$ | $\#\text{comp} = n - 1$ |
| $R = n - 1$ | $\Rightarrow$ | $2 \leq j \leq n - 1$ | $\Rightarrow$ | $\#\text{comp} = n - 2$ |
| $R = n - 2$ | $\Rightarrow$ | $2 \leq j \leq n - 2$ | $\Rightarrow$ | $\#\text{comp} = n - 3$ |
| $\vdots$ | | $\vdots$ | | $\vdots$ |
| $\vdots$ | | $\vdots$ | | $\vdots$ |
| $R = 3$ | $\Rightarrow$ | $2 \leq j \leq 3$ | $\Rightarrow$ | $\#\text{comp} = 2$ |
| $R = 2$ | $\Rightarrow$ | $2 \leq j \leq 2$ | $\Rightarrow$ | $\#\text{comp} = 1$ |

The total number of comparisons (in best, worst, and average cases) is again:

$$1 + 2 + 3 + \cdots\cdots + (n-2) + (n-1) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Notice that both Selection Sort and Bubble Sort do the same number of basic operations in best, worst, and average cases. Thus neither one is able to take advantage of an input list that is already sorted, so as to do less work. The next sorting algorithm, called *Insertion Sort*, is more efficient in this respect. Insertion Sort also splits the list in two sections, sorted and unsorted, with the sorted section on the left, unsorted on the right. The boundary between the two is encoded by the index *L*, of the leftmost element in the unsorted section. The variable $L$ begins at 2, so that initially the sorted section has one element, then walks off the list to the right. The algorithm works by taking $a_L$, stepping through the sorted section from right to left swapping consecutive pairs as it goes, until it locates the position where $a_L$ belongs, thereby 'inserting' it into its correct position. The index *L* is then incremented to enlarge the sorted section by one.

**Example** $n = 6$. Illustrating again on the same example: 7 9 5 0 4 3, we use '*' for the boundary between sorted and unsorted section, while *L* is the leftmost index in the unsorted section. We show the list after every swap.

| | $L$ |
|---|---|
| 7 * 9 5 0 4 3 | 2 |
| 7 9 * 5 0 4 3 | 3 |
| 7 5 * 9 0 4 3 | |
| 5 7 * 9 0 4 3 | |
| 5 7 9 * 0 4 3 | 4 |
| 5 7 0 * 9 4 3 | |
| 5 0 7 * 9 4 3 | |
| 0 5 7 * 9 4 3 | |
| 0 5 7 9 * 4 3 | 5 |
| 0 5 7 4 * 9 3 | |
| 0 5 4 7 * 9 3 | |
| 0 4 5 7 * 9 3 | |
| 0 4 5 7 9 * 3 | 6 |
| 0 4 5 7 3 * 9 | |
| 0 4 5 3 7 * 9 | |
| 0 4 3 5 7 * 9 | |
| 0 3 4 5 7 * 9 | |
| 0 3 4 5 7 9 * | 7 |

Input: $n \geq 1$ (the number of numbers in the list), $a_1, \cdots, a_n$ (the list of numbers to be sorted).
Output: The modified list arranged in increasing order.

InsertionSort
1.) $L = 2$
2.) while $L \leq n$
3.) $j = L$
4.) while $j \geq 2$ and $a_j < a_{j-1}$
5.) swap $a_j \leftrightarrow a_{j-1}$
7.) $j = j - 1$
8.) $L = L + 1$
9.) stop

The run-time analysis of Insertion Sort is a little more complicated that that of Selection Sort and Bubble Sort, if only because it behaves differently in best and worst cases. We confine our attention here to best and worst, and leave the average case as an exercise.

A moment's thought shows that the best case occurs when the input list $a_1, \cdots, a_n$ is already sorted. In this case, the while loop test on line 4 is always false, since $a_j < a_{j-1}$ is false. The inner while loop (lines 4-7) body is therefore never entered, and each iteration of the outer while loop (lines 2-8) entails exactly one comparison of list elements, namely the test $a_j < a_{j-1}$ occurring in the while loop (4-7) heading. We see then that total number of comparisons performed by Insertion Sort in this case is the number of iterations of the outer while loop (2-8), which is $n-1$.

The worst case performance occurs when the list $a_1, \cdots, a_n$ is initially anti-sorted, i.e. sorted in decreasing order. In this case, the comparison $a_j < a_{j-1}$ is always true, so that the inner while loop test (4) only becomes false when $j = 1$. In this loop, $j$ ranges from $L$ down to 2, hence the number of list comparisons performed on the $L^{\text{th}}$ iteration of the outer while loop (2-8) is $L-1$. The analysis of Insertion Sort in worst case is thus similar to that of Selection Sort and Bubble Sort.

| Outer loop | | Inner loop | | Comparisons |
|---|---|---|---|---|
| $L = 2$ | $\Rightarrow$ | $2 \geq j \geq 2$ | $\Rightarrow$ | $\#\text{comp} = 1$ |
| $L = 3$ | $\Rightarrow$ | $3 \geq j \geq 2$ | $\Rightarrow$ | $\#\text{comp} = 2$ |
| $\vdots$ | | $\vdots$ | | $\vdots$ |
| $\vdots$ | | $\vdots$ | | $\vdots$ |
| $L = n-1$ | $\Rightarrow$ | $n-1 \geq j \geq 2$ | $\Rightarrow$ | $\#\text{comp} = n-2$ |
| $L = n$ | $\Rightarrow$ | $n \geq j \geq 2$ | $\Rightarrow$ | $\#\text{comp} = n-1$ |

The total number of comparisons in worst case is therefore:

$$1 + 2 + 3 + \cdots\cdots + (n-2) + (n-1) \;=\; \frac{n(n-1)}{2} \;=\; \frac{1}{2}n^2 - \frac{1}{2}n$$

**Exercise** Argue that the average case number of comparisons performed by Insertion Sort on lists of length *n* is approximately $\frac{n(n+1)}{4} = \frac{1}{4}n^2 + \frac{1}{4}n$.

**Exercise** Recall that the swap on line 5 of Insertion Sort, when refined, entails 3 separate assignments, by using a temporary variable. This swap can be replaced by a single assignment statement by first storing $a_L$ in a temporary, replacing the comparison $a_j < a_{j-1}$ in the heading of loop 4-7 by the comparison $\text{temp} < a_{j-1}$, then after the inner while is complete, overwrite $a_j$ with temp. Carry out these modifications and verify that the resulting algorithm is equivalent to the version stated above.

Summarizing our run time analysis of these three sorting algorithms, we have

| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Selection Sort | $\frac{1}{2}n^2 - \frac{1}{2}n$ | $\frac{1}{2}n^2 - \frac{1}{2}n$ | $\frac{1}{2}n^2 - \frac{1}{2}n$ |
| Bubble Sort | $\frac{1}{2}n^2 - \frac{1}{2}n$ | $\frac{1}{2}n^2 - \frac{1}{2}n$ | $\frac{1}{2}n^2 - \frac{1}{2}n$ |
| Insertion Sort | $n-1$ | $\frac{1}{4}n^2 + \frac{1}{4}n$ | $\frac{1}{2}n^2 - \frac{1}{2}n$ |

Focusing on just the worst case, all algorithms performed $\frac{1}{2}n^2 - \frac{1}{2}n$ comparisons on lists of length *n*. If we had included other peripheral operations in our analysis like swaps, assignments, or other types of comparisons such as index comparisons, our results would have been different. However, the worst case cost would still be of the form $an^2 + bn + c$ for some constants *a*, *b*, and *c*. These constants are ultimately machine dependent, since they depend on the relative costs of the various 'basic' operations. In order to obtain a truly machine independent measure of run time, we wish to disregard these machine influences and just concentrate on the '$n^2$'. This motivates the following.

Informal Definition: Given functions $f(n)$ and $g(n)$ we will say that $f(n)$ is *asymptotically equivalent* to $g(n)$, and write $f(n) = \Theta(g(n))$, if it is the case that

$$f(n) = \text{constant} \cdot g(n) + \text{ (lower order terms)}$$

where the constant is a positive real number, and the "lower order terms" constitute a function $h(n)$ which grows slower than $g(n)$ in the sense that $\frac{h(n)}{g(n)} \to 0$ as $n \to \infty$. If we divide the above equation by $g(n)$ we have $\frac{f(n)}{g(n)} = \text{constant} + \frac{h(n)}{g(n)}$, and hence

$$\frac{f(n)}{g(n)} \to \text{constant as } n \to \infty.$$

This limit could therefore serve as an alternative "informal" definition of asymptotic equivalence.

There are a number of different ways to express the notion of asymptotic equivalence verbally. The statements: $f(n)$ is *of order* $g(n)$, $f(n)$ has *asymptotic growth rate* $g(n)$, and $f(n)$ is *asymptotically equivalent* to $g(n)$ all say the same thing, i.e. that $f(n) = \Theta(g(n))$.

**Examples**

$2n^2 + 3n + 5 = \Theta(n^2)$

$5n^3 - n + 100 = \Theta(n^3)$

$3\sqrt{n} + \sqrt[3]{n} + \sqrt[4]{n} = 3n^{1/2} + n^{1/3} + n^{1/4} = \Theta(n^{1/2})$

$6n\sqrt{n} + 100n + \sqrt{n} = \Theta(n^{3/2})$

$6n\sqrt{n} + 100n^2 + \sqrt{n} = \Theta(n^2)$

$10n^{7/3} + n^{5/2} + 1 = \Theta(n^{5/2})$

Evidently, an even more informal definition of asymptotic equivalence would be the following: to obtain a simple function that is asymptotically equivalent to $f(n)$, drop any lower order terms, retain only the highest order term, and replace the coefficient of that term by 1. (Admittedly this "definition" is circular since it uses the word "order".) To see a fully rigorous (i.e. formal) definition of asymptotic equivalence, you must take CMPS 101.

Replacing the run times of our three sorting algorithms by their asymptotic growth rates, we have what are called their *asymptotic run times*. Summarizing:

| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Selection Sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Bubble Sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Insertion Sort | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |

This notion of *asymptotic* run time is the fully machine independent measure of run time which we sought.