**Definition:**

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

**Vocabulary:**

- **Input:**
- **Output:**
- **Correctness:**
- **Generality:**
- **Finiteness:**
- **Effectiveness:**
- **Time complexity:**

**input values from a specified set**

**solution to the problem**

**an algorithm should produce the correct output values for all input values**

The algorithm should be applicable for all problems of the desired form, not just for a particular set of input values.

An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input.

Expressed in terms of the number of operations used by the algorithm when the input has a particular size. (usually expressed as big-o, or big-$\Theta$)
How to present an algorithm:

- plain English
- computer language
- pseudo code

Pseudo code we will use:

**Assignment:**

- \( i \leftarrow 0 \) Assign the value 0 to \( i \)

**Loops:**

- For \( (\text{Start}; \text{End}; \text{Step}) \)
  
  Example: For \( (i=1; i \leq n; \text{Step}=1) \) executes the loop for all \( i \) in \( \{1, \ldots, n\} \)

- While \( (\text{proposition}) \)
  
  Example: \( \text{while } (i \leq n) \)

**Conditional:**

\[ \text{If } (\text{proposition}) \text{ then} \]
\[ \text{Body 1} \]
\[ \text{else if } (\text{proposition 2}) \text{ then} \]
\[ \text{Body 2} \]
\[ \text{else} \]
\[ \text{Body 3} \]
\[ \text{end if} \]
3. Searching algorithms

**Definition:** Searching algorithms solve the problem of locating an element in an ordered list.

This problem occurs in many contexts (think for example spellcheck...)

The simplest way to solve this problem is to test each element of the list → **linear search**

**Pseudo code:**

```
procedure LinearSearch (Input: N (integer), a_1, ... a_N (integers))
    x (integer)
    Output: index (integer)

    Index ← 0
    For (i = 1; i ≤ N; Step = 1)
    {
        if (a_i == x) then
            index ← i
            return
        endif
    }
    return
```
What is the complexity of this algorithm?

It depends!

- If \( x = a_1 \) \( \rightarrow \) 1 comparison
- If \( x = a_N \) \( \rightarrow \) \( N \) comparisons (expected behavior)

We define two measures of complexity:

- **Worst case complexity**: largest number of operations needed to solve the given problem of a specified size.
- **Average case complexity**: average number of operations.

**Worst case complexity for linear search**:

When \( x \) is in the list. The worst case occurs when \( x \) is the last element of the list: the loop is fully executed.

- Number of comparisons: 2 per step, \( N \) steps.
- Hence, the total number of comparisons is \( 2N \), which is \( O(N) \).

**Average case complexity**:

- If \( x = a_1 \), then the number of comparisons is 2
- If \( x = a_2 \), then the number of comparisons is 4
- \( \vdots \)
- If \( x = a_N \), then the number of comparisons is \( 2N \)

Average number:

\[
\frac{1}{N} \left( 2 + \cdots + 2N \right) = \frac{2}{N} \left( 1 + \cdots + N \right) = \frac{2}{N} \frac{N(N+1)}{2} = N + 1, \text{ i.e } O(N)
\]
Linear search however is not optimal, as it does not take into account the fact that the data are ordered.

Another approach: **Binary search**

Example: Let \( S = \{1, 3, 7, 11, 17, 23, 32, 48, 49, 51, 53, 60\} \)

and \( x = 53 \).

If we use binary search, we need 22 comparisions.

Binary search proceeds by dichotomy:

\[
\left\lfloor \frac{N+1}{2} \right\rfloor = 6
\]

\[
1, 3, 7, 11, 17, 23, 32, 48, 49, 51, 53, 60
\]

53 is bigger than 23: we only need to look at the right of 23.

\[
\left\lfloor \frac{N+1}{2} \right\rfloor = 3
\]

\[
32, 48, 49, 51, 53, 60
\]

53 is bigger than 49: we only need to look at the right of 49.

\[
\left\lfloor \frac{N+1}{2} \right\rfloor = 2
\]

\[
51, 53, 60
\]

We are done!

We only need 3 comparisions... + 3 auxiliary operations to find which number to compare.


Pseudo code for binary search

Procedure Binary Search

\[ \text{Input: } \begin{aligned} N \text{ (integer)} \\ a_1, \ldots, a_N \text{ (integers)} \\ x \text{ (integer)} \end{aligned} \]
\[ \text{Output: } \text{Index} \text{ (integer)} \]

Index \leftarrow 0
Left \leftarrow 1
Right \leftarrow N

While (Left \leq Right)

Center \leftarrow \text{floor} \left( \frac{Right + Left}{2} \right)

if \( x = a_{\text{center}} \) then
    Index \leftarrow \text{center}
    return

else if \( x < a_{\text{center}} \) then
    Right \leftarrow \text{Center} - 1

else
    Left \leftarrow \text{Center} + 1

end if

end while

return
What is the worst-case complexity of the binary search?

Each step requires 2 comparisons + 1 "operation". For simplicity, we assume that the total cost of one step is "3".

How many steps do we need?

Let us assume \( n = 2^k \) (otherwise \( 2^k \leq n < 2^{k+1} \) and we set \( k = \log_2 n \)). Then \( k = \log_2 n \).

At each step, the region in which \( x \) is to be found is divided by 2:

\[
2^k \rightarrow 2^{k-1} \rightarrow \ldots \rightarrow 1
\]

Hence, after at most \( k \) steps, we know if \( x \) is not in the list, or we have located it.

Therefore, in the worst case we need \( 3 \times \log_2 (n) \) operations, i.e. the worst case complexity is \( O(\log_2 n) \).

Example: to find a name in a phone book containing 250 million entries, required at most 28 steps.
4. Sorting

Ordering elements in a list is a very common problem. There are many, many methods to solve this problem. We will look at one of the simplest, the insertion sort method.

How does it work? Think of arranging cards in increasing order. Take one card after the other and insert in slack of cards already ordered. Repeat until all cards have been ordered.

Example: 2, 4, 1, 3, 8, 5

Steps:
1) Start with 2 assumed to be correctly placed.

   \[ 2, 4, 1, 3, 8, 5 \]

2) Move to second position:

   Compare 2 with 4; order is correct so no changes.

   \[ 2, 4, 1, 3, 8, 5 \]

3) Third position:

   Find position of 1 in ordered list. Insert 1 and shift other numbers.
4) Fourth position
   Find position of 3. Insert and shift other numbers.
   \[ 1, 2, 4, 8, 5 \]

5) Fifth position
   Find position of 8. Here, order is correct.
   \[ 1, 2, 3, 4, 8, 5 \]

6) Sixth position
   Find position of 5 and shift other numbers
   \[ 1, 2, 3, 4, 8 \]
   \[ 1, 2, 3, 4, 5, 8 \leftarrow \text{final order} \]
Pseudo code for Insertion Sort

Procedure Insertion Sort \((N \text{ (integer)})\) \((a_1, \ldots, a_N \text{ (integers)})\)

# Initialise loop: from 2 to N
For \((i = 2; i \leq N; \text{ Step}=1)\)
{
    # Store a local copy of current number \(a_i\)
    copy \(\leftarrow a_i\)

    # Locate position to insert number \(a_i\) in the ordered numbers \(1, \ldots, a_{i-1}\); use linear search, stop when a number greater than \(a_i\) is found
    \(IP \leftarrow 1\)
    \(J \leftarrow 1\)
    While \((a_i > a_j) \land (j < i)\)
        \(IP \leftarrow IP + 1\)
        \(J \leftarrow J + 1\)
    \}

# Now \(IP\) indicates where \(a_i\) should be inserted. First, move up all numbers from \(a_{IP}\) to \(a_{i-1}\) by 1:
For \((j = i-1; j \geq IP; \text{ Step}=-1)\)
    \(a_{j+1} \leftarrow a_j\)

# Insert \(a_i\) at position \(IP\)
\(a_{IP} \leftarrow \text{copy}\)

# end loop
Worst case complexity of insertion sort:

\[
\begin{align*}
& i = 2 \\
& i = 3 \\
& i = 4 \\
& \vdots \\
& i = N \\
2 \text{ comparisons} \\
4 \text{ comparisons} \\
6 \text{ comparisons} \\
& \text{total: } 2(1+2+\ldots+N) = N(N+1)
\end{align*}
\]

Insertion sort is of order \( N^2 \)

5. Understanding complexity of algorithms.

- \( O(1) \) constant
- \( O(\log(n)) \) logarithmic
- \( O(n) \) linear
- \( O(n^b) \) polynomial
- \( O(b^n) \) exponential

A problem solvable with an algorithm of worst case complexity that is polynomial is called \underline{tractable}, otherwise it is \underline{intractable}.

Types of intractable problems:

- unsolvable
- intractable, but with a solution that can be checked in polynomial time: class \( \text{NP} \) (non deterministic polynomial)
NP-complete: class of problems with equivalent complexity. If one of them is proven to be polynomial, all are polynomial.

One example of NP-complete problem:

Given a finite set \( S \) of integers, determine whether any non-empty subset of \( S \) has its elements that sum to 0.

A supposed answer is very easy to verify, but no one knows a way to solve this problem other than testing every single possible subset.