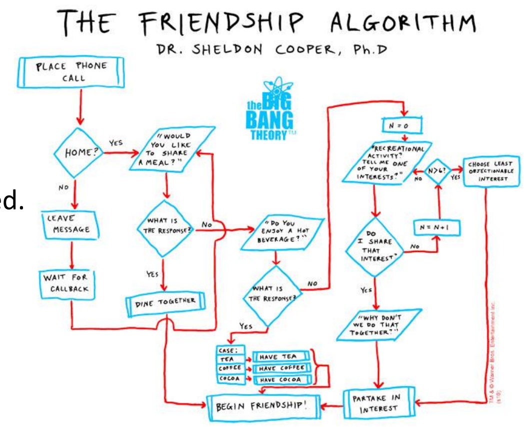
Algorithms - Introduction

1. Algorithm

Computers are information processing machines. While data structures are needed to systematically organize the data, an algorithm is needed to perform a data processing task in a **finite amount of time**.

- ➢ An algorithm is a **sequence of steps** to solve a given problem.
- ➢ Steps can consist of smaller steps and can be repeated.



Algorithms are closely dependent on the used data structure(s)!

2. Algorithm Representation

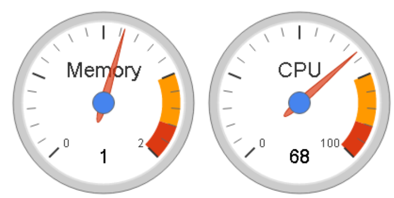
The result of an algorithm execution must not depend on how it is represented.
The most common forms of representation are:

- **described** – natural language (informal; ambiguity is possible) or pseudo code (somehow informal; some ambiguity is possible)
- **visualized graphically** - flowchart (easy to understand)
- **clearly defined** - program code

3. Choosing an Algorithm

Any problem can usually be solved by several different algorithms. How do we choose which one to use?

- ➢ compare required resources
  - memory
  - CPU time (**running time**)
- ➢ compare performance characteristics
  - precision
  - simplicity



4. Linear Search Algorithms

Linear search usually starts at index 0 and looks at each item one by one.
At each index, we ask this question: **"Is the item we are looking for at the current index?"**
One of <u>**two conditions**</u> breaks the loop: **the end of the list is reached**, or **the item is found**.

a) Linear Search with while-loop

- set the **index** to **0**
- while **index < length of the list** and **item[index] isn't the searched item**
  - add 1 to the index
- if **index** has reached the end of the list the value was not found; **return -1**
- otherwise **return the index** at which the value was found

This algorithm evaluates both conditions at each step through the loop. The evaluation of the first condition wastes time unnecessarily.

b) Linear Search with for-loop

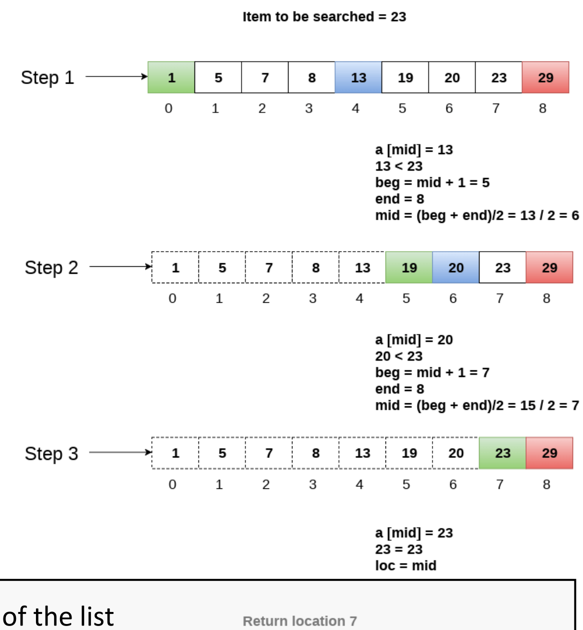
- for **index** from **0** to the last valid index in the list,
  - if **item[index] is the searched item**,
    - **return the index** at which the value was found
- the value was not found; **return -1**

Direct evaluation of the condition for the end of the list is no longer needed. However, this evaluation is hidden in the loop in most languages (e.g. Java).

c) Linear Search with sentinel

- append the value we are searching to end of the list **(add the sentinel)**
- set the **index** to **0**
- while **item[index] isn't the searched item**,
  - add 1 to the index
- remove the value from the end of the list **(remove the sentinel)**
- if **index** has reached the end of the list (the sentinel) the value was not found; **return -1**
- otherwise **return the index** at which the value was found

By ensuring that the value is in the list we no longer need to check whether the end of the list has been reached. Thus, the constant evaluation of the first condition is not needed.



5. Binary Search Algorithms

This algorithm repeatedly divides an **ordered** list and ignores one half of it.
The search continues in the other half.

a) Iterative Binary Search

- set **left** and **right** indices to the beginning and the end of the list
- while **left** index is not bigger than **right** index,
  - calculate index **middle**
  - if the item at index **middle** is the searched item:
    - **return middle**
  - if the item at index **middle** is greater than the searched item:
    - **right** index becomes middle - 1 **(ignore the right half)**
  - if the item at index **middle** is smaller than the searched item:
    - **left** index becomes middle +1 **(ignore the left half)**
- **return -1**

b) Recursive Binary Search

- if **right** index is smaller than **left** index, **(base case - list was exhausted)**
  - **return -1**
- calculate index **middle**
- if the item at index **middle** is the searched item:
  - **return middle** **(base case - the item was found)**
- if the item at index **middle** is greater than the searched item:
  - **call** the recursive function for the list between **left** and **middle-1** **(ignore the right half)**
- if the item at index **middle** is smaller than the searched item:
  - **call** the recursive function for the list between **middle+1** and **right** **(ignore the left half)**