1 month

Algorithms Fundamentals

Introduction, Design & Types

Author: Eng. Carlos Andrés Sierra, M.Sc.

cavirguezs@udistrital.edu.co

Full-time Adjunct Professor
Computer Engineering Program
School of Engineering
Universidad Distrital Francisco José de Caldas

2026-I

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Outline

Outline

What is an Algorithm?

Complete

NP-Hard

steps

Definition

An algorithm is a finite sequence of well-defined instructions that can be mechanically executed to solve a computational problem or perform a task.

family

Key Components:

- **Input:** Zero or more quantities supplied externally
- **Output:** One or more quantities produced
- **Instructions:** Precise and unambiguous steps
- **Execution:** Can be performed mechanically

Remember

An algorithm is a *method* for solving problems, not the implementation itself!

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

What is an Algorithm?

Definition

An algorithm is a finite sequence of well-defined instructions that can be mechanically executed to solve a computational problem or perform a task.

Key Components: – values → context

- **Input:** Zero or more quantities supplied externally
- **Output:** One or more quantities produced
- **Instructions:** Precise and unambiguous steps
- **Execution:** Can be performed mechanically

0–n $\quad$ 1–m

$I \rightarrow \boxed{\text{Alg.}} \rightarrow O$

$x_0 \longrightarrow y_0$

$x_1 \longrightarrow y_1$

x

$x_p \longrightarrow y_p$

Remember

An algorithm is a *method* for solving problems, not the implementation itself!

What is an Algorithm?

Definition

An algorithm is a finite sequence of well-defined instructions that can be mechanically executed to solve a computational problem or perform a task.

Key Components:

- **Input:** Zero or more quantities supplied externally
- **Output:** One or more quantities produced
- **Instructions:** Precise and unambiguous steps
- **Execution:** Can be performed mechanically

design

Remember

An algorithm is a [*method* for solving problems] not the implementation itself!

Algorithm Characteristics

Essential Properties

1. **Finiteness:** Must terminate after finite number of steps
2. **Definiteness:** Each step must be precisely defined
3. **Input:** Zero or more input values
4. **Output:** One or more output values
5. **Effectiveness:** Operations must be basic and feasible

1 2

Example

Making Coffee Algorithm:

1. Boil water (definite action)
2. Add coffee grounds (precise amount)
3. Wait 4 minutes (finite time)
4. Pour into cup (effective operation)

Algorithm Characteristics

Essential Properties

1. **Finiteness:** Must terminate after finite number of steps
2. **Definiteness:** Each step must be precisely defined
3. **Input:** Zero or more input values
4. **Output:** One or more output values
5. **Effectiveness:** Operations must be basic and feasible

Example

Making Coffee Algorithm:

1. Boil water (definite action)
2. Add coffee grounds (precise amount)
3. Wait 4 minutes (finite time)
4. Pour into cup (effective operation)

ml

90°

CONSTANT temp-water

P water

amount? gr.? 95/cup?

Algorithm vs. Program vs. Process

design   code   Top   Ctrl + Alt + Supr

Algorithm

- Abstract concept
- Language-independent
- Mathematical description
- Design phase

Program

- Concrete implementation
- Language-specific
- Code representation
- Implementation phase

Process

- Runtime execution
- System-dependent
- Active computation
- Execution phase

compilation   execution   logic   errors   error

Example

[illegible] **Example:** Finding the maximum number

- **Algorithm:** "Compare each element with current maximum"
- **Program:** Python code implementing the comparison
- **Process:** Program running in memory, using CPU cycles

Algorithm vs. Program vs. Process

Algorithm

- Abstract concept
- Language-independent
- Mathematical description
- Design phase

Program

- Concrete implementation
- Language-specific
- Code representation
- Implementation phase

Process

- Runtime execution
- System-dependent
- Active computation
- Execution phase

Example

Example: Finding the maximum number

if > replace

- **Algorithm:** "Compare each element with current maximum"
- **Program:** Python code implementing the comparison
- **Process:** Program running in memory, using CPU cycles

Types of Algorithmic Thinking

Fundamental Approaches

1. **Sequential Thinking:** Step-by-step execution
2. **Conditional Thinking:** Decision-based branching
3. **Iterative Thinking:** Repetitive operations loops
4. **Recursive Thinking:** Self-referential solutions
5. **Parallel Thinking:** Concurrent execution

automation

Example (Daily Life)

- Getting dressed (sequential)
- Choosing route to university (conditional)
- Studying until understanding (iterative)
- Recalling a recipe (recursive)
- Cooking multiple dishes (parallel)

Types of Algorithmic Thinking

Fundamental Approaches

1. **Sequential Thinking:** Step-by-step execution
2. **Conditional Thinking:** Decision-based branching
3. **Iterative Thinking:** Repetitive operations
4. **Recursive Thinking:** Self-referential solutions
5. **Parallel Thinking:** Concurrent execution

Example (Daily Life)

- Getting dressed (sequential)
- Choosing route to university (conditional)
- Studying until understanding (iterative)
- Recalling a recipe (recursive)
- Cooking multiple dishes (parallel)

Study Case: Euclidean GCD Algorithm

Greatest Common Divisor Problem

Input: Two positive integers a and b
Output: The largest integer that divides both a and b

Algorithm 1 Euclidean GCD Algorithm

Input: Integers a, b where $a \geq b > 0$
while $b \neq 0$ **do**
  $r \leftarrow a \bmod b$
  $a \leftarrow b$
  $b \leftarrow r$
end while
return a

Study Case: Euclidean GCD Algorithm

Greatest Common Divisor Problem

Input: Two positive integers a and b
Output: The largest integer that divides both a and b

Algorithm 2 Euclidean GCD Algorithm

Input: Integers a, b where $a \geq b > 0$
while $b \neq 0$ **do**
  $r \leftarrow a \bmod b$
  $a \leftarrow b$
  $b \leftarrow r$
end while
return a

define f as a
• def f

Exercise Time! [1]

Exercise 1: Square Root Calculation

Design an algorithm to calculate the square root of a positive number using the Babylonian method:

1. Start with an initial guess x_0 (e.g., $S/2$ for number S)
2. Improve the guess: $x_{n+1} = \frac{1}{2}(x_n + \frac{S}{x_n})$
3. Repeat until convergence

```
input: S
x = S/2
DIFF = 0,001
DO{
    x_next = (1/2)*(x+(S/x))
    diff_x = abs(x - x_next)
    x = x_next
} while (diff_x > DIFF)
return x
```

Exercise 2: Find Maximum in an Array

Write an algorithm that finds the maximum element in an array of integers. **Consider:** What if the array is empty? What about duplicate maximums?

Exercise Time! [1]

Exercise 1: Square Root Calculation

Design an algorithm to calculate the square root of a positive number using the Babylonian method:

1. Start with an initial guess x_0 (e.g., $S/2$ for number S)
2. Improve the guess: $x_{n+1} = \frac{1}{2}(x_n + \frac{S}{x_n})$
3. Repeat until convergence

if length A == 0 {
return 'error'
}

Exercise 2: Find Maximum in an Array

Write an algorithm that finds the maximum element in an array of integers.
Consider: What if the array is empty? What about duplicate maximums?

Input: [a_0 a_1 ... a_n]
max ← A[0]
for (i = 1 → n-1) {
if (A[i] > max) {
max = A[i] }

return max

Exercise Time! [2]

Exercise 3: Algorithms in Daily Life

Identify and describe three algorithms you use in your daily life. For each:

- Define clear inputs and outputs
- List the precise steps
- Verify all algorithm characteristics

Outline

Mathematical Notation for Algorithms [I]

Mathematical Notation

Mathematical notation provides a formal way to express algorithms using symbols, formulas, and structures from mathematics.

Universal

Why Mathematical Notation?

- **Precision:** Unambiguous specification
- **Universality:** Language-independent
- **Conciseness:** Compact representation
- **Analysis:** Enables formal verification

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Mathematical Notation for Algorithms [I]

Mathematical Notation

Mathematical notation provides a formal way to express algorithms using symbols, formulas, and structures from mathematics.

Why Mathematical Notation?

- **Precision:** Unambiguous specification
- **Universality:** Language-independent
- **Conciseness:** Compact representation
- **Analysis:** Enables formal verification

+ / > <=

Mathematical Notation for Algorithms [II]

Example

Set Operations: $S = \{2, 3, 5, 7, 11, 13, 17, 19\}$

- $S = \{x \in \mathbb{N} : x$ is prime and $x < 20\}$
- $\forall x \in S, P(x)$ (for all elements in S, property P holds) Map
- $\exists x \in S : Q(x)$ (there exists an element in S with property Q)

Filter

Example

Function Definition:

$$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$f(a, b) = \begin{cases} a & \text{if } b = 0 \\ f(b, a \bmod b) & \text{otherwise} \end{cases}$$

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Mathematical Notation for Algorithms [II]

Example

Set Operations:

- $S = \{x \in \mathbb{N} : x \text{ is prime and } x < 20\}$
- $\forall x \in S, P(x)$ (for all elements in S, property P holds)
- $\exists x \in S : Q(x)$ (there exists an element in S with property Q)

Example

Function Definition:

$$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$f(a, b) = \begin{cases} a & \text{if } b = 0 \\ f(b, a \bmod b) & \text{otherwise} \end{cases}$$

Euclidean GCD

Converting Mathematical Formulas to Pseudocode

Mathematical Expression

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Another Example

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$sum \leftarrow 0$
for $i = 1$ to n **do**
  $sum \leftarrow sum + i^2$
end for
$formula \leftarrow \frac{n(n+1)(2n+1)}{6}$
if $sum = formula$ **then**
  return `true`
else
  return `false`
end if

$result \leftarrow 1$
$term \leftarrow 1$
$n \leftarrow 1$
while $|term| > \epsilon$ **do**
  $term \leftarrow \frac{term \cdot x}{n}$
  $result \leftarrow result + term$
  $n \leftarrow n + 1$
end while
return $result$

Converting Mathematical Formulas to Pseudocode

Mathematical Expression

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

```
sum ← 0
for i = 1 to n do
    sum ← sum + i^2
end for
formula ← n(n+1)(2n+1)/6
if sum = formula then
    return true
else
    return false
end if
```

range(1, n+1)

validation

Another Example

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

```
result ← 1
term ← 1
n ← 1
while |term| > ε do
    term ← term·x/n
    result ← result + term
    n ← n + 1
end while
return result
```


UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Converting Mathematical Formulas to Pseudocode

Mathematical Expression

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

```
sum ← 0
for i = 1 to n do
    sum ← sum + i^2
end for
formula ← n(n+1)(2n+1)/6
if sum = formula then
    return true
else
    return false
end if
```

Another Example

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

```
result ← 1
term ← 1
n ← 1
while |term| > ϵ do
    term ← (term·x)/n
    result ← result + term
    n ← n + 1
end while
return result
```

Converting Mathematical Formulas to Pseudocode

Mathematical Expression

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

```
sum ← 0
for i = 1 to n do
    sum ← sum + i²
end for
formula ← n(n+1)(2n+1)/6
if sum = formula then
    return true
else
    return false
end if
```

Another Example

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

DIFF

```
result ← 1
term ← 1
n ← 1
while |term| > ε do
    term ← (term·x)/n
    result ← result + term
    n ← n + 1
end while
return result
```

Study Case: Factorial Using Different Notations

Mathematical

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

$\Sigma \; \Pi$

$$n! = \prod_{i=1}^{n} i$$

n not > 0 ⇒ error

Iterative Pseudocode

function FACTORIAL(n)
  $result \leftarrow 1$
  for $i = 1$ **to** n **do**   (1, n, +1)
    $result \leftarrow result \times i$
  end for
  return $result$
end function

Pseudocode Standards and Conventions

reservadas

Basic Elements

- **Keywords:** `if, then, else, while, for, return`
- **Assignment:** $x \leftarrow value$ or $x = value$
- **Comparison:** $=, \neq, <, >, \leq, \geq$
- **Logic:** `and, or, not`
- **Comments:** // Single line, /* Multi-line */

type(var)

type

&& || ! ~

bool

≠ --- !=

< --- <=

> --- >=

C1	C2	&&	||
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

comp1 and comp2

|| or

not comp1

"...."

"...."

#

Algorithm Documentation Best Practices

Essential Documentation Elements

1. **Purpose:** What does the algorithm do? → This function...
2. **Preconditions:** What must be true before execution? Fcn: $n \geq 0$
3. **Postconditions:** What is guaranteed after execution? Fcn: $a > b$
4. **Parameters:** Input and output specifications * Args: Returns:
5. **Complexity:** Time and space requirements
6. **Examples:** Sample inputs and outputs

Doc.

I O

Programming Contest

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Study Case: Fibonacci Sequence

Problem Definition

The Fibonacci sequence: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

Purpose: Compute the nth Fibonacci number

Preconditions: $n \geq 0$ (non-negative integer)

Postconditions: Returns F_n where $F_0 = 0, F_1 = 1$

Parameters:

- n: position in Fibonacci sequence

Complexity: $O(n)$ time, $O(1)$ space

Example: $\textsc{FibonacciIterative}(5) = 5$

0 1 1 2 3 5 8 13 21 ...

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

n

n^2

for $i = 2$ to n

$f(i) = f[i-1] + f[i-2]$

Study Case: Fibonacci Sequence

Problem Definition

The Fibonacci sequence: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

Purpose: Compute the nth Fibonacci number

Preconditions: $n \geq 0$ (non-negative integer)

Postconditions: Returns F_n where $F_0 = 0, F_1 = 1$

Parameters:

- n: position in Fibonacci sequence → 7-th

Complexity: $O(n)$ time, $O(1)$ space

Example: $\textsc{FibonacciIterative}(5) = 5$

0 | 0
6 | 8

Control Structures

Sequential

1: Step 1
2: Step 2
3: Step 3
4: ...

Execution: One after another

Conditional

1: **if** condition **then**
2: Action A
3: **else**
4: Action B
5: **end if**

Execution: Based on condition

Iterative

1: **while** condition **do**
2: Action
3: **end while**
4: **for** i = 1 to n **do**
5: Action
6: **end for**

Execution: Repeated

Typical

Control Structures

$< \quad > \quad \leq \quad \geq \quad \neq \quad ==$

Sequential

```
Step 1
Step 2
Step 3
...
```

Execution: One after another

Conditional

```
if condition then
    Action A
else
    Action B
end if
```

Execution: Based on condition

Iterative

```
while condition do
    Action
end while
for i = 1 to n do
    Action
end for
```

Execution: Repeated

Python ==

2 == 2.0

JavaScript

2 === 2.0 → false

2 == 2.0 → true

Java - C

```
x = y > 5 ? 3 : 4;
```

Python

```
x = 3 if y > 5 else 4
```

```
if (y > 5) {
    x = 3;
} else {
    x = 4;
}
```

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Control Structures

Sequential

Step 1
Step 2
Step 3
...

Execution: One after another

Conditional

```
if condition then
    Action A
else
    Action B
end if
```

Execution: Based on condition

Iterative

```
while condition do
    Action
end while
for i = 1 to n do
    Action
end for
```

Execution: Repeated

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Control Flow Concepts

- **Entry Point:** Where execution begins
- **Exit Point:** Where execution ends
- **Decision Points:** Where flow branches
- **Loop Control:** How iterations are managed

inputs - preconditions

output

break | continue

continue

Study Case: Decision-Making Algorithms

Grade Classification Algorithm

Problem: Convert numerical grade to letter grade

Large

Num → letter

Decision Tree

- $Score \geq 90? \rightarrow A$
- $Score \geq 80? \rightarrow B$
- $Score \geq 70? \rightarrow C$
- $Score \geq 60? \rightarrow D$
- $Otherwise \rightarrow F$

Key Concepts

- **Mutually exclusive** conditions
- **Order matters** in if-elsif chains
- **Default case** handling

Study Case: Decision-Making Algorithms

Grade Classification Algorithm

Problem: Convert numerical grade to letter grade

Decision Tree

- $Score \geq 90? \rightarrow A$
- else $Score \geq 80? \rightarrow B$
- else $Score \geq 70? \rightarrow C$
- else $Score \geq 60? \rightarrow D$
- $Otherwise \rightarrow F$

switch

Key Concepts

- **Mutually exclusive** conditions
- **Order matters** in if-elsif chains
- **Default case** handling

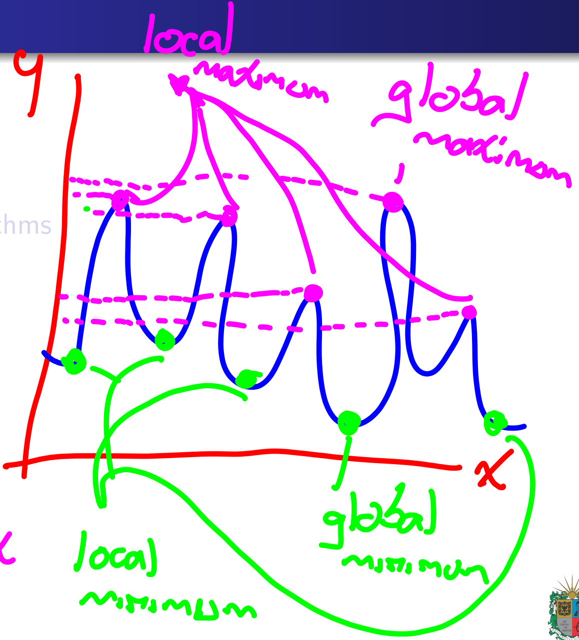
Outline

$f(x) = a_n x^n + a_{n-1} x^{n-1}$

$y = \ldots + a_0 x^0$

Opt. max

min



Greedy Algorithms: Philosophy and Approach

Definition

A greedy algorithm makes locally optimal choices at each step, hoping to find a global optimum.

Greedy Strategy

1. **Greedy Choice:** At each step, choose the best available option
2. **Optimal Substructure:** Optimal solution contains optimal solutions to subproblems
3. **No Backtracking:** Never reconsider previous choices

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Greedy Algorithms: Philosophy and Approach

Definition

A greedy algorithm makes locally optimal choices at each step, hoping to find a global optimum.

j. most cheaper   God most ch...

Greedy Strategy

1. **Greedy Choice:** At each step, choose the best available option
2. **Optimal Substructure:** Optimal solution contains optimal solutions to subproblems

   local
3. **No Backtracking:** Never reconsider previous choices

Greedy Algorithms: When to Use Them

When Greedy Works:

- Activity selection
- Minimum spanning trees → Ware
- Huffman coding

compression

text

18bytes

PATTERN

PATTERN

PATTERN

1. all words
2. sort words by size

2.1 sort by occurrences

PATTERN → ♪

When Greedy Fails:

- 0/1 Knapsack problem
- Traveling salesman
- Graph coloring

Greedy Algorithms: When to Use Them

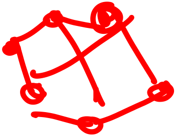
When Greedy **Works**:

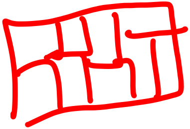
- Activity selection
- Minimum spanning trees
- Huffman coding

When Greedy **Fails**:

- 0/1 Knapsack problem
- Traveling salesman
- Graph coloring

S + ele v p





Study Case: Coin Change Problem

Problem Statement

Given: Coin denominations $[d_1, d_2, \ldots, d_k]$ and amount n
Goal: Find minimum number of coins to make amount n

1000 - 500 - 200 - 100 - 50

1. sort denominations → great to less

2. loop → coins = n/d_i → integer

$n = n - (\text{coins} * d_i)$

Study Case: Coin Change Problem [Solution]

Denominations: [25, 10, 5, 1] cents, **Amount:** 67 cents

```
Algorithm GreedyCoinChange(denominations, amount)
result ← []
for each denomination d in descending order do
    while amount ≥ d do
        result.append(d)
        amount ← amount − d
    end while
end for
return result
```

Solution: $67 = 25 + 25 + 10 + 5 + 1 + 1$ (6 coins)

Divide and Conquer: Methodology

Definition

Divide and Conquer breaks a problem into smaller subproblems, solves them recursively, then combines solutions.

similar

Three-Step Process

1. **Divide:** Break problem into smaller subproblems
2. **Conquer:** Solve subproblems recursively
3. **Combine:** Merge solutions to get final answer

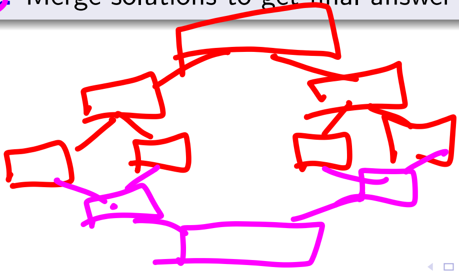
Divide and Conquer: Methodology

Definition

Divide and Conquer breaks a problem into smaller subproblems, solves them recursively, then combines solutions.

Three-Step Process

1. **Divide:** Break problem into smaller subproblems 1
2. **Conquer:** Solve subproblems recursively 2
3. **Combine:** Merge solutions to get final answer



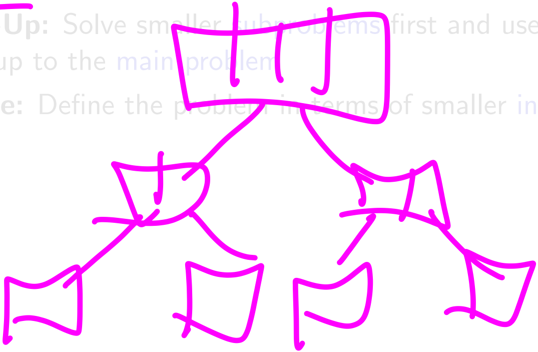
solved

Divide and Conquer: Algorithm Template

```
Algorithm DivideAndConquer(problem)
if problem is small enough then
    return SolveDirectly(problem)
else
    subproblems ← Divide(problem)
    for each subproblem do
        solution ← DivideAndConquer(subproblem)
    end for
    return Combine(solutions)
end if
```

Problem Decomposition Strategies

- **Top-Down:** Start with the main problem and break it down into subproblems
- **Bottom-Up:** Solve smaller subproblems first and use their solutions to build up to the main problem
- **Recursive:** Define the problem in terms of smaller instances of itself



Problem Decomposition Strategies

- **Top-Down:** Start with the main problem and break it down into subproblems
- **Bottom-Up:** Solve smaller subproblems first and use their solutions to build up to the main problem
- **Recursive:** Define the problem in terms of smaller instances of itself

Problem Decomposition Strategies

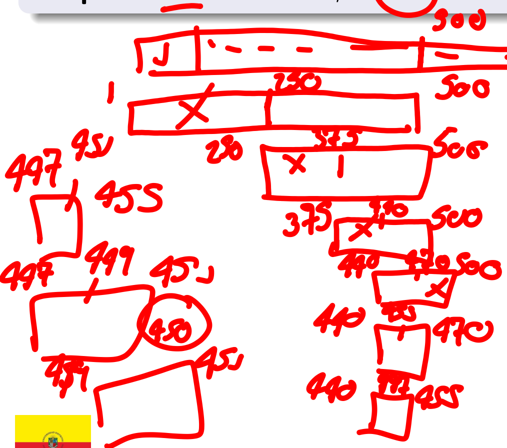
- **Top-Down:** Start with the main problem and break it down into subproblems
- **Bottom-Up:** Solve smaller subproblems first and use their solutions to build up to the main problem
- **Recursive:** Define the problem in terms of smaller instances of itself

Study Case: Binary Search

Problem

Input: Sorted array $A[1..n]$ and search value x

Output: Index of x in A, or -1 if not found



$2^{10} \sim 1000$

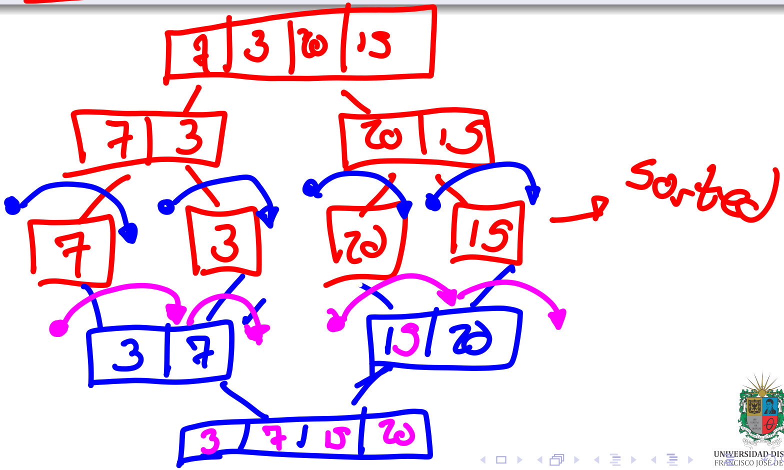
$2^{20} \sim 1000000$

7 3 2 1

Study Case: Merge Sort Approach

Sorting Problem

Input: Array $A[1..n]$ of comparable elements
Output: Array sorted in ascending order



UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Demo Time! Clean the House Exercise

$$J = \left(\frac{y_1 - y_2}{x_1 - x_2}\right) \rightarrow (x_1 - x_2) = (y_1 - y_2)$$

Problem

You need to clean your house efficiently. You have different rooms and different cleaning tasks.

Greedy Strategy: Always clean the dirtiest room first

1. Assess dirt level of all rooms
2. Choose room with highest dirt level
3. Clean that room completely
4. Repeat until all rooms clean

Advantage: Maximum immediate impact

Divide & Conquer Strategy: Split house systematically

1. Divide house into sections (floors/wings)
2. Recursively clean each section
3. Within each room, divide into areas
4. Clean areas systematically

Advantage: Systematic coverage

Queen $\rightarrow$ cases?

$$\text{Queen}((x_1,y_1),(x_2,y_2)) = \begin{cases} 0 & \text{si } x_1 = x_2 \text{ and } y_1 = y_2 \\ 1 & \text{si misma Fila } x_1 = x_0 \text{ ó misma col. } y_1 = y_2 \text{ ó } abs(x_1 - x_2) = abs(y_1 - y_2) \\ 2 & \text{otherwise} \end{cases}$$

Demo Time! Clean the House Exercise

Problem

You need to clean your house efficiently. You have different rooms and different cleaning tasks.

Greedy Strategy: Always clean the dirtiest room first

1. Assess dirt level of all rooms
2. Choose room with highest dirt level
3. Clean that room completely
4. Repeat until all rooms clean

Advantage: Maximum immediate impact

Divide & Conquer Strategy: Split house systematically

1. Divide house into sections (floors/wings)
2. Recursively clean each section
3. Within each room, divide into areas
4. Clean areas systematically

Advantage: Systematic coverage

Exercise: Knapsack Problem

Problem Statement

Given: Knapsack capacity W, items with weights w_i and values v_i
Goal: Maximize total value without exceeding weight capacity

array → sorter desc (g * 1)
sum first n items:
return sum

Exercise: Knapsack Problem [Solution]

0/1 Knapsack (Greedy Approach)

Cannot take fractions - either take entire item or leave it

- **Strategy:** Sort by value/weight ratio, take whole items
- **Optimal:** No! (Greedy doesn't guarantee optimal solution)

Sort items by v_i/w_i descending
for each item i **do**
  if $w_i \leq$ remaining capacity **then**
    Take entire item
    Update remaining capacity
  end if
end for

Brute Force: Exhaustive Search

Definition

Brute force algorithms try all possible solutions until finding the correct one.

Balotto /

45 44 43 42 41 40

43!

39!

Characteristics:

- **Exhaustive:** Examines every possibility
- **Guaranteed:** Always finds optimal solution (if exists)
- **Expensive:** Often exponential time complexity
- **Simple:** Easy to understand and implement

Brute Force: Exhaustive Search

Definition

Brute force algorithms try all possible solutions until finding the correct one.

Characteristics:

- **Exhaustive:** Examines every possibility
- **Guaranteed:** Always finds optimal solution (if exists)
- **Expensive:** Often exponential time complexity
- **Simple:** Easy to understand and implement

resources

negative

Brute Force: When to Use It

1. Problem size is small — tiny small
2. No efficient algorithm is known
3. Correctness is more important than efficiency
4. As baseline for comparing other algorithms
5. When optimization overhead exceeds brute force cost

Brute Force
D&C
Greedy

mem

Study Case: Password Cracking

Problem

Given: Encrypted password hash (1) and character set (2)

Goal: Find the original password

3

cannot

a → [aaa,
b   aab,
c   aac,
d   aad,
⋮   aae,
z   ⋮
      aaz]

recognize?

Study Case: Password Cracking [Solution]

Algorithm BRUTEFORCEPASSWORD($hash$, $maxlength$)
  for $length = 1$ to $maxlength$ **do**
    for each possible string s of given $length$ **do**
      if HASH(s) $= hash$ **then**
        return s // *Password found!*
      end if
    end for
  end for
  return *null* // *Password not found*

2) for ~

recursive?

Character set: [a-z] (26 characters)
Password length: 8 characters
Total attempts: $26^8 \approx 208$ billion
Time estimate: Weeks on single computer!

Study Case: Password Cracking [Solution]

```
Algorithm BruteForcePassword(hash, maxlength)
for length = 1 to maxlength do
    for each possible string s of given length do
        if Hash(s) = hash then
            return s // Password found!
        end if
    end for
end for
return null // Password not found
```

DDOS

max 7 attemps

Character set: [a-z] (26 characters)
Password length: 8 characters
Total attempts: $26^8 \approx 208$ billion
Time estimate: Weeks on single computer!

Backtracking: Systematic Trial and Error

Definition

Backtracking explores solution space systematically, **abandoning partial solutions** that cannot lead to valid solutions.

Backtracking Strategy

1. **Choose:** Make a choice from available options
2. **Explore:** Recursively explore consequences
3. **Unchoose:** If path fails, backtrack and try another

Backtracking: Systematic Trial and Error

Definition

Backtracking explores solution space systematically, **abandoning partial solutions** that cannot lead to valid solutions.

Backtracking Strategy

1. **Choose:** Make a choice from available options
2. **Explore:** Recursively explore consequences
3. **Unchoose:** If path fails, backtrack and try another

Backtracking: Algorithm

```
Algorithm BACKTRACK(partial_solution)
if ISCOMPLETE(partial_solution) then
    return partial_solution
end if
for each possible choice c do
    if ISVALID(partial_solution, c) then
        partial_solution.add(c)
        result ← BACKTRACK(partial_solution)
        if result ≠ null then
            return result
        end if
        partial_solution.remove(c) // Backtrack!
    end if
end for
return null
```

Study Case: N-Queens Problem

Problem

Place N queens on an $N \times N$ chessboard such that no two queens attack each other.

Study Case: N-Queens Problem [Solution]

```
Algorithm SolveNQueens(board, row)
if row = N then
    return board // Solution found!
end if
for each column col in 0 to N − 1 do
    if IsSafe(board, row, col) then
        board[row][col] ← Q // Place queen
        result ← SolveNQueens(board, row + 1)
        if result ≠ null then
            return result
        end if
        board[row][col] ← . // Remove queen (backtrack)
    end if
end for
return null // No solution found
```

Implementation Patterns for Backtracking

- **Recursive Structure:** Backtracking algorithms are typically implemented using recursion.
- **State Representation:** Maintain a representation of the current state (e.g., partial solution).
- **Choice Points:** Identify points where decisions are made (choices to explore).
- **Validity Checks:** Implement checks to determine if the current state is valid.
- **Base Case:** Define a base case for when a complete solution is found.
- **Backtracking Mechanism:** Ensure that the algorithm can revert to previous states when necessary.

Implementation Patterns for Backtracking

- **Recursive Structure:** Backtracking algorithms are typically implemented using recursion.
- **State Representation:** Maintain a representation of the current state (e.g., partial solution).
- **Choice Points:** Identify points where decisions are made (choices to explore).
- **Validity Checks:** Implement checks to determine if the current state is valid.
- **Base Case:** Define a base case for when a complete solution is found.
- **Backtracking Mechanism:** Ensure that the algorithm can revert to previous states when necessary.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Implementation Patterns for Backtracking

- **Recursive Structure:** Backtracking algorithms are typically implemented using recursion.
- **State Representation:** Maintain a representation of the current state (e.g., partial solution).
- **Choice Points:** Identify points where decisions are made (choices to explore).
- **Validity Checks:** Implement checks to determine if the current state is valid.
- **Base Case:** Define a base case for when a complete solution is found.
- **Backtracking Mechanism:** Ensure that the algorithm can revert to previous states when necessary.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Implementation Patterns for Backtracking

- **Recursive Structure:** Backtracking algorithms are typically implemented using recursion.
- **State Representation:** Maintain a representation of the current state (e.g., partial solution).
- **Choice Points:** Identify points where decisions are made (choices to explore).
- **Validity Checks:** Implement checks to determine if the current state is valid.
- **Base Case:** Define a base case for when a complete solution is found.
- **Backtracking Mechanism:** Ensure that the algorithm can revert to previous states when necessary.

Implementation Patterns for Backtracking

- **Recursive Structure:** Backtracking algorithms are typically implemented using recursion.
- **State Representation:** Maintain a representation of the current state (e.g., partial solution).
- **Choice Points:** Identify points where decisions are made (choices to explore).
- **Validity Checks:** Implement checks to determine if the current state is valid.
- **Base Case:** Define a base case for when a complete solution is found.
- **Backtracking Mechanism:** Ensure that the algorithm can revert to previous states when necessary.

Implementation Patterns for Backtracking

- **Recursive Structure:** Backtracking algorithms are typically implemented using recursion.
- **State Representation:** Maintain a representation of the current state (e.g., partial solution).
- **Choice Points:** Identify points where decisions are made (choices to explore).
- **Validity Checks:** Implement checks to determine if the current state is valid.
- **Base Case:** Define a base case for when a complete solution is found.
- **Backtracking Mechanism:** Ensure that the algorithm can revert to previous states when necessary.

Summary: Algorithm Types

Key Takeaways

- **Greedy:** Make locally optimal choices (works when greedy choice property holds)
- **Divide & Conquer:** Break into subproblems, solve recursively, combine solutions
- **Brute Force:** Try all possibilities (guarantees correctness but expensive)
- **Backtracking:** Systematic search with pruning (intelligent brute force)

Summary: Algorithm Types

Algorithm Selection Guide

- **Small problem size?** → Consider brute force
- **Optimization problem with greedy choice property?** → Try greedy
- **Problem naturally divides into subproblems?** → Use divide & conquer
- **Constraint satisfaction or search problem?** → Apply backtracking

Outline

Thanks!

Questions?

Repo: *https://github.com/EngAndres/ud-public/tree/main/courses/computer-sciences-i*