Algorithm efficiency can be measured in terms of:

- Time
- Space
- Other resources such as processors, network packets, etc.

Algorithms analysis tends to focus on time:

- Techniques for measuring all resources are basically the same
- Historically, time dominated
The efficiency of a program or system is affected by:

- Programming Language
- Physical characteristics of the computer
- Room temperature
- Amount of data to process (or actual parameters)
- Algorithms used

In the real world all factors are considered, and any one can dominate in any particular situation.

When developing algorithms in the abstract, most such factors are ignored because they are:

- Out of our control, as programmers and algorithm developers.
- Difficult to predict.
- For that reason we say they are arbitrary.

Consequently, we are interested in algorithm efficiency, not program or system efficiency.
The efficiency of an algorithm is specified as a **running time**, sometimes also called **complexity**.

In our abstract world, the **input length** or parameters have the most influence on running time of an algorithm.

The running time of an algorithm measures:

- Total number of *operations* executed by an algorithm, or
- Total number of *statements* executed by an algorithm

Running times can be based on:

- Worst case (most frequently used)
- Average case (most difficult)
- Best case
Example – Inputting $n$ integers:

```java
public static void inputInts(int n) {
    int i;

    i = 1;
    while (i <= n) {
        x = kb.nextInt();
        i = i + 1;
    }
}
```

Total number of statements executed is approximately $T(n) = 3n+2$

Running time is therefore $O(n)$

How many operations are executed by the above algorithm?
Another Example:

```java
public static void inputInts(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println(i);
    }
    for (int i = 0; i < 2*n; i++) {
        System.out.println(2*n-i+1);
    }
}
```

Total number of statements executed is approximately $T(n) = 9n+4$

Running time is therefore $O(n)$
Another Example:

```java
public static void inputInts(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println(i);
            System.out.println(i+j);
        }
        System.out.println(j);
    }
    System.out.println("All done!");
}
```

Total number of statements executed is approximately $T(n) = 4n^2 + 5n + 2$

Running time is therefore $O(n^2)$
Another Example:

```java
public static void inputInts(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                System.out.println(i);
                System.out.println(j);
                System.out.println(k);
            }
            System.out.println(“All done inner loop!”);
        }
        System.out.println(“All done middle loop!”);
    }
    System.out.println(“All done outer loop!”);
}
```

Total number of statements executed is approximately … *good luck with that!!!*

Running time is therefore $O(n^3)$
Another Example:

```java
public static void inputInts(int n) {
    int x, y;

    x = n;
    y = 10 * n;
    x = x + y;
    System.out.println(x);
}
```

Total number of statements executed is $T(n) = 4$

Running time is therefore $O(1)$
“Big Oh” Notation

- Note that it is tempting to think that $O(f(n))$ means “approximately $f(n)$.”

- Although used in this manner frequently, this interpretation or use of “big Oh” is absolutely incorrect.

- Most frequently, $O(f(n))$ is pronounced “on the order of $f(n)$” or “order $f(n)$,” but keep in mind that this also does not mean “approximately $f(n)$.”
Definitions:

Let $f(n)$ and $g(n)$ be functions:
- $f(n)$ is said to be less than $g(n)$ if $f(n) \leq g(n)$ for all $n$.

For example, $n^2$ is less than $n^4 + 1$. 
To within a constant factor, \( f(n) \) is said to be less than \( g(n) \) if there exists a positive constant \( c \) such that \( f(n) \leq cg(n) \) for all \( n \).
Example:

\[ g(n) = 3n^2 + 2 \]
\[ f(n) = 6n^2 + 3 \]

Note:

6\(n^2 + 3\) is not less than 3\(n^2 + 2\)

However:

To within a constant factor 6\(n^2 + 3\) is less than 3\(n^2 + 2\)

Proof:

Let \(c=9\), then we see that:

\[ 6n^2 + 3 <= 9(3n^2 + 2) \]
\[ = 27n^2 + 18 \]
By the way, for these two functions it is also the case that:

To within a constant factor $3n^2 + 2$ is less than $6n^2 + 3$

**Proof:**
Let $c=1$, then we see that:

$$3n^2 + 2 \leq 1(6n^2 + 3)$$
$$= 6n^2 + 3$$

In fact $3n^2 + 2$ is actually less than $6n^2 + 3$

In other words, *asymptotically*, there ain’t much difference in the growth of the two functions, as $n$ goes to infinite.

(always enjoy using that word in class)
Question:

\[ g(n) = n^2 \]
\[ f(n) = 2^n \]

Is \( f(n) \), to within a constant factor, less than \( g(n) \)?

In other words, is there a constant \( c \) such that:

\[ f(n) \leq cg(n) \]

No! Not even if we let \( c = 1,000,000, \) or larger!
Definition: $f(n)$ is said to be $O(g(n))$ if there exists two positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Intuitively, we say that, as $n$ goes to infinity, $f(n)$ is, to within a constant factor, less than $g(n)$.

Stated another way, as $n$ goes to infinity, $f(n)$ is, to within a constant factor, bounded from above by $g(n)$.
- From the definition of big-O, it should now be clear that saying $T(n)$ is $O(g(n))$ means that $g(n)$ is an *upper bound* on $T(n)$, and does not mean approximately.

- An analogy:
  - Consider two integers $x$ and $y$, where $x \leq y$.
  - Is $y$ approximately equal to $x$?

  Of course not!

- Similarly for the definition of big-O.
Example:

\[ g(n) = n^2 \]
\[ f(n) = n^2 + 1 \]

Claim:

\[ n^2 + 1 \text{ is } O(n^2) \]

Proof:

Let \( c = 2 \) and \( n_0 = 1 \), then we see that:

\[ n^2 + 1 \leq 2n^2 \text{ for all } n \geq n_0 = 1. \]

*Note that in this case \( f(n) \) is not less than \( g(n) \), even to within a constant.*
The original procedure is still correct (choose the highest order term, drop the constant).

More specifically, this procedure is consistent with the formal definition of big-O.

\[ f(n) = n^2 + 1 \quad f(n) \text{ is } O(n^2) \]
\[ g(n) = n + 6 \quad g(n) \text{ is } O(n) \]
\[ h(n) = n^{35} + n^{16} + n^4 + 3n^2 + 1025 \quad h(n) \text{ is } O(n^{35}) \]
\[ r(n) = (1/2)n + 300 \quad r(n) \text{ is } O(n) \]
\[ s(n) = 4n^2 + 2n + 25 \quad s(n) \text{ is } O(n^2) \]

And yes, the depth of loop nesting, at least on simple programs, does indicate the running time.
More generally:

If: \[ f(n) = a_{m-1}n^{m-1} + a_{m-2}n^{m-2} + \ldots + a_1n^1 + a_0 \]

Then: \[ f(n) \text{ is } O(n^{m-1}). \]

Explanation:

Let: \[ c = |a_{m-1}| + |a_{m-2}| + \ldots |a_1| + |a_0| \]
\[ n_0 = 1 \]

Then it will be the case that:

\[ f(n) <= cn^{m-1} \]
\[ \text{for all } n >= n_0 \]
Why is $f(n) \leq cn^{m-1}$ for all $n \geq n_0 = 1$?

$$cn^{m-1} = |a_{m-1}|n^{m-1} + |a_{m-2}|n^{m-1} + \ldots + |a_1|n^{m-1} + |a_0|n^{m-1}$$

$$f(n) = a_{m-1}n^{m-1} + a_{m-2}n^{m-2} + \ldots + a_1n + a_0$$
Another thing worth noting:

If, for example, \( f(n) = 5n^3 + 35n^2 + 3n + 1025 \)

Then \( f(n) \) is \( O(n^3) \), but also \( O(n^4) \), \( O(n^5) \), etc.

This may seem counter-intuitive, but it is exactly what we want big-oh to be, i.e., an upper bound.
### Common Running Times:

- **Polynomial**
  - $O(1)$: constant
  - $O(\log n)$: logarithmic
  - $O(n)$: linear
  - $O(n \log n)$
  - $O(n^2)$: quadratic
  - $O(n^2 \log n)$
  - $O(n^3)$
  - $O(n^4)$

- **Exponential**
  - $O(2^n)$
  - $O(3^n)$
Let $n$ be the length of the array $A$. 

What is the (worst case) running time of sequential/linear search?  
\[ \Rightarrow \text{What are the worst case scenario?} \]
\[ \Rightarrow \text{We will focus on the “==“ operation (not that its special, or anything...)} \]

```java
public boolean inList(int x; int[] A)  
{  
    int i;  
    i=0;  // notice for -> while  
    while (i<=A.length-1) {  
        if (x == A[i])  
            return true;  
        i++;  
    }  
    return false;  
}  
```
What is the (worst case) running time of selection sort?

⇒ What are the worst case scenario?
⇒ Again, we will focus on the “<“ operator.

// Selection sort
public static void sort(int[] a) {
    int i, j, minPos, temp;

    for (i=0; i<a.length-1; i++) {
        // Find the position of the value that belongs in position i
        minPos = i;
        for (j=i+1; j<=a.length-1; j++)
            if (a[j] < a[minPos])
                minPos = j;

        // Swap the values in positions i and min
        temp = a[i];
        a[i] = a[minPos];
        a[minPos] = temp;
    }
}
Consider the number of comparisons made in the *worst case*:

- $n - 1$ iterations performed by the outer loop
- First iteration: how many comparisons?
- Second iteration: how many comparisons?

\[
(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N-1)/2 = (N^2 - N)/2
\]

Which is $O(n^2)$

*Question:*  
- What about the other operations – shouldn’t they be counted also?  
- We could count those also, but the end result would be the same, at least *asymptotically* it would.
Other Questions:

- Should we distinguish between different types of operations, such as comparisons vs. assignments?
- In some cases, yes, but here the distinction is not important; all operations are in main memory, and operate on a small, fixed number of bytes.
- File I/O or network message operations would be more time consuming.
- What about best and average cases?

Best case:

- The assignment to \texttt{minPos} never takes place, would appear more efficient, but the \texttt{<} operator is still executed the same number of times.
- Consequently, the end result is the same - $O(n^2)$

Average case:

- Same thing - $O(n^2)$
What is the (worst case) running time of insertion sort?

```java
public static void insertionSort(int[] a) {
    int j;

    for (int i=1; i<=a.length-1; i++) {
        j=i;
        v = a[j];
        while ((j>0) && (v<a[j-1])) {
            a[j]=a[j-1];
            j=j-1;
        }
        a[j] = v;
    }
}
```
Consider the number of times the condition in the while-loop is evaluated in the worst case:

- \( N - 1 \) iterations performed by the outer loop.
- First iteration: how many evaluations?
- Second iteration: how many evaluations?

: 

- \( 2 + \ldots + (N - 2) + (N - 1) + N = \)
- \( (1 + 2 + \ldots + N) - 1 = \)
- \( N(N+1)/2 - 1 = \)
- \( (1/2)N^2 + (1/2)N - 1 \)

Which is \( O(n^2) \)

- How about best case?
- How about average case?
What is the worst-case running time of bubble sort?

```java
public static void bubbleSort1(int[] a)
{
    int temp;

    for (int i=1; i<a.length; i++) {
        for (int j=0; j<a.length-i; j++) {
            if (a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

Question: Is there a distinction between worst, best and average cases?
Consider the number of comparisons made by the if-statement in the \textit{worst case}:

- $n - 1$ iterations performed by the outer loop
- First iteration: how many comparisons?
- Second iteration: how many comparisons?

\[
(N - 1) + (N - 2) + \ldots + 2 + 1 = \frac{N(N-1)}{2} = \frac{N^2 - N}{2}
\]

Which is $O(n^2)$

What about the other versions of bubble sort?
Second version: (fewer bubbles)

// This version stops when a pass occurs with no swaps.
public static void bubbleSort1(int[] a) {
    int i, temp;
    boolean doMore;

    i = 1;
    doMore = true;
    while ((i<a.length) && (doMore)) {
        doMore = false;
        for (int j=0; j<a.length-i; j++)
            if (a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                doMore = true;
            }
        i = i + 1;
    }
}
Analysis of Binary Search

- Recall binary search:

- Suppose we are searching for:
  - 45
  - 23
  - 29
  - 31
Summary of binary search:

- At each step there is a segment of the array being searched, which is indicated by a “low” position and a “high” position.
- Look at the middle element between the “low” and “high.”
- If the middle element is the one you are looking for, stop.
- If not, repeat the search on the upper or lower half of the current segment of the array being searched.

See the program at:

- [http://cs.fit.edu/~pbernhar/teaching/cse1001/binarySearch.txt](http://cs.fit.edu/~pbernhar/teaching/cse1001/binarySearch.txt)
Logarithms

“Fear not the logarithm, for it is your friend!”

- The *logarithm* of a number to a given base is the exponent to which the base must be raised in order to produce that number.

- In other words, \( \log_a(b) \) is the power to which “\( a \)” must be raised in order to produce “\( b \).”

- What is:

  \[
  \begin{align*}
  \log_{10}(1000) &= \ ? \\
  \log_3(9) &= \ ? \\
  \log_5(1) &= \ ? \\
  \log_2(16) &= \ ? \\
  \log_2(32) &= \ ? \\
  \log_2(1024) &= \ ?
  \end{align*}
  \]
Of particular interest in computer science are base-2 logarithms.

\[
\begin{align*}
\log_2(1) &= 0 & \log_2(64) &= 6 \\
\log_2(2) &= 1 & \log_2(128) &= 7 \\
\log_2(4) &= 2 & \log_2(256) &= 8 \\
\log_2(8) &= 3 & \log_2(512) &= 9 \\
\log_2(16) &= 4 & \log_2(1024) &= 10 \\
\log_2(32) &= 5 & \log_2(2048) &= 11 \\
\end{align*}
\]

Notice that the logarithm is the inverse of exponentiation.

\[\log_2(2^k) = k\]

The logarithm function is therefore a very slowly growing function.

- Exponentiation is a very fast growing function.
- See the wikipedia page on logarithms for the graph.
As stated previously, $\log_2(n)$ is the power to which 2 must be raised in order to produce “$n$.”

Another way to state this is:

“$\log_2(n)$ is the number of times you can divide $n$ by 2 and still get a number $>= 1$.”

$log_2(8) = 3$  
$log_2(16) = 4$  
$log_2(32) = 5$  
$log_2(2^k) = k$

$log_2(9) = 3$  
$log_2(30) = 4$  
$log_2(2000) = 10$

Why is this important? Recall binary search…

➤ [http://cs.fit.edu/~pbernhar/teaching/cse1001/binarySearch.txt](http://cs.fit.edu/~pbernhar/teaching/cse1001/binarySearch.txt)
Running Time of Binary Search

- Each iteration of the main loop, performs one comparison, and divides the array segment to be searched in half.
  - At the start of the first iteration, the segment has length n.
  - At the start of the second iteration, the segment has length n/2.
  - At the start of the second iteration, the segment has length n/4.
  - ...
- In the worst-case, the loop will continue until the array has length < 1.

- How many times does the loop iterate, i.e., how many times can the length of the array be reduced by ½ and still result in an array of length >= 1?
  - \( \log_2(n) \)

- Since each iteration performs a constant number of operations, the running time of binary search is therefore \( O(\log n) \)