Module 1 - Introduction

Comparing Algorithms

One of the primary goals of this course is to develop and apply analytic techniques to enable comparing two or more algorithms. The purpose of such a comparison is to help computer scientists determine the “best” algorithm for their particular problem. Typically, such a comparison focuses on one of two possible areas—time and space. In other words, from a resource optimization perspective, we are concerned with minimizing the amount of time required for an algorithm to run, as well as minimizing the space required to store the data for processing.

How should we go about comparing algorithms? In general, we may want to assess performance empirically where we implement the algorithms and run a series of experiments. This is an excellent way to assess performance on specific implementations of algorithms but can be problematic when attempting to draw general conclusions about the underlying algorithms themselves. To make such generalizations, careful experimental design is a must! A second approach is to apply formal mathematical analysis to the algorithm to determine what the complexity bounds are. Here, mathematical expressions characterizing time and space requirements are formulated and then solved.

Given the need to compare algorithms, we also need to ask appropriate questions to guide that comparison. For example, what performance bounds should be the focus of the analysis? Are we interested in the worst-case conditions for the algorithm and the associated complexity? Perhaps what we really want to know is how the algorithm can be expected to perform “on average.” But this raises issues of what we mean by “average” in that it suggests knowledge of some distribution of the data can be determined. At times we might be interested in the best case conditions of the algorithm, perhaps because such analysis tells us the theoretical limit of what we can expect.

For best and worst case analysis, we often do not care about the true, closed-form solution to the mathematical expression we devised, but we don’t want to rely on empirical assessment either. This leads to the goal of finding an “asymptotic bound” on performance where we find the complexity within some constant multiplier. In other words, we seek to find an expression that “bounds” the performance as tightly as possible without necessarily providing a detailed mathematical expression. This is because many of the constants and “low-order” terms in the expression are driven by specific implementation of the algorithms . . . something of little interest to us when attempting to determine the theoretical properties of the algorithm.

Analyzing Algorithms

A major component of this class will be developing the skills necessary to analyze algorithms in support of the above comparisons. In addition to analyzing the resource requirements (i.e., time and space) for the algorithm, we will often be concerned with determining the correctness of the algorithm. In other words, we will want to prove that the algorithm does what we think it does. Elements of such analysis include determining the conditions under which we can expect the algorithm to terminate, and verifying the extent to which we get the right answer on data provided as input.

Example - Analyzing Insertion Sort

Let’s return to our insertion sort example and provide an analysis of the time required for the algorithm to run. For convenience, we repeat the pseudo-code here, but this time we will number the lines in the algorithm.

Algorithm 1 Insertion Sort

```
1. for $j \leftarrow 2$ to $length(\mathbf{A})$ do
2.    $key \leftarrow \mathbf{A}[j]$
3.    $i \leftarrow j - 1$
4.    while $i > 0$ and $\mathbf{A}[i] > key$ do
5.       $\mathbf{A}[i+1] \leftarrow \mathbf{A}[i]$
6.       $i \leftarrow i - 1$
7.    end do
8.    $\mathbf{A}[i+1] \leftarrow key$
9. end for
```

When reading this code, we need to define some terms. Specifically, let $n = length(\mathbf{A})$, and let $t_j =$ the number of times the while loop runs, where j indicates the current position in the array $\mathbf{A}$. From this, we want to know the time required on an input of size n, and we denote that time $T(n)$.

To guide our analysis, observe that each line, by itself, takes a constant amount of time to execute. The actual time to execute will depend on implementation and specific characteristics of the underlying compiler, so let's suppose the time required for each line is c_1, c_2, c_3, c_4, c_5, c_6, and c_8 respectively. We skip line 7 and line 9 since they end blocks. From here, we need to consider how many times each line will be executed. In this case we find the following:

- Line 1: n times, because we must consider all elements except the first, and we must terminate after finishing the list.
- Line 2: $n-1$ times, because there are only $n-1$ elements to be examined.
- Line 3: $n-1$ times, for the same reason as line 2.
- Line 4: $\sum_{j=2}^{n} t_j$, because we have $n-1$ passes through the loop, and the loop itself depends on the position of the ith item in the final list.
- Line 5: $\sum_{j=2}^{n}(t_j - 1)$, because we only shift up to the required position and are bound by the length of the loop.
- Line 6: $\sum_{j=2}^{n}(t_j - 1)$, for the same reason as line 5.
- Line 8: $n-1$, for the same reason as line 2.

Now we are ready to consider all of the steps and all of the costs in the algorithm. This yields the following:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j \\ &+ c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1). \end{aligned}$$

This provides a nice mathematical expression for the cost of the equation, but the expression in this form is not very useful. Specifically, we are still left with questions such as, "What does this mean for the best case and what does it mean for the worst case?"

Examining the algorithm, we can see that the best case occurs when the input data is already sorted. In this situation, no elements of the array need to be shifted, so the inner loop is never executed. In this case, the above expression reduces to the following:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8) \\ &= an + b. \end{aligned}$$

From this, we see that the best case performance is **linear** in the input. But what happens in the worst case? In the worst, the data is in reverse order, and we have to shift **all** of the members of the array with every insertion. Thus, the inner loop is always executed, and it is always executed to the maximum length possible. In other words, for every execution of the loop, $t_j = j$, and we need to use the complete equation given above.

To solve this expression, we need to make use of a couple of identities, which will be proven later. Specifically, we need to make use of the identities:

$$\begin{aligned}\sum_{j=2}^{n} j &= \frac{n(n+1)}{2} - 1 \\ \sum_{j=2}^{n} (j-1) &= \frac{n(n-1)}{2}.\end{aligned}$$

Using these identities, the full expression reduces to

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_8(n-1 \\ &= \frac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{1}{2}(c_4 - c_5 - c_6) + c_8)n - (c_2 + c_3 + c_4 + c_8) \\ &= an^2 + bn + c\end{aligned}$$

From this, we see that the worst case performance is quadratic in the input. As we will see later, this is poor performance for a sorting algorithm.

From here, it can be expected that we will perform similar types of analyses on various algorithms. However, this example illustrates that such detailed analyses can be tedious, so we look for an alternative approach. That approach will be based on deriving asymptotic bounds on performance.

In addition, most of our future analyses will emphasize worst-case behavior since we would still want to design algorithms that perform well under adverse conditions. Empirically, our experience is that the worst case occurs frequently. Some argue that "average case" analysis would be more useful, except that a) average case requires an understanding of the distribution of the data, and b) often the average case is asymptotically equivalent to the worst case. Therefore, except for cases where the distribution can be determined, it is rare when average case analysis yields useful additional information. Even so, on occasion we will consider the average case because the performance is indeed different from the worst case.

Asymptotic Notation

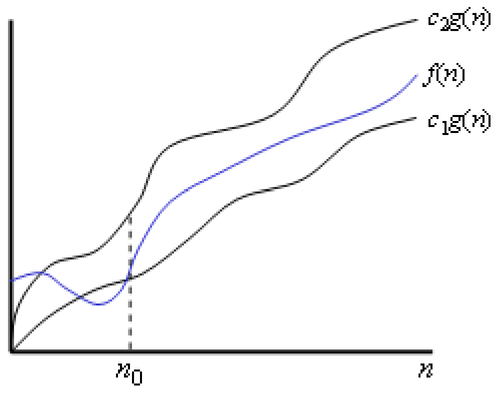
In most complexity analyses we will perform in this coure, we will make use of a notation, called asymptotic notation, to simplify our analysis. Note that in some cases we can probably derive an "exact" bound on the cost, but such analysis is often not necessary. Typically, we are only concerned with an order-of-magnitude analysis and tend to treat algorithms with the same order-of-magnitude complexity as being "equivalent." In this section, we will define the asymptotic notation we will use more precisely.

There are five forms of asymptotic notation that we will use throughout the course. This notation can be summarized as follows:

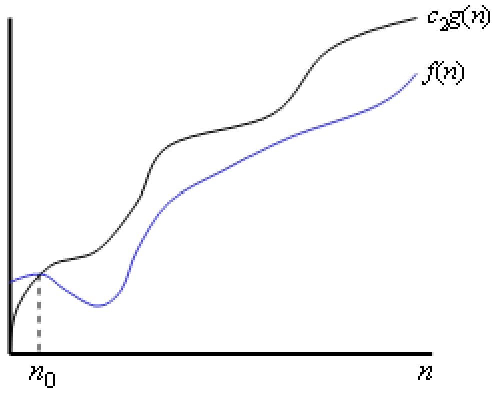
$\Theta(n)$	Provides a tight upper and lower bound on performance.
$O(n)$	Provides a tight upper bound on performance
$o(n)$	Provides a loose upper bound on performance
$\Omega(n)$	Provides a tight lower bound on performance
$\omega(n)$	Provides a loose lower bound on performance

Let's define these forms mathematically.

1. For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions $f(n)$ where there exist positive constants c_1, c_2, and n_0, such that for all $n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. Intuitively, this definition says that $f(n)$ is "sandwiched" between two multiples of the function g(n)as illustrated here.

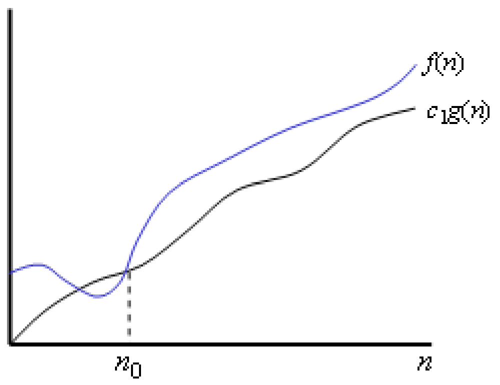


2. For a given function $g(n)$, we denote $O(g(n))$ the set of functions f(n) where there exist positive constants c and n_0 such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$. Intuitively this definition says that $f(n)$ is bound from above by some multiple of $g(n)$ and that this bound is tight (as indicated by the "$\leq$" in the definition). This is illustrated here.



3. For a given function $g(n)$, we denote $o(g(n))$ the set of functions $f(n)$ where there exist positive constants c and n_0 such that for all $n \geq n_0$, $0 \leq f(n) < cg(n)$. This definition is very similar to $O(g(n))$, except the bound is not necessarily tight (as indicated by the "$<$" in the definition).

4. For a given function $g(n)$, we denote Ω(g(n)) the set of functions $f(n)$ where there exist positive constants c and n_0 such that for all $n \geq n_0$, $0 \leq cg(n) \leq f(n)$. Intuitively, this definition says that $f(n)$ is bound from below by some multiple of $g(n)$ and that this bound is tight (as indicated by the "$\leq$" in the definition). This is illustrated here:



5. For a given function $g(n)$, we denote $\omega(g(n))$ the set of functions $f(n)$ where there exist positive constants c and n_0 such that for all $n \geq n_0$, $0 \leq cg(n) < f(n)$. This definition is very similar to $\Omega(g(n))$, except the bound is not necessarily tight (as indicated by the "$<$" in the definition).

Returning to both the insertion sort example, recall that we showed the complexity to be $T(n) = an + b$ in the best case and $T(n) = an^2 + bn + c$ in the worst case. Applying asymptotic notation, we can rewrite these bounds to indicate the complexity of insertion sort is $T(n) = \Omega(n)$ and $T(n) = O(n^2)$ respectively. Notice that, since $n \neq n^2$ in terms of the complexity bound, we cannot say that insertion sort is either $O(n)$ or $\Theta(n)$. In the next section, we will look at another algorithm—merge sort—and we will show that the complexity of merge sort is $\Theta(n \lg n)$. This suggests that, for merge sort, $T(n) = \Omega(n \lg n)$ and $T(n) = O(n \lg n)$. Whenever $O(g(n)) = \Omega(g(n))$, we can say the complexity is $\Theta(g(n))$.

Designing Algorithms

It is important to be able to analyze an algorithm to determine its performance and thereby assess the efficiency of the algorithm on a given problem. The other side of the problem is designing the algorithm in the first place. In keeping with the focus of analysis, we will explore techniques for designing algorithms that maximize efficiency, and we will back that design up with the analysis to prove the expected bounds are met. Throughout the remainder of the course, we will consider several different algorithm designs as case studies to provide examples of good algorithms. We will also consider various design methods and apply those methods in designing our own algorithms.

In this unit, we will have considered one method for algorithm design already—the incremental method. In the incremental method, we tackle a problem by considering each of the data items one at a time and process that data item until finished. We found, for the sorting procedure, this incremental approach was not efficient; however, we may find that such incremental methods are satisfactory, depending on the requirements of the problem being solved. Here, we focus on an additional design method as an example of one that frequently yields efficient solutions. That method is referred to as the divide-and-conquer method.

Fundamentally, the divide-and-conquer method is a recursive method. In other words, as a method, we construct an algorithm, that solves a problem by applying itself to smaller and smaller problems until it reaches a point where solution is trivial. The standard form of a recursive algorithm is as follows:

Algorithm 2 Recursive Functions

```
RecursiveFunction($argList$)
  if $problem$ is small enough then
    $solution \leftarrow$ SimpleSolution($argList$)
  else
    reduce problem size through division, processing, etc.
    $solution \leftarrow$ RecursiveFunction($argList$ for subproblem)
  return $solution$
```

Algorithm 3 Merge Sort

```
MergeSort($\mathbf{A}, p, r$)
  if $p < r$ then
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    MergeSort($\mathbf{A}, p, q$)
    MergeSort($\mathbf{A}, q + 1, r$)
    Merge($\mathbf{A}, p.q.r$)
```

Given this template, the divide-and-conquer design method constructs such recursive algorithms by following three steps:

1. **Divide**: Split the problem into two or more smaller subproblems.
2. **Conquer**: Solve each of the subproblems recursively (or directly if the problem is simple enough).
3. **Combine**: Take the solutions to the subproblems and combine them into a solution for the current problem.

Example - Merge Sort

Let's apply the divide-and-conquer method to sort a sequence of numbers, just as we did with insertion sort. The difference in this case, however, is that we are not going to consider the elements of the array one at a time. Instead, we will sort the array as a whole by splitting the array into smaller sorting problems and then combining the results of the sorted sub-arrays into a new sorted array. Specifically, we will take the following steps:

1. **Divide**: Start by dividing the n-element sequence into two $n/2$-element sequences each. We do this by simply splitting the sequence in half.
2. **Conquer**: Apply merge sort to each of the $n/2$-element sequences to get back two sorted sequences.
3. **Combine**: Merge the two sorted sub-sequences into a new sorted sequence to solve the problem.

We say the recursion "bottoms out" when we can no longer divide the sequence. This occurs when we have a sequence with one element in it. By definition, that one-element sequence must be sorted. This leads to the following algorithm:

For this algorithm, $\mathbf{A}$ represents the whole array, and p and r are pointers into the array to define the bounds on the sub-array to be sorted. Notice that the algorithm follows the template described previously. Specifically, there is a test to see if the data set is small enough to solve directly. In this case, when p r, the sub-array is either empty or contains one element. At that point, the algorithm, simply returns since the trivial sub-array is already sorted. Otherwise, the sub-array is split in half ($q := \lfloor (p + r)/2 \rfloor$), and a merge sort is performed on each of those sub-arrays. The results are then merged together.

The merge function operates by scanning each of the two sub-arrays in order and performing a pair-wise comparison of an element from each array. In the simplest case, an auxiliary data structure is used to hold the merged sub-array, and then the merged sub-array is copied back into the array being sorted. The merge can also be performed "in-place" by swapping elements between the two sub-arrays and keeping track on whether sorted order is violated. Details are provided in the textbook.

Divide-and-conquer algorithms suggest a different approach to analysis than we saw with the iterative insertion sort algorithm. Specifically, one can see that it is difficult to associate a cost with a recursive call such as MergeSort($\mathbf{A}$,p,q). In fact, this call hides the cost of the work being done within and must be represented using a mathematical equation called a recurrence. Let's analyze the merge sort algorithm by considering the three steps of the divide-and-conquer process.

1. **Divide**: This step simply splits the array by calculation new pointers. Therefore, the divide step can be completed in constant time, which we denote $O(1)$. (Note: Sometimes, we will denote constant time with $\Theta(1)$. These two forms are completely interchangeable in this case only.)

2. **Conquer**: We are now faced with solving two subproblems, each with size (approximately) $n/2$. There are two subproblems to be solved, so the conquer step requires $T(n/2)$ time.

3. **Combine**: The combine step uses the MERGE function, and this is linear in that it simply scans both sub-arrays. We denote this linear complexity as $\Theta(n)$.

So the complexity of merge sort is the sum of the complexities of these three steps. In other words $T(n) = O(1) + T(n/2) + \Theta(n)$. As we will see later, the $\Theta(n)$ term "dominates" the $O(1)$ term, so we can say that $O(1) + \Theta(n) = \Theta(n)$. At the bottom of the recursion, we can also observe that $T(1) = O(1)$, and this forms the "base case" for the recurrence. Now we are ready to write the full recurrence.

As we will see later, this recurrence can be solved to show that the complexity of merge sort is $T(n) = \Theta(n \lg n)$.