Program Performance

- **Performance**: Amount of memory and time to run a program.

- **Space Complexity**: amount of memory needed to run a program. Why important?
  - Running in multiuser environment
  - Is there enough memory?
  - Smaller programs can be run with other programs
  - Estimate the largest program we can run

- **Time Complexity**: Amount of time needed to run a program. Why important?
  - May need to provide a time limit.
  - May need to provide a real time response
  - Use appropriate program when several alternatives exist.
template<class T>
int SequentialSearch(T a[], const T& x, int n)
{
    // Search the unordered list a[0:n-1] for x.
    // Return position if found; return -1 otherwise.
    int i;
    for (i = 0; i < n && a[i] != x; i++);
    if (i == n) return -1;
    return i;
}

- total number of steps executed by SequentialSearch depends on the input.
  - Worst case: loop executed $n$ times
  - Best case: loop executed zero times
  - Average case: loop executed $\frac{n}{2}$ times (for successful search assuming ...)
Step Count

- Program Step: (loosely defined) a syntactically or semantically meaningful segment of a program for which the execution time is independent of the instance characteristics. (e.g. $a+b*c+d*r$)

- Initially set count to zero and each time a program step is executed count is increased.
\[x = x + 1;\]
for \((i = 1; i <= n; i = i + 1)\)
\[
x = x + 1;
\]
for \((i = 1; i <= n; i = i + 1)\)
\[
\text{for}(j = 1; j <= i; j = j + 1)\]
\[
x = x + 1;
\]
for \((i = 1; i <= n; i = i + 1)\)
\[
\text{for}(j = 1; j <= i; j = j + 1)\]
\[
\text{for}(k = 1; k <= j; k = k + 1)\]
\[
x = x + 1;\]

1 unit
\[
\sum_{i=1}^{n} 1 = n \quad \text{units}
\]
\[
\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\]
\[
\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1
= \sum_{i=1}^{n} \sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2}
= c_1 n^3 + c_2 n^2 + c_3 n + c_4
\]
Big Oh Notation

\[ f(n) = O(g(n)) \iff \text{there exists a positive constant } c \text{ and an } n_0 \text{ s.t.} \]
\[ f(n) \leq cg(n) \text{ for all } n, n \geq n_0. \]

\[
\begin{align*}
  f(n) &= 3n + 2 \quad \rightarrow \quad f(n) = O(n) \\
  f(n) &= 10n^2 + 4n + 2 \quad \rightarrow \quad f(n) = O(n^2) \\
  f(n) &= 6 \times 2^n + n^2 \quad \rightarrow \quad f(n) = O(2^n) \\
  f(n) &= 9 \text{ (or 8, 933, 849)} \quad \rightarrow \quad f(n) = O(1) \\
  f(n) &= 9n^2 + 4n + 2 \quad \rightarrow \quad f(n) = O(n^4), \text{ but not tight}
\end{align*}
\]

\[ O \text{ is used for Upper Bounds} \]
\Omega\text{ Notation}

\[ f(n) = \Omega(g(n)) \iff \text{there exists a positive constant } c \text{ and an } n_0 \text{ s.t.} \]

\[ f(n) \geq cg(n) \text{ for all } n, n \geq n_0. \]

\begin{align*}
  f(n) & = 3n + 2 \quad \rightarrow \quad f(n) = \Omega(n) \\
  f(n) & = 10n^2 + 4n + 2 \quad \rightarrow \quad f(n) = \Omega(n^2) \\
  f(n) & = 6 \times 2^n + n^2 \quad \rightarrow \quad f(n) = \Omega(2^n) \\
  f(n) & = 9 \text{ (or 8, 363, 456)} \quad \rightarrow \quad f(n) = \Omega(1) \\
  f(n) & = 9n^2 + 4n + 2 \quad \rightarrow \quad f(n) = \Omega(n), \text{ but not tight}
\end{align*}

\Omega\text{ is used for Lower Bounds}
\[ f(n) = \Theta(g(n)) \iff f(n) \text{ is } \mathcal{O}(n), \text{ and } f(n) \text{ is } \Omega(n). \]

\[
\begin{align*}
   f(n) &= 3n + 2 & \implies f(n) &= \Theta(n) \\
   f(n) &= 10n^2 + 4n + 2 & \implies f(n) &= \Theta(n^2) \\
   f(n) &= 6 \cdot 2^n + n^2 & \implies f(n) &= \Theta(2^n) \\
   f(n) &= 9 \ (\text{or } 8, 363, 456) & \implies f(n) &= \Theta(1) \\
   f(n) &= 9n^2 \text{ if } n \text{ is odd, and} \\
   \quad & 4n + 2 \text{ when } n \text{ is even} & \implies f(n) \text{ is not } \Theta(n) \text{ nor } \Theta(n^2)
\end{align*}
\]

\(\Theta\) is used for \text{Tight Bounds}
Practical Complexities

<table>
<thead>
<tr>
<th>Input Size n</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log n$</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>$n \log n$</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$\log n$</td>
<td></td>
</tr>
</tbody>
</table>
Fibonacci Numbers

n is non negative integer

\[
fib(n) = \begin{cases} 
n & \text{if } n \leq 1 \\
fib(n-1) + fib(n-2) & \text{if } n > 1 
\end{cases}
\]

fib(int n)

{ if (n <= 1) return n;
    else return (fib(n-1) + fib(n-2));
}

void main(void)

{ int n;
    cin >> n ;
    cout << n << " " << fib(n) << endl;
}
Time complexity of above method is $\Omega(2^{n/2})$. But it can be computed in $O(n)$ time and constant space.
Performance Measurement

- Choose problem instance size.
- Test data that exhibits worst case.
- Test data that exhibits best case.
- Test data that exhibits average case.
- Test other data.

Timing

- Use user time in “time a.out”
- Or use the following strategy.
```cpp
#include <iostream>
#include "insort.h"

int main(void)
{
    // Program 2.31
    int a[100000], step = 1000;
    clock_t start, finish;
    for (int n = step; n <= 1000; n += step) {
        // get time for size n
        for (int i = 0; i < n; i++)
            a[i] = n - i; // initialize
        start = clock();
        InsertionSort(a, n);
        finish = clock();
        cout << n << ' ' << (finish - start) / CLOCKS_PER_SEC << endl;
    }
}
```
Sometimes Analysis Is Not Important

- Program is run a few times
- Input size is always small
- Efficient programs are sometimes hard to maintain
- Sometimes efficient algorithms use too much space
- Stability and accuracy issues in numerical algorithms

But, most of the time it is very useful !!!