Terminology:

problem - question we seek an answer for, e.g., "what is the largest item in an array?"

instance - the specific input to the problem

array S:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>15</td>
<td>20</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

largest: ?

n: 7

(number of elements)

Our Goal: We want to derive a function for the number of times that the basic operation is performed related to the problem size.

Example sum - Add all the numbers in the array S of n numbers.

Inputs: positive integer n, array of numbers S indexed from 1 to n.

Outputs: sum, the sum of the numbers in S.

number sum ( int n, const keytype S[ ] )
{
    index i;
    number result;

    result = 0;
    for ( i = 1; i <= n; i++ )
        result = result + S[ i ];
    return result;
}

(Note the text uses bold to indicate an array, e.g., a, and subscripts to indicate an array element, e.g., a₃ or aᵢ. However, I’ll usually use square brackets to denote an array element a[3].)

For algorithm sum, what is the basic operation?

What is the problem size?
Run-time Analysis of *sum*

```cpp
number sum ( int n, const keytype S[ ] )
{
    index i;
    number result;
    result = 0;  // Done once. Assume time to do once is \( c_1 = 100 \)
    for (i = 1;  i <= n;  i++)
        result = result + S [ i ];  // Done n times. Assume time to do loop once is \( c_2 = 10 \)
    return result;
}
```

*Every-case time complexity analysis*, \( T(n) = c_1 + c_2 \cdot n = 100 + 10 \cdot n \)

What determines the length of \( c_1 \) and \( c_2 \)?

When \( n \) is "sufficiently large," the \( 10 \cdot n \) term dominates. For what value of \( n \), does \( 10 \cdot n = 100 \)?

**Big-oh Definition** - asymptotic upper bound

**Definition 2.1:** A function \( f(x) \) is “Big-oh of \( g(x) \)” or “\( f(x) \) is \( O(g(x)) \)” or “\( f(x) = O(g(x)) \)” if there are positive, real constants \( c \) and \( x_0 \) such that for all values of \( x \geq x_0 \),

\[
f(x) \leq c \times g(x).
\]

![Execution Time vs Problem Size](diagram)

\( T(n) = c_1 + c_2 \cdot n = 100 + 10 \cdot n \) is \( O(n) \).

"Proof": Pick \( c = 110 \) and \( x_0 = 1 \).
Then \( 100 + 10 \cdot n \leq 110 \cdot n \) for all \( n \geq 1 \) since
\begin{align*}
    100 + 10 \cdot n &\leq 110 \cdot n \\
    100 &\leq 100 \cdot n \\
    1 &\leq n \text{ which true for all } n \geq 1.
\end{align*}
Problem with big-oh:
If $T(n)$ is $O(n)$, then it is also $O(n^2)$, $O(n^3)$, $O(n^3)$, $O(2^n)$, .... since these are also upper bounds.

**Omega Definition** - asymptotic lower bound

**Definition 2.2:** A function $f(x)$ is “omega of $g(x)$” or “$f(x)$ is $\Omega(g(x))$” or “$f(x) = \Omega(g(x))$” if there are positive, real constants $c$ and $x_0$ such that for all values of $x \geq x_0$,

\[ f(x) \geq c \times g(x). \]

Execution problem size, $x$

T(n) = $c_1 + c_2 \ n = 100 + 10 \ n$ is $\Omega(n)$.

"Proof": We need to find a $c$ and $N$ so that the definition is satisfied, i.e.,

$100 + 10 \ n \geq c \ n$ for all $n \geq N$.

What $c$ and $x_0$ will work?
**Theta Definition** - asymptotic upper and lower bound, i.e., a "tight" bound, i.e., "best" big-oh

**Definition 2.3:** A function $f(x)$ is “theta of $g(x)$” or “$f(x)$ is $\theta(g(x))$” or “$f(x) = \theta(g(x))$” if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

**Alternate Definition 2.3:** A function $f(x)$ is “theta of $g(x)$” or “$f(x)$ is $\theta(g(x))$” or “$f(x) = \theta(g(x))$” if there are positive, real constants $c$, $d$, and $x_0$ such that for all values of $x \geq x_0$,

$$c \times g(x) \leq f(x) \leq d \times g(x).$$

**Execution Time**

Problem size, $x$

$$T(n) = c_1 + c_2 n = 100 + 10 n$$

is $\theta(n)$. 

---

Lecture 3 - 4
A problem is *intractable* if it is impossible to solve it with *any* polynomial-time algorithm.

**Three general categories of problems:**
1. Problems with known polynomial-time algorithms, e.g., sorting an array
2. Problems that have been proven to be intractable, e.g., listing every subset of an n-element set.
3. Problems that have not proven to be intractable, but for which no known polynomial-time algorithm exists, e.g., traveling salesperson problem, TSP

**Traveling Salesperson Problem, TSP** - find the shortest path through a set of cities, visiting each city only one time.

**Input:** A map of cities including the roads between the cities, and distances along the roads. (i.e., a graph problem)

**Output:** A sequence of roads that will take a salesperson through every city on the map and return them to the starting city, such that they will visit each city exactly once, and will travel the shortest total distance.

The TSP is in a special class of algorithms called *NP-complete* problems. *NP-complete* problems have a critical property that if a polynomial-time algorithm can be found for one *NP-complete* problem, then all *NP-complete* problems can be solved in polynomial-time by “minor” modifications of the same algorithm.

Many bioinformatics problems are *NP-complete* problems (and thousands of non-bioinformatics problems). Since thousands of Computer Scientists have tried for about 50 years to find a polynomial-time algorithm to any of these *NP-complete* problems, it is unlikely that an exist.

**NOTE:** This is one of the biggest open questions in Computer Science!!!