Algorithm Analysis

• How can we demonstrate that one algorithm is superior to another without being misled by any of the following problems:
  – Special cases
    • Every algorithm has certain inputs that allow it to perform far better than would be expected.
  – Small data sets
    • An algorithm may not display its inefficiencies if the data set on which it is run is too small.
  – Hardware differences
  – Software differences

Performance Factors

• There are many factors that influence the performance of an algorithm
  – Number of copies/comparisons made
  – Number of statements executed
  – Varying implementations on different machines
  – Variations in the input set
• Most often we will estimate the performance of an algorithm by analyzing the algorithm
Measuring Efficiency

• The efficiency of an algorithm is a measure of the amount of resources consumed in solving a problem of size n.
  – The resource we are most interested in is time
  – We can use the same techniques we will present to analyze the consumption of other resources, such as memory space.
• It would seem that the most obvious way to measure the efficiency of an algorithm is to run it and measure how much processor time is needed.

Asymptotic Analysis

• The time it takes to solve a problem is usually an increasing function of the size of the problem
  – The bigger the problem the longer it will take to solve.
• We need a formula that associates $n$, the problem size, with $t$, the time required to obtain a solution.
• We can get this type of information using asymptotic analysis to develop expressions of the form:
  – $t \approx O[f(n)]$
Formal Definition

• $t \approx O(f(n))$ if there are constants $c$ and $n_0$ such that $t \leq cf(n)$ when $n \geq n_0$

• Note that the constant is not part of the function used to describe the performance of the algorithm (i.e. there is no $O(2n)$, only $O(n)$)

• Big “O” gives the worst case performance, the average performance of the algorithm might be better.

Big Oh

• If the order of the algorithm were, for example, $O(n^3)$, then the relationships between $t$ and $n$ would be given by a formula of the form
  - $t \leq M n^3$

• Although we often do not know anything about the value of the constant $M$, we can say that
  - If the size of the problem doubles, the total time needed to solve the problem will increase about eightfold
  - If the problem size triples, the overall running time will be about 27 times greater.
Performance Classification

<table>
<thead>
<tr>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed</td>
</tr>
<tr>
<td>$\log n$</td>
<td>Logarithmic: when n increases, so does run time, but much slower. When n doubles, $\log n$ increases by a constant, but does not double until n increases to $n^2$. Common in programs which solve large problems by transforming them into smaller problems.</td>
</tr>
<tr>
<td>$n$</td>
<td>Linear: run time varies directly with n. Typically, a small amount of processing is done on each element.</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>When n doubles, runtime slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions</td>
</tr>
<tr>
<td>$n^2$</td>
<td>Quadratic: when n doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).</td>
</tr>
<tr>
<td>$n^3$</td>
<td>Cubic: when n doubles, runtime increases eightfold</td>
</tr>
<tr>
<td>$2^n$</td>
<td>Exponential: when n doubles, run time squares. This is often the result of a natural, &quot;brute force&quot; solution.</td>
</tr>
</tbody>
</table>

How To Use It

• For reasonably large problems we always want to select an algorithm of the lowest order possible.
  – If algorithm A is $O[f(n)]$ and algorithm B is $O[g(n)]$, then algorithm A is said to be of a lower order than B if $f(n) < g(n)$ for all n greater than some constant k.

• So given a choice of an algorithm that is $O(n^2)$ and an algorithm than $O(n^3)$ we would go for the $O(n^2)$ algorithm.
Algorithm Efficiencies

The Critical Section

- When analyzing an algorithm, we do not care about the behavior of each statement
  - We focus our analysis on the part of the algorithm where the greatest amount of its time is spent
- A critical section has the following characteristics:
  - It is an operation central to the functioning of the algorithm, and its behavior typifies the overall behavior of the algorithm
  - It is contained inside the most deeply nested loops of the algorithm and is executed as often as any other section of the algorithm.
The Critical Section

• The critical section can be said to be at the "heart" of the algorithm
  – We can characterize the overall efficiency of an algorithm by counting how many times this critical section is executed as a function of the problem size

• The critical section dominates the completion time
  – If an algorithm is divided into two parts, the first taking \( O(f(n)) \) followed by a second that takes \( O(g(n)) \), then the overall complexity of the algorithm is \( O(\max\{f(n), g(n)\}) \). The slowest and most time-consuming section determines the overall efficiency of the algorithm.

Analogy

• Imagine that you have the job of delivering a gift to someone living in another city thousands of miles away
• The job involves the following three steps:
  – wrapping the gift
  – driving 2,000 miles to the destination city
  – delivering the package to the proper person
• The critical section obviously step 2.
  – Changing step 2 to flying instead of driving would have a profound impact of the completion time of the task.
Sequential Search

```java
public static int search( int array[], int target ) {
    // Assume the target is not in the array
    int position = -1;
    // Step through the array until the target is found
    // or the entire array has been searched.
    for ( int i=0; i < array.length && position == -1; i=i+1 ) { // Is the current element what we are looking for?
        if ( array[ i ] == target ) {
            position = i;
        }
    }
    // Return the element's position
    return position;
}
```

$O(n)$ where $n$ is the length of the array $\rightarrow$ Linear

---

Binary Search

```java
public static int search( int array[], int target ) {
    int start = 0;           // The start of the search region
    int end = array.length; // The end of the search region
    int position = -1;       // Position of the target
    // While there's something to search and we have not found it
    while ( start <= end && position == -1 ) {
        int middle = (start + end) / 2; // Determine where the target should be
        if ( target < array[ middle ] ) { // Smaller must be in left half
            end = middle - 1;
        } else if ( target > array[ middle ] ) { // Larger must be in right half
            start = middle + 1;
        } else { // Found it!!
            position = middle;
        }
    }
    return position;
}
```

$O(\log_2 n)$ where $n$ is the length of the array $\rightarrow$ Logarithmic
Selection Sort

\( O(n^2) \) where \( n \) is the length of the array \( \rightarrow \) Quadractic

\[
\begin{align*}
\text{for ( int } i = 0; i < \text{array.length - 1; } i = i + 1 \} \{ \\
\text{ for ( int } j = i + 1; j < \text{array.length; } j = j + 1 \} \{ \\
\text{ if ( array[ } j \} < \text{array[ } i \} \} \\
\text{ // Elements out of order -- swap } \\
\text{ int tmp = array[ } i \}; \\
\text{ array[ } i \} = \text{array[ } j \}; \\
\text{ array[ } j \} = \text{tmp; }
\}
\}
\]

Bubble Sort

\( O(n^2) \) where \( n \) is the length of the array \( \rightarrow \) Quadractic

\[
\begin{align*}
\text{public static void bubbleSort( int array[]) } \{ \\
\text{ int i = 0; } \quad \text{// How many elements are sorted - initially none } \\
\text{ boolean swap; } \quad \text{// Was a swap made during this pass? } \\
\text{ // Keep making passes through the array until it is sorted } \\
\text{ do } \{ \\
\text{ swap = false; } \\
\text{ // Swap adjacent elements that are out of order } \\
\text{ for ( int } j = 0; j < \text{array.length - i - 1; } j++ \} \{ \\
\text{ // If the two elements are out of order - swap them } \\
\text{ if ( array[ } j \} > \text{array[ } j + 1 \} } \\
\text{ int tmp = array[ } j \}; \\
\text{ array[ } j \} = \text{array[ } j + 1 \}; \\
\text{ array[ } j + 1 \} = \text{tmp; } \\
\text{ swap = true; } \quad \text{// Made a swap - array might not be sorted } \\
\}
\}
\]

9/10/2003 Algorithm Analysis
Insertion Sort

\[ O(n^2) \text{ where } n \text{ is the length of the array } \rightarrow \text{ Quadratic} \]

```java
public static void insertionSort(int[] array)
{
    // i marks the beginning of the unsorted array
    for (int i = 1; i < array.length; i = i + 1)
    {
        // Insert the first element in the unsorted portion of
        // the array into the sorted portion. Works from the
        // bottom of the sorted portion to the top
        for (int j = i - 1; j >= 0 && array[j + 1] < array[j]; j = j + 1)
        {
            // Since we have not found the correct position for the
            // new element - move the elements in the sorted
            // portion down one
            int tmp = array[j];
            array[j] = array[j + 1];
            array[j + 1] = tmp;
        }
    }
}
```