# 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.

# Example (Operation Count)

```
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.

| | |
|---|---|
| $x = x + 1;$ | 1 unit |
| $\text{for}(i = 1; i <= n; i = i + 1)$ | |
| $\quad x = x + 1;$ | $\sum_{i=1}^{n} 1 = n$ units |
| $\text{for}(i = 1; i <= n; i = i + 1)$ | |
| $\quad \text{for}(j = 1; j <= i; j = j + 1)$ | |
| $\quad\quad x = x + 1;$ | $\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ |
| $\text{for}(i = 1; i <= n; i = i + 1)$ | |
| $\quad \text{for}(j = 1; j <= i; j = j + 1)$ | |
| $\quad\quad \text{for}(k = 1; k <= j; k = k + 1)$ | |
| $\quad\quad\quad x = x + 1;$ | $\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1$ |
| | $= \sum_{i=1}^{n} \sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2}$ |
| | $= c_1 n^3 + c_2 n^2 + c_3 n + c_4$ |

# Big Oh Notation

$f(n) = O(g(n)) \Leftrightarrow$ there exists a positive constant $c$ and an $n_0$ s.t. $f(n) \leq cg(n)$ for all $n$, $n \geq n_0$.

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

$O$ is used for Upper Bounds

# $\Omega$ Notation

$f(n) = \Omega(g(n)) \Leftrightarrow$ there exists a positive constant $c$ and an $n_0$ s.t. $f(n) \geq cg(n)$ for all $n$, $n \geq n_0$.

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

$\Omega$ is used for Lower Bounds

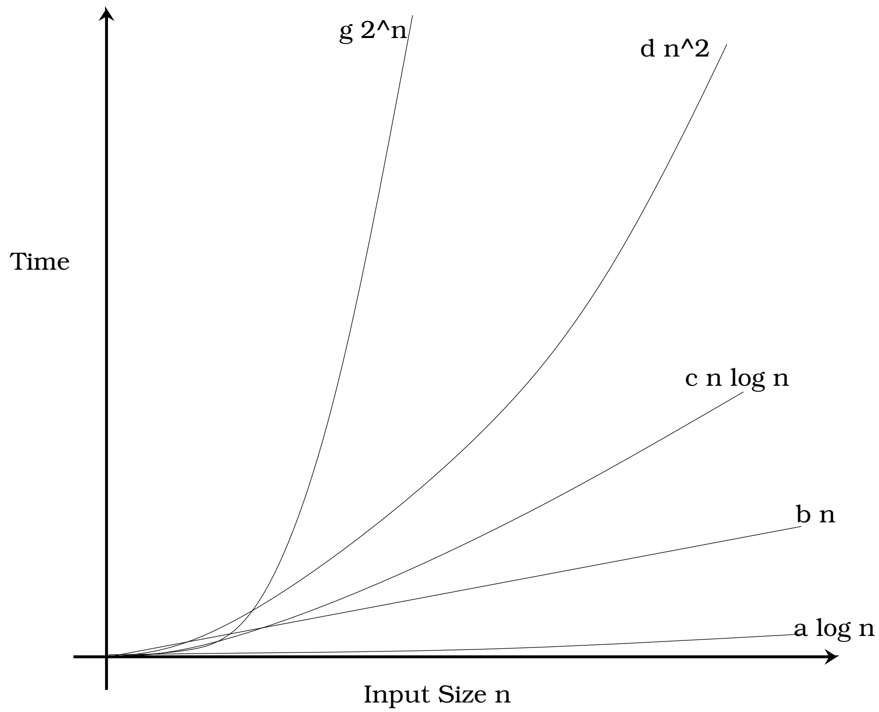
# $\Theta$ Notation

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) \text{ is } O(n)\text{, and } f(n) \text{ is } \Omega(n).$$

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

$\Theta$ is used for Tight Bounds

# Practical Complexities

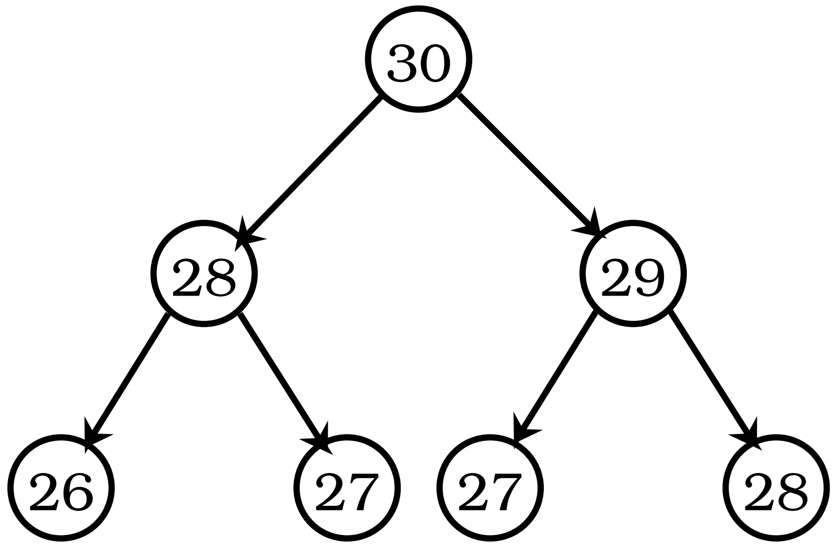


# Fibonacci Numbers

n is non negative integer

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

```
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;
   }
```




| $n$ | 36 | 40 | 44 | 50 | 54 | 60 | 80 | 88 | 100 |
|---|---|---|---|---|---|---|---|---|---|
| Time | 2s | 15s | 1.6m | 8m | 1h | 1d | 6y | 1c | 64c |

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.

```
#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