

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



Outline

1 Introduction to Algorithms



2 Algorithm Design



3 Algorithm Types



Outline

1 Introduction to Algorithms

2 Algorithm Design

3 Algorithm Types



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:** Performed mechanically



Remember

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



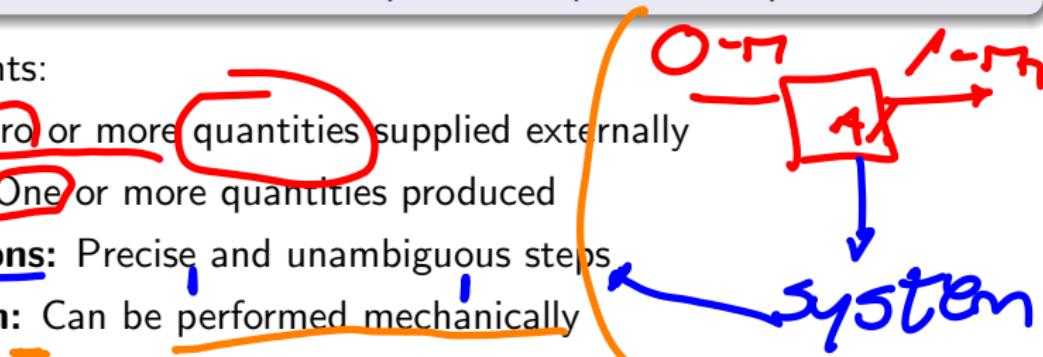
What is an Algorithm?

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



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

Remember

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

design



Algorithm Characteristics

Essential Properties

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

⑤

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)

Value → Constant

Value

Algorithm vs. Program vs. Process

Algorithm *design*

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

Program *code*

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

Process *FUJ*

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

Works like

Example

terminate 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

~~terminate~~ **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

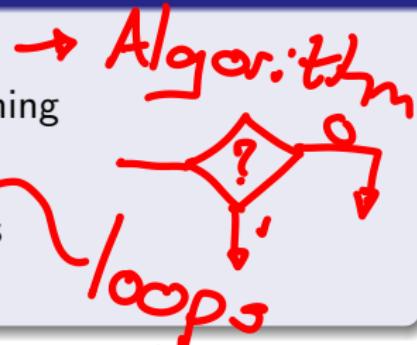
if replace

(ctrl + Alt + Söp) — top

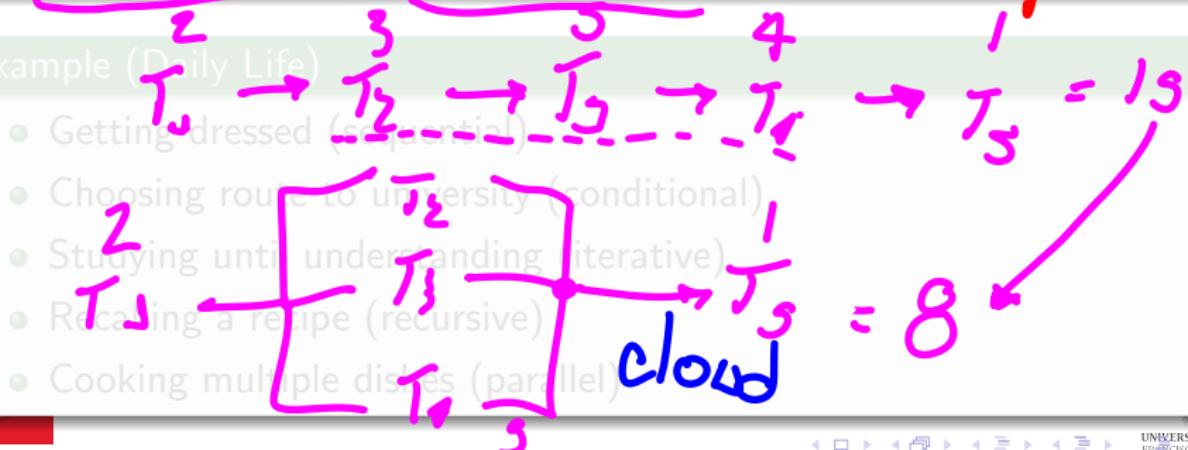
Types of Algorithmic Thinking

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)

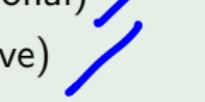


Types of Algorithmic Thinking

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

pre-requisite
Let
Define f as a funct --



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

Alg Input : 5

return x

Exercise Find max element in an Array

DIFF = 0.005

Write an algorithm that finds the maximum element in an array of integers.

Consider: What if the array is empty? What about duplicate maximums?

$x = 5/2$

DOE

$$x_{\text{next}} = (\frac{1}{2})(x + (S/x))$$

$$\text{dF_temp} = \text{abs}(x - x_{\text{next}})$$

} while ($\text{dF_temp} > \text{DIFF}$)



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

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: $a \rightarrow [a_0, a_1, \dots, a_n]$

if length.a == 0:
return "No"
max = a[0]

for i in range(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 ~~x~~
- Verify all algorithm characteristics

- Wake - About sound time?
- Take a shower - Time
- Get some breakfast - ~~fast~~ time
- Put on dress - weather schedule
- Toothbrush
- Pick Up bag - Schedule
- Go out - time



Outline

1 Introduction to Algorithms

2 Algorithm Design

3 Algorithm Types



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 [I]

Mathematical Notation

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

$$x + y$$

Why Mathematical Notation?

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

$$\begin{array}{rcl} 2 \times 3 & = & 6 \\ 3 \times 3 & = & 9 \\ 4 \times 3 & = & 12 \end{array}$$



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) \rightarrow map
 - $\exists x \in S : Q(x)$ (there exists an element in S with property Q)



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:

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

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



Converting Mathematical Formulas to Pseudocode

Mathematical Expression

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

(The term i^2 is circled in red.)

Another Example

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

```

1: sum ← 0
2: for i = 1 to n do
3:   sum ← sum + i2
4: end for
5: formula ←  $\frac{n(n+1)(2n+1)}{6}$ 
6: if sum = formula then
7:   return true
8: else
9:   return false
10: end if

```

```

1: result ← 1
2: term ← 1
3: n ← 1
4: while |term| > ε do
5:   term ←  $\frac{term \cdot x}{n}$ 
6:   result ← result + term
7:   n ← n + 1
8: end while
9: return result

```



Converting Mathematical Formulas to Pseudocode

Mathematical Expression

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

```

1: sum ← 0
2: for i = 1 to n do
3:   sum ← sum + i2
4: end for
5: formula ←  $\frac{n(n+1)(2n+1)}{6}$ 
6: if sum = formula then
7:   return true
8: else
9:   return false
10: end if

```

Another Example

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

```

1: result ← 1
2: term ← 1
3: n ← 1
4: while |term| > ε do
5:   term ←  $\frac{term \cdot x}{n}$ 
6:   result ← result + term
7:   n ← n + 1
8: end while
9: return result

```

Converting Mathematical Formulas to Pseudocode

Mathematical Expression

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

Another Example

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

Difficult

```

1: sum ← 0
2: for  $i = 1$  to  $n$  do
3:   sum ← sum +  $i^2$ 
4: end for
5: formula ←  $\frac{n(n+1)(2n+1)}{6}$ 
6: if sum = formula then
7:   return true
8: else
9:   return false
10: end if

```

Diagram showing the conversion of the mathematical formula for e^x into pseudocode. A pink circle highlights the term x^n , and a pink arrow points from it to the pseudocode line "term ← term · x". Another pink arrow points from the label "difficult" to the pseudocode line "while |term| > ε do".

```

1: result ← 1
2: term ← 1
3: n ← 1
4: while |term| >  $\epsilon$  do
5:   term ←  $\frac{term \cdot x}{n}$ 
6:   result ← result + term
7:   n ← n + 1
8: end while
9: return result

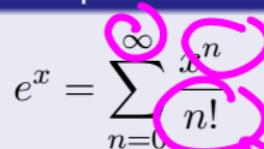
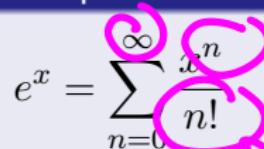
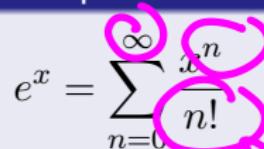
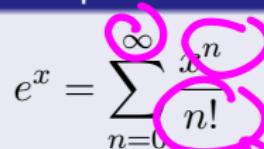
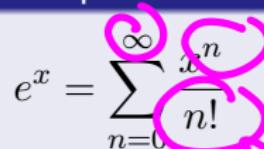
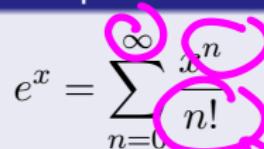
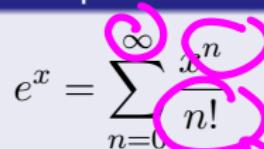
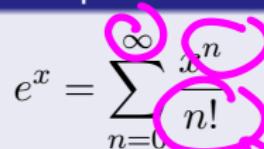
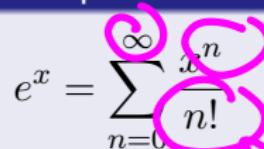
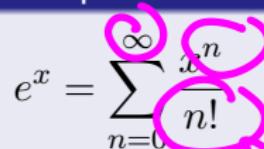
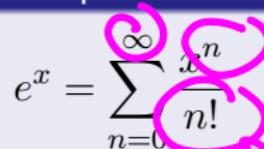
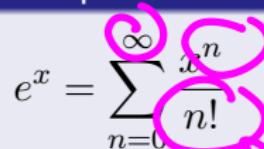
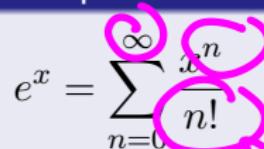
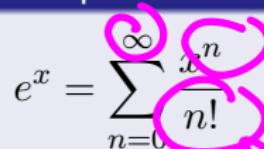
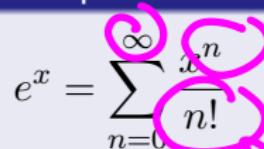
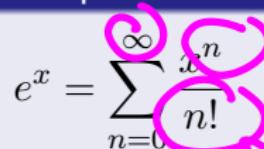
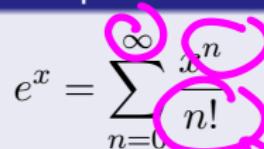
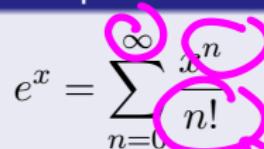
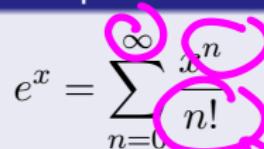
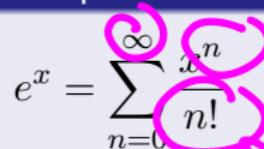
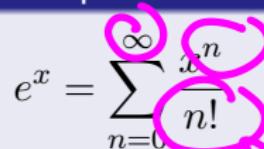
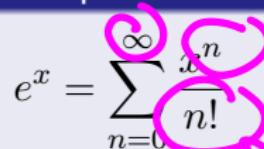
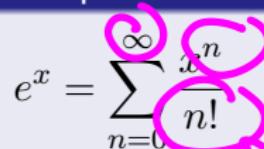
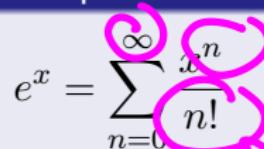
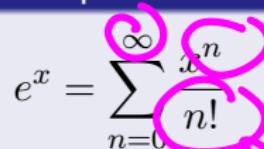
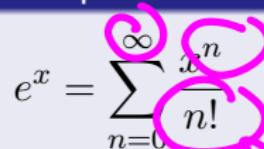
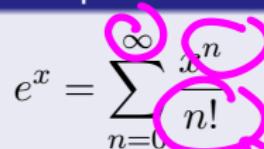
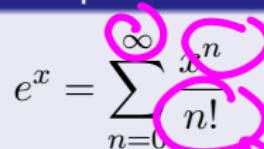
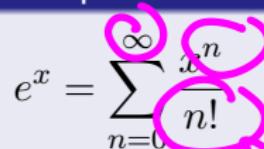
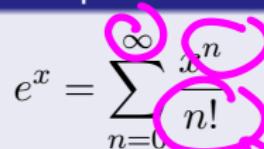
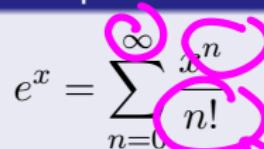
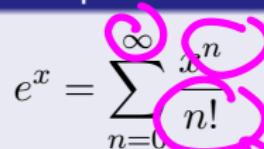
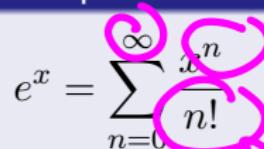
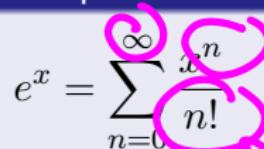
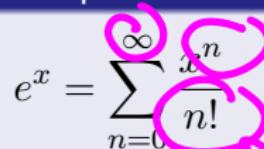
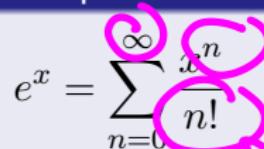
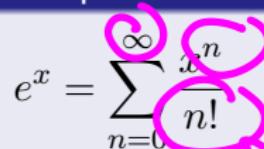
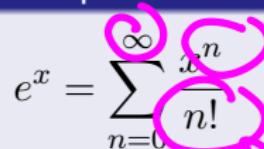
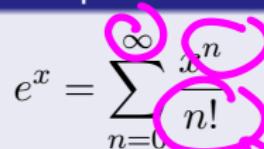
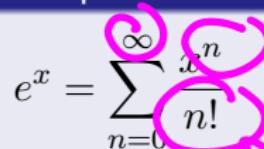
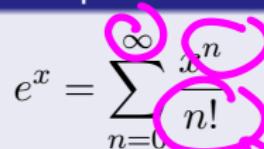
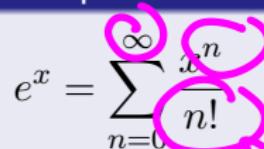
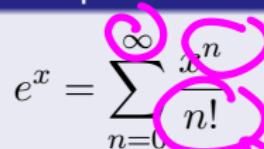
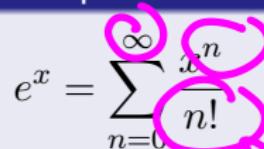
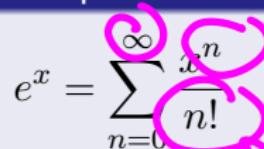
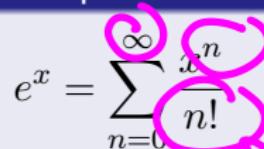
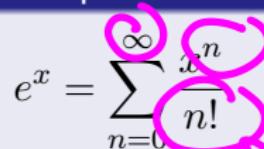
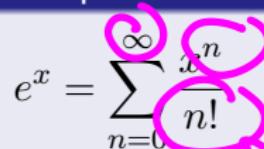
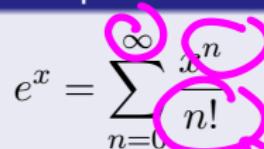
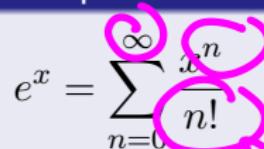
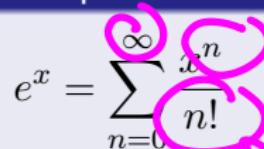
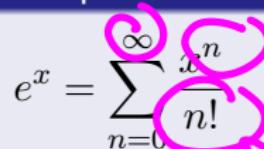
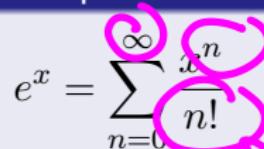
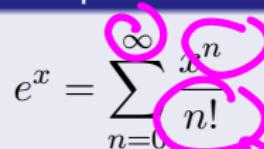
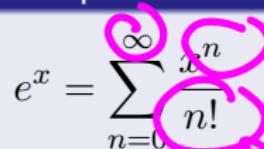
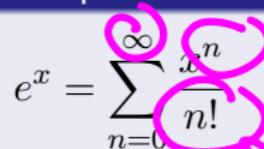
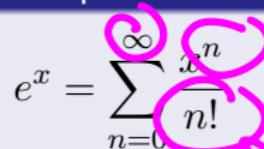
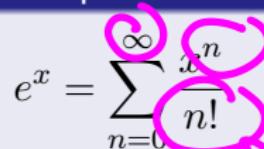
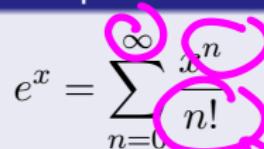
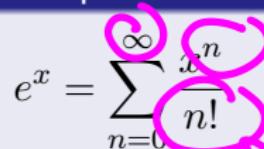
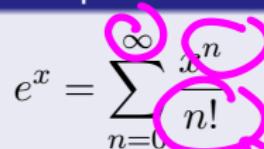
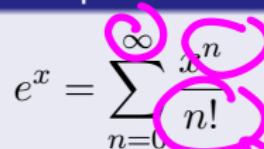
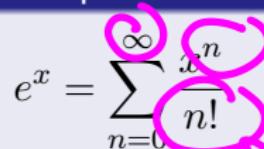
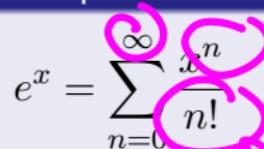
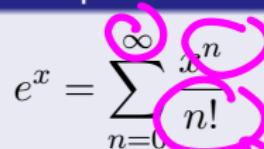
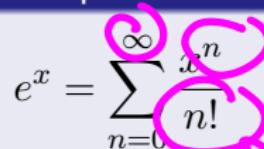
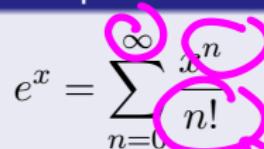
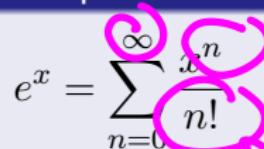
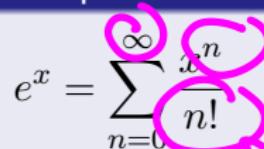
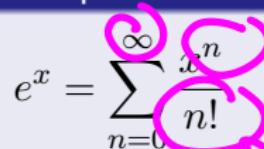
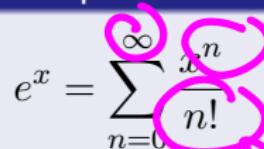
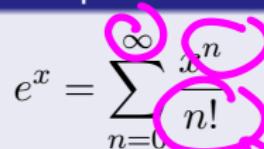
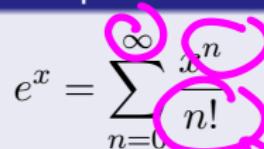
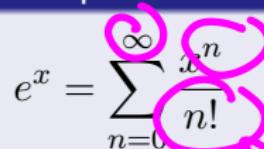
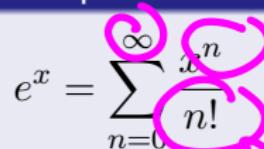
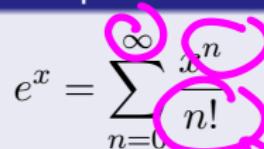
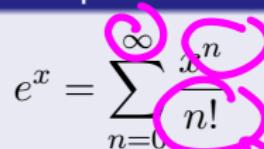
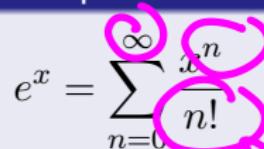
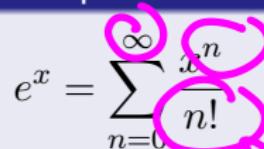
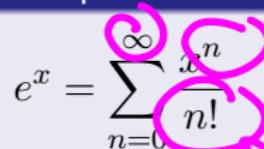
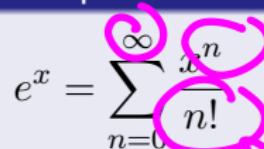
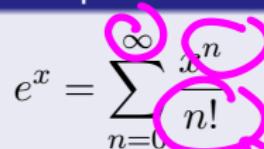
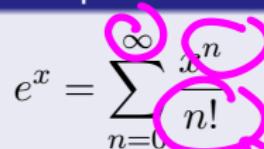
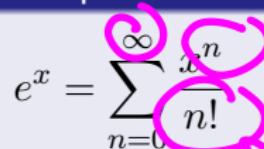
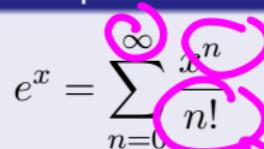
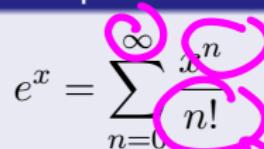
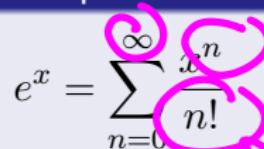
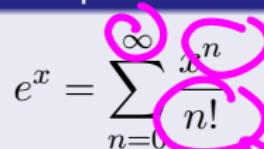
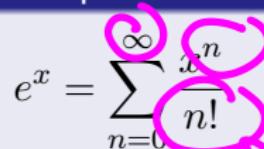
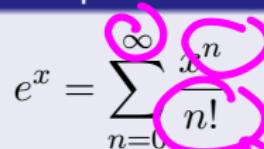
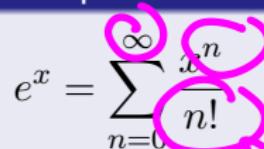
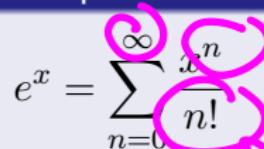
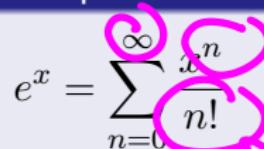
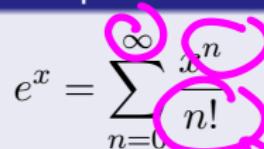
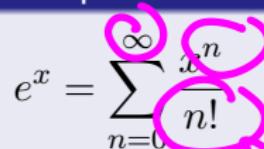
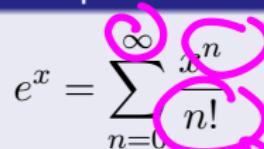
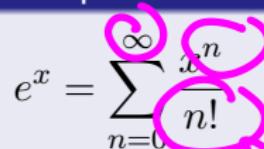
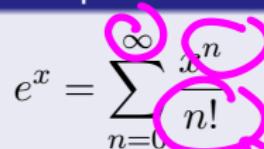
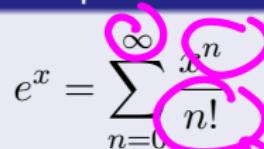
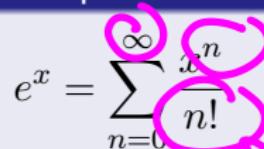
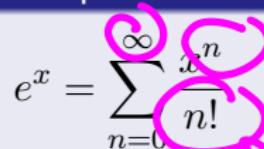
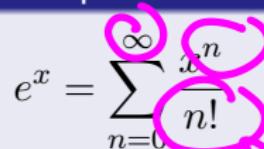
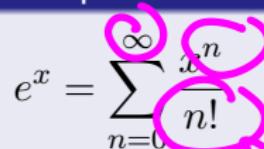
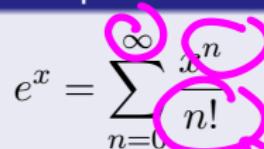
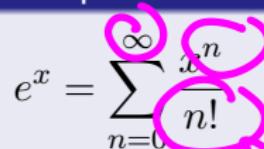
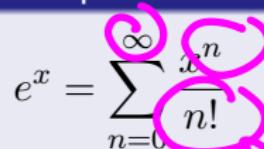
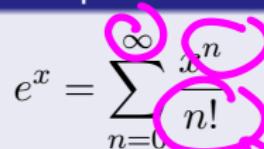
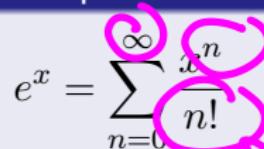
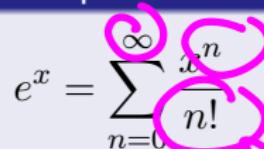
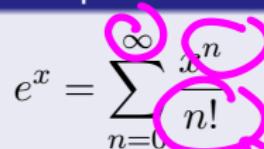
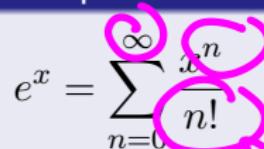
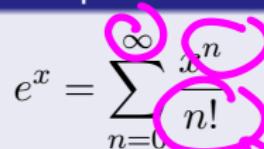
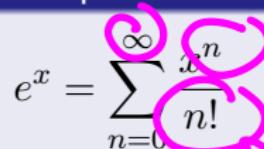
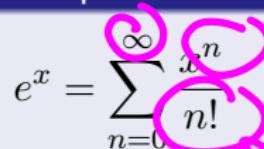
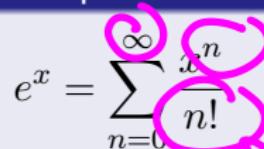
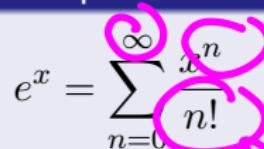
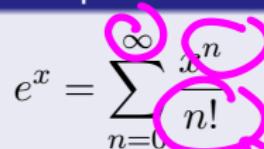
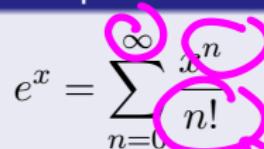
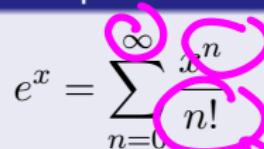
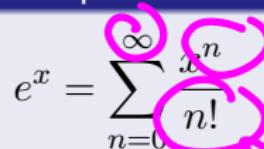
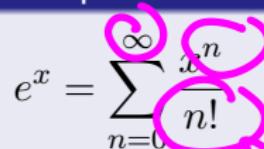
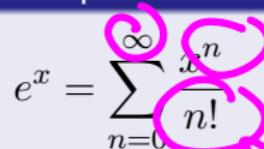
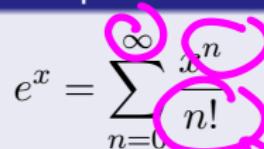
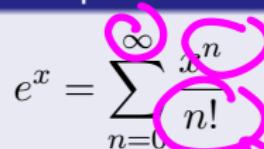
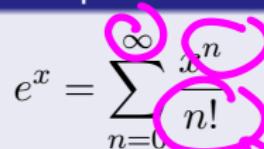
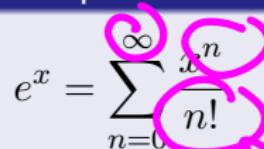
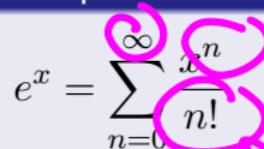
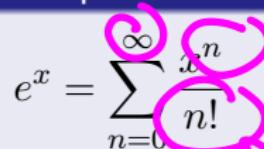
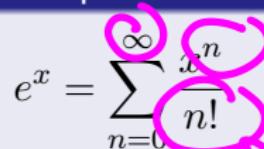
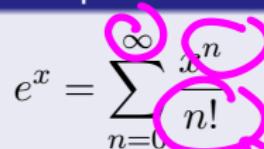
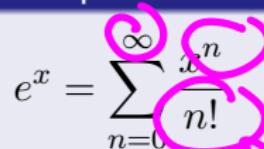
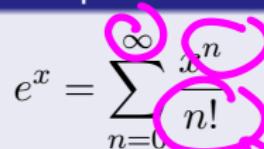
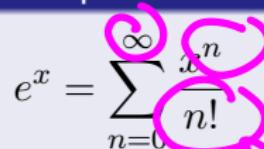
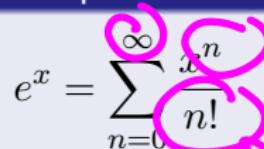
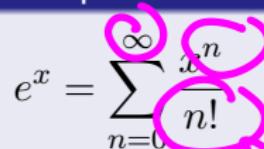
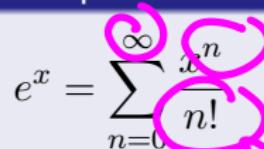
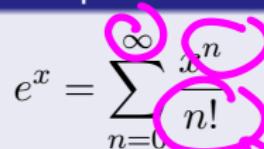
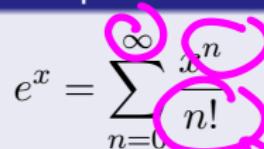
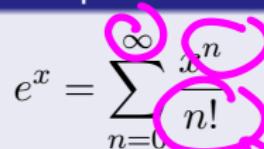
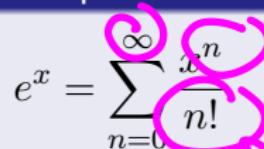
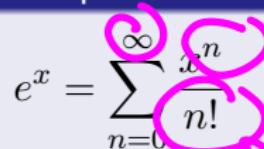
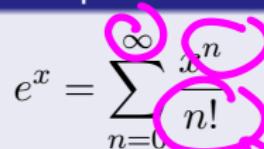
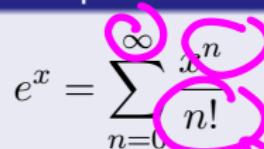
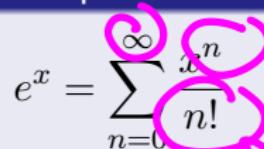
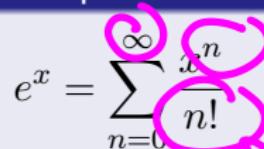
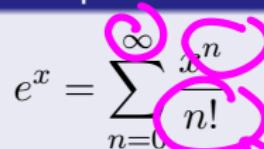
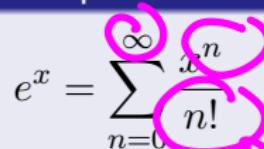
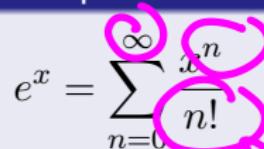
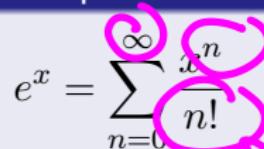
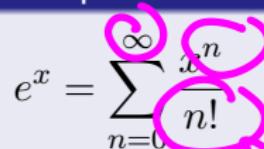
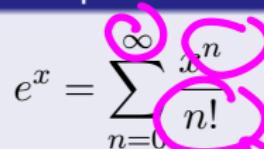
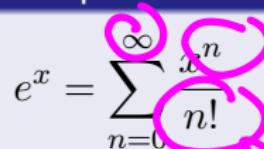
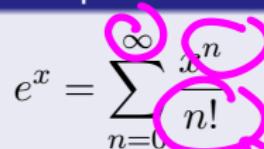
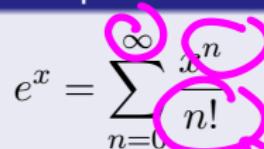
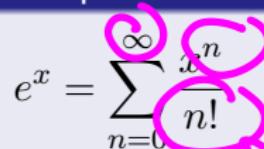
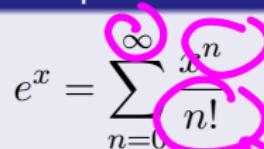
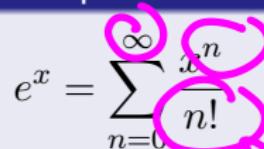
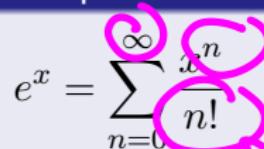
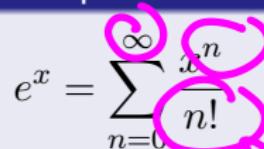
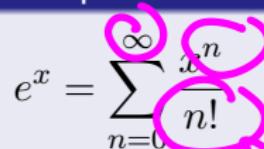
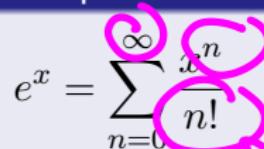
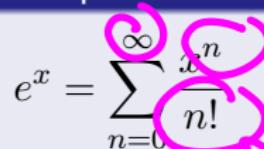
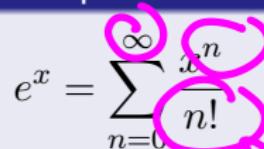
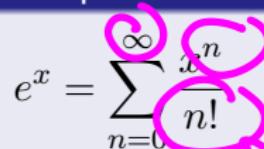
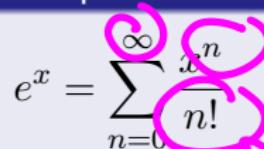
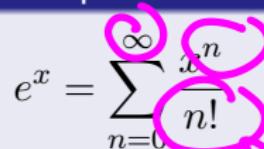
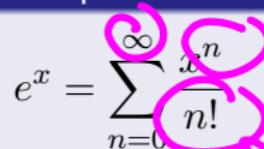
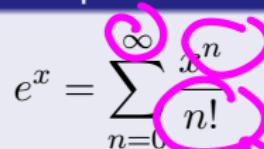
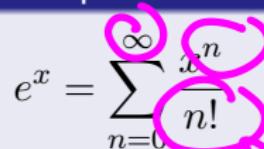
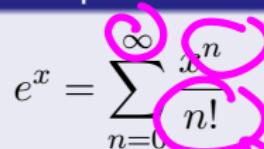
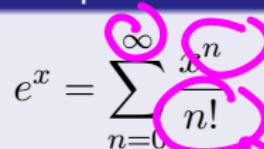
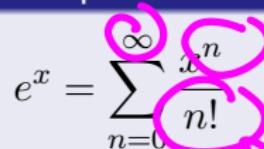
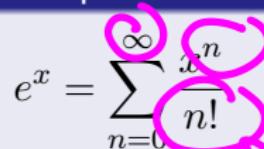
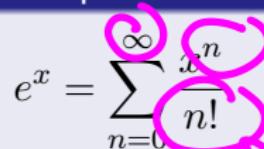
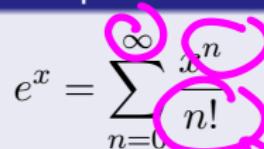
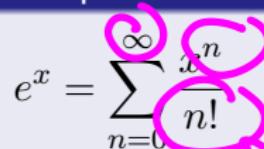
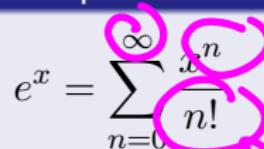
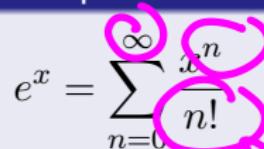
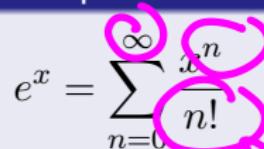
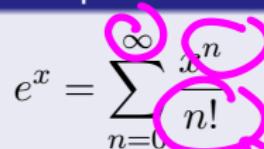
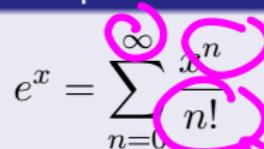
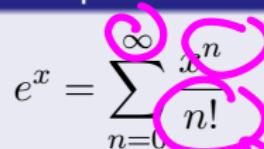
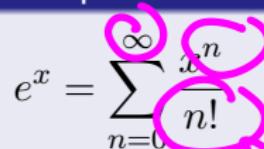
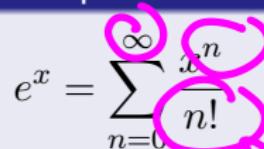
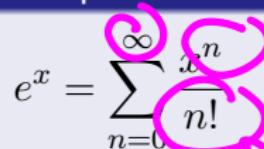
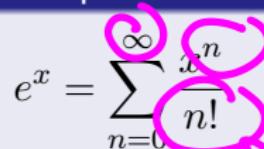
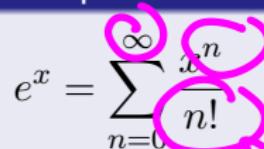
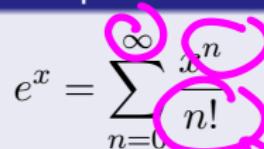
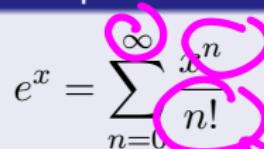
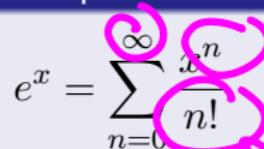
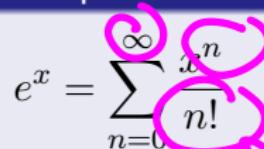
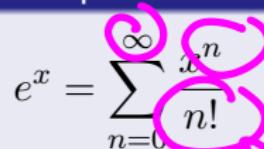
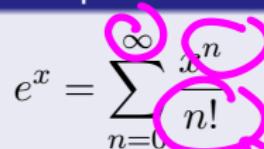
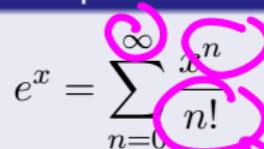
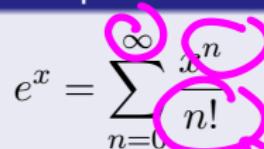
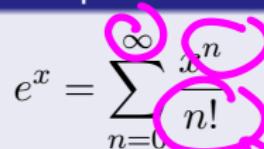
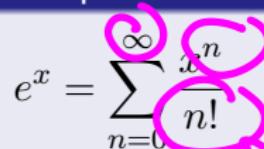
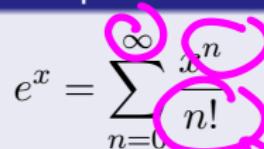
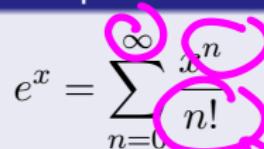
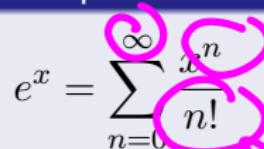
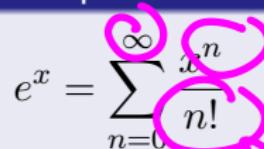
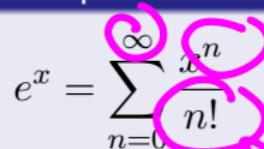
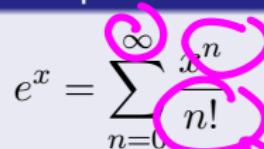
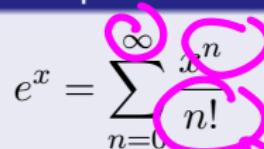
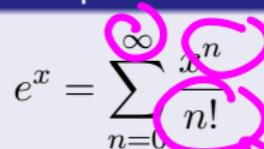
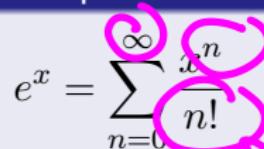
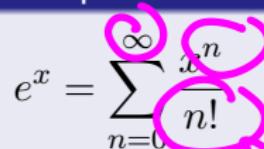
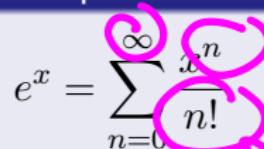
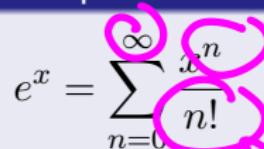
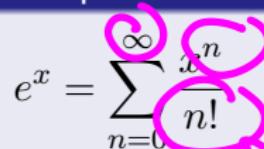
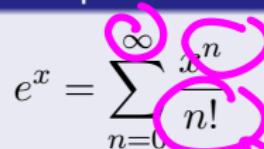
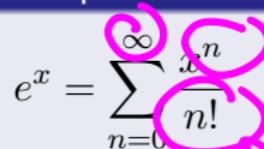
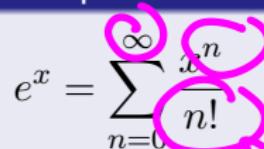
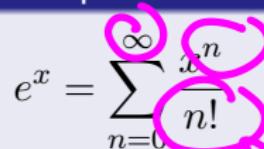
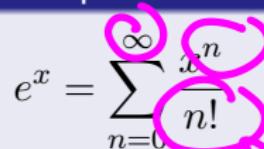
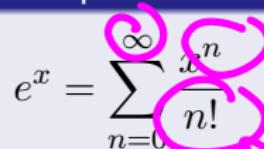
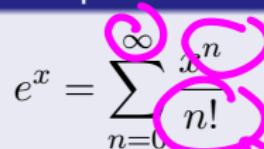
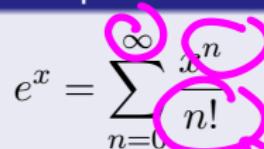
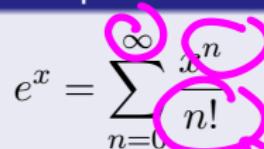
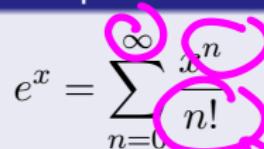
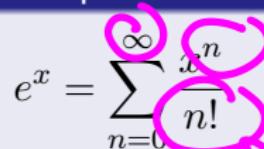
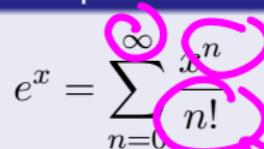
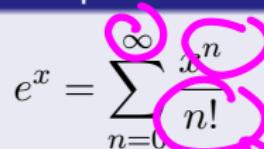
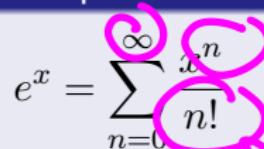
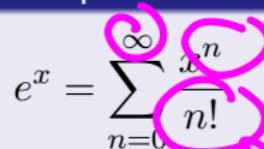
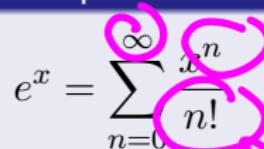
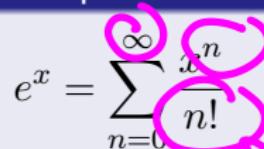
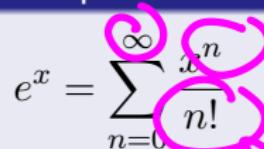
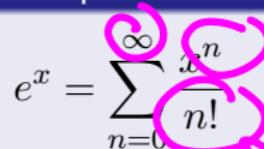
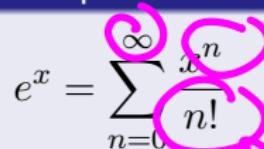
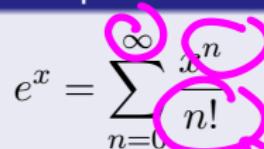
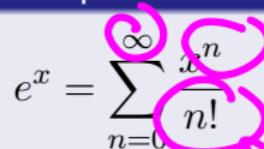
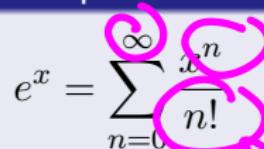
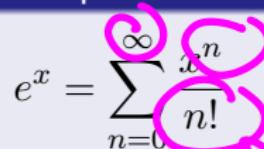
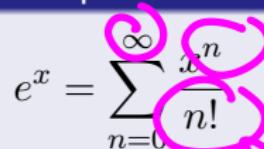
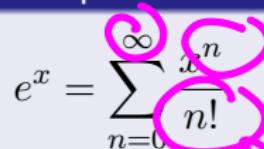
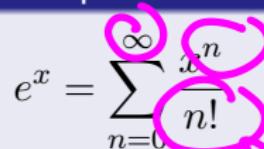
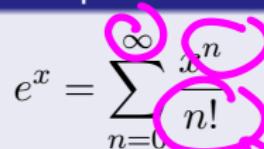
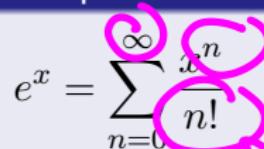
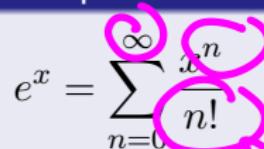
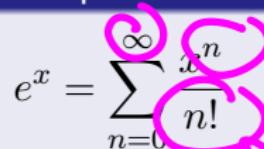
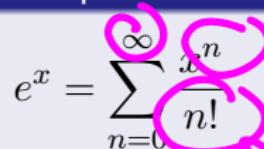
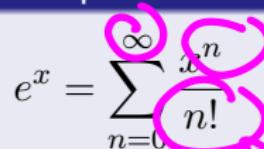
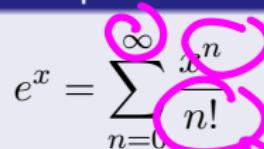
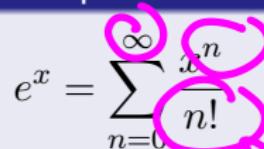
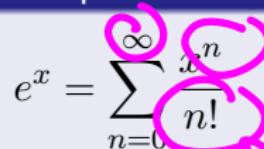
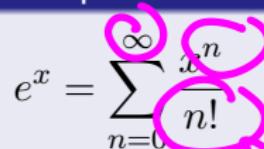
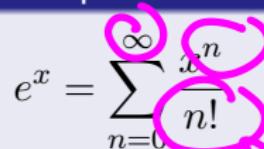
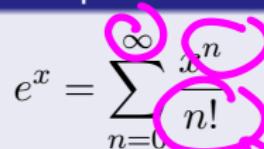
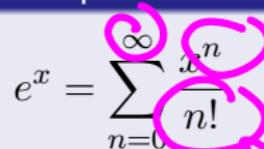
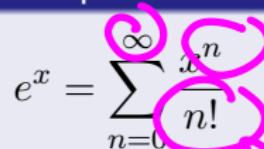
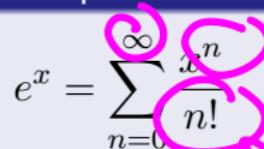
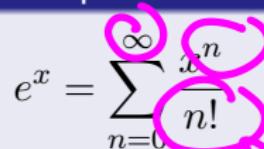
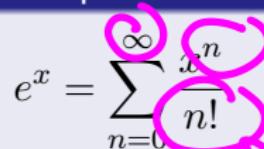
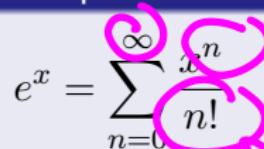
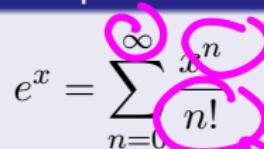
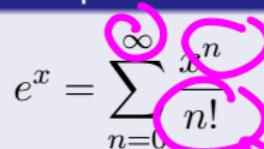
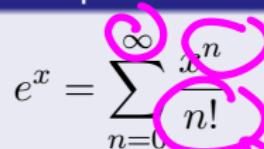
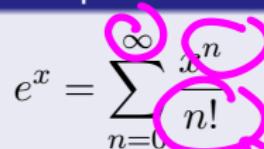
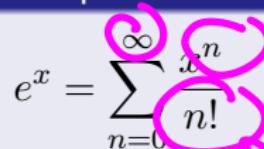
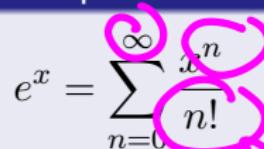
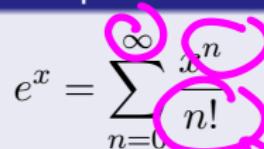
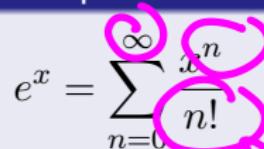
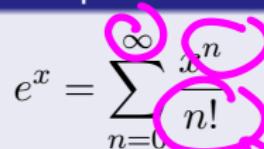
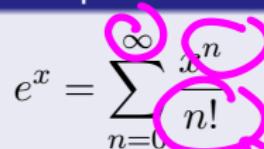
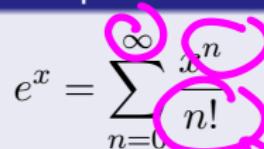
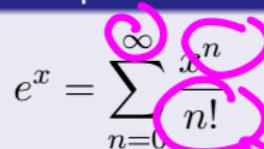
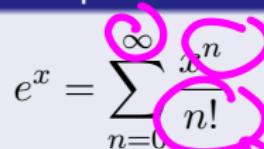
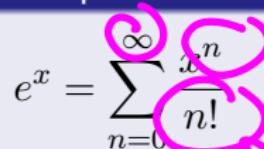
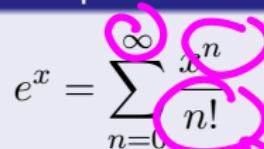
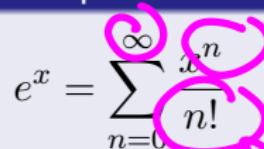
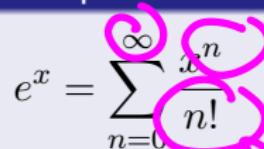
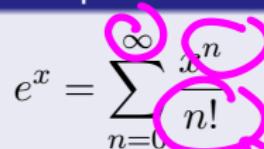
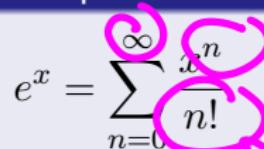
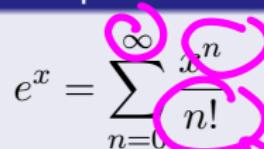
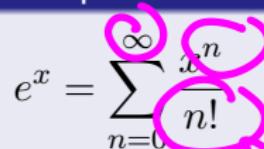
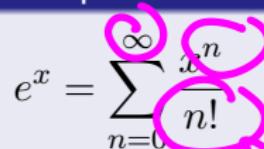
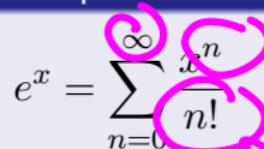
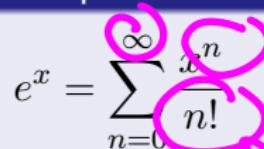
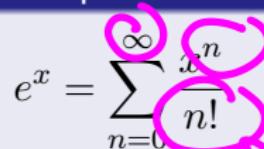
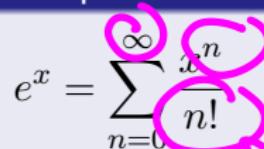
```

Converting Mathematical Formulas to Pseudocode

Mathematical Expression

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

Another Example

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

























































































































































































































































































<img alt="A hand-drawn pink circle highlights the term n! in the denominator of the

Study Case: Factorial Using Different Notations

Mathematical

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

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

$$\Sigma + + * *$$

Iterative Pseudocode

```

1: function FACTORIAL(n)
2: result  $\leftarrow$  1
3: for i = 1 to n do
4:   result  $\leftarrow$  result  $\times$  i
5: end for
6: return result
7: end function
```

RECURSIVE

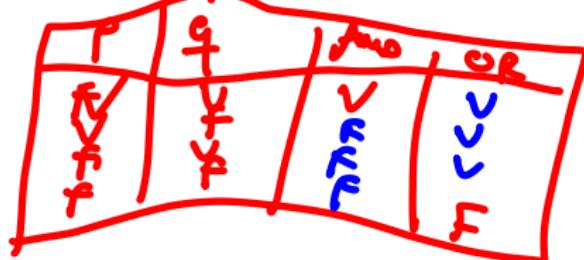


Pseudocode Standards and Conventions

~~Basic Elements~~

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

- and & or // not ! ~
comp & comp



Algorithm Documentation Best Practices

Essential Documentation Elements

- ① **Purpose:** What does the algorithm do?
- ② **Preconditions:** What must be true before execution?
- ③ **Postconditions:** What is guaranteed after execution?
- ④ **Parameters:** Input and output specifications
- ⑤ **Complexity:** Time and space requirements
- ⑥ **Examples:** Sample inputs and outputs

For $\rightarrow \eta$ steps

$\eta \rightarrow$ steps?

For
for $\} \rightarrow \eta^2$ steps

BeeCrowd



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 n th 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: FIBONACCI ITERATIVE(5) = 5



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 n th 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

n -th

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

Example: $\text{FIBONACCIITERATIVE}(5) = 5$

for $i = 0$ to $n-1$ do
 $f[i] = f[i-1] + f[i-2]$



Control Structures

Sequential

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

Execution: One after another

Traditional

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



Control Structures

Sequential

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

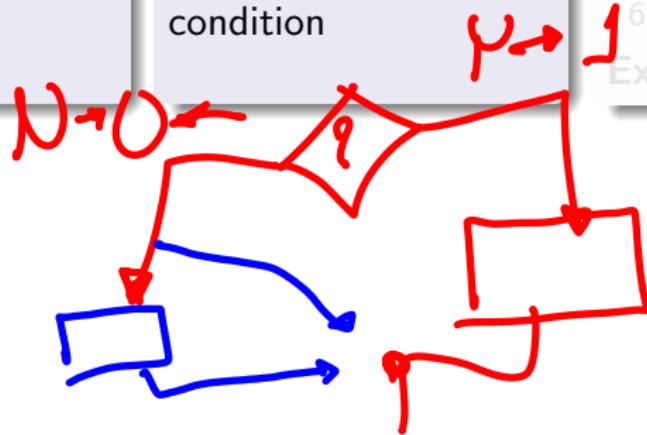
Execution: One after another

Conditional

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

**Execution:** Based on condition ( )

## Execution: Repeated



# 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



# Control Flow Concepts

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



# Study Case: Decision-Making Algorithms

## Grade Classification Algorithm

**Problem:** Convert numerical grade to letter grade

Decision Tree  
Range?  $1 - 100$

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

Range by letter?

A

B

C

D

E

100-90

89-80

79-70

69-60

lesión  
60

Key Concepts

- Mutually exclusive
- 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

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

else



### Key Concepts

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

at least "error"



## Outline

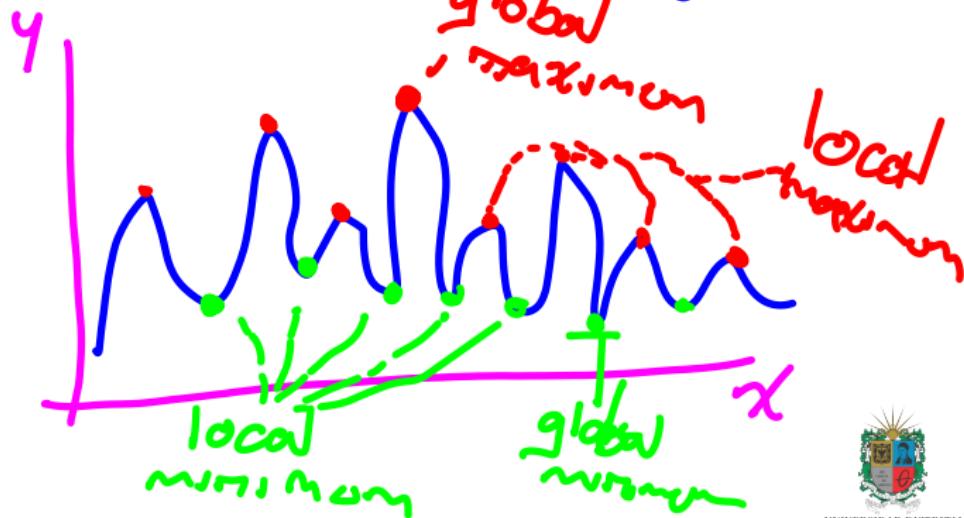
Optimization → objective function  
handle variable by  $x$   $\leftarrow \max_{m, M}$

- 1 Introduction to Algorithms

## 2 Algorithm Design

Max

- ### 3 Algorithm Types



# 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



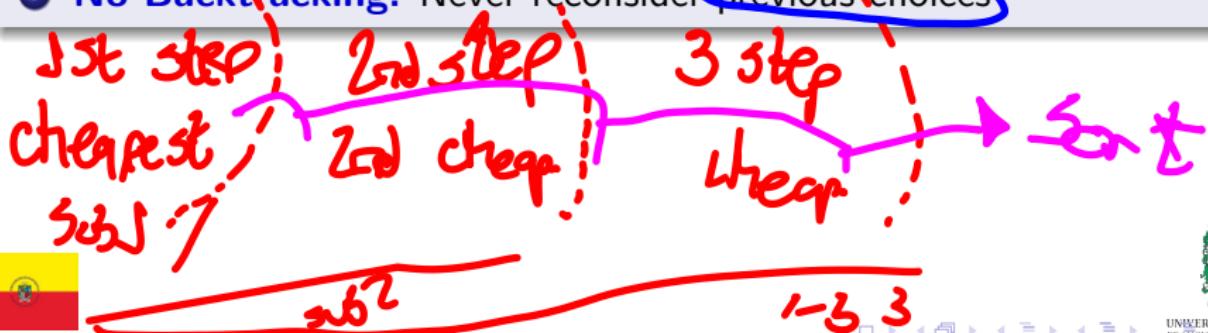
# Greedy Algorithms: Philosophy and Approach

## 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



## Greedy Algorithms: When to Use Them

## When Greedy Works.

- Activity selection
  - Minimum spanning trees
  - Huffman coding

# ~ ~ PATTERN ~

Traveling salesman

## • Graph Patterns --

## - P A T R E N -

11

• zip    • rar } PATTERN = J

~ ~ ~

J. Sort occurrences

↳ sort length



# Greedy Algorithms: When to Use Them

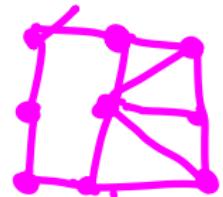
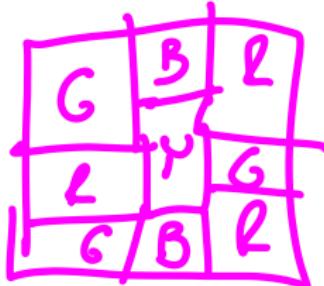
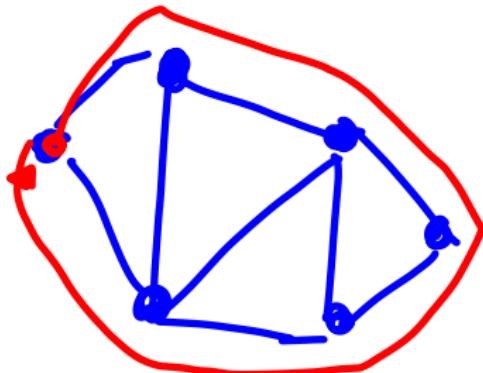
When Greedy Works:

- Activity selection
- Minimum spanning trees
- Huffman coding

When Greedy Fails:

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

2 shades



# Study Case: Coin Change Problem

## Problem Statement

**Given:** Coin denominations  $[d_1, d_2, \dots, d_k]$  and amount  $n$

**Goal:** Find minimum number of coins to make amount  $n$

Handwritten notes:  
 - Coin denominations: 50, 200, 1000, 100, 50  
 - Amounts:  $n_0 \rightarrow 1000?$ ,  $n_1 \rightarrow 500?$ ,  $n_2 \rightarrow 200?$   
 - An exclamation mark (!) is placed below the question mark of  $n_2$ .



# Study Case: Coin Change Problem [Solution]

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

```

1: Algorithm GREEDYCOINCHANGE(denominations, amount)
2: result ← []
3: for each denomination d in descending order do
4: while amount ≥ d do
5: result.append(d)
6: amount ← amount - d
7: end while
8: end for
9: return result

```

*not know* → *Sort*  
*integer*  
 $\text{coins}_i = (\text{amount} / d_i)$   
 $\text{amount} = \text{amount} \% d_i$

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

2 14 7 2 1 0 x



# Divide and Conquer Methodology

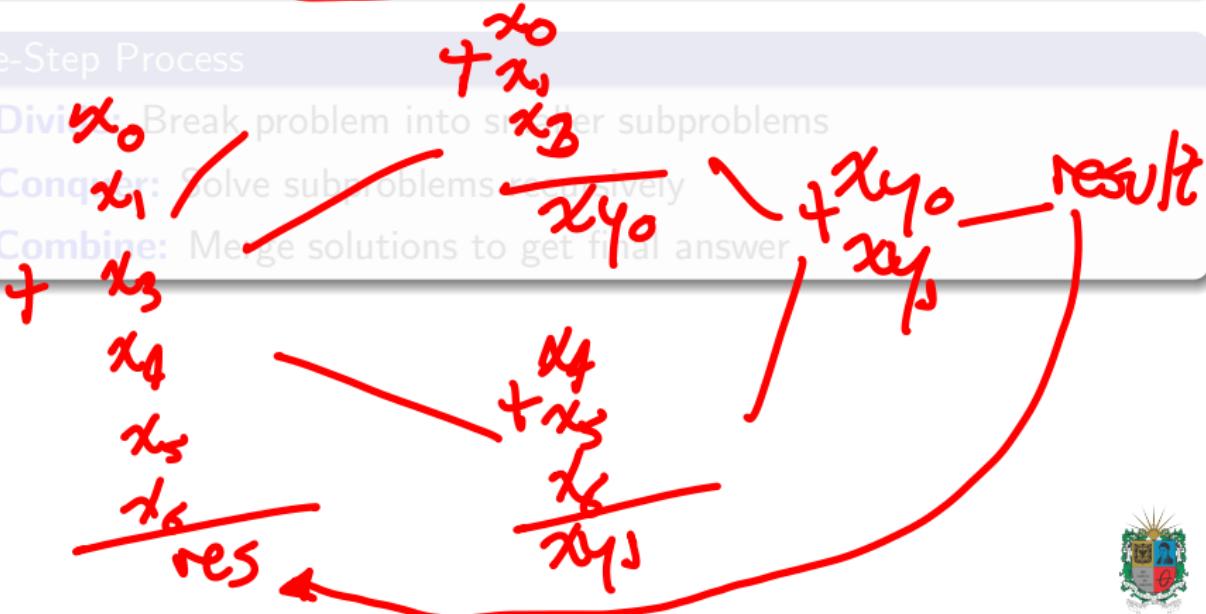
## Definition

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

same structure

## 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
- ② **Conquer:** Solve subproblems recursively
- ③ **Combine:** Merge solutions to get final answer



# Divide and Conquer: Algorithm Template

```
1: Algorithm DIVIDEANDCONQUER(problem)
2: if problem is small enough then
3: return SOLVEDIRECTLY(problem)
4: else
5: subproblems \leftarrow DIVIDE(problem)
6: for each subproblem do
7: solution \leftarrow DIVIDEANDCONQUER(subproblem)
8: end for
9: return COMBINE(solutions)
10: end if
```



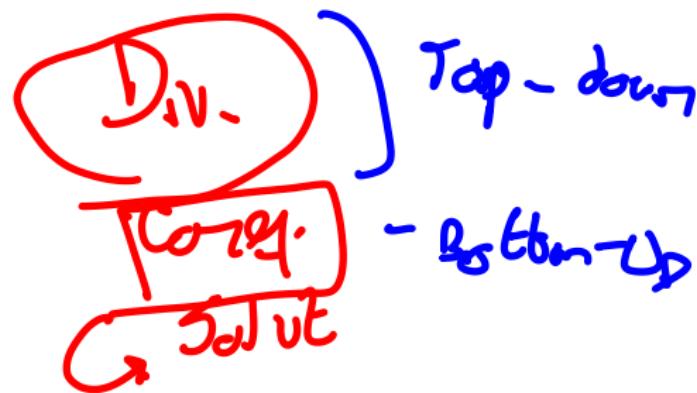
# Problem Decomposition Strategies

- **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

$$2^{10} = 1024$$

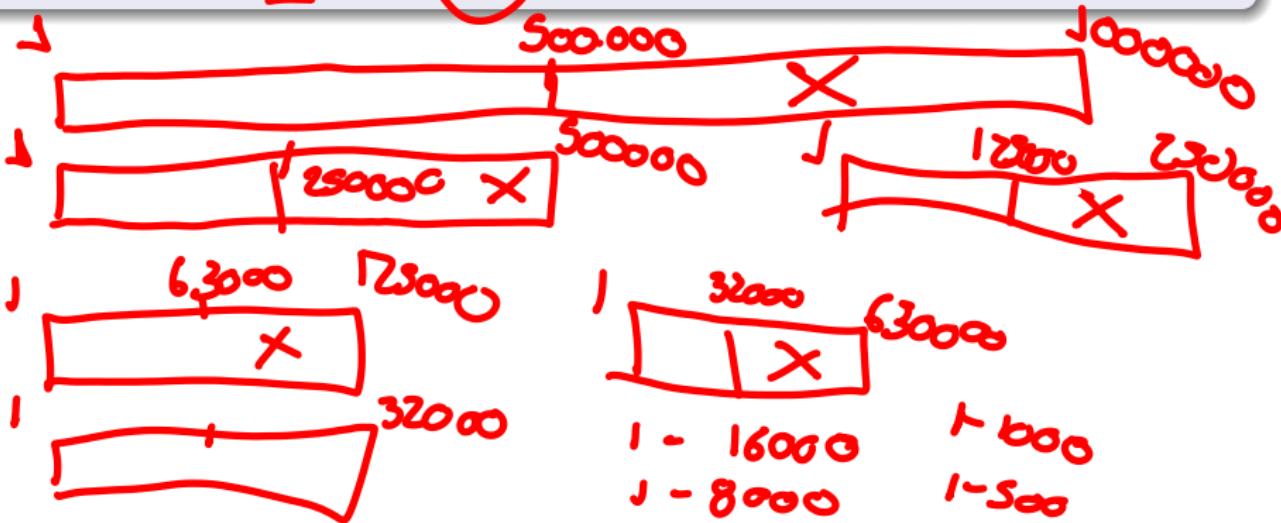
$$2^{30} \sim 10^9$$

## Problem

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

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

$$2^{30} \sim 10^9$$



|           |          |         |
|-----------|----------|---------|
| I - 16000 | J - 8000 | I - 500 |
| J - 8000  | I - 500  | I - 250 |
| I - 4000  | I - 250  | I - 125 |
| I - 2000  |          |         |

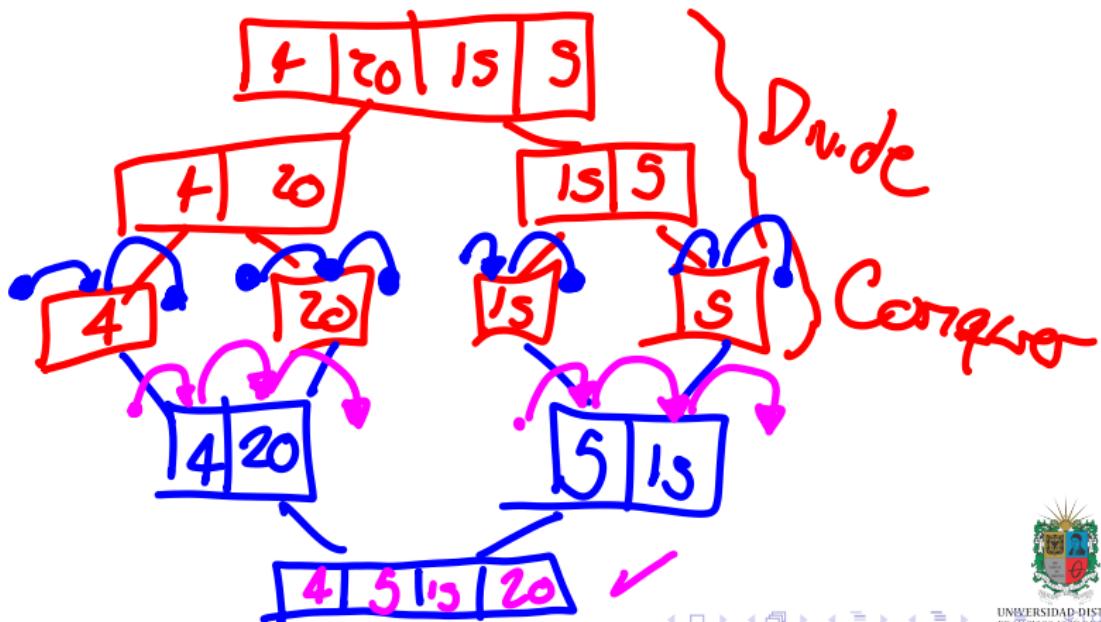


# Study Case: Merge Sort Approach

## Sorting Problem

**Input:** Array  $A[1..n]$  of comparable elements

**Output:** Array sorted in ascending order



# 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

else if ( $x_1 == x_2$ ) or ( $y_1 == y_2$ ) or  
 $(\text{abs}(x_1 - x_2) == \text{abs}(y_1 - y_2))$   
 return 1  
 else:  
 return 2



Divide & Conquer Strategy:

- 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



# 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



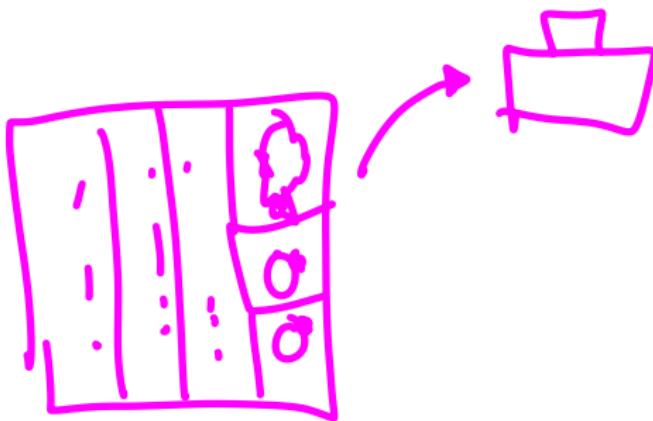
# Exercise: Knapsack Problem

## Problem Statement

Given: Knapsack capacity  $W$ , items with weights  $w_i$  and values  $v_i$

Goal: Maximize total value without exceeding weight capacity

values  
/ price



# 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)

- 1: Sort items by  $v_i/w_i$  descending
- 2: for each item  $i$  do
- 3:   if  $w_i \leq$  remaining capacity then
- 4:     Take entire item
- 5:     Update remaining capacity
- 6:   end if
- 7: end for

most value  
by we/unt



# Brute Force: Exhaustive Search

## Definition

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

## Characteristics:

Exhaustive: Examining every possibility

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

$$= \sim 5 \times 10^9$$



# Brute Force: Exhaustive Search

## Definition

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

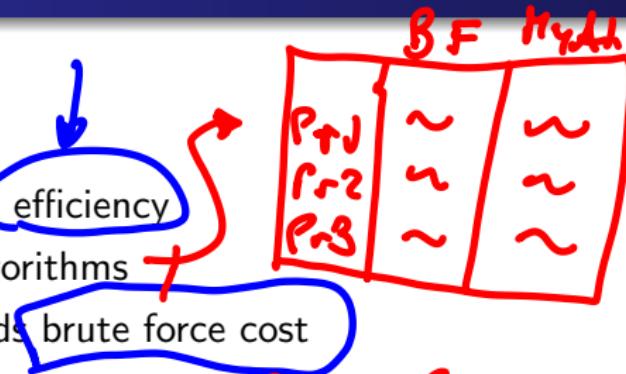
## Characteristics:

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



# Brute Force: When to Use It

- ① Problem size is 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



LMs

↳ 3 weeks - 20000 - 25000  
GPJ - 4 → J90M. USD      GPLI3



# Study Case: Password Cracking

## Problem

Given: Encrypted password hash and character set

Goal: Find the original password

$$\Sigma = \{a, b\}$$

alphabet

recursion?  
validations?



# Study Case: Password Cracking [Solution]

```

1: Algorithm BRUTEFORCEPASSWORD(hash, maxlength)
2: for length = 1 to maxlength do
3: for each possible string s of given length do
4: if HASH(s) = hash then
5: return s // Password found!
6: end if
7: end for
8: end for
9: return null // Password not found

```



s

*recursion*

Character set: [a-zA-Z] (26 characters)

Password length: 8 characters

Total attempts:  $26^8 \approx 208$  billion

Time estimate: Weeks on single computer!

*Brutal force*



# Study Case: Password Cracking [Solution]

```
1: Algorithm BRUTEFORCEPASSWORD(hash, maxlen)
2: for length = 1 to maxlength do
3: for each possible string s of given length do
4: if HASH(s) = hash then
5: return s // Password found!
6: end if
7: end for
8: end for
9: return null // Password not found
```

**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

```
1: Algorithm BACKTRACK(partial_solution)
2: if ISCOMPLETE(partial_solution) then
3: return partial_solution
4: end if
5: for each possible choice c do
6: if ISVALID(partial_solution, c) then
7: partial_solution.add(c)
8: result \leftarrow BACKTRACK(partial_solution)
9: if result \neq null then
10: return result
11: end if
12: partial_solution.remove(c) // Backtrack!
13: end if
14: end for
15: 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]

```
1: Algorithm SOLVENQUEENS(board, row)
2: if row = N then
3: return board // Solution found!
4: end if
5: for each column col in 0 to N – 1 do
6: if ISSAFE(board, row, col) then
7: board[row][col] ← Q // Place queen
8: result ← SOLVENQUEENS(board, row + 1)
9: if result ≠ null then
10: return result
11: end if
12: board[row][col] ← . // Remove queen (backtrack)
13: end if
14: end for
15: 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.



# 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.



# 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

1 Introduction to Algorithms

2 Algorithm Design

3 Algorithm Types



# Thanks!

# Questions?



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

