LEC02
PROGRAM ANALYSIS (40-51)

Lecture slide courtesy of Prof. Steven Skiena
2-4. [8] What value is returned by the following function? Express your answer as a function of $n$. Give the worst-case running time using Big Oh notation.

```plaintext
function conundrum(n)
    r := 0
    for i := 1 to n do
        for j := i + 1 to n do
            for k := i + j - 1 to n do
                r := r + 1
        return(r)
```
r<-1:10
r^3
1 8 27 64 125 216 343 512 729 1000
REASONING ABOUT EFFICIENCY: SELECTION SORT

```java
selection_sort(int s[], int n)
{
    int i,j;    /* counters */
    int min;    /* index of minimum */
    for (i=0; i<n; i++) {
        min=i;
        for (j=i+1; j<n; j++)
            if (s[j] < s[min]) min=j;
        swap(&s[i], &s[min]);
    }
}
```

```java
void selectionSort(int[] array, int startIndex)
{
    if (startIndex >= array.length - 1) return;
    int minIndex = startIndex;
    for (int index = startIndex + 1; index < array.length; index++)
    {
        if (array[index] < array[minIndex])
            minIndex = index;
    }
    int temp = array[startIndex];
    array[startIndex] = array[minIndex];
    array[minIndex] = temp;
    selectionSort(array, startIndex + 1);
}
```
The outer loop goes around $n$ times.

The inner loop goes around at most $n$ times for each iteration of the outer loop.

Thus selection sort takes at most $n \times n \rightarrow O(n^2)$ time in the worst case.

```c
selection_sort(int s[], int n)
{
    int i, j; /* counters */
    int min; /* index of minimum */

    for (i=0; i<n; i++)
    {
        min=i;
        for (j=i+1; j<n; j++)
            if (s[j] < s[min]) min=j;
        swap(&s[i], &s[min]);
    }
}
```
MORE CAREFUL ANALYSIS

- An exact count of the number of times the *if* statement is executed is given by:

\[
S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1
\]

\[
S(n) = (n - 2) + (n - 3) + \ldots + 2 + 1 + 0 = (n - 1)(n - 2)/2
\]

- Thus the worst case running time is \( \Theta(n^2) \).

- Can we say \( O(n^2) \) or \( \Omega(n^2) \) ?
REASONING ABOUT EFFICIENCY: INSERTION SORT

```java
for (i=1; i<n; i++) {
    j=i;
    while ((j>0) && (s[j] < s[j-1])) {
        swap(&s[j],&s[j-1]);
        j = j-1;
    }
}

public static int[] RecursiveInsertionSort(int[] array, int n) {
    int i;
    if (n > 1)
        RecursiveInsertionSort(array, n - 1);
    else {
        int k = array[n];
        i = n - 1;
        while (i >= 0 && array[i] > k) {
            array[i + 1] = array[i];
            i = i - 1;
        }
        array[i + 1] = k;
    }
    return array;
}
```
How often does the inner \textit{while} loop iterate?

- two different stopping conditions:
  - One to prevent us from running off the bounds of the array \((j > 0)\)
  - One to mark when the element finds its proper place in sorted order \((s[j] < s[j-1])\).

Since \textbf{worst-case analysis} seeks an upper bound on the running time, we ignore the early termination and assume that this loop always goes around \(i\) times.

Insertion sort must be a quadratic-time algorithm, i.e. \(O(n^2)\).
**Problem:** Substring Pattern Matching

*Input:* A text string \(t\) and a pattern string \(p\).

*Output:* Does \(t\) contain the pattern \(p\) as a substring, and if so where?

```c
int findmatch(char *p, char *t)
{
    int i,j;
    int m, n;
    m = strlen(p);
    n = strlen(t);
    for (i=0; i<=(n-m); i=i+1) {
        j=0;
        while ((j<m) && (t[i+j]==p[j]))
            j = j+1;
        if (j == m) return(i);
    }
    return(-1);
}
```

Searching for the substring *abba* in the text *aababba*. 
What is the worst-case running time of these two nested loops?

```
int findmatch(char *p, char *t)
{
    int i,j;    /* counters */
    int m, n;    /* string lengths */

    m = strlen(p);
    n = strlen(t);

    for (i=0; i<=(n-m); i=i+1) {
        j=0;
        while ((j<m) && (t[i+j]==p[j]))
            j = j+1;
        if (j == m) return(i);
    }
    return(-1);
}
```

$\Theta(n)$  
$\Theta(m)$  
$\Theta(n - m)$  
$O(m + 2)$  
$O(n + m + (n - m)(m + 2))$  

$n \geq m, n \leq nm,$  

$O(nm)$
Problem: Matrix Multiplication

Input: Two matrices, $A$ (of dim. $x \times y$) and $B$ (dim. $y \times z$).

Output: An $x \times z$ matrix $C$ where $C[i][j]$ is the dot product of the $i$th row of $A$ and the $j$th column of $B$.

```java
for (i=1; i<=x; i++)
    for (j=1; j<=y; j++) {
        C[i][j] = 0;
        for (k=1; k<=z; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```
WORST CASE ANALYSIS

The number of multiplications $M(x, y, z)$ is given by the following summation:

$$M(x, y, z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1$$

Sums get evaluated from the right inward. The sum of $z$ ones is $z$, so

$$M(x, y, z) = \sum_{i=1}^{x} \sum_{j=1}^{y} z$$

The sum of $y$ $z$s is just as simple, $yz$, so

$$M(x, y, z) = \sum_{i=1}^{x} yz$$

Finally, the sum of $x$ $yz$s is $xyz$.

Thus the running of this matrix multiplication algorithm is $O(xyz)$.

If we consider the common case where all three dimensions are the same, this becomes $O(n^3)$—i.e., a cubic algorithm.
Logarithms

- It is important to understand deep in your bones what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.
- Logarithms reflect how many times we can double something until we get to $n$, or halve something until we get to 1.
- Logarithms arise in any process where things are repeatedly halved.
**In binary search we throw away half the possible number of keys after each comparison. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!**

- How many time can we halve \( n \) before getting to 1?
- **Answer:** \( \lceil \log_2 n \rceil \)

A height \( h \) tree with \( d \) children per node as \( d^h \) leaves.

Here \( h = 2 \) and \( d = 3 \).
LOGARITHMS AND TREES

- How tall a binary tree do we need until we have n leaves? The number of potential leaves doubles with each level.
- How many times can we double 1 until we get to n?
- Answer: ceiling(lgn)
A binary tree of height 1 can have up to 2 leaf nodes
What is the height $h$ of a rooted binary tree with $n$ leaf nodes?
• Note that the number of leaves doubles every time we increase the height by one.
• To account for $n$ leaves, $n = 2^h$ which implies that $h = \log_2 n$.

What if we generalize to trees that have $d$ children, where $d = 2$ for the case of binary trees?
• A tree of height 1 can have up to $d$ leaf nodes, while one of height two can have up to $d^2$ leaves.

The number of possible leaves multiplies by $d$ every time we increase the height by one, so to account for $n$ leaves, $n = d^h$ which implies that $h = \log_d n$,
How many bits do you need to represent the numbers from 0 to $2^i - 1$?

Each bit you add doubles the possible number of bit patterns,

so the number of bits equals $\lg(2^i) = i$
Recall that

\[ \log_a(xy) = \log_a(x) + \log_a(y) \]

This is how people used to multiply before calculators, and remains useful for analysis.

What if \( x = a \)?

\[ \log_a n^b = b \cdot \log_a n \]
Recall the definition, $c^{\log_c x} = x$ and that

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Thus $\log_2 n = \frac{1}{\log_{100} 2} \times \log_{100} n$. Since $\frac{1}{\log_{100} 2} = 6.643$ is just a constant, it does not matter in the Asymptotic notations.
2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.

- (a) Base offense Level: 6
- (b) Specific offense Characteristics

(1) If the loss exceeded $2,000, increase the offense level as follows:

<table>
<thead>
<tr>
<th>Loss (Apply the Greatest)</th>
<th>Increase in Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) $2,000 or less</td>
<td>no increase</td>
</tr>
<tr>
<td>(B) More than $2,000</td>
<td>add 1</td>
</tr>
<tr>
<td>(C) More than $5,000</td>
<td>add 2</td>
</tr>
<tr>
<td>(D) More than $10,000</td>
<td>add 3</td>
</tr>
<tr>
<td>(E) More than $20,000</td>
<td>add 4</td>
</tr>
<tr>
<td>(F) More than $40,000</td>
<td>add 5</td>
</tr>
<tr>
<td>(G) More than $70,000</td>
<td>add 6</td>
</tr>
<tr>
<td>(H) More than $120,000</td>
<td>add 7</td>
</tr>
<tr>
<td>(I) More than $200,000</td>
<td>add 8</td>
</tr>
<tr>
<td>(J) More than $350,000</td>
<td>add 9</td>
</tr>
<tr>
<td>(K) More than $500,000</td>
<td>add 10</td>
</tr>
<tr>
<td>(L) More than $800,000</td>
<td>add 11</td>
</tr>
<tr>
<td>(M) More than $1,500,000</td>
<td>add 12</td>
</tr>
<tr>
<td>(N) More than $2,500,000</td>
<td>add 13</td>
</tr>
<tr>
<td>(O) More than $5,000,000</td>
<td>add 14</td>
</tr>
<tr>
<td>(P) More than $10,000,000</td>
<td>add 15</td>
</tr>
<tr>
<td>(Q) More than $20,000,000</td>
<td>add 16</td>
</tr>
<tr>
<td>(R) More than $40,000,000</td>
<td>add 17</td>
</tr>
<tr>
<td>(Q) More than $80,000,000</td>
<td>add 18</td>
</tr>
</tbody>
</table>
The increase in punishment level grows logarithmically in the amount of money stolen.

Thus it pays to commit one big crime rather than many small crimes totaling the same amount.