Complexity and Problem Solving

BEKKOUCHE Mohammed

National Higher School of Computer Science, Sidi Bel Abbès

2nd Year Second Cycle

Specialization: Artificial Intelligence and Data Science (IASD), Cybersecurity (Cys)

m.bekkouche@esi-sba.dz

Website: https://mdbekkouche.github.io

August 22, 2026

This course covers:

- The theoretical and practical aspects related to complexity and problem solving.
  $\Longrightarrow$ It enables students to acquire the necessary skills to analyze the complexity of algorithms and problems, allowing them to design correct and efficient algorithms for solving a given problem.
- An introduction to the field of artificial intelligence.

Objectives

The objectives of this module are to focus on several aspects:

- Acquire the concepts of algorithmic complexities.
- Understand the different concepts for classifying problems.
- Identify problems that cannot be solved and those for which it is difficult to conceive an efficient solution.
- Introduce the problem-solving approach and solution construction.
- Present several methods for problem-solving.

These concepts will be presented through problems from various domains in computer science.

Studying the complexity of problems helps determine if a problem can be solved by an algorithm and how much resources (in terms of time and space) will be required to solve this problem by an algorithm. Problem solving can be carried out using different methods. Some strategies will solve a problem more efficiently than others.

Module Contents

Module Contents

Chapter 3: Artificial Intelligence for Problem Solving

1. Introduction
2. Approach to Problem Representation and Solution

Chapter 4: Solution Methods in State Spaces

1. Blind Search Strategies
   Breadth-First Search
   Depth-First Search
   Limited Depth-First Search
   etc.
2. Heuristic Search Strategies
   Concept of Heuristics
   Greedy Algorithms

Chapter 5: Problem Decomposition Solution Methods

1. Representation by AND/OR Tree
2. Strategies for Solving Decomposable Problems

An uninformed search (Blind Search) is a searching technique that has no additional information about the distance from the current state to the goal. Informed Search (Heuristic Search) is another technique that has additional information about the estimate distance from the current state to the goal.

Prerequisites

- Algorithms and static data structures
- Algorithms and dynamic data structures
- Knowledge of graph theory and languages

Introduction

Contents:

- Generalities
- Computing Fibonacci Numbers
- Elementary Operations
- Algorithm: Correctness and Complexity
- Recursion and Divide and Conquer Approach

Generalities

Algorithm

- The word "algorithm" comes from the name of the 9th-century Arab mathematician, Mohammed Ibn-Moussa Al-Khuwarizmi.
- The first algorithm is the Euclidean Algorithm (300 BC) for computing the greatest common divisor of two integers.

Definition

An algorithm is a well-defined computational procedure that takes an input value or a set of values and produces an output value or a set of values. An algorithm is, therefore, a sequence of computational steps that transforms the input value into the output value.

1. Al Khwarizmi proposed fundamental methods for adding, multiplying, and dividing integers, as well as calculating the roots of an equation and the decimals of the number π. These procedures were precise, unambiguous, mechanical, efficient, and correct: they were simply algorithms.
2. It is worth noting that algorithms were known and used long before the advent of computing.
3. There are various ways to define an algorithm. Here, we present a definition that can provide an idea of the concept of an algorithm.

Generalities

Algorithm Complexity

- Performance is a central concern in computer science. If a problem is solved within a reasonable time, the user is satisfied.
- Generally, precise time measurement is not essential; only an approximation of the time matters. If the obtained time is perceived as relatively long, the question of a more efficient solution arises.
- With the machine being unmodified, improvements will focus on how algorithms are designed based on the data volume n to be processed.
- For example, if by doubling the volume of data, an algorithm returns a result by multiplying the execution time by 2 and another returns it by multiplying the execution time by $n > 2$, we will obviously opt for the first: we says that its time complexity is better
- Additionally, space complexity can be defined to account for the memory space used during algorithm execution. The larger the complexity, the more memory zones the program implementing the algorithm will require to store data.

1. In this course, we focus on the study of time complexity.
2. Why do we mainly address time complexity over space complexity?
   - Modern machines are highly robust with extraordinary hardware capabilities.
   - Algorithms with poor time complexity may become infeasible to run on large datasets, even if they use less memory.
   - We mainly concentrate on time complexity as space complexity is generally less critical.
3. This doesn't mean space complexity is ignored, especially in cases where memory is constrained (e.g., embedded systems)

Computing Fibonacci Numbers

Recursive Definition

Let (F_n) be the Fibonacci sequence:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

The sequence generates the following numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

A sequence defined by recurrence is a sequence defined by its first term (or first terms) and a recurrence relation, which defines each term based on the previous one or ones, if they exist.

The Fibonacci sequence is a sequence of integers where each term is the sum of the two preceding terms.

Computing Fibonacci Numbers

A First Version of the Algorithm

Algorithm 1: Fib1(n)

```
1 if n = 0 then
2 |   return 0;
3 else
4 |   if n = 1 then
5 |   |   return 1;
6 |   else
7 |   |   return Fib1(n-1)+Fib1(n-2);
8 |   end
9 end
```

Three essential questions are raised:

1. Is the algorithm correct?
2. How much time does it take to terminate, depending on n?
3. Can it be improved?

1. In programming, pseudo-code is an Algorithm Description Language. It is a way of describing an algorithm in almost natural language, without reference to a particular programming language.
2. An algorithm is correct if it performs as expected.

Computing Fibonacci Numbers

Computing Fibonacci Numbers

1. **Correctness:** The algorithm is correct since it precisely implements the recursive definition of the Fibonacci sequence.
2. **Execution Time:** Let T(n) be the number of steps needed for the algorithm to compute F_n.

$$T(n) \leq 2 \text{ for } n \leq 1$$

$$T(n) = T(n-1) + T(n-2) + 3 \text{ for } n > 1$$

- An approximate solution: $T(n) \approx 2^{0.694n}$
- T(n) exhibits exponential growth
  $\Longrightarrow$ What would be the required time for this calculation on a computer?

1. The answer to the first question is quite evident.
2. The second question is intricate.

Computing Fibonacci Numbers

A First Version of the Algorithm

- For example, let's consider the calculation of F_{200}.
- The computation of Fib1(n) executes approximately $T(200) \approx 2^{138}$ elementary steps.
- Let's take a computer, executing 40 trillion 40×10^{12} instructions per second.
- On this computer, we would need 2^{92} seconds!
  $\Longrightarrow$ Waiting until the theoretical extinction of the sun!
- Thus, we are completely discouraged by the harsh reality of exponential growth, which cannot be overcome by any current machine!

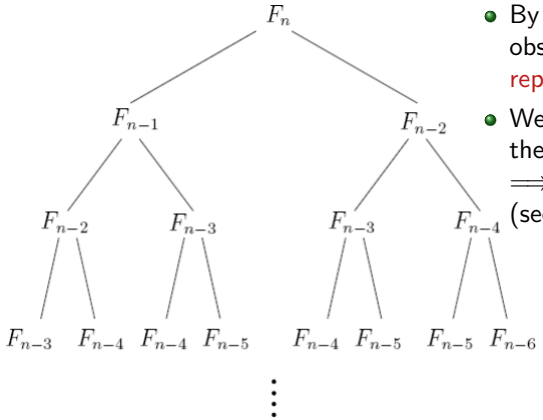
3. Can we do better in terms of computation cost?
   $\Longrightarrow$ And without relying on machine improvements, which we have just seen are inadequate!

1. According to Moore's hypothesis which assumes that computers double their power every 18 months, then, we could arrive at the fact that: if we could reasonably calculate F_{100}, then with Moore's hypothesis , we could probably calculate with the same power F_{101}, ... Meaning that we will gain a number each year.

Computing Fibonacci Numbers

A Polynomial Version

The behavior of Fib1(n) (the trace):



- By analyzing this trace, we can observe that many calls are repeated.
- We can adopt the trick of storing the already computed values $\Longrightarrow$ This leads to the version Fib2 (see next slide).

Dynamic programming consists of storing the values of subproblems to avoid redundant computations. There are two methods to effectively compute a solution: bottom-up method and top-down method.

1. In the top-down method, we write a recursive algorithm, but we use memoization to avoid solving the same problem multiple times. In other words, we store the results of recursive calls in an array F[.]:

Algorithm 2: Fib3(n)

```
1 if F[n] ≠ null then return F[n] ;
2 if n = 0 or n == 1 then F[n] = n ;
3 else F[n] = Fib3(n − 1) + Fib3(n − 2) ;
4 return F[n] ;
```

2. In the bottom-up method, we start by calculating solutions for the smallest subproblems, then, step by step, we compute solutions for larger problems and store the results in an array F[.] (see Fib2).

Computing Fibonacci Numbers

A Polynomial Version

- The algorithm is obviously correct since, once again, it follows the recurrent definition of Fibonacci.
- How much does this algorithm require to compute F_n as a function of n?
  $\Longrightarrow$ The loop in the algorithm requires n - 1 calculation steps. Therefore, the execution time of this algorithm is linear in n.
- It is now possible to compute F_{200000} with little effort!

Algorithm 3: Fib2(n)

```
1  if n = 0 or n == 1 then
2  |  return n;
3  end
4  Create an array F[0..n];
5  F[0] ← 0;
6  F[1] ← 1;
7  for i ← 2 to n do
8  |  F[i] ← F[i − 1] + F[i − 2] ;
9  end
10 return F[n];
```

1. Designing a good algorithm can lead to exponential savings!

Elementary Operations

- An algorithm is a set of elementary computational operations, organized according to specific rules to solve a given problem.
- For each input of the problem, the algorithm produces an output after a finite number of operations.
- Elementary operations include usual arithmetic operations, data transfers, comparisons between data, etc.
- It is useful to consider as truly elementary only those operations with constant computation time, i.e., independent of the size of operands. For example:
  - Addition of integers with a bounded size (e.g., 'int' in C) is an elementary operation.
  - Addition of integers with arbitrary size is not an elementary operation.
  - Similarly, testing membership of an element in a set is not an elementary operation in this sense because its execution time depends on the size of the set, even if some programming languages provide basic instructions to perform this operation.

Algorithm: Correctness and Complexity

Insertion Sort: Algorithm, Correctness, and Complexity

Sorting Problem

Input A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$

Output A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

- Given an input sequence such as $\langle 5, 90, 45, 89 \rangle$, a sorting algorithm will produce an output of $\langle 5, 45, 89, 90 \rangle$
  $\Longrightarrow$ Such an input sequence is referred to as an instance of the problem

1. To illustrate our approach, we address a frequently encountered practical problem: sorting a sequence of numbers in ascending order.
2. The slide presents a formal definition of this problem.

Algorithm: Correctness and Complexity

Insertion Sort: Algorithm

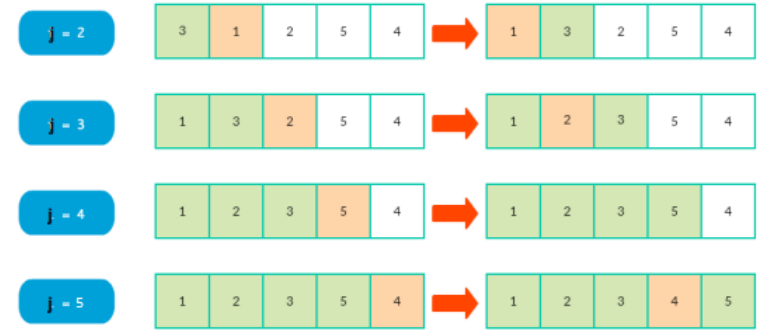
A first solution to the sorting problem: Insertion Sort

Algorithm 4: Insertion-Sort(A)

Data: A sequence of n numbers $A = \langle a_1, a_2, ..., a_n \rangle$
Result: A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
1  for j ← 2 to length(A) do
2      pivot ← A[j] ;
3      {Insert A[j] into the sorted sequence A[1..j − 1]};
4      i ← j − 1 ;
5      while (i > 0) ∧ (A[i] > pivot) do
6          A[i + 1] ← A[i] ;
7          i ← i − 1 ;
8      end
9      A[i + 1] ← pivot ;
10 end
```

Insertion sort considers each element of the array and inserts it at the correct position among the already sorted elements. Therefore, when considering an element, the elements preceding it are already sorted, while the elements following it are not yet sorted.



Algorithm: Correctness and Complexity

Algorithm Correctness

- An algorithm is correct if, for every input instance, it terminates with the correct desired output.
- The loop invariant in an algorithm is a property that holds true at:
  1. Before starting the loop
  2. After each iteration of the loop
  3. At the end of the loop

  To prove the invariant:

INITIALIZATION Show that the loop invariant is true before the first loop iteration.

MAINTENANCE Show that if the loop invariant is true before a loop iteration, it remains true before the next iteration.

TERMINATION Once the loop is finished, the invariant provides a useful property that helps demonstrate the algorithm's correctness.

Establishing this proof systematically provides the proof of the algorithm's correctness.

1. Maintenance is also referred to as Conservation.

Algorithm: Correctness and Complexity

Insertion Sort: Correctness

Loop Invariant of Insertion Sort: The subarray A[1..j-1] is sorted.
Proof of correctness of Insertion Sort:

- **INITIALIZATION:** Before the first iteration of loop 1, the subarray A[1..j-1] contains only one element (j=2). An array with a single element is inherently sorted (trivial), hence our invariant "the subarray A[1..j-1] is sorted" holds true.
- **MAINTENANCE:** Overall, what does loop 1 do? It moves the elements A[j-1], A[j-2], A[j-3], etc. one position to the right until the correct position for A[j] is found. After these right shifts are performed, the value A[j] is inserted at the right place. During an iteration of loop 1, we transition from an unsorted array A[1..j] (subarray A[1..j-1] is sorted, but element A[j] is not yet in the correct place, so A[1..j] is not sorted yet) to a sorted array (shifting elements A[j-1], A[j-2], A[j-3], etc. to the right and placing A[j] at the correct position).

We will utilize this property (the invariant) to demonstrate the correctness of the Insertion Sort algorithm.

Algorithm: Correctness and Complexity

Insertion Sort: Correctness

Proof of correctness of Insertion Sort (continued):

At the end of an iteration (just before "j ← j+1"), A[1..j] is sorted, and right after "j ← j+1", the previously mentioned array A[1..j] becomes the array A[1..j-1]. Thus, we can affirm that if the invariant "the subarray A[1..j-1] is sorted" is true before an iteration of loop 1, it remains true before the next iteration.

- **TERMINATION:** When the loop terminates, we must have j = n + 1 (for an array of n elements). If we substitute n+1 with j in the loop invariant, we obtain the subarray A[1..n]. When the loop ends, the statement "the subarray A[1..j-1] is sorted" becomes "the subarray A[1..n] is sorted." Now, the subarray A[1..n] corresponds to the array composed of n elements. Hence, the original array is now sorted, demonstrating the correctness of the algorithm.

Exercise. Prove the correctness of the `while` (inner) loop using the loop invariant method.

Exercise solution.

Loop Invariant At the start of each iteration of the `while` loop:

$$A[i+1 \dots j]$$

contains the elements that are greater than or equal to `pivot`.

Initialization Before the first iteration, we have $i = j - 1$ and $A[j] = pivot$. Thus the subtable $A[i+1 \dots j]$ contains one element, which trivially satisfies the invariant.

Maintenance Suppose the invariant holds at the start of an iteration. If $A[i] > pivot$, then we set $A[i+1] = A[i]$ and decrement i. Now, $A[i+1 \dots j]$ still contains exactly those elements greater than `pivot`. Hence, the invariant is maintained.

Termination The loop terminates when either:

1. $i = 0$, or
2. $A[i] \leq pivot$.

At this point, $A[i+1 \dots j]$ contains all elements greater than `pivot`, and the correct position for `pivot` is exactly at $A[i+1]$. Finally, the instruction

$$A[i+1] = pivot$$

places the pivot in its proper location, maintaining the sorted order of $A[1 \dots j]$.

Algorithm: Correctness and Complexity

Insertion Sort: Complexity

Concept of input size

- The time taken by the Insertion Sort procedure depends on the size of its input: sorting a thousand numbers takes more time than sorting three numbers.
- The concept of input size depends on the problem under study; sometimes it is more appropriate to describe the input size with two numbers
  $\Longrightarrow$ For example, in a problem related to graphs, the input size could be the number of vertices and the number of edges.

For each problem studied, we will specify the utilized measure of size.

In the context of computational complexity, the size of a problem refers to a quantitative mesure that indicates the amount of the input required to define the problem.

Algorithm: Correctness and Complexity

Insertion Sort: Complexity

- The execution time of an algorithm on a particular input is the number of elementary operations executed.
- The execution time of the algorithm is the sum of the execution times of each instruction executed
  $\Longrightarrow$ An instruction i that executes in time c_i and is executed n times contributes $c_i n$ to the total.
- To calculate $T(n)$, the execution time of Insertion Sort, we sum the products of costs by the number of occurrences.
- If the sequence is already ordered, we get:
  $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) =$
  $(c_1 + c_2 + c_4 + c_5 + c_9)n + (-c_2 - c_4 - c_5 - c_9)$
- We can express this final formula in the form $an + b$, where a and b are constants
  $\Longrightarrow$ We say the algorithm is in $O(n)$.

We have calculated the cost of the Insertion Sort algorithm in the best-case scenario.

Algorithm: Correctness and Complexity

Insertion Sort: Complexity

- If the sequence is in reverse order, we are in the worst-case scenario:
  $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_9(n-1)$
  where t_j is the number of iterations of the second loop at iteration j of the first loop.
- We need to compare each element $A[j]$ with each element of the sorted subarray $A[1..j-1]$, so $t_j = j$ for $j = 2, 3, ..., n$.
- Given that: $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$
- We get: $T(n) =$
  $c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_9(n-1)$
  $T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9)n + (-c_2 - c_4 - c_5 - c_9)$
- This final cost is of the form $an^2 + bn + c$ where a, b, and c are constants
  $\Longrightarrow$ We say the algorithm is in $O(n^2)$.

1. A sequence (U_n) is called arithmetic if, for every natural number n, we have: $U_{n+1} = U_n + r$, where r is the common difference of this sequence.
2. We have calculated the cost of the Insertion Sort algorithm in the worst-case scenario, which is the longest execution time for any input of size n. This is the most commonly used cost measure in practice. In the following, we will explore several methods for calculating this cost.

Recursion and Divide and Conquer Approach

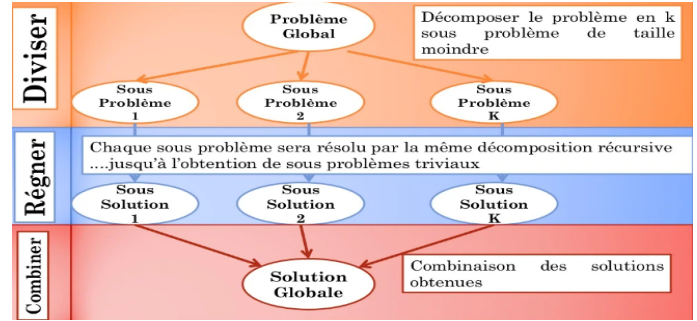
This problem-solving paradigm involves three steps at each level of recursion:

- **Divide** the problem into a number of subproblems
- **Conquer** the subproblems by solving them recursively. Moreover, if the size of a subproblem is small enough, it can be solved directly.
- **Combine** the solutions to the subproblems into a complete solution for the original problem.

Algorithm 5: D&C_Recursive(P)

```
1 if Small(P) then Solution(P) ;
2 else
3     Divide P into P_1, P_2, ..., P_k ;
4     D&C_Recursive(P_1) ; D&C_Recursive(P_2) ; ... ; D&C_Recursive(P_k) ;
5     Combine(D&C_Recursive(P_1), D&C_Recursive(P_2), ...,
        D&C_Recursive(P_k)) ;
6 end
```

Many useful algorithms follow a recursive structure: to solve a given problem, they call themselves recursively one or more times on smaller, very similar subproblems. These algorithms embrace the "divide and conquer" approach: they break the problem into multiple subproblems similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to recover a solution to the original problem.



Recursion and Divide and Conquer Approach

Merge Sort and Divide and Conquer Approach

The merge sort algorithm closely follows the "divide and conquer" rule:

- **Divide** Divide the array $T[1,..n]$ to be sorted into two subarrays $T[1,..n/2]$ and $T[n/2+1,..n]$
- **Conquer** Sort the two subarrays $T[1,..n/2]$ and $T[n/2+1,..n]$ (recursively, or do nothing if they have size 1)
- **Combine** Merge the two sorted subarrays $T[1,..n/2]$ and $T[n/2+1,..n]$

Algorithm 6: MergeSort(A, p, r)

Data: A sequence of n numbers $A = \langle a_1, a_2, ..., a_n \rangle$
Result: A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
1 if p < r then
2 |   q ← ⌊(p + r)/2⌋;
3 |   MergeSort(A, p, q);
4 |   MergeSort(A, q + 1, r);
5 |   Merge(A, p, q, r);
6 end
```

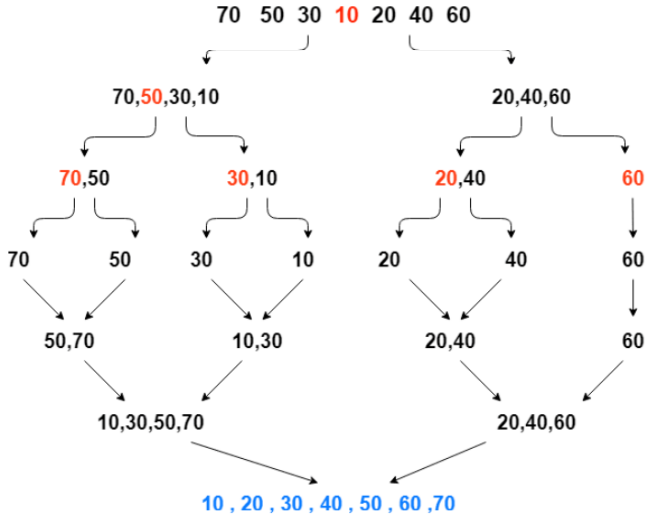
Algorithm 7: Merge$(Tab, start, middle, end)$

```
1  Tmp: temporary array of size end − start + 1 ;
2  i ← 1 ; i1 ← start ; i2 ← middle + 1 ;
3  while (i1 ≤ middle) ∧ (i2 ≤ end) do
4  |   if (Tab[i1] ≤ Tab[i2]) then
5  |   |   Tmp[i] ← Tab[i1] ; i1 ← i1 + 1 ;
6  |   else
7  |   |   Tmp[i] ← Tab[i2] ; i2 ← i2 + 1 ;
8  |   end
9  |   i ← i + 1 ;
10 end
11 while (i1 ≤ middle) do Tmp[i] ← Tab[i1] ; i1 ← i1 + 1 ; i ← i + 1;
12 while (i2 ≤ end) do Tmp[i] ← Tab[i2] ; i2 ← i2 + 1 ; i ← i + 1;
13 for i ← start to end do Tab[i] ← Tmp[i − start + 1];
```

Recursion and Divide and Conquer Approach

Merge Sort and the Divide-and-Conquer Approach

Illustration of Merge Sort:



Recursion and Divide-and-Conquer Approach

Merge Sort: Complexity

- When an algorithm contains a recursive call to itself, its execution time can often be described by a recurrence equation.
- This equation describes the overall execution time for a problem of size n in terms of the execution time for smaller-sized inputs.
- We can use mathematical tools to solve the recurrence and find bounds for the algorithm's performance.

Recursion and Divide-and-Conquer Approach

Merge Sort: Complexity

- Let $T(n)$ be the execution time of a problem of size n.
- If the problem size is sufficiently small, say $n \leq c$ for some constant c, the solution takes constant time, denoted as $\Theta(1)$.
- Suppose we divide the problem into a subproblems, each of size $1/b$ of the initial problem size.
- If it takes time $D(n)$ to divide the problem into subproblems and time $C(n)$ to combine the solutions to the subproblems into the final solution, we obtain the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Recursion and Divide-and-Conquer Approach

Merge Sort: Complexity

- We assume that the size of the initial problem is a power of two.
- Each "divide" step then generates two sub-sequences of exactly size $n/2$. We will see later that this assumption does not affect the order of magnitude of the recurrence solution.
- Merge sort on a single element takes constant time.
- With $n > 1$ elements, we segment the execution time as follows:

  Divide It only involves computing the middle of the subsequence, which takes constant time, $D(n) = \Theta(1)$.

  Conquer We recursively solve two subproblems, resulting in $2T(n/2)$.

  Combine It can be shown that the merging procedure on a subproblem of size n is of the form $C(n) = an + b$, where a and b are constants. We will denote this as $C(n) = \Theta(n)$.

1. Our recurrence-based analysis is simplified if we assume that the size of the initial problem is a power of two.

Recursion and Divide-and-Conquer Approach

Merge Sort: Complexity

- Therefore, we have:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- We will prove later that $T(n)$ is of the form $an\log(n) + bn + c$, where a, b, and c are constants.
  $\Longrightarrow$ We denote this form as $\Theta(n\log(n))$.
- **Note:** This cost is significantly lower than that of insertion sort, as $\exists n_0, \forall n, n > n_0, n^2 > n\log(n)$.

Recursion and Divide-and-Conquer Approach

Fast Exponentiation

- The goal is to compute x^n for given values of x and n, by calculating the complexity with respect to n.
- The naive method using the definition below results in linear complexity.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n-1} \times x & n > 0 \end{cases}$$

- By using the following observations:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^2)^{n/2} & \text{if n is even} \\ x(x^2)^{(n-1)/2} & \text{if n is odd} \end{cases}$$

The first recursive definition of the power function:
$x^3 = x^2 \times x = (x^1 \times x) \times x = ([x^0 \times x] \times x) \times x$ (this definition requires 3 multiplications to compute x^3)
Recursive definition of multiplication count for power function:

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

Second definition

- If n is even, x^n is equivalent to computing the power of x^2 to $n/2$.
- If n is odd, x^n is equivalent to multiplying x by the power of x^2 to $(n-1)/2$

$$T'(n) = \begin{cases} 0 & \text{if } n = 0 \\ T'(\frac{n}{2}) + 1 & \text{if n is even} \\ T'(n-1) + 1 & \text{if n is odd} \end{cases}$$

This definition speeds up the process towards the base case.

Recursion and Divide-and-Conquer Approach

Fast Exponentiation

- Hence, the algorithm:

Algorithm 8: FastExp(x,n)

```
1 if (n = 0) then
2 |   return 1;
3 else
4 |   if n is even then
5 |   |   return FastExp(x × x, n/2);
6 |   else
7 |   |   return x × FastExp(x × x, (n − 1)/2);
8 |   end
9 end
```

Hence, the complexity of fast exponentiation is $O(\log n)$.

Complexity in the number of multiplications for x^{100}

T'(100) = 1 + T'(50)
= 1 + 1 + T'(25)
= 1 + 1 + 1 + T'(24)
= 1 + 1 + 1 + 1 + T'(12)
= 1 + 1 + 1 + 1 + 1 + T'(6)
= 1 + 1 + 1 + 1 + 1 + 1 + T'(3)
= 1 + 1 + 1 + 1 + 1 + 1 + 1 + T'(2)
= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + T'(1)
= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + T'(0)

The complexity of x^{100} is equal to 9. There is a significant improvement from the linear complexity (100 multiplications) to the complexity of this second definition, which requires only 9 multiplications to compute x^{100}.

Complexity in the number of multiplications for x^{100}

$$
\begin{aligned}
T'(100) = \quad & 1 + \quad T'(50) \\
& 1 + \quad T'(25) \\
& \quad 1 + \quad T'(24) \\
& \quad\quad 1 + \quad T'(12) \\
& \quad\quad\quad 1 + \quad T'(6) \\
& \quad\quad\quad\quad 1 + \quad T'(3) \\
& \quad\quad\quad\quad\quad 1 + \quad T'(2) \\
& \quad\quad\quad\quad\quad\quad 1 + \quad T'(1) \\
& \quad\quad\quad\quad\quad\quad\quad 1 + \quad T'(0)
\end{aligned}
$$

The complexity of x^{100} is equal to 9. There is a significant improvement from the linear complexity (100 multiplications) to the complexity of this second definition, which requires only 9 multiplications to compute x^{100}.