Dynamic Programming

In the Dynamic Programming,

1. We divide the large problem into multiple subproblems.

2. Solve the subproblem and store the result.

3. Using the subproblem result, we can build the solution for the large problem.

4. While solving the large problem, if the same subproblem occurs again, we can reuse the already stored result rather than recomputing it again. This is also called **memoization**.

Dynamic Programming Approaches

1. Bottom-Up approach

2. Top-Down approach

The top-down design approach

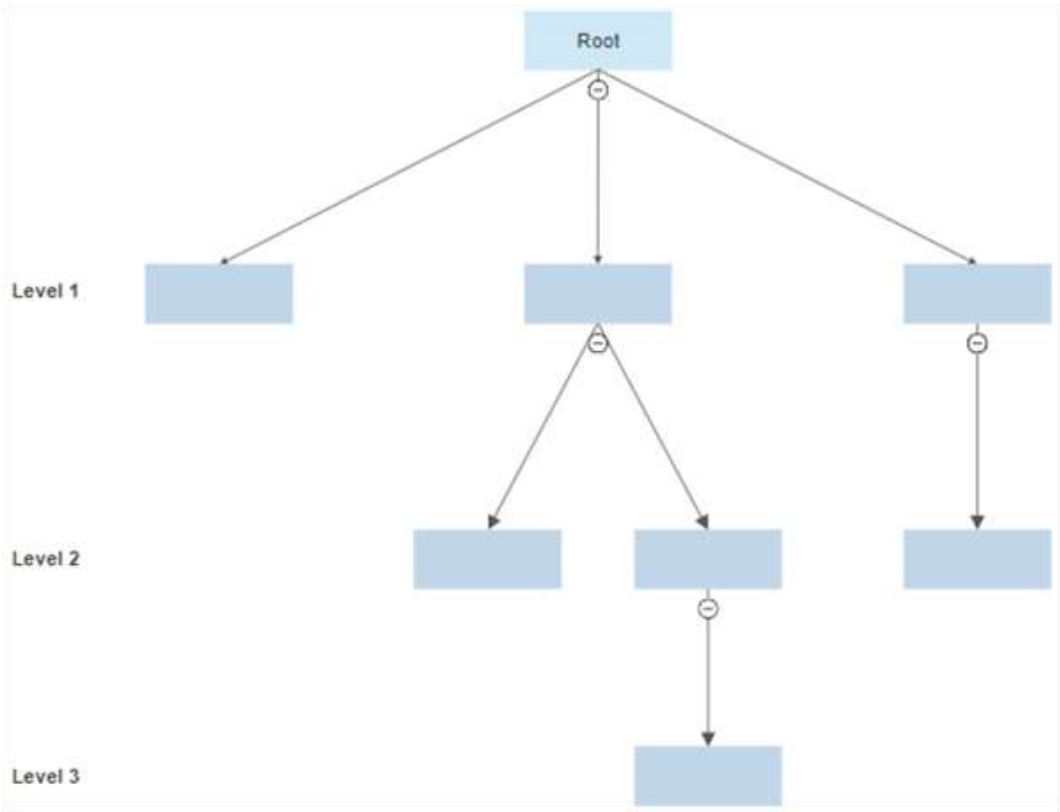
The top-down design approach, also called stepwise refinement, is essential to developing a well-structured program [2]. This approach is a problem-solving technique that systematically breaks a complicated problem into smaller, more manageable pieces. If any of these subproblems is not easier to solve, we further divide the subproblem into smaller parts. We repeat the subdividing process until all small parts are not dividable. Then, we can use a few lines of code to solve every trivial problem. In the end, we put all these little pieces together as a solution to the original problem. This approach makes it easy to think about the problem, code the solution, read the code, and identify bugs.

To demonstrate how to apply the top-down approach to design a program, we tackle a real-world problem:

- We work in the IT department of a manufacturing company. The company web site provides an email address that collects questions and comments from visitors. Customer relationship managers want to extract all email messages into a SQL server database for data mining.

1 – Introducing the Top-Down Design

The top-down design approach is to produce a plan for solving a complicated problem. The output of this design is a tree-like structure chart



The root is the entry point of the program. The first level contains several major subtasks. These subtasks work together to solve the initial problem. If a subtask is not trivial at the first level, we divide it

into smaller tasks, and then we place all these smaller tasks at the second level. This process continues until we reach a level at which breakdown is not necessary. There are several levels between the root and leaves. At each level, the dividable subtasks reference subtasks at the lower level. We define interfaces (or signatures) of these subtasks to communicate with other subtasks at the different levels. We use the following procedure to walk through the top-down design [3]:

1. Start with an initial problem statement.
2. Define subtask at the first level.
3. Divide subtasks at a higher level into more specific tasks.
4. Repeat step (3) until each subtask is trivial.
5. Refine the algorithm into real code.

The program that implements the top-down design reflects the quality code's characteristics: readable, extendable, extensible, and debuggable [4]. Because of descriptive names of functions and variables, and human-readable syntax, people who do not know Python language may infer what the program is doing. When we have new requirements, we can add a piece of code without modifying existing functions. If we find the program does not work correctly, we only need to run suspicious functions to locate the problematic function. Besides, this program allows other people to improve one function without touching code in other functions.

Top-down design and object-oriented programming (OOP) can co-exist [5]. The top-down design approach answers a question about how to start tackling a complicated problem. However, the top-down design approach has some limits [6]. If we do not divide a task into several independent subtasks carefully, we may face a challenge when implementing those subtasks at the lower levels.

Performing the Top-Down Design

Every activity we perform at work is in response to business needs [9]. Programmers should write programs to satisfy business

requirements. Before designing a program, we should perform a business requirement analysis.

From the business requirement analysis, we can define an initial problem statement for the top-down design. Then, we divide the initial problem into several general tasks at the first level in the tree-like structure. Next, we break each dividable task into more specific tasks. This process repeats until we can use several lines of code to implement each subtask. In the end, we put all these little pieces together to solve the initial problem.

3.1 Starting with an Initial Problem Statement

The first step is to perform business analysis and understand what the specifications are asking. Then, we can define an initial problem statement for the design:

- Read new emails from an email box and save email messages into a database table.

3.2 Defining Subtasks at the First Level

Every Python function written in this exercise adopts the input-process-output pattern, a widely used approach in systems analysis and software engineering, to describe a problem-solving process [10]. The input-process-out pattern includes these three steps: (1) Get input, (2) Process, and (3) Output. We should also choose descriptive function names; consequently, others can easily understand the program [11]. When a program can clearly express our intent, we do not need to add comments to the program very much [12].

Since Python's syntax is human-readable, we directly use Python code to express these subtasks. The main() function, the program's entry point, is the root of the structure chart. We breakdown the initial problem statement into two general subtasks

We have decomposed the original problem into two independent subtasks. This decomposition process breaks a complex problem into small parts that are easier to conceive, understand, program, and maintain. We do not need to worry about the detailed definitions of these subtasks in this step. We assume that all the subtasks work correctly, and we use them to solve the initial problem. We also call this process abstraction.

We can visualize this step with the tree-like structure chart; each task in the design is a rectangle. An arrow connecting two rectangles indicates that the one at a higher level uses the one below. We use Python functions to implement these subtasks.

A Python function usually has a name, arguments, and an expected return value. We call this information the interface or signature of the function. To add more information to the chart, we place arguments and return value along the edges. The chart produced from this step should look like Figure 5.

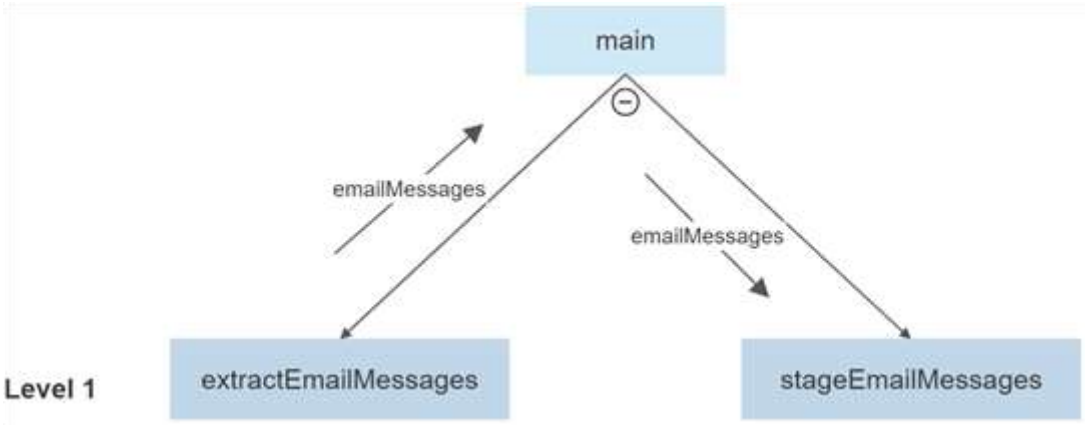
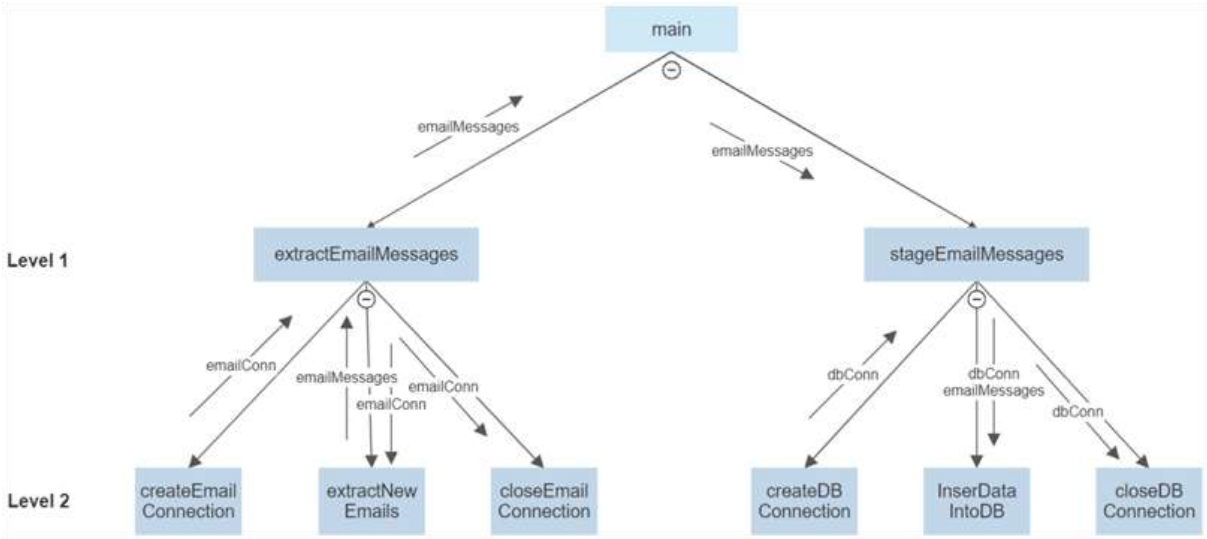


Figure 5 The First Level of the Tree Structure

Since we extract email messages from an email server and then load these messages into a SQL server database, we can store email messages into a list of dictionaries. The keys in the dictionary correspond to columns in the database table.

3.3 Dividing Subtasks at the Higher Level into More Specific Tasks

If any task at a higher level is trivial (i.e., indivisible), we can create a prototype of this function to implement the task, and we do not further divide the task. However, these two tasks at a higher level are still complicated



Summary:

We used the top-down design approach to solve the real-world problem. The top-down design approach uses the divide-and-conquer algorithm that expresses a complex problem in terms of small, simple problems.

We started at the first level in the tree-like structure chart and then worked way down to lower levels. At each level, we broke down a general task to several smaller, more straightforward subtasks. Through this approach, the complicated task is no longer a complicated thing to perform.

Tasks at a higher level may call their subtasks. The interface of a function served as a contract to connect the caller and callees. We immediately produced prototypes to implement subtasks that are not dividable.

The last step of this approach is to refine all algorithms into a real Python program. We adopted the bottom-up implementation method. Following this method, we started code implementation and unit testing at the lowest levels of the tree-like structure and then moved upwards.

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
1.	In this approach We focus on breaking up the problem into smaller parts.	In bottom up approach, we solve smaller problems and integrate it as whole and complete the solution.
2.	Mainly used by structured programming language such as COBOL, Fortran, C, etc.	Mainly used by object oriented programming language such as C++, C#, Python.
3.	Each part is programmed separately therefore contain redundancy.	Redundancy is minimized by using data encapsulation and data hiding.
4.	In this the communications is less among modules.	In this module must have communication.
5.	It is used in debugging, module documentation, etc.	It is basically used in testing.
6.	In top down approach, decomposition takes place.	In bottom up approach composition takes place.

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
7.	In this top function of system might be hard to identify.	In this sometimes we can not build a program from the piece we have started.
8.	In this implementation details may differ.	This is not natural for people to assemble.
9.	**Pros-** • Easier isolation of interface errors • It benefits in the case error occurs towards the top of the program. • Defects in design get detected early and can be corrected as an early working module of the program is available.	**Pros-** • Easy to create test conditions • Test results are easy to observe • It is suited if defects occur at the bottom of the program.
10.	**Cons-** • Difficulty in observing the output of test case. • Stub writing is quite crucial as it leads to setting of output parameters. • When stubs are located far from the top level module, choosing test cases	**Cons-** • There is no representation of the working model once several modules have been constructed. • There is no existence of the program as an entity without the addition of the last module. • From a partially integrated system, test engineers cannot observe system-

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
	and designing stubs become more challenging.	level functions. It can be possible only with the installation of the top-level test driver.

1.Top-Down approach

Top-Down breaks the large problem into multiple subproblems.

if the subproblem solved already just reuse the answer.

Otherwise, Solve the subproblem and store the result.

Top-Down uses memoization to avoid recomputing the same subproblem again.

Let's solve the same Fibonacci problem using the top-down approach.

Top-Down starts breaking the problem unlike bottom-up.

Like,

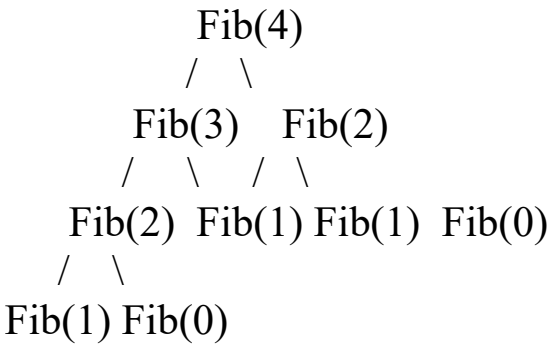
If we want to compute Fibonacci(4), the top-down approach will do the following

Fibonacci(4) -> Go and compute Fibonacci(3) and Fibonacci(2) and return the results.

Fibonacci(3) -> Go and compute Fibonacci(2) and Fibonacci(1) and return the results.

Fibonacci(2) -> Go and compute Fibonacci(1) and Fibonacci(0) and return the results.

Finally, Fibonacci(1) will return 1 and Fibonacci(0) will return 0.



Algorithm

```
Fib(n)
  If n == 0 || n == 1 return n;
  Otherwise, compute subproblem results recursively.
  return Fib(n-1) + Fib(n-2);
```

1. Bottom-Up approach

Start computing result for the subproblem. Using the subproblem result solve another subproblem and finally solve the whole problem.

Example

Let's find the nth member of a Fibonacci series.

Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(2) = 1 (Fibonacci(0) + Fibonacci(1))

Fibonacci(3) = 2 (Fibonacci(1) + Fibonacci(2))

We can solve the problem step by step.

1. Find Oth member
2. Find 1st member
3. Calculate the 2nd member using 0th and 1st member
4. Calculate the 3rd member using 1st and 2nd member
5. By doing this we can easily find the nth member.

Algorithm

1. set Fib[0] = 0
2. set Fib[1] = 1
3. From index 2 to n compute result using the below formula

   Fib[index] = Fib[index - 1] + Fib[index - 2]
4. The final result will be stored in Fib[n].

Asymptotic Analysis: Big-O Notation and More

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.
An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

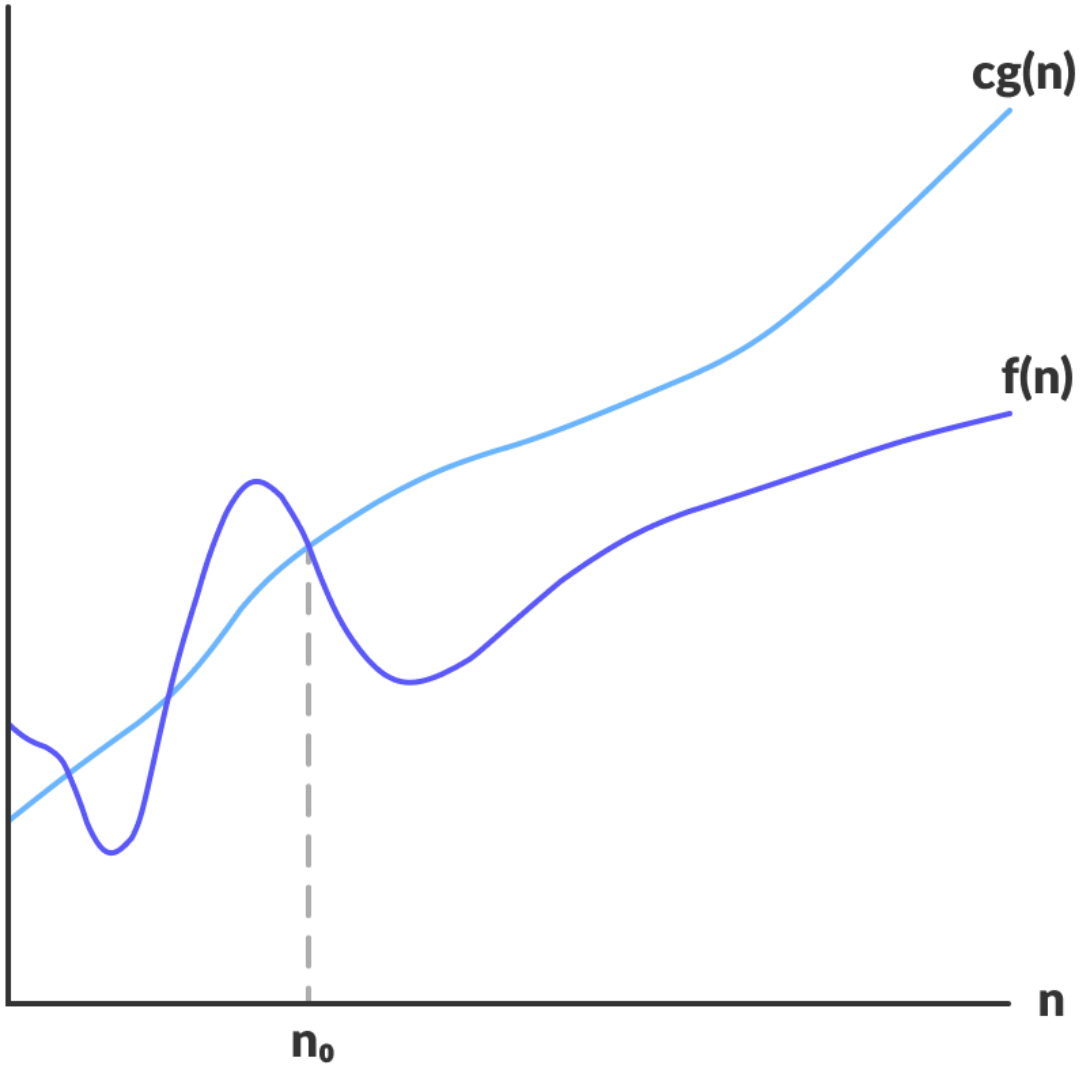
Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case. When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.
There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



f(n) = O(g(n))

Big-O gives the upper bound of a function

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0
such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

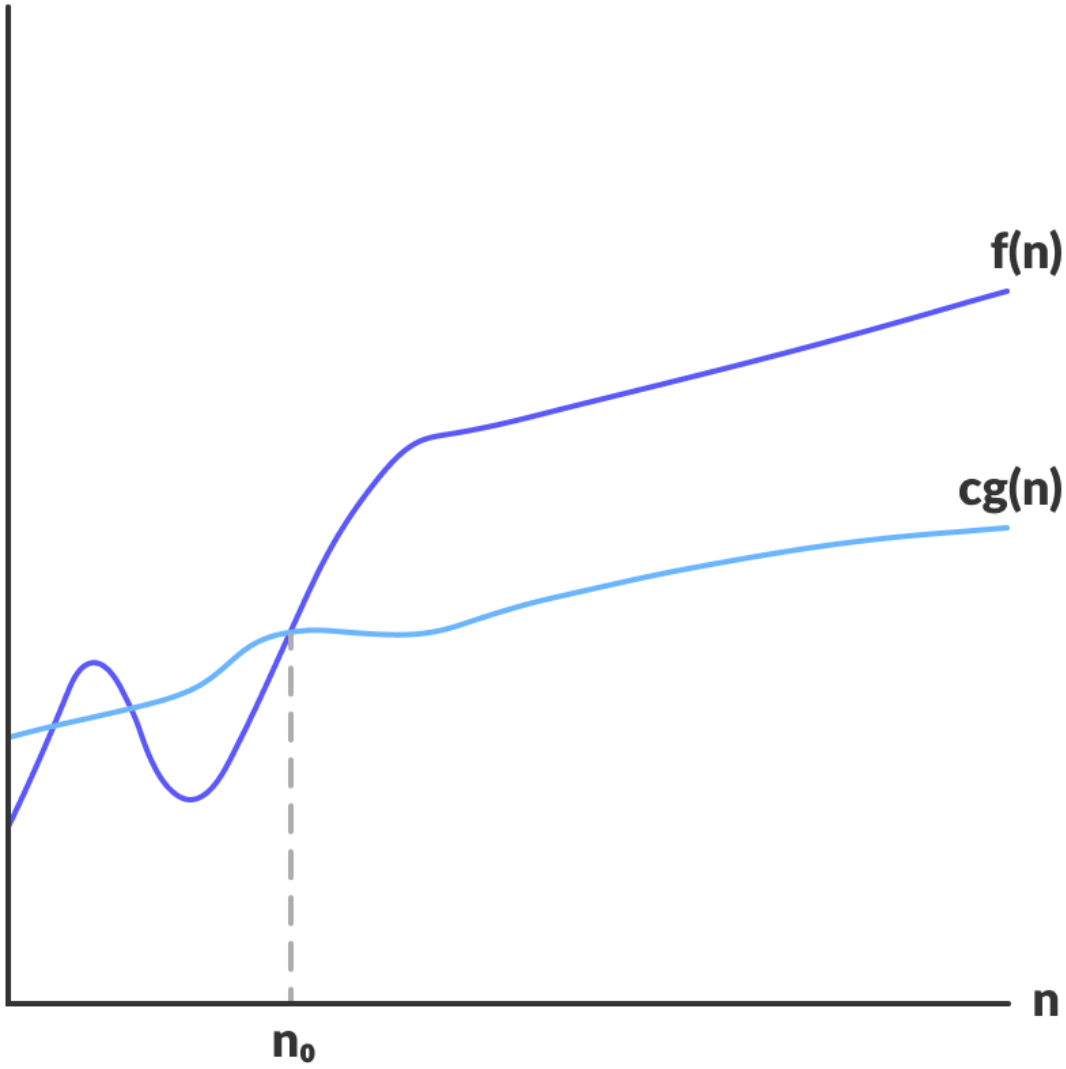
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Omega gives the lower bound of a function

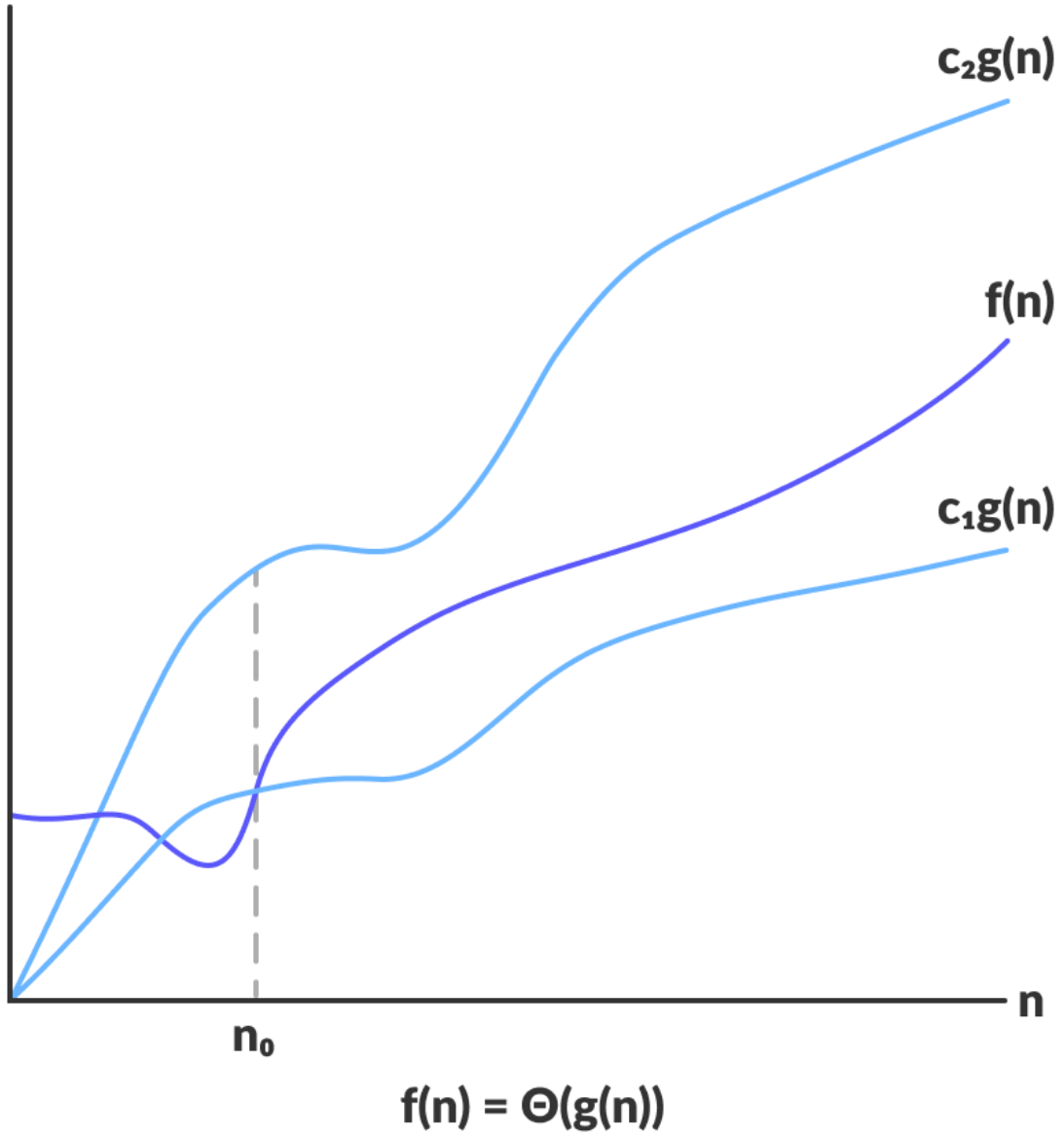
$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c_1, c_2 and n_0

such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }

The above expression can be described as a function f(n) belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. If a function f(n) lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.