Algorithmic Complexity

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

April 2, 2026

Content:

Introduction

- An algorithm is said to be correct if, for each input data, it must produce the correct output.
- In this case, it is said that the (correct) algorithm solves the given problem.
- To computationally solve a given problem, an algorithm is implemented on a computer. That is, the algorithm is rewritten using a programming language.
- However, for a given problem, there are often several algorithms that can solve it.
- These algorithms differ from each other in terms of efficiency. These differences can be much more significant than those due to hardware and software.

Is there any benefit in choosing algorithms? And if so, how do we choose them?

Introduction

In computer science, the notion of complexity refers to two concepts:

Algorithmic Complexity: This is the study of efficiency when comparing algorithms.

- Indeed, complexity introduces the concept of cost and shows that it is not sufficient to find a correct method to solve a problem; it must also be efficient.
- Efficiency is measured by the time and space required by an algorithm to solve a problem.

Problem Complexity: This is the study that allows the classification of problems based on the performance of the best known algorithms that solve them.

This chapter focuses on algorithmic complexity.

Notion of Algorithm Complexity

- Studying the complexity of an algorithm involves predicting the necessary resources (memory, computation time) for the program that implements that algorithm. Sometimes, the relevant resources are:
  $\Longrightarrow$ Used memory (spatial complexity)
  $\Longrightarrow$ But most often, we want to measure computation time (time complexity)
  - **Time Complexity:** It quantifies the time required for an algorithm to execute based on the length of the input.
  - **Space Complexity:** It quantifies the amount of memory space taken by an algorithm to execute based on the length of the input.

In this course:

- We emphasize time complexity, and it's this notion that we will delve into.
- We will use a model of computation called a Random Access Machine (RAM) with a single processor. $\Longrightarrow$ In this model, instructions are executed sequentially without simultaneous operations.

Algorithm Analysis $\Longleftrightarrow$ Study of Algorithm Complexity

Notion of Algorithm Complexity

- Consider a given problem and an algorithm to solve it.
- For input data x of size n, the algorithm requires a certain amount of time, measured in the number of elementary operations, denoted as $c(x)$.
- The time cost obviously varies with the data size, but it can also vary among different data of the same size n.

For example:

Let's consider the bubble sort algorithm, which takes a sequence $\langle a_1, ..., a_n \rangle$ of distinct real numbers to be sorted in ascending order. It searches for the first descent, i.e., the smallest index i such that $a_i > a_{i+1}$, swaps these two elements, and repeats the process on the resulting sequence.
The number of inversions performed:

- 0 for a sorted sequence.
- $\frac{n(n-1)}{2}$ for a decreasing sequence.

Our goal is to evaluate the cost of an algorithm according to certain criteria, and based on the data size n.

1. Elementary operations include common arithmetic operations, data transfers, data comparisons, etc. It is useful to consider as truly elementary only those operations with constant computation time, i.e., independent of operand size. **For example:**
   - Addition of bounded-size integers (e.g., `int` in C) is an elementary operation.
   - Addition of arbitrary-size integers is not elementary.
   - Similarly, testing membership of an element in a set is not an elementary operation in this sense, as its execution time depends on the set size, even if some programming languages have basic instructions to perform this operation.
2. Bubble sort is a sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
3. $\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$

Notion of Algorithm Complexity

Fundamental Operations

- For certain problems, we can identify one or more operations that are fundamental (the most important elementary operations) in the sense that the execution time of an algorithm solving this problem is always proportional to the number of these operations.
- It is then possible to compare algorithms dealing with this problem using this simplified measure.

Examples:

Problem	**Fundamental Operations**
Searching for an element in a list	Comparison between this element and entries in the list
Sorting a list of elements	Comparison between elements and element rearrangement
Multiplying two matrices of numbers	Multiplication and addition of two numbers
Primality testing of a given integer n	Division

If a very precise microanalysis of program execution time is desired, one can decide that all elementary operations of the program are fundamental.

Notion of Algorithm Complexity

Rules for Counting the Number of Fundamental Operations

Sequence When operations are in a sequence of instructions, their numbers are added together.

If-Then-Else For conditional branches, it is generally difficult to determine which branch of the condition is executed, and therefore which operations to count. However, the number of operations can be upper-bounded.

Loops For loops, the number of operations in the loop is $\sum P(i)$, where i is the loop control variable, and $P(i)$ is the number of fundamental operations during the execution of the ith iteration. If the number of iterations is difficult to calculate, one can use a good upper bound.

After identifying the fundamental operations, the next step is to count the number of operations of each type. There is no complete system of rules for counting the number of operations based on the syntax of algorithms, but some observations can be made.

1. **Complexity of a sequence of instructions:**
   $S =$ Begin
   $i_1; i_2; ...; i_n$
   End
   Therefore, $Nb(S) = \sum_{p=1}^{n} Nb(i_p)$
   $Nb(i_k) =$ the number of fundamental operations denoted as Opf contained in algorithmic instruction i_k.
2. **Complexity of a conditional statement:**
   Cond = if Expr then E1 else E2
   Therefore, $Nb(Cond) \leq Nb(Expr) + \max(Nb(E_1), Nb(E_2))$

Notion of Algorithm Complexity

Rules for Counting the Number of Fundamental Operations

Example (Loops)

Iter = Iteration Expr(i)
S
endIteration

Therefore: Nb(Iter) = [Nb(S) + Nb(Expr(i))] × Nb_Iterations
So, for the case of a For loop:

Iter = For I := a to b do
Begin
i_1 **;** i_2 **; ... ;** i_n
End;

$Nb(Iter) = (|b - a| + 1) \times (\sum_{p=1}^{n} Nb(i_p) + 1)$
The complexity of the continuation condition $I \leq b$ (Expr(i)) is: 1

Notion of Algorithm Complexity

Rules for Counting the Number of Fundamental Operations

Procedure Calls For non-recursive procedure and function calls, we can calculate the complexity of these calls and take them into account based on their nesting in the algorithm.

Recursive Calls For recursive procedure and function calls, counting the number of fundamental operations generally leads to solving recurrence relations. Indeed, the number $T(n)$ of operations in a procedure call with an argument of size n is expressed, based on recursion, in terms of various $T(k)$ for $k < n$. The example of Fibonacci given in the introductory chapter illustrates this case.

Complexity of Subroutine Calls:

1. Without recursion:
   Example:
   Subroutine A:
   I_1 ; I_2 ; Call of B; I_3 ;
   $Nb(A) = Nb(I_1) + Nb(I_2) + Nb(B) + Nb(I_3)$
2. In case of recursion, calculating Nb(A) leads to the resolution of a recurrence relation.
   Example:
   Function fact(n:integer):integer
   Begin
   If (n=0) then fact ← 1 else fact ← n × fact(n-1) ;
   End;
   The number $T(n)$ of fundamental operations ($\times$) satisfies:
   $T(0) = 0$ and $T(n) = T(n-1) + 1$ for $n \geq 1$
   By direct resolution, we obtain: $T(n) = n$

Notion of Algorithm Complexity

Data Size

It is evident that the calculation of an algorithm's cost depends on the data it operates on.

1. First, it is necessary to define a measure of size for the data that reflects the amount of contained information.

Example

When adding or multiplying integers, a meaningful measure is the number of digits in the numbers.

2. For certain algorithms, the execution time depends not only on the size of the data, but most of the time, the complexity also varies for fixed data size based on the specific data. Thus, the number of operations will vary according to the nature of the data, leading to the use of complexities in the worst-case, best-case, and average scenarios.

Notion of Algorithm Complexity

Data Size

Example

Let's consider a problem $P(n)$ with data size n. Let x be an instance of problem P. For example, let $P(n)$ be the problem of arranging a list of length n in ascending order. An instance x could be sorting the array $\langle 5, 7, 12, 6, 8 \rangle$ where $n = 5$. Another instance could be $\langle 154, 45, 89, 12, 1547, 4 \rangle$, ...

- The problem is formally defined for a unique understanding, while its number of instances is generally infinite.

The challenge in computing complexity is to reason about this infinity of instances by establishing a function $c(n)$ that characterizes the number of operations based on the variable n, representing the data size.

Notion of Algorithm Complexity

Types of Analysis (Best and Worst Cases)

1. **Best-Case Complexity**
   The cost $Min_A(n)$ of an algorithm A in the best-case scenario for a problem $P(n)$ is defined as:
   $$Min_A(n) = \min_{|x|=n} c(x)^1$$

2. **Worst-Case Complexity**
   The cost $Max_A(n)$ of an algorithm A solving problem $P(n)$ in the most unfavorable or worst-case scenario is defined as the maximum cost over all data of size n:
   $$Max_A(n) = \max_{|x|=n} c(x)$$

[1] The execution cost of the algorithm on data x

Notion of Algorithm Complexity

Types of Analysis (Average Case)

3 Average-Case Complexity

- In situations where the worst-case scenario is expected to be rare, we are more interested in the average cost of the algorithm.
- A correct formulation of this average cost assumes that a probability distribution over data of size n is known.
- If $p(x)$ is the probability of data x, the average cost $Avg_A(n)$ of an algorithm A solving problem $P(n)$ on data of size n is defined as:

$$Avg_A(n) = \sum_{|x|=n} p(x)c(x)$$

- Often, a uniform distribution is assumed, meaning $p(x) = \frac{1}{|D_n|}$[2]. In this case, the expression for average cost becomes:

$$Avg_A(n) = \frac{1}{|D_n|} \sum_{|x|=n} c(x)$$

- Average-case complexity and extreme complexities are related by the inequality: $Min_A(n) \leq Avg_A(n) \leq Max_A(n)$

[2] D_n is the set of data of size n

1. Average-case complexity is much harder to determine than worst-case complexity, partly because the analysis becomes mathematically challenging and also because it is not always easy to establish a suitable probability model for the problem.
2. Uniform distribution means that all configurations of data with a fixed size n are equally probable.
3. If the algorithm's behavior depends solely on the data size (as in the example of matrix multiplication), then these three quantities coincide. However, in general, this is not the case, and we don't even know if the average cost is closer to the minimal or maximal cost.

Asymptotic Analysis and Function Magnitudes

- We have determined the complexity of an algorithm as a function of the size of the data.
- It is crucial to understand the rate of growth of this function as the data size increases.
  $\Longrightarrow$ For solving a small-sized problem, the chosen method matters less.
  $\Longrightarrow$ However, for a large-sized problem, performance differences between algorithms can be substantial.
- Often, a simple approximation of the complexity function is sufficient to determine whether an algorithm is usable or not, or to compare different algorithms.

Example

For a large n, it is often of secondary importance whether an algorithm performs $n+1$ or $n+2$ operations.

Asymptotic Measurement and Function Magnitudes

Sometimes, multiplicative constants are also of little importance:

- Let's consider comparing algorithm A_1 with complexity $M_1(n) = n^2$ and algorithm A_2 with complexity $M_2(n) = 2n$. A_2 is better than A_1 for almost all n ($n > 2$).
- Similarly, if $M_1(n) = 3n^2$ and $M_2(n) = 25n$, A_2 is better than A_1 for $n > 8$.
- **Neglecting a term in an addition:** Regardless of the multiplicative constants k_1 and k_2 such that $M_1(n) = k_1 n^2$ and $M_2(n) = k_2 n$, algorithm A_2 is always better than A_1 for some value of n, as the function $f(n) = n^2$ grows much faster than the function $g(n) = n$. Indeed:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

We say that the asymptotic order of magnitude of $f(n)$ is strictly larger than that of $g(n)$. In this case, we can write that $f(n)$ is of the same order as $f(n) + g(n)$.

Asymptotic Measurement and Function Magnitudes

- **Neglecting a Constant:** On the other hand, regardless of the multiplicative constants k_1 and k_2 such that $M_1(n) = k_1 f(n)$ and $M_2(n) = k_2 f(n)$, algorithm A_2 is always of the same order as A_1. In fact:

$$\lim_{n \to \infty} \frac{k_1 f(n)}{k_2 f(n)} = \frac{k_1}{k_2}$$

  In this case, we can state that $M_1(n)$ is of the same order as $M_2(n)$.

Asymptotic Measurement and Function Magnitudes

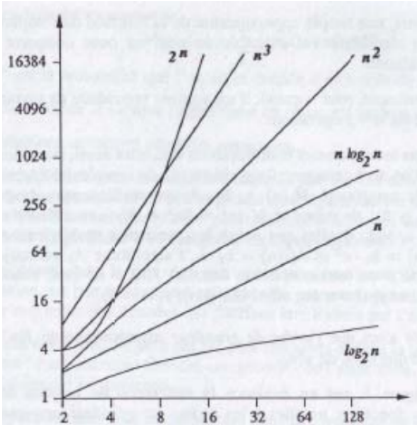


Figure: Growth of Certain Common Functions

The asymptotic orders of magnitude of the functions 1, $\log_2(n)$, $n\log_2(n)$, n^2, n^3, 2^n strictly increase; these functions form a scale of comparison.

A scale of comparison is a set of reference functions ordered by asymptotic order of magnitude.

Asymptotic Measure and Function Magnitudes

- To analyze the complexity $M_A(n)$ of an algorithm A, we aim to determine the asymptotic order of magnitude of $M_A(n)$
  $\Longrightarrow$ we search within a comparison scale, which may be more comprehensive than that formed by the functions in the previous figure, for a function that has a similar growth rate to $M_A(n)$

Suppose we need to compare two algorithms A_1 and A_2 with complexities $M_{A_1}(n)$ and $M_{A_2}(n)$. If the order of magnitude of $M_{A_1}(n)$ is strictly greater than the order of magnitude of $M_{A_2}(n)$, then we can immediately conclude that A_2 is better than A_1 for large n. However, if $M_{A_1}(n)$ and $M_{A_2}(n)$ have the same asymptotic order of magnitude, a more detailed analysis is required to compare A_1 and A_2.

Landau Notations

- To compare the asymptotic order of magnitude of complexity functions, we commonly use the following notion:

Definition

Given two functions f and g from $\mathbb{N}$ to $\mathbb{R}^+$,

$$f = O(g) \Leftrightarrow \exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ such that } \forall n > n_0, f(n) \leq c.g(n)$$

It is also said that:

$$f = \Omega(g) \Leftrightarrow g = O(f)$$

- Thus, $f = O(g)$ means that the asymptotic order of magnitude of f is less than or equal to that of g
  $\Longrightarrow$ We say that f is asymptotically dominated by g
  $\Longrightarrow$ **For example:** $2n = O(n^2)$, and also $2n = O(n)$
- This notion provides an upper bound on the asymptotic order of magnitude of f

1. $f = O(g)$ means that $f(n)$ is bounded above by $g(n)$ with a positive constant factor for sufficiently large n.
   $f = \Omega(g)$ means that $f(n)$ is bounded below by $g(n)$ with a positive constant factor for sufficiently large n.
2. Interpretation of the mathematical definition: Calculating the exact number of operations $f(n)$ is often out of reach for various reasons. This difficulty is overcome by finding a function g that bounds and dominates the number of operations.

Landau Notations

- Bounding the complexity's magnitude is not always sufficient to compare the performance of different algorithms with the same magnitude, as one needs to know the exact orders of magnitudes, not just upper bounds.
- When we say that the complexity $M_A(n)$ of an algorithm A is in $h(n)$, we mean that its asymptotic order of magnitude is precisely $h(n)$[3].
- This notion is formalized as follows:

Definition

Given two functions f and g from $\mathbb{N}$ to $\mathbb{R}^+$,

$$f = \Theta(g) \Leftrightarrow \exists c, d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ such that } \forall n > n_0, d.g(n) \leq f(n) \leq c.g(n)$$

Or equivalently,

$$f = \Theta(g) \Leftrightarrow f = O(g) \text{ and } g = O(f)$$

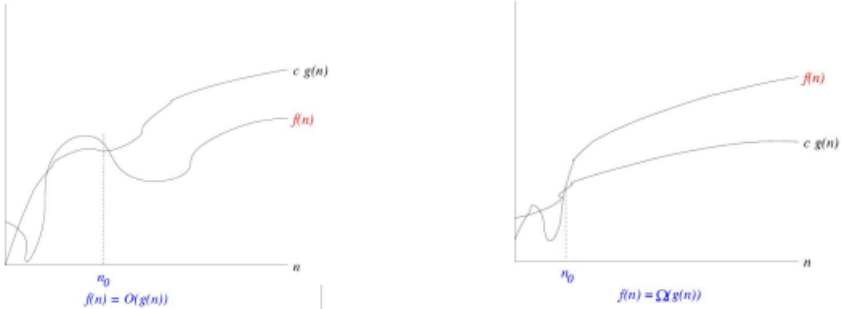
[3] $h(n)$ is the smallest upper bound

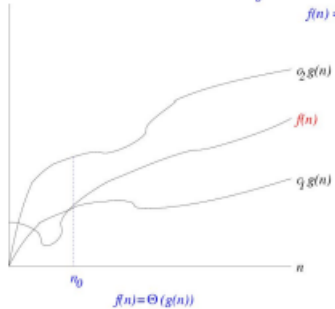
1. $f = \Theta(g)$ means that the ratio $\frac{f(n)}{g(n)}$ is bounded between two positive constants for sufficiently large n.

Landau Notations

- If $f = \Theta(g)$, we say that f and g have the same asymptotic order of magnitude.
- The Θ notation is more precise than the O notation.
  $\Longrightarrow$ **For example:** $2n = \Theta(n)$, but $2n$ is not in $\Theta(n^2)$.
- However, in most algorithmic works, analysis results are presented in the form of $O(f(n))$, even though a precise count of fundamental operations often leads to the conclusion that the complexity is exactly $\Theta(f(n))$.
  $\Longrightarrow$ **For example:** It is often stated that heapsort is in $O(n \log(n))$, but in fact, it is in $\Theta(n \log(n))$.

Landau Notations (Graphical Illustration)





$f = O(g)$

If there exists a strictly positive constant c and an integer n_0 such that:
$n > n_0$, $f(n) \leq c.g(n)$

$f = \Omega(g)$

If there exists a strictly positive constant c and an integer n_0 such that:
$n > n_0$, $f(n) \geq c.g(n)$

$f = \Theta(g)$

If there exist two strictly positive constants c and d and an integer n_0 such that:
$n > n_0$, $d.g(n) \leq f(n) \leq c.g(n)$

Execution Time and Data Size

Complexity / size	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	$\simeq 1\mu s$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	$\simeq 1\mu s$	9,9 μs	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	$\simeq 1\mu s$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	$\simeq 1\mu s$	16,6 μs	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	$\simeq 1\mu s$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

- This table provides an estimate of the execution time for each algorithm for different data sizes n of the problem on a computer capable of performing 10^6 operations per second.

It clearly demonstrates that as the size of the data increases, the differences between the various execution times become more pronounced.

The notion of order of magnitude of the complexity of algorithms has significant practical importance. Let's consider a scenario where we have seven algorithms to solve a given problem, and their worst-case complexities have the following order of magnitude: 1 (constant function, independent of data size), $log_2(n)$ (logarithmic function base 2), n (linear function), $nlog_2(n)$ (quasi-linear function), n^2 (polynomial function of degree 2), n^3 (polynomial function of degree 3), and 2^n (exponential function).

1μs (microseconds) $= 10^{-6}s$
1ms (milliseconds) $= 10^{-3}s$

Execution Time and Data Size

Complexity / Calculation time	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

- This table provides an estimate of the maximum data size that can be processed by each of the algorithms within a fixed execution time (and still on a computer performing 10^6 operations per second).

According to these two tables, it is evident that certain algorithms are usable for solving problems on computers, while others are not or are less practical.

Execution Time and Data Size

Algorithms that are usable for large data sizes are those that execute in the following time complexities:

Constant	Example: Accessing the k-th element of an array
Logarithmic	Example: Binary search, or in binary search trees
Linear	Example: Sequential search
Quasi-Linear (n.log(n))	Example: Efficient sorting algorithms

- Algorithms that take polynomial time, i.e., $\Theta(n^k)$ with $k > 0$, are truly usable only for $k < 2$.
- When $2 \leq k \leq 3$, they are only suitable for medium-sized problems, and when k exceeds 3, they can handle only small problems.
- Algorithms with exponential time complexity, like $\Theta(2^n)$ for instance, are almost unusable, except for very small-sized problems. $\Longrightarrow$ Such algorithms have been labeled as inefficient.

Execution Time and Data Size

- The table below shows how data size and execution time vary in relation to each other.

If the computer's processing speed is increased by a factor of 10
$\Longrightarrow$ The maximum data size that can be processed by an exponential algorithm is hardly affected
$\Longrightarrow$ While the data size that can be processed by a linear algorithm is clearly multiplied by 10.

Complexity	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
Evolution of time when the size is multiplied by 10	t	t+3,32	10×t	(10+ε)×t	100×t	1000×t	t^{10}
Evolution of the size when the time is multiplied by 10	∞	n^{10}	10×n	(10−ε)×n	3,16×n	2,15×n	n+3,32

It is therefore still relevant to seek efficient algorithms, even as technological advancements enhance hardware performance.

Optimality of an Algorithm

- Suppose we have an algorithm A to solve a given problem
  $\Longrightarrow$ It is natural to wonder if we can find an even better algorithm than A
  $\Longrightarrow$ Hence, it is interesting to know the complexity of the best possible algorithm for solving a problem
- Consider a problem P, and let's look at the class C of all algorithms solving P
- We also have a complexity measure, which is the number of fundamental operations (evaluated either in the worst case or on average)

- We define the optimal complexity of class C as the lower bound of complexities among the algorithms in the class
- An algorithm A in class C is called optimal if its complexity is equal to the optimal complexity of C, denoted as $M_{opt}(n)$
- Therefore, an algorithm A in class C is optimal if there is no algorithm B in C that solves problem P with fewer operations than A

Optimality of an Algorithm

- Sometimes, we cannot precisely determine $M_{opt}(n)$, but we can know the order of magnitude of the optimal complexity of the class

In this case, an algorithm A in class C is considered optimal if its complexity is of lower or equal order of magnitude compared to the complexity of any algorithm in class C:

$$\forall B \in C, M_A = O(M_B)$$

The order of magnitude of $M_{opt}(n)$ is then $\Theta(M_A)$

- It is clear that in this situation, multiple algorithms with the same order of complexity can be optimal

Optimality of an Algorithm

Example: Finding the Largest Element in a List

Algorithm 1: FindMax(L)

Data: L: a non-empty list with n integer elements
Result: Finds the largest element M in the non-empty list L

```
1 M ← L[1] ;
2 for i ← 2 to n do
3 |   if L[j] > M then
4 |   |   M ← L[j] ;
5 |   end
6 end
```

- $Min_A(n) = Avg_A(n) = Max_A(n) = n - 1$
- Now, we might wonder if this number $n - 1$ can be improved? In fact, it can be proven that this algorithm is optimal[4].

[4] The proof will be covered in the tutorial session

1. It can be observed that the number of comparisons is equal to the number of iterations in the for loop (minus one). Regardless of the evaluation measure considered ($Min_A(n)$, $Avg_A(n)$, or $Max_A(n)$), the complexity remains the same.

Recurrence Equations and Solving Techniques

- When analyzing a recursive algorithm, we often obtain a complexity function that is itself recursive.
- It frequently happens that, in order to analyze the complexity of an algorithm, we need to solve a recurrence.
- The goal of this final part of the chapter is to study methods for solving recurrence equations.

QuickSort:

Average number of comparisons:

$$\begin{cases} C_0 & = 0 \\ C_1 & = 0 \\ C_N & = N + 1 + \sum_{k=1}^{N} \frac{1}{N}(C_{k-1} + C_{N-k}) \text{ for } N > 1 \end{cases}$$

Merge Sort:

Number of comparisons in the worst case:

$$\begin{cases} C_1 & = 0 \\ C_N & = C_{\lceil \frac{N}{2} \rceil} + C_{\lfloor \frac{N}{2} \rfloor} + N \text{ for } N > 1 \end{cases}$$

```
1 function QuickSort(A,lo,hi):
2     if lo < hi then
3         p ← choose the pivot
            element ;
4         swap A[hi] and A[p] ;
5         i ← Partition(A,lo,hi) ;
6         QuickSort(A,lo,i − 1) ;
7         QuickSort(A,i + 1,hi) ;
8     end
```

- Performs quick sort on the elements of array A from index lo to index hi.
- Chooses a pivot, places it at position hi, and calls the partition function.
- Recursively calls itself on the left and right partitions.

The function Partition(A,lo,hi) partitions the elements of array A from index lo to index hi using $A[hi]$ as the pivot. It returns the position of the pivot in the partitioned array.

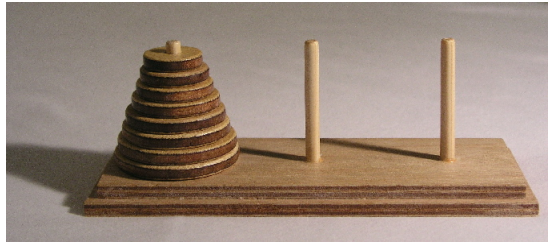
Complexity of QuickSort:

Best case: $O(n \log n)$ (pivot chosen partitions the array into two equal parts).

Worst case: $O(n^2)$ (pivot partitions the array with one empty side and the other side with $n-1$ elements). Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Recurrence Equations and Solving Techniques

Towers of Hanoi:



Goal: Move the entire tower from the first peg to one of the other two pegs.
Constraints:

- Only one disk can be moved at a time.
- A disk can only be placed on a disk with a larger diameter.

The number of disk movements for a tower of n disks from one peg to another satisfies the following recurrence relation:

$$\begin{cases} T_1 & = 1 \\ T_N & = 2\,T_{N-1} + 1 \text{ for } N > 1 \end{cases}$$

QuickSort Complexity (continued from previous slide):

Average Complexity: The average number of comparisons C_N to quickly sort an array of N distinct elements is in $O(2n \ln n)$
C_N satisfies the recurrence:

$$\begin{cases} C_0 & = 0 \\ C_1 & = 0 \\ C_N & = N + 1 + \frac{C_0 + C_{N-1}}{N} + \frac{C_1 + C_{N-2}}{N} + \ldots + \frac{C_{N-1} + C_0}{N} \text{ for } N > 1 \end{cases}$$

The partition requires $N + 1$ comparisons.
If the pivot is in the first position, the left call will receive an array of 0 elements, and the right call an array of $N - 1$ elements.
If it's in the second position, the left call will receive an array of 1 element, and the right call an array of $N - 2$ elements.
... etc
It is necessary to consider the probability of each pivot position occurring (the probability of all pivot positions). We assume a uniform distribution ($p(x) = \frac{1}{N}$).

Recurrence Equations and Solving Techniques

Calculating Fibonacci Numbers :

Algorithm 2: FIB1(n)

```
1 if n = 0 then
2 |   return 0;
3 else
4 |   if i = 1 then
5 |   |   return 1;
6 |   else
7 |   |   return FIB1(n-1)+FIB1(n-2);
8 |   end
9 end
```

Number of additions performed by the FIB1 algorithm for an input of n :

$$\begin{cases} F_0 & = 0 \\ F_1 & = 0 \\ F_N & = F_{N-1} + F_{N-2} + 1 \text{ for } N > 1 \end{cases}$$

Recurrence Equations and Solving Techniques

Solving Techniques for Recurrence Equations

We aim to obtain an analytical solution to a recurrence equation
Several possible approaches:

- → Guess-and-verify
- → Plug-and-chug (telescoping)
- → Recursion trees
- → Characteristic equations (linear)
- → Master theorem (general theorem)
- → Change of variable

Recurrence Equations and Solving Techniques

Guess-and-Verify Method

Principle:

1. Calculate the initial values of T_n;
2. Guess an analytical solution;
3. Prove its correctness, for example, by induction

Application:

$T_n = 2T_{n-1} + 1$ (Tower of Hanoi):

n	1	2	3	4	5	6	...
T_n	1	3	7	15	31	63	...

$\Longrightarrow$ We guess $T_n = 2^n - 1$

- We must prove (by induction) that the solution is correct

Recurrence Equations and Solving Techniques

Guess-and-Verify Method

Theorem

$T_n = 2^n - 1$ *satisfies the recurrence:*

$$\begin{cases} T_1 & = 1 \\ T_N & = 2\,T_{N-1} + 1 \text{ for } N > 1 \end{cases}$$

Proof.

By induction on n. Let $P(n) =$ " $T_n = 2^n - 1$"
Base case: $P(1)$ is true since $T_1 = 1 = 2^1 - 1$
Inductive case: We will show that $T_n = 2^n - 1$ (for $n \geq 2$) is true whenever $T_{n-1} = 2^{n-1} - 1$ is true:

$$\begin{aligned} T_n &= 2\,T_{n-1} + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 1 \end{aligned}$$

Recurrence Equations and Solving Techniques

"Plug-and-Chug" Method (Brute Force)

(also known as telescoping or method of summation factors)

1. "Plug" (apply the recurrence equation) and "Chug" (simplify)

$$\begin{aligned} T_n &= 1 + 2\,T_{n-1} \\ &= 1 + 2(1 + 2\,T_{n-2}) \\ &= 1 + 2 + 4\,T_{n-2} \\ &= 1 + 2 + 4(1 + 2\,T_{n-3}) \\ &= 1 + 2 + 4 + 8\,T_{n-3} \\ &= \ldots \end{aligned}$$

Note: Simplify moderately.

2. Identify and Verify a "Pattern"
   - Identification:

$$T_n = 1 + 2 + 4 + \ldots + 2^{i-1} + 2^i\,T_{n-i}$$

- Verification by expanding one more step:

$$\begin{aligned} T_n =& 1 + 2 + 4 + ... + 2^{i-1} + 2^i(1 + 2T_{n-(i+1)}) \\ =& 1 + 2 + 4 + ... + 2^{i-1} + 2^i + 2^{i+1} T_{n-(i+1)} \end{aligned}$$

1. Express the n-th term in terms of previous terms
   Note that the base case, $T_1 = 1$, occurs when $n - i = 1$. That is, $i = n - 1$, which gives:

$$\begin{aligned} T_n =& 1 + 2 + 4 + ... + 2^{n-2} + 2^{n-1} T_1 \\ =& 1 + 2 + 4 + ... + 2^{n-2} + 2^{n-1} \end{aligned}$$

2. Find an analytical solution for the n-th term

$$\begin{aligned} T_n =& 1 + 2 + 4 + ... + 2^{n-2} + 2^{n-1} \\ =& \sum_{i=0}^{n-1} 2^i \\ =& \frac{1 - 2^n}{1 - 2} \\ =& 2^n - 1 \end{aligned}$$

Recurrence Equations and Solving Techniques

"Plug-and-Chug" Method (First Order Linear Recurrence)

- The plug-and-chug applied to a recurrence of the form:

$$T_n = T_{n-1} + f(n)$$

directly yields:

$$T_n = T_0 + \sum_{k=1}^{n} f(k) \text{ or } T_n = T_1 + \sum_{k=2}^{n} f(k)$$

- If the coefficient of T_{n-1} is not 1:

$$T_n = g(n)\,T_{n-1} + f(n)$$

dividing both sides by $h(n) = g(n).g(n-1).g(n-2)...g(1)$ reduces it to a recurrence of the form:

$$\frac{T_n}{h(n)} = \frac{T_{n-1}}{h(n-1)} + \frac{f(n)}{h(n)} \Rightarrow T_n = h(n)\left(\frac{T_0}{h(0)} + \sum_{k=1}^{n} \frac{f(k)}{h(k)}\right)$$

Useful if the product $h(n)$ has a simple form.

1. Recurrences are classified based on the combination of terms, the nature of coefficients, as well as the number and nature of the preceding terms involved.
   Examples:
   First-order linear: $a_n = na_{n-1} - 1$
   First-order nonlinear: $a_n = 1/(1 + a_{n-1})$

   Second-order linear: $a_n = a_{n-1} + 2a_{n-2}$
   Second-order nonlinear: $a_n = a_{n-1}2a_{n-2} + \sqrt{a_n - 2}$
   Second-order with variable coefficients: $a_n = na_{n-1} + (n-1)a_{n-2} + 1$
2. Non-linear recurrences involve non-linear operations like multiplication, division, or other functions that are not simple sums or differences of previous terms.

Recurrence Equations and Solving Techniques

"Plug-and-Chug" Method (First Order Linear Recurrence)

Example 1 : $T_0 = 0,\ T_n = 2T_{n-1} + 2^n$ (for $n > 0$)
By dividing both sides by $h(n) = 2 \cdot 2 \cdot ... \cdot 2 = 2^n$, we obtain:

$$\frac{T_n}{2^n} = \frac{T_{n-1}}{2^{n-1}} + 1 \Rightarrow T_n = 2^n(0 + \sum_{k=1}^{n} 1) = n \times 2^n$$

Example 2: $T_0 = 0,\ T_n = (1 + \frac{1}{n})T_{n-1} + 2$ (for $n > 0$) (Average Case Complexity of Quicksort)
By dividing both sides by:

$$h(n) = \frac{n+1}{n} \times \frac{n}{n-1} \times ... \times \frac{3}{2} \times \frac{2}{1} = n + 1$$

we obtain:

$$\frac{T_n}{n+1} = \frac{T_{n-1}}{n} + \frac{2}{n+1} = \sum_{k=1}^{n} \frac{2}{k+1} \Rightarrow T_n \approx (n+1)2[\ln(n+1) + \gamma - 1]^5$$

$$\Rightarrow T_n = O(n \log n)$$

[5] where $\gamma \approx 0.577$ is the Euler's constant.

1. Average Complexity of QuickSort:
   $C_N = N + 1 + \sum_{k=1}^{N} \frac{1}{N}(C_{k-1} + C_{N-k})$ for $N > 1$
   We bring this formula into the form: $C_N = g(n)C_{N-1} + f(n)$

$$\begin{aligned} C_N &= N + 1 + \tfrac{2}{N}(C_0 + C_1 + ... + C_{N-1}) \\ N \times C_N &= N^2 + N + 2(C_0 + C_1 + ... + C_{N-1}) \\ (N-1) \times C_{N-1} &= (N-1)^2 + (N-1) + 2(C_0 + C_1 + ... + C_{N-2}) \end{aligned}$$

$$N \times C_N - (N-1) \times C_{N-1} = 2N + 2C_{N-1}$$

$$\begin{aligned} N \times C_N &= 2N + 2C_{N-1} + (N-1) \times C_{N-1} \\ &= 2N + C_{N-1} + N \times C_{N-1} \end{aligned}$$

$$\begin{aligned} C_N &= 2 + \tfrac{1}{N} C_{N-1} + C_{N-1} \\ &= (1 + \tfrac{1}{N})C_{N-1} + 2 \end{aligned}$$

2. The Euler-Mascheroni constant γ is defined as follows:

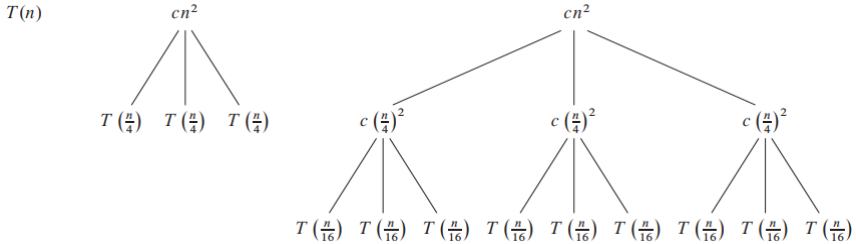
$$\gamma = \lim_{n \to \infty} \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} - \ln n\right)$$

Recurrence Equations and Solving Techniques

"Recursion Trees" Method

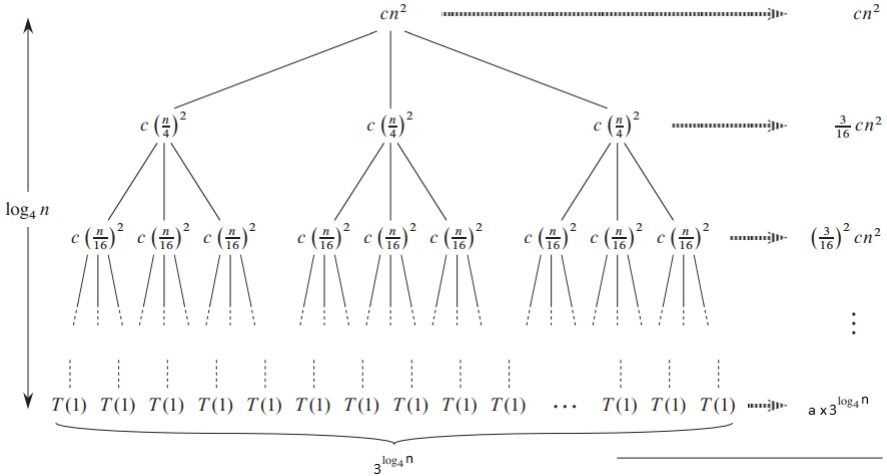
- Graphical approach to guess an analytical solution (or an asymptotic bound) for a recurrence
- The solution must always be subsequently proven (typically by induction)
- Illustrated with the following recurrence:

$$\begin{cases} T(1) = & a \\ T(n) = & 3T(\frac{n}{4}) + cn^2 \text{ for } n > 1 \end{cases}$$



Recurrence Equations and Solving Techniques

"Recursion Trees" Method"



Recurrence Equations and Solving Techniques

"Recursion Trees" Method"

- The total cost is the sum of the cost at each level of the tree:

$$
\begin{aligned}
T(n) =& cn^2 + \frac{3}{16}cn^2 + ... + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + a3^{\log_4 n} \\
=& \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + a3^{\log_4 n} \\
<& \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + a3^{\log_4 n} \\
\approx& \frac{1}{1 - \left(\frac{3}{16}\right)} cn^2 + a3^{\log_4 n} \\
=& O(n^2)
\end{aligned}
$$

Recurrence Equations and Solving Techniques

Summary

Three empirical approaches to find an analytical solution:

- → "Guess-and-verify"
- → "Plug-and-Chug"
- → "Recursion Trees"

These approaches are generic but

- → It's not always easy to find the pattern
- → The obtained sum needs to be solved
- → The solution must be verified by induction

We will explore more systematic methods to solve specific recurrences:

- → Linear recurrences of order $k \geq 1$ with constant coefficients (exact analytical solution)
- → "Divide-and-conquer" recurrences (asymptotic bounds only)

Empirical: Based solely on experience, observation, not on theory or reasoning.

Recurrence Equations and Solving Techniques

Linear Recurrences

Definition

A uniform linear recurrence of degree k with constant coefficients is a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k)$$

where $a_1, a_2, ..., a_k \in \mathbb{R}$ are constants, $a_k \neq 0$, and with k initial conditions:

$$f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$$

Example: The following recurrence $f(n) = 8f(n-2) - 16f(n-4)$ is a uniform linear recurrence with constant coefficients.

Recurrence Equations and Solving Techniques

Linear Recurrences

Definition

A (general) non-uniform linear recurrence of degree k with constant coefficients is a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

with k initial conditions:

$$f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$$

where $a_1, a_2, ..., a_k \in \mathbb{R}$ are constants, $a_k \neq 0$, and $g(n) \neq 0$ is a function depending only on n. The associated uniform recurrence relation (AURR) is:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k)$$

Example: The non-uniform linear recurrence relation $f(n) = 3f(n-1) + 2n$ has an associated AURR $f(n) = 3f(n-1)$.

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Consider a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

with k initial conditions: $f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$

Step 1: Find the roots of the characteristic equation of the associated uniform recurrence relation (AURR)

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - ... - a_{k-1} r - a_k = 0$$

Note: The term $g(n)$ is not included in the characteristic equation

- **Characteristic root:** Solution r_0 of the characteristic equation
- **Multiplicity m of a root r_0:**

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - ... - a_{k-1} r - a_k = (r - r_0)^m Q(r)$$

where r_0 is not a root of $Q(r)$

AURR: Associated uniform Recurrence Relation.

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Step 2: Find a uniform solution $u(n)$, disregarding initial conditions

- If the characteristic equation has t distinct roots $r_1, r_2, ..., r_t$ with multiplicities $m_1, ..., m_t$ respectively ($m_i \geq 1$ and $m_1 + m_2 + + m_t = k$), then the solution $\{u(n)\}$ of the recurrence relation is of the form:

$$u(n) = p_1(n)r_1^n + p_2(n)r_2^n + ... + p_t(n)r_t^n$$

where $p_i(n)$ is a polynomial of degree $m_i - 1$ in the variable n

Example

If the characteristic equation of a uniform linear recurrence has roots:

r_1 with multiplicity 1, r_2 with multiplicity 3, r_3 with multiplicity 5

then the solution is of the form:

$$u(n) = \alpha_1 r_1^n + (\alpha_2 + \alpha_3 n + \alpha_4 n^2)r_2^n + (\alpha_5 + \alpha_6 n + \alpha_7 n^2 + \alpha_8 n^3 + \alpha_9 n^4)r_3^n$$

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Step 3: Find a particular solution $p(n)$, disregarding initial conditions
In the case where $g(n) = Q(n)s^n$, with $Q(n)$ being a polynomial of degree t and $s \in \mathbb{R}$, then

- if s is not a root of the characteristic equation of the AURR, then
$$p(n) = (\beta_0 + \beta_1 n + ... + \beta_{t-1} n^{t-1} + \beta_t n^t) s^n$$
- otherwise, if s is a root of the characteristic equation with multiplicity m of the AURR, then
$$p(n) = n^m (\beta_0 + \beta_1 n + ... + \beta_{t-1} n^{t-1} + \beta_t n^t) s^n$$

Example

What is the form of the particular solution of the recurrence relation $f(n) = 5f(n-1) - 6f(n-2) + g(n)$ if:
a) $g(n) = 7^n$?
b) $g(n) = 2^n$?
c) $g(n) = n^2(-1)^n$?
d) $g(n) = (n-1)3^n$?

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Solution

The AURR is $f(n) = 5f(n-1) - 6f(n-2)$, the characteristic equation is $r^2 - 5r + 6 = 0$. Therefore, the characteristic roots are $r_1 = 2$ and $r_2 = 3$, both with a multiplicity of 1.
a) $p(n) = \beta_0 7^n$ ($Q(n) = 1$; $t = 0$; $s = 7$ is not a root)
b) $p(n) = n\beta_0 2^n$ ($Q(n) = 1$; $t = 0$; $s = 2$ is a root with multiplicity 1)
c) $p(n) = (\beta_0 + \beta_1 n + \beta_2 n^2)(-1)^n$ ($Q(n) = n^2$; $t = 2$; $s = -1$ is not a root)
d) $p(n) = n(\beta_0 + \beta_1 n)3^n$ ($Q(n) = n - 1$; $t = 1$; $s = 3$ is a root with multiplicity 1)

When solving a non-uniform linear recurrence relation, the values of the coefficients in the particular solution (the β_i in the example above) do not depend on the initial conditions. Their values can be directly determined from the recurrence relation. We then use the fact that the particular solution is a solution of the initial recurrence relation.

Roots of a Quadratic Polynomial:

If $P = aX^2 + bX + c$ with $a \neq 0$, three cases arise:

- $\Delta = b^2 - 4ac < 0$, the polynomial has no roots in $\mathbb{R}$.
- $\Delta = b^2 - 4ac > 0$, the polynomial has two distinct roots in $\mathbb{R}$ which are: $x_1 = \dfrac{-b - \sqrt{\Delta}}{2a}$ and $x_2 = \dfrac{-b + \sqrt{\Delta}}{2a}$
- $\Delta = b^2 - 4ac = 0$, the polynomial has a double root in $\mathbb{R}$ which is: $x = \dfrac{-b}{2a}$

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Example

In example (a), the particular solution is of the form $p(n) = \beta_0 7^n$. Thus, we know that $\beta_0 7^n$ is a solution of the recurrence relation $f(n) = 5f(n-1) - 6f(n-2) + 7^n$. By assigning a value to n, we obtain an equation in terms of β_0. For instance, we can choose $n = 2$:

$$\begin{aligned} p(n) &= 5p(n-1) - 6p(n-2) + 7^n \\ p(2) &= 5p(1) - 6p(0) + 7^2 \\ 49\beta_0 &= 5(7\beta_0) - 6(\beta_0) + 49 \end{aligned}$$

Thus, we find $\beta_0 = \frac{49}{20}$. Consequently, we have $p(n) = \frac{7^{n+2}}{20}$.

In this example, the particular solution is expressed in terms of a single unknown. Thus, only one equation was needed to determine its value. In general, the particular solution is expressed in terms of multiple unknowns; in that case, as many equations as unknowns are required. To generate more equations, it's enough to substitute different integer values for n in the initial recurrence relation.

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Step 4: Form a general solution, disregarding initial conditions
Just add the uniform solution and the particular solution:

Any solution $\{s(n)\}$ of the recurrence relation:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

is of the form:

$$s(n) = p(n) + u(n)$$

where $\{p(n)\}$ is a particular solution and $\{u(n)\}$ is a solution of the AURR

Step 5: Determine the values of the constants introduced in Step 2

- For each initial condition, apply the general solution to that condition. This yields an equation in terms of the constants to be determined.
- Solve the system of equations formed by these equations

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Example

Find the solution of $f(n) = 4f(n-1) + 3n$, with the initial condition $f(1) = 2$

Solution

Step 1: *Find the roots of the characteristic equation of the associated uniform recurrence relation (AURR)*
We identify the **AURR***:* $f(n) = 4f(n-1)$
We determine the **characteristic equation of the AURR***:* $r - 4 = 0$
We calculate **the roots of the characteristic polynomial of the AURR***:* $r_1 = 4$
Step 2: *Find a uniform solution* $u(n)$*, without considering initial conditions*
We find **the form of the solution of the AURR***:* $u(n) = \alpha_1 4^n$

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Solution

Step 3: *Find a particular solution $p(n)$, without considering initial conditions Since $g(n) = 3n = 3n(1)^n$, we have $s = 1$ (which is not a root) and $t = 1$ (the degree of $3n$). The solution $p(n)$ is therefore of the form:*

$$p(n) = \beta_0 + \beta_1 n$$

We determine **the values of the coefficients** β_0 *and* β_1*: since $p(n)$ is a particular solution, it satisfies the recurrence relation, i.e., $p(n) = 4p(n-1) + 3n$. Thus,*

$$\begin{aligned} \beta_0 + \beta_1 n &= 4(\beta_0 + \beta_1(n-1)) + 3n \\ &= (4\beta_0 - 4\beta_1) + (4\beta_1 + 3)n \end{aligned} \Rightarrow \begin{cases} \beta_0 = 4\beta_0 - 4\beta_1 \\ \beta_1 = 4\beta_1 + 3 \end{cases} \Rightarrow \begin{aligned} \beta_0 &= -\frac{4}{3} \\ \text{et } \beta_1 &= -1 \end{aligned}$$

Recurrence Equations and Solving Techniques

Solving Linear Recurrences

Solution

Step 4: *Form a general solution $s(n)$, without considering initial conditions*

$$s(n) = p(n) + u(n) = \alpha_1 4^n - \frac{4}{3} - n$$

Step 5: *Determine the values of the constants introduced in step 2*
We determine the value of the coefficient α_1. Since there is only one coefficient to find, we need one equation:

$$s(1) = \alpha_1 \cdot 4 - \frac{4}{3} - 1 = 2 = f(1) \Longrightarrow \alpha_1 = \frac{13}{12}$$

The solution is:

$$\boxed{s(n) = \left(\tfrac{13}{12}\right) 4^n - \tfrac{4}{3} - n}$$

Recurrence Equations and Solving Techniques

Master theorem (General Theorem)

Theorem

If the recurrence is of the following form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where n: the size of the problem, $a \geq 1$, $b > 1$, and $f(n) = \Theta(n^k \log^p n)$ is a function with $k \geq 0$ and p is a real number
$T(n)$ can then be asymptotically bounded as follows:

1. *If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $a = b^k$, then:*
   1. $p > -1 \longrightarrow T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   2. $p = -1 \longrightarrow T(n) = \Theta(n^{\log_b a} \log \log n)$
   3. $p < -1 \longrightarrow T(n) = \Theta(n^{\log_b a})$
3. *If $a < b^k$, then:*
   1. $p \geq 0 \longrightarrow T(n) = \Theta(n^k \log^p n)$
   2. $p < 0 \longrightarrow T(n) = \Theta(n^k)$

This theorem can be used to determine the time complexity of divide and conquer algorithms if the recurrence is in the form specified in the theorem.

$$T(n) = aT(\frac{n}{b}) + \Theta(n^k \log^p n)$$

where n: the size of the problem
a: the number of sub-problems in the recursion and $a \geq 1$
$\frac{n}{b}$: the size of each sub-problem
$b > 1$, $k \geq 0$, and p is a real number

Recurrence Equations and Solving Techniques

Master theorem (General Theorem)

Example

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 1$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = 1 = \Theta(n^0 \log^0 n)$ ($k = 0$ and $p = 0$)

- $a > b^k (2 > 2^0) \longrightarrow T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

$$T(n) = 4\,T\left(\frac{n}{2}\right) + n$$

For this recurrence, we have $a = 4$, $b = 2$ and $f(n) = n = \Theta(n^1 \log^0 n)$ ($k = 1$ and $p = 0$)

- $a > b^k (4 > 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

Recurrence Equations and Solving Techniques

Master theorem (General Theorem)

Example

$$T(n) = 8\,T\left(\frac{n}{2}\right) + n\log n$$

For this recurrence, we have $a = 8$, $b = 2$ and $f(n) = n\log n = \Theta(n^1 \log^1 n)$ ($k = 1$ and $p = 1$)

- $a > b^k (8 = 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

$$T(n) = 2\,T\left(\frac{n}{2}\right) + n$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = n = \Theta(n^1 \log^0 n)$ ($k = 1$ and $p = 0$)

- $a = b^k (2 = 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$ $(p > -1)$

Recurrence Equations and Solving Techniques

Master theorem (General Theorem)

Example

$$T(n) = 4\,T\left(\frac{n}{2}\right) + n^2 \log n$$

For this recurrence, we have $a = 4$, $b = 2$ and $f(n) = n^2 \log n = \Theta(n^2 \log^1 n)$ ($k = 2$ and $p = 1$)

- $a = b^k (4 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 4} \log^2 n) = \Theta(n^2 \log^2 n)$ $(p > -1)$

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = \frac{n}{\log n} = \Theta(n^1 \log^{-1} n)$ ($k = 1$ and $p = -1$)

- $a = b^k (2 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 2} \log \log n) = \Theta(n \log \log n)$ $(p = -1)$

Recurrence Equations and Solving Techniques

Master theorem (General Theorem)

Example

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \frac{n}{\log^2 n}$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = \frac{n}{\log^2 n} = \Theta(n^1 \log^{-2} n)$ ($k = 1$ and $p = -2$)

- $a = b^k (2 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$ $(p < -1)$

$$T(n) = T\left(\frac{n}{2}\right) + n^2$$

For this recurrence, we have $a = 1$, $b = 2$ and $f(n) = n^2 = \Theta(n^2 \log^0 n)$ ($k = 2$ and $p = 0$)

- $a < b^k (1 < 2^2) \longrightarrow T(n) = \Theta(n^2 \log^0 n) = \Theta(n^2)$ $(p \geq 0)$

Recurrence Equations and Solving Techniques

Master theorem (General Theorem)

Example

$$T(n) = 2\,T\left(\frac{n}{2}\right) + n^2 \log n$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = n^2 \log n = \Theta(n^2 \log^1 n)$ ($k = 2$ and $p = 1$)

- $a < b^k (2 < 2^2) \longrightarrow T(n) = \Theta(n^2 \log^1 n) = \Theta(n^2 \log n)$ $(p \geq 0)$

$$T(n) = 4\,T\left(\frac{n}{2}\right) + \frac{n^3}{\log n}$$

For this recurrence, we have $a = 4$, $b = 2$ and $f(n) = \frac{n^3}{\log n} = \Theta(n^3 \log^{-1} n)$ ($k = 3$ and $p = -1$)

- $a < b^k (4 < 2^3) \longrightarrow T(n) = \Theta(n^3)$ $(p < 0)$

Recurrence Equations and Solving Techniques

Change of Variables

- Sometimes, a change of variables can transform a "divide and conquer" recurrence into a linear recurrence.

Example

Consider the following recurrence:

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

By setting: $n = 2^m$ and $S(m) = T(2^m)$, we obtain:

$$S(m) = T(2^m) = 7T(2^{m-1}) + (2^m)^2 = 7S(m-1) + 4^m$$

Recurrence Equations and Solving Techniques

Change of Variables

- A change of variable allows solving recurrences that may initially appear complex.

Example

Consider the following recurrence:

$$T(n) = 2\,T(\sqrt{n}) + \log n$$

Setting $n = 2^m$ ($m = \log n$), we obtain:

$$T(2^m) = 2\,T\left(2^{\frac{m}{2}}\right) + m$$

Let $S(m) = T(2^m)$. We have:

$$S(m) = 2S\left(\frac{m}{2}\right) + m \Rightarrow S(m) = \Theta(m \log m)$$

Finally:

$$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log n \log\log n)$$

Recurrence Equations and Solving Techniques

Summary

- Tools for solving recurrence equations:
  - → Generic methods: "Guess-and-Verify," "Plug-and-Chug," recursion trees
  - → Linear recurrences of order k (with constant coefficients)
  - → "Divide and Conquer" recurrences: "Master Theorem"

  The last two are the most systematic approaches.
- The most challenging part remains translating a real-world problem into a recurrence equation.