Algorithms

Dr. Tuğberk Kocatekin
Istanbul Arel University

what is an algorithm?

- Formal:
    - An algorithm is an ordered set of **unambiguous**, **executable** steps that defines a **terminating** process.
- Informal:
    - A set of steps that define how a task is performed.
- Not only for *computer science*, can also be used for other tasks.
    - Recipe
- Can be represented in various forms:
    - Flowcharts
    - Pseudocodes
    - etc.

details

- Formal definition
    - **unambiguous**: information must be enough for execution. It should not require any creative skills. It must be clear.
    - **terminating**: execution of algorithms must come to an end.
- Sometimes we refer to non-terminating processes too.
    - Some say that these terminate and repeat.
- There are also non-terminating processes which are important
    - Monitoring (vital signs of a patient)
    - Maintaining (altitude on an aircraft)

youtube videos & games

- Father and children on following an algorithm
  - https://www.youtube.com/watch?v=PlWAFPGN5W0
- scratch from mit.edu
- cargo-bot
  - there are similar games

algorithm representations

- Algorithms are not enough for computer to understand. We should put it in certain terms.
- Algorithms are constructed using **primitives** which are a set of building blocks for computers to understand.
- A collection of primitives and a **collection of rules** combined constitutes a **programming language**.
  - A programming language consists of primitives and rules so that we can turn algorithms into meaningful programs which computer can understand.

pseudocode

- Provides a description of the steps in an algorithm in plain language.
- Similar to a programming language but not a specific language.

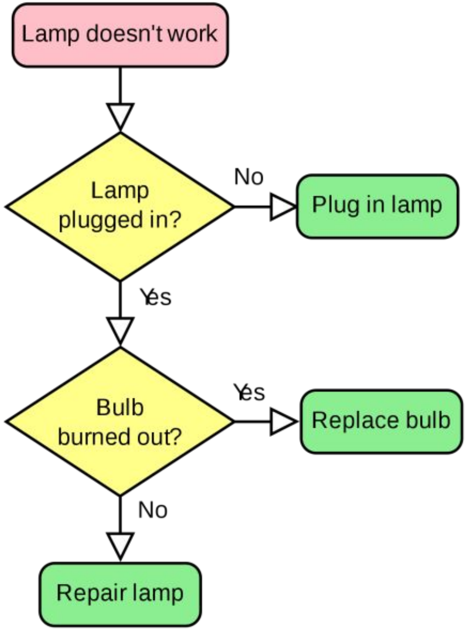
```
Organise everything together;
Plug in kettle;
Put teabag in cup;
Put water into kettle;
Wait for kettle to boil;
Add water to cup;
Remove teabag with spoon/fork;
Add milk and/or sugar;
Serve;
```

Algorithm 1 Intent Communication Algorithm

procedure DEC-MDP(S, A, P, R, O, Ω)
  $A \leftarrow A_1 \times A_2$
  $s_1, s_2 \leftarrow S$
  $a_1, a_2 \leftarrow A$
  $R(s_i, a_i) = 0, i = 0, j = 0$
  repeat
    $i \leftarrow i + 1, j \leftarrow j + 1$
    for o_1, o_2 **do**
      Determine scenario $\in [1, 4]$
      $p_1, p_2 \leftarrow P(s' \mid s, a_1, a_2)$
      $a_1, a_2 \leftarrow A$
      $\max_{a_1,a_2} r_{1,2}(s_1, s_2, a_1, a_2)$
      for s_1, s_2 **do** check
        if $d(s_1, s_2) \leq$ *scenario threshold* **then**
          Update θ_i, θ_j using $d(s_1, s_2)$
        end if
        $\pi[s_1, s_2] = \arg\max_{a_1,a_2} r_{1,2}$
      end for
    end for
  until $s_1 = s_{g_1}$ or $s_2 = s_{g_2}$
  return $\pi, R(s_i, a_i)$
end procedure

flowchart

- Type of diagram that represents a **workflow** or **process**.
- Represents a solution model to a given problem.
- Usage precedes computers, used in several different areas
  - engineering, quality control, etc
- First used by **John von Neumann** for computer programs.
  - *Fun fact: His wife was the first programmer to implement a flowchart to a computer program. That was Monte-Carlo simulation.*



algorithmic thinking & problem solving

- Before programming you need to be able to think algorithmically. Programs are just representations of algorithms.
- Mathematician G.Polya presented basic principles of problem solving in 1945:

1. Understand the problem
2. Get an idea of how an algorithmic function might solve the problem.
3. Formulate the algorithm and represent it as a program.
4. Evaluate the program for accuracy and for its potential as a tool for solving other problems.

loops

- An implementation of a repetition.
  - **while**: a loop repeated as long as an expression is true.
  - **for**: a loop that runs for a specific number of times.
  - **nested loops**: loop inside loop.
  - **infinite loop**: a loop running indefinitely because there is no terminating condition

```

#while loop
while(i<5):
        print(i)
        i += 1

#for loop
for i in range(10):
        print(i)

#nested loops
for i in range(10):
        for j in range(5):
                print(j)
#infinite loop
while(true):
        print('this is a loop')

```

loop control

- break:
  - When you write break, the loops terminates.
- continue:
  - This statement jumps to next iteration skipping anything in between. Generally used with *if*.

```
while true:
        print('hello')
        break
        print('world')

```

```
while(x<10):
        if(x==5):
                continue
        else:
                print('hello world')

```

algorithm classes

- Algorithms with similar problem-solving approaches can be grouped together.
  - brute-force algorithm
  - recursive algorithm
  - backtracking algorithm
  - searching algorithm
  - sorting algorithm
  - divide&conquer algorithm
  - greedy algorithm
  - dynamic programming
- They can also be grouped by the problem they are trying to solve: *searching*, *sorting*, etc.

brute force algorithm

- Simplest approach for a problem. When we try to find a solution, we try every scenario until we find the answer.
- https://www.geeksforgeeks.org/fundamentals-of-algorithms/?ref=shm
- Advantages:
  - Good for small and simple problems
  - Generic method and can be applied to almost every scenario
  - Guaranteed way to find correct solution
- Disadvantages:
  - Inefficient. Most of the time it takes too much time.
  - Relies on computing power and not the algorithm quality.
  - Slow

recursive algorithm

- A problem is broken into several sub-parts and called the same function again and again.

- The following shows the recursive and iterative versions of the factorial function:

Recursive version

```
int factorial (int n)
{
    if (n == 0)
        return 1;
    else
        return  n * factorial (n-1);
}
```

Recursive Call

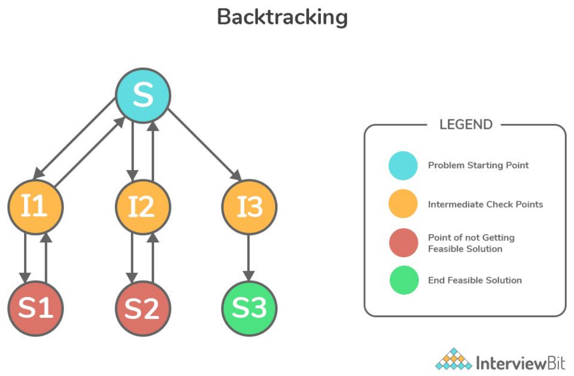
Iterative version

```
int factorial (int n)
{
    int i, product=1;
    for (i=n; i>1; --i)
        product=product * i;

    return product;
}
```

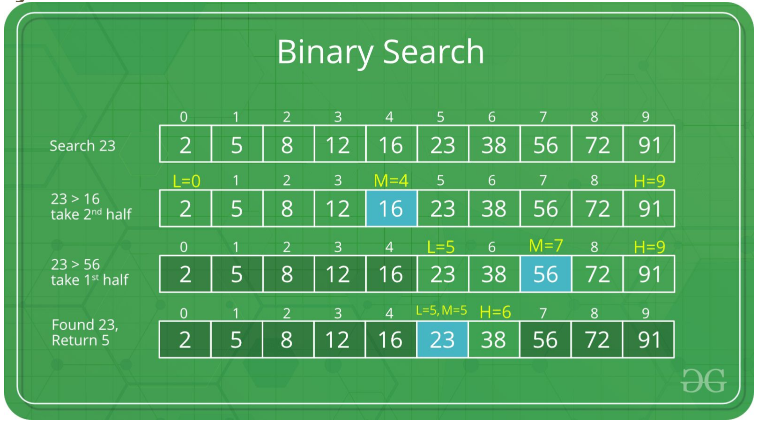
backtracking algorithm

- Builds the solution by searching among all possible solutions.
- Whenever a solution fails we trace back to the failure point and build on the next solution.
- This is done until the solution is found or all possible solutions are checked.



searching algorithms

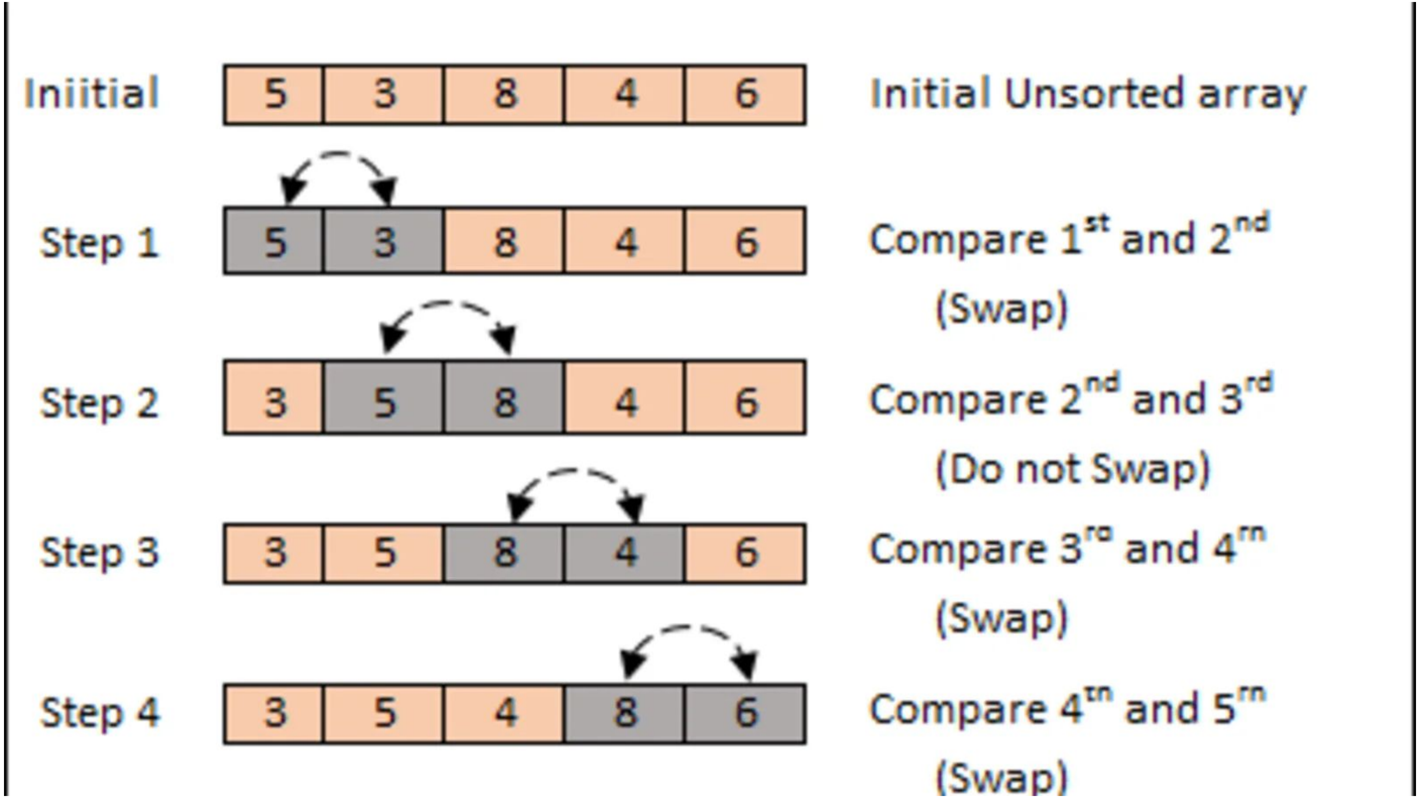
- These algorithms are used for searching elements or groups of elements in data structure (array, list, etc).
- Two categories:
  - Sequential Search
    - Start by looking at every number sequentially.
  - Interval Search
    - Binary search
      - fastest
      - performed on ordered list



sorting algorithms

- Sorting: arranging a group of data.
- Several sorting algorithms:
  - merge sort
  - selection sort
  - bubble sort
  - insertion sort
  - quick sort (fastest)
  - …
- Visualization of sorting algorithms:
  - https://www.youtube.com/watch?v=ZZuD6iUe3Pc

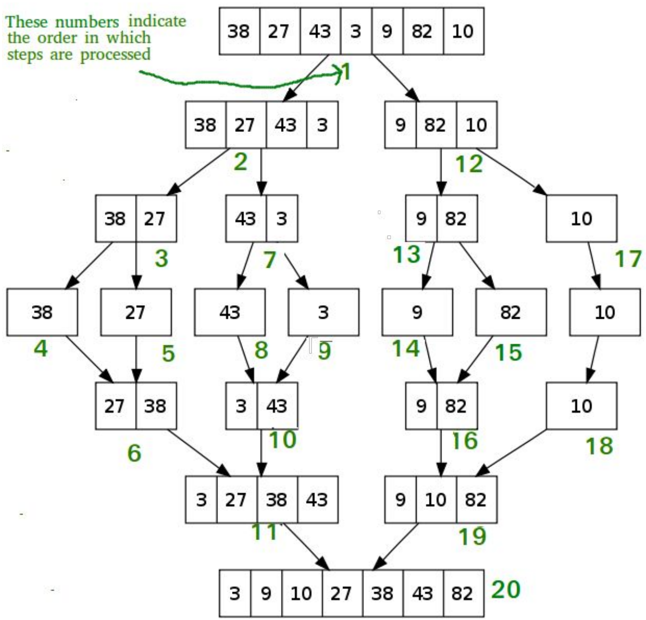
bubble sort



Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd
(Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd
(Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th
(Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th
(Swap)

divide & conquer algorithms (DAC)

- Breaks a problem into sub-problems, solves that and merges all together.
  - Divide
    - Dividing the problem into smaller sub-problems.
  - Solve
    - Solve sub-problems by calling recursively.
  - Combine
    - Combine the sub-problems to get the final solution.
- examples:
  - quicksort
  - merge sort
  - karatsuba algorithm



algorithm complexity

- Defined by the amount of space and time it consumes.
- **space complexity:**
  - amount of memory required by the algorithm.
- **time complexity:** amount of time the algorithm requires to execute and get result.
  - best, average and worst. We usually care about the **worst**.
  - expressed by using **big O notation**. O(n), O(log n), etc.
  - O(n): *linear time*
  - $O(n^2)$: *quadratic time (cubic time for 3)*
  - $O(2^n)$: *exponential time complexity*
  - O(log n): *logarithmic time*
  - ...

naming conventions

- camelCase
- snake_case
- PascalCase
- kebab-case