Algorithms & Complexity

SEDDIK Mohamed Taki Eddine

Batna 2 UNIVERSITY
Faculty of Mathematics and Computer Science
Department of Computer Science

November 8, 2023

Contents of chapter 1:Algorithmic Complexity and Review I

1 Recapitulation of Algorithm Basics
- Overview of Algorithm Characteristics
  - Definition and Importance of Algorithms
  - Key Characteristics: Correctness, Efficiency, etc.
  - General Structure and Components of Algorithms

2 Complexity
- Definitions
- Type of Complexity
- Landau's Notation
- Complexity Classes
- Complexity Calculation

Content

1 Recapitulation of Algorithm Basics
- Overview of Algorithm Characteristics

2 Complexity

Overview of Algorithm Characteristics

“Algorithms are the heartbeats of computer science;they define the pulse of problem-solving.”

Let’s embark on this journey to understand, analyze, and master algorithms!

Definition of Algorithms

An **algorithm** is:

- A step-by-step procedure for solving a problem.
- Employed in various domains: sorting, searching, ML, AI.
- Designed to reach a specific outcome.
- Often optimized for efficient problem-solving.

Importance of Algorithms

Why Algorithms are Crucial:

- Enable efficient problem-solving.
- Offer a clear procedural solution.
- Form the basis for programmatic implementations.
- Underpin various technologies.

Essential Algorithm Characteristics

- **Correctness:** Accurate outcomes for all possible inputs.
- **Efficiency:** Optimal use of time and space resources.
- **Simplicity:** Ease of understanding and implementation.
- **Unambiguousness:** Clear and unequivocal at each step.

Additional Algorithm Characteristics

- **Finiteness:** Concludes after a limited, defined number of steps.
- **Definiteness:** Every step is explicitly stated and unambiguous.
- **Effectiveness:** Every operation must be basic, precise, and performable in a practical manner.
- **Language Independence:** Implementable in various programming languages.

Components of Algorithms

- **Input:** Values (or no values) from a specified set.
- **Output:** Values from a specified set, often different from the input.
- **Processing Steps:** Precise instructions that transform input to output.

Algorithm Structure Example

Typical Algorithm Structure:

1. Start
2. Retrieve input data
3. Execute processing steps
4. Deliver output
5. End

Implementing Algorithms: An Example

A sorting algorithm might involve:

1. Comparing pairs of numbers in a list.
2. Swapping them to be in ascending order.
3. Repeating until the entire list is sorted.

Note: Control structures, such as loops and conditionals, guide algorithm steps.

Recursion

Recursion: A Journey Within

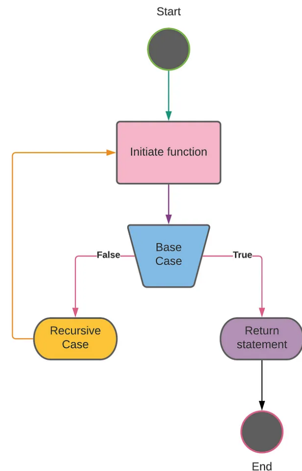
"Recursion is the process of defining something in terms of itself."

Recursion in Real Life

- Like Russian nesting dolls: each doll contains a smaller doll inside.
- Each smaller problem is a version of the larger problem.

Basic Concept of Recursion

- **Recursive Case:** The part of the function that calls itself.
- **Base Case:** The condition under which the recursion stops.



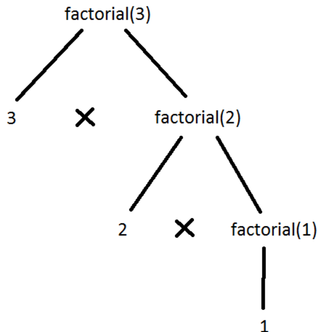
Example: Calculating Factorial

Factorial of a number (n!)

- Base Case: $0! = 1$
- Recursive Case: $n! = n \times (n-1)!$

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

Visualizing Recursion



- Each function call creates a branch.
- Base case is the leaf of the tree.

Advantages and Disadvantages of Recursion

Advantages:

- Simplifies code for complex problems.
- Natural fit for problems involving subproblems.

Disadvantages:

- Can be less efficient (memory).
- Risk of stack overflow with deep recursion.

Recap: Understanding Recursion

- Recursion is a function calling itself.
- Consists of a base case and a recursive case.
- Useful for problems that can be broken down into smaller subproblems.
- Requires careful consideration to avoid performance issues.

"Recursion: See Recursion."

```
char Understand_Recursion(char b) {
    if (b=='y') {
        return 'y';
    } else {
        for(int slide =12;slide
            <=18;slide++)
            read(page);
        printf("do you understand 
            recusion?y or n");
        scanf("%c",&b);
        return Understand_Recursion
            (b);
    }
}
```

Content

1 Recapitulation of Algorithm Basics

2 Complexity
- Definitions
- Type of Complexity
- Landau's Notation
- Complexity Classes
- Complexity Calculation

Definitions

Algorithm Complexity

The complexity of an algorithm is the measure of the number of fundamental operations it performs on a dataset. It is expressed as a function of the size of the dataset.

Optimal Algorithm

An algorithm is said to be optimal if its complexity is the minimum among the algorithms of its class.

Type of Complexity

Best-Case Complexity

- Smallest number of operations for a dataset of size n.
- $T_{\min}(n) = \min_{d \in D_n} T(d)$

Average-Case Complexity

- Average complexities for datasets of size n.
- $T_{\text{avg}}(n) = \frac{\sum_{d \in D_n} T(d)}{|D_n|}$

Worst-Case Complexity

- Largest number of operations for a dataset of size n.
- $T_{\max}(n) = \max_{d \in D_n} T(d)$

Notations:

- D_n : Set of data of size n.
- $T(n)$: Operations on a dataset of size n.

LANDAU'S NOTATION

Introduction

- Landau's notation, often referred to as "Big O", is commonly used to formally describe the performance of an algorithm.

Expression

- This notation represents the upper bound of a function up to a constant factor.
- $f = O(g) \Leftrightarrow \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n)$

Example

- $T(n) = O(n^2)$ means there exists a constant $c > 0$ and a constant $n_0 > 0$ such that for all $n > n_0$, $T(n) \leq c \times n^2$.

LANDAU'S NOTATION

Rules of the Big O Notation

- Constant terms: $O(c) = O(1)$
- Multiplicative constants are omitted:
  $O(cT) = cO(T) = O(T)$
- Addition is carried out by taking the maximum:
  $O(T1) + O(T2) = O(T1 + T2) = \max(O(T1); O(T2))$
- Multiplication remains unchanged:
  $O(T1)O(T2) = O(T1T2)$

Landau's Notation

Assumption

Assuming the execution time of an algorithm is described by the function $T(n) = 3n^2 + 10n + 10$, calculate $O(T(n))$?

$$\begin{aligned} O(T(n)) &= O(3n^2 + 10n + 10) \\ &= O(\max(3n^2, 10n, 10)) \\ &= O(3n^2) \\ &= O(n^2) \end{aligned}$$

Landau's Notation

Remark

For $n = 10$ we have:

- Execution time for $3n^2$: $\frac{3(10)^2}{3(10)^2+10(10)+10} = 73.2\%$
- Execution time for $10n$: $\frac{10(10)}{3(10)^2+10(10)+10} = 24.4\%$
- Execution time for 10: $\frac{10}{3(10)^2+10(10)+10} = 2.4\%$

The weight of $3n^2$ becomes even greater when $n = 100$, i.e., 96.7%.
Quantities $10n$ and 10 can be neglected.
This explains the rules of the Big O notation.

Complexity Classes

Class	**Notation and Example**
Constant	$O(1)$ - Access the first element
Linear	$O(n)$ - Traverse a data set
Logarithmic	$O(\log(n))$ - Splitting data set
Quasi-linear	$O(n\log(n))$ - Repeated splitting
Quadratic	$O(n^2)$ - Two nested loops
Polynomial	$O(n^P)$ - P nested loops
Exponential	$O(a^n)$ - All possible subsets

Calcul de la Complexité

Case of a simple instruction (writing, reading, assignment):

The execution time of each simple instruction is $O(1)$.

Case of a sequence of simple instructions:

The execution time of a sequence of instructions is determined by the sum rule. Therefore, it's the time of the sequence which has the highest execution time:
$O(T) = O(T1 + T2) = \max(O(T1); O(T2))$.

Complexity Calculation

Example :

Permutation(S: array, i: index, j: index)

$T1$:	tmp $\leftarrow S[i]$	$O(1)$
$T2$:	$S[i] \leftarrow S[j]$	$O(1)$
$T3$:	$S[j] \leftarrow$ tmp	$O(1)$
$T4$:	return S	$O(1)$

Complexity

$$O(T) = \max(O(T1 + T2 + T3 + T4)) = O(1)$$

Complexity Calculation

Case of a conditional treatment:

The execution time of a SI (IF) instruction is the execution time of the executed instructions under condition, plus the time to evaluate the condition. For an alternative, we consider the worst case.

```
If (condition) Then
   Treatment1
Else
   Treatment2
End If
/* O(T_condition) + max(O(T_treatment1),
O(T_treatment2)) */
```

Complexity Calculation

Case of an Iterative Treatment

The execution time of a loop is the sum of the time to evaluate the body and the time to evaluate the condition. Often, this time is the product of the number of loop iterations by the longest possible execution time of the body.

For Loop

```
For (i from indStart to indEnd) do
Treatment
End For
```

$$\sum_{i=\text{indStart}}^{\text{indEnd}} O(\text{Treatment})$$

While Loop

```
While (condition) do Treatment Done
```

$$\text{number of iterations} \times (O(\text{condition}) + O(\text{Treatment}))$$

Complexity Calculation

Example 2: Sequential Search (x: value, S: array, n: size)

```
i <- 0
Found <- false  /*O(1)*/
While (i < n) and /*Condition = O(1)*/
(not Found) do  /*number of iterations = n*/
    i <- i + 1    /*O(1)*/
    If (S[i] = x) then /*O(1)*/
        Found <- true /*O(1)*/
    End If
End While
Return Found /*O(1)*/
```

$$O(T) = \max(O(1) + O(n \times 1) + O(1)) = O(n)$$

Complexity Calculation

Example 3: Bubble Sort (T: Array, n: size)

```
For i from 1 to n do
    / n times /
    For j from i+1 to n do
        / n - i times /
        If (T[i] > T[j]) Then
            tmp <- T[i];
            T[i] <- T[j];
            T[j] <- tmp;
        End If
    End For
End For
```

Bubble Sort Complexity Analysis

Mathematical Explanation:

Suppose we have N elements in our list.

- In the first pass, we need $(N-1)$ comparisons.
- In the second pass, we need $(N-2)$ comparisons.
- $\ldots$
- In the penultimate pass, we need 1 comparison.

Total number of comparisons (C):

$$C = (N-1) + (N-2) + (N-3) + \ldots + 2 + 1$$

Using the formula for the sum of an arithmetic series:

$$C = \frac{N \times (N-1)}{2}$$