CS1010X: Programming Methodology I
Chapter 3: Wishful Thinking

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 3: Wishful Thinking

Learning Objectives

By the end of this chapter, you should be able to:

- **Apply** the Top-Down design strategy (*Wishful Thinking*) to decompose complex problems into manageable sub-problems.
- **Demonstrate** principles of good abstraction by writing code that is readable, reusable, and free of *magic numbers*.
- **Trace** the execution of recursive functions using the *Call Stack* model to understand memory usage and stack frames.
- **Understand** the idea of time and space complexity, in the context of recursive algorithms.

3.1 Wishful Thinking (Top-Down Design)

Wishful Thinking is a powerful strategy in computer science, formally known as **Top-Down Design**.

The core philosophy is to break a large, complex problem into smaller, manageable sub-problems. When you encounter a difficult sub-task, you *pretend* (wish) that a function solving it already exists. This allows you to focus on the high-level logic (how to *use* the solution) before worrying about the implementation details (how to *build* the solution).

Dynamic Programming

We will revisit the idea of a **top-down** and **bottom-up** approach when we eventually cover dynamic programming. In essence, bottom-up solutions start building solutions to the smallest problems before combining them to obtain the solution for the larger problems.
However, this is often harder to do intuitively, since our solution to solving problem often revolves around breaking difficult tasks down into simpler ones.

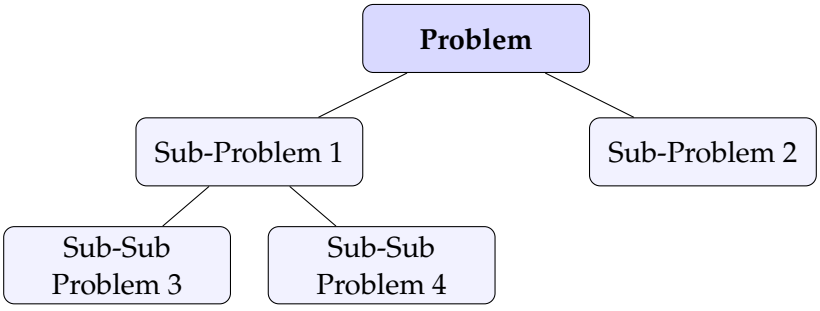


Figure 3.1: Top-Down decomposition of a problem into sub-problems.

3.1.1 Calculating the Hypotenuse of a Triangle

Suppose we have a right-angled triangle, with side lengths a and b. We now want to determine the length of hypotenuse of the triangle, given by:

$$H = \sqrt{a^2 + b^2}$$

We notice that there are a few parts to this problem:

- We need a way to take square roots. Fortunately, there are two ways about this, one is using the `math.sqrt()` function in the math library, or by using `**0.5`. So we make use of the **abstraction** `math.sqrt()` in this case. We assume that the implementers of the square root function have tested it properly to ensure it works as intended.
- Then, we need the ability to sum two squares ($a^2 + b^2$). Then, we realise we do not actually have a way to obtain the square yet.
- So, we also need the ability to obtain a square. This is the `square(x)` function, which we previously have written.

Therefore, by employing wishful thinking, we have successfully written code to compute the hypotenuse of a right-angled triangle, as shown below.

```
1  import math
2
3  def hypotenuse(a, b):
4      """Return the length of the hypotenuse of a right triangle."""
5      # Wishful thinking: I wish I had a 'sum_of_squares' function...
6      return math.sqrt(sum_of_squares(a, b))
7
8  def sum_of_squares(x, y):
9      """Return x^2 + y^2."""
10     # Wishful thinking: I wish I had a 'square' function...
11     return square(x) + square(y)
12
13 def square(x):
14     """Return x^2."""
15     # Primitive level: I can solve this directly.
16     return x * x
```

3.2 Principles of Good Abstraction

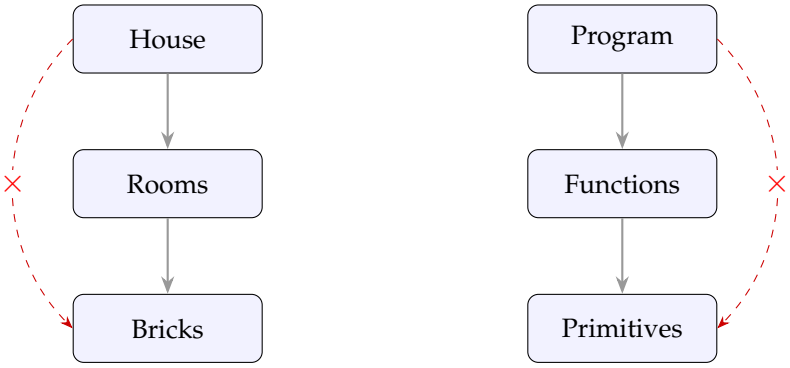
Abstraction is an important concept to make your code more maintainable and reusable.

A *good* abstraction:

- hides unnecessary detail,
- matches the way we naturally think about the problem, and
- can be reused and changed without breaking everything else.

3.2.1 Write Code the Way You Think

When an architect designs a house, they do *not* start by arranging individual bricks. They start with **rooms** and **layouts**, then eventually someone else worries about the bricks.



Similarly, when we design a program, we should not jump straight from *program* to low-level *primitives* (operators, loops, etc.). Instead, we write the program in terms of **functions** that match the way we naturally think about the problem, and only then implement those functions using primitives.

Consider the code below. The distance formula is written directly in terms of coordinates. Line 8 is where the code is extremely hard to understand for others.

```
1 # We want the distance between two positions on a 2D grid.
2 import math
3
4 player_x, player_y = 1, 2
5 enemy_x, enemy_y  = 4, 6
6
7 # First attempt: write the math directly.
8 dist = math.sqrt((enemy_x - player_x)**2 + (enemy_y - player_y)**2)
```

To understand this, the reader has to mentally reconstruct:

- which variables belong together,
- that this is really "distance between two points",
- and which part of the expression means "difference in x" versus "difference in y".

Now we introduce a function that represents the *concept* we care about: `distance`. The highlighted lines are the new abstraction and its use.

```
import math
def distance(x1, y1, x2, y2):
    """Return the distance between (x1, y1) and (x2, y2)."""
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx*dx + dy*dy)

player_x, player_y = 1, 2
enemy_x, enemy_y   = 4, 6

dist = distance(player_x, player_y, enemy_x, enemy_y)
```

Now, the main code captures the intent more clearly: It is clear that we are trying to determine the distance between two points on a grid. This also leads into our next point about *readability*.

Takeaway: Design your program in terms of meaningful sub-tasks (functions) first, and only then implement those sub-tasks with primitives.

3.2.2 Make Programs Easier to Understand

A good abstraction allows a reader to understand *what* the code is doing without forcing them to immediately parse *how* it is doing it. Complex logic is a prime candidate for this type of abstraction.

Below, we are checking if a user's input is acceptable. To understand this, you must mentally parse three different conditions and how they combine.

```
user_input = "Pass123!"
# First attempt: Checking logic inline
if len(user_input) >= 8 and any(c.isupper() for c in user_input) \
   and any(c.isdigit() for c in user_input):
    print("Access Granted")
```

It requires a lot of mental overhead to understand what exactly is going on in this code. Instead, we can abstract the specific rules into a function named `is_valid_password`. The main program flow now reads like English.

```
def is_valid_password(password):
    """Check if password has length, uppercase, and digit."""
    has_len   = len(password) >= 8
    has_upper = any(c.isupper() for c in password)
    has_digit = any(c.isdigit() for c in password)
    return has_len and has_upper and has_digit

user_input = "Pass123!"

if is_valid_password(user_input):
    print("Access Granted")
```

Case Study: Fast Inverse Square Root Algorithm (from Quake III)

An especially pertinent example of this is drawn from the fast inverse square root algorithm from Quake III arena, written in C.

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;
    i  = 0x5f3759df - ( i >> 1 );
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
    this can be removed

    return y;
}
```

There are a few takeaways from this function alone:

- **Function Naming.** Suppose there was no information about it being a fast inverse square root algorithm. Would you have been able to determine what this function did? Probably not. This is why naming your functions appropriately is important.
- **Abstraction.** Next, it is worth noting that this is precisely why abstraction is important. If this code were to appear in your main file directly, without being abstracted into a function, peopple would find it difficult to understand what is going on!
  Giving your function proper names and abstracting appropriately allows people to simply use your functions without having to understand the underlying behaviour!
- **Magic Constants.** As we will see later on, it is important to give variable proper names too. In particular, what does the constant `0x5f3759df` even represent? Others reading your code would probably not understand.

Takeaway: Use clear function names, and abstract functions appropriately, so readers do not need to reverse-engineer your expressions. Abstracting code appropriately helps to improve the readability of your code.

3.2.3 Capture Common Patterns

If you are writing the same code several times, that is a sign to generalise it for more inputs. We should avoid copy-pasting code as much as possible. This is also known as the *DRY (Don't Repeat Yourself)* principle.

In the code below, squaring appears in several places (highlighted).

```
import math

width = 3
radius = 5
mass = 2
c = 3_000_000_000  # speed of light (approx.)

# Same idea (x*x) written in multiple places:
square_area = width * width
circle_area = math.pi * radius * radius
energy = mass * mass * c * c
```

We can introduce a reusable `square` function, then use it everywhere the pattern appears.

```
import math

def square(x):
    """Return x squared."""
    return x * x

width = 3
radius = 5
mass = 2
c = 3_000_000_000

square_area = square(width)
circle_area = math.pi * square(radius)
energy = square(mass) * square(c)
```

As an additional benefit, if the function breaks, we do not need to change it in multiple places. This will be explored in more detail in the section about *ease of debugging*.

Takeaway: Avoid copy-paste logic. Capture it once as a function.

3.2.4 Code Reusability

In software engineering, requirements often change, and therefore code might have to be adapted to slightly different scenarios.

A good function should therefore be able to be used for similar purposes, beyond just its initial purpose.

For example, consider the tax calculation below (line 3). Notice that if we now wanted to calculate the invoice for multiple tax rates, the program is not very extensible. We would need to write a new function for that.

```
1  def print_invoice(amount):
2      """Print the invoice for a single purchase."""
3      tax = amount * 0.08        # 8% VAT is hardcoded here.
4      total = amount + tax
5      print(f"Total due: ${total:.2f}")
```

We can pull the tax calculation out into its own function, then call it from `print_invoice`. If we eventually wanted to compute the value for a different tax rate, it would be possible.

```
1  def calc_tax(amount, tax_rate):
2      """Return the tax for the given amount."""
3      return amount * tax_rate
4
5  def print_invoice(amount):
6      TAX_RATE = 0.08
7      tax = calc_tax(amount, TAX_RATE)
8      total = amount + tax
9      print(f"Subtotal: ${amount:.2f}")
10     print(f"Tax:      ${tax:.2f}")
11     print(f"Total:    ${total:.2f}")
```

Takeaway: Try to make your function as flexible as possible to anticipate future changes.

3.2.5 Hide Irrelevant Details

When you allow parts of your program to directly touch variables, you create a dependency. If you ever change how those variables are organized, every single line of code that touches them will break.

Imagine we are writing a video player. We track the movie's length using two variables: hours and minutes.

```
1  # LIBRARY CODE (How we store data)
2  hours = 2
3  minutes = 15
```

A user (or another part of the program) wants to know the total length in minutes. Since they have direct access, they write the math themselves.

```
# USER CODE (Calculates total time)
# The user relies on 'hours' and 'minutes' existing.
total_time = (hours * 60) + minutes
print(total_time)  # Output: 135
```

Later, we decide that storing two separate variables is annoying. We update the library to store just one variable: `total_minutes`.

```
# LIBRARY CODE (Updated for efficiency)
# We deleted 'hours' and 'minutes'!
total_minutes = 135
```

Since the user calculated it manually using our internal variables, the code instantly crashes.

```
# USER CODE (Run on the new library)
# CRASH: NameError: name 'hours' is not defined.
total_time = (hours * 60) + minutes
```

Instead, if we had forced the user to use a function from the start, we could have changed the internal storage without them ever knowing.

Version A (Hours/Minutes): The use can call `get_total_minutes()`

```
def get_total_minutes():
    return (hours * 60) + minutes
```

Version B (Internal storage changed): We simply update the function logic. The user's code `get_total_minutes()` continues to work perfectly.

```
def get_total_minutes():
    return total_minutes
```

Looking Forward: Encapsulation

This is the idea of **encapsulation** in Object-Oriented Programming. We generally follow the "Information Hiding" and "Tell, Don't Ask" principles, which will be core principles in CS2030/S.

Takeaway: Expose the smallest, simplest interface you can. Hide algorithmic details inside.

3.2.6 Separate Specification from Implementation

The *specification* (what the function promises to do) should stay the same even if you change the implementation (how it does it).

For example, initially, we might implement a sum using a standard loop.

```
def sum_to_n(n):
    if n == 0:
        return 0
    return n + sum_to_n(n - 1)

print(sum_to_n(5)) # The caller gets the result: 15
```

Later, we realize we can use the arithmetic progression formula, $\frac{n(n+1)}{2}$, to make our calculations faster. We update the code, but the *interface* remains exactly the same.

```
def sum_to_n(n):
    # Implementation changed to use the formula.
    return (n * (n + 1)) // 2

# The caller code does not need to change!
print(sum_to_n(5))
```

Takeaway: Because we separated the implementation behind a function, we were able to upgrade the code from a slow loop to a fast formula without breaking the rest of the program.

3.2.7 Ease of Debugging

Good abstractions make it obvious where to look when something breaks.

The speed formula appears in multiple lines. If the calculations are buggy, it is difficult to tell where the bug lies.

```
# Assume we already have a distance(p1, p2) function.

player_speed = distance(player_start, player_end) / time_elapsed
enemy_speed  = distance(enemy_start, enemy_end)  / time_elapsed
```

If the speed calculation is wrong, there are multiple places to fix.

If we introduce a helper `speed_travelled` and call it everywhere we need this logic, we only need to change the function if you have found a bug in your logic.

```
def speed_travelled(start, end, elapsed_time):
    """Return average speed between two positions."""
    return distance(start, end) / elapsed_time

player_speed = speed_travelled(player_start, player_end, time_elapsed)
enemy_speed  = speed_travelled(enemy_start, enemy_end, time_elapsed)
```

Takeaway: When logic is hidden behind a single helper, you can debug and test in one place, and the fix automatically applies everywhere that uses it.

3.2.8 Managing Complexity

In summary, we use three main pillars to keep large systems manageable:

1. **Information Hiding via Abstraction:** Users of a function should not need to know how it works internally. They only need to know the *interface* (inputs/outputs). This reduces cognitive load.
2. **High Cohesion, Low Coupling:**
   - **Cohesion:** A function should do **one thing** well.
   - **Coupling:** Parts of the system should be independent. Changing one module should not break an unrelated module.
3. **Top-Down Design (Wishful Thinking):** We can break complex problems down into simpler subproblems to solve them.

3.2.9 Avoiding Magic Numbers

A "magic number" is a raw constant hardcoded without context.

Suppose you were to read the code below, what do the numbers mean? Without more context, it is difficult to tell.

```
# BAD: What do 3.2 and 0.5 mean?
total_fare = 3.2 + (distance * 0.5)
```

Instead, if we were to express these constants as variables, the code becomes a lot easier to understand for others. This helps to keep our program maintainable, especially when you are working with other developers.

```
# GOOD: Abstraction via constants.
BASE_FARE = 3.2          # dollars
COST_PER_KM = 0.5        # dollars per kilometer

total_fare = BASE_FARE + distance * COST_PER_KM
```

3.3 Introduction to Recursion

Recursion is a very useful application of Wishful Thinking, which you may already have been inadvertently using. It is based on the idea that a problem can be solved by reducing it to a smaller instance of *the same problem*. We already saw examples of this when we were dealing with the Picture Language. In this section, we shall flesh the idea out more.

3.3.1 Structure of a Recursive Function

Every proper recursive function must have two parts:

1. **Base Case(s):** The simplest instance that can be solved directly (e.g., $n = 1$ or $n = 0$). This stops the infinite loop.
2. **Recursive Step:** Calls the function itself with a smaller input (e.g., $n - 1$).

3.3.2 Recursion as Wishful Thinking

When writing the recursive step, we ask:

> *"If I assume the function works correctly for the smaller problem $(n - 1)$, how can I use that result to solve the current problem (n)?"*

For example, suppose I wanted to compute the factorial of a number:

$$n! = n \times (n-1) \times (n-2) \cdots \times 1$$

which can be rewritten as

$$n! = n \times (n-1)!.$$

```
def factorial(n):
    """Return n! for n >= 1."""
    if n == 1:
        # Base case
        return 1
    else:
        # Recursive step: assume factorial(n-1) works correctly.
        return n * factorial(n - 1)
```

Note that just by assuming that `factorial(n-1)` is available and written correctly, we can effectively solve the problem just by breaking it down into smaller subproblems!

3.4 Call Stack vs The Heap (Not Strictly in Syllabus)

While Wishful Thinking is how we *write* recursion, the **Call Stack** is how the computer *executes* it.

When a function is called, the computer allocates a block of memory called a **stack frame** to store its parameters and local variables. In a recursive process, the first call cannot finish until the second call returns, which cannot finish until the third call returns, and so on.

These frames "stack up" in memory. This is why a recursive process requires memory proportional to the number of active calls.

Deep Dive: Python Memory Model (Not in Syllabus)

Under the hood, in most programming language, variables and functions must be handled by the computer's memory. In Python, we have:

1. **The Call Stack**
   The stack manages the *execution context* of the program. It is organized into **stack frames**. In other words, this is how Python works when we call functions.
   - **Creation:** Every time a function is called, a new frame is created (pushed) on top.
   - **Isolation:** Each frame is private. A variable named x in one function is distinct from x in another.
   - **Lifecycle:** The frame exists only while the function is running. When the function returns, the frame is destroyed (popped), and its local variable names are forgotten.
2. **The Heap**
   The heap is a large pool of memory used for dynamic allocating objects.
   - **Content:** *Every* value in Python (integers, lists, functions) lives here as an **object**.
   - **Persistence:** Unlike stack frames, objects in the heap do not disappear automatically when a function ends. They remain as long as there is at least one active reference pointing to them (managed by Garbage Collection).
   - **Unordered:** While the stack is ordered (last in, first out), the heap is an unstructured "bag" of objects accessed via memory addresses.
3. **References**
   Python variables (on the stack) are **pointers** that hold **addresses**. Everything in Python is an object, so everything is stored as a reference.

Looking Forward: Stack and Heaps

The concept of memory models is very central to Computer Science.
The concept of stack, heaps and metaspaces in terms of Java's memory model will be covered in CS2030/S. C++'s memory model will also be further covered in CS3210/CS3211.

3.5 Motivation: Complexity and Big-O Notation

Before using formal notation, let us understand complexity intuitively.

Intuitively, we know that some programs run slower or faster than others. But how exactly can we quantify this, when better hardware causes programs to run faster? The solution to this is the study of "complexity", and in it, the concept of Big-O Notation.

We shall get an intuition for determining the time complexity before we formally explore it in future.

3.5.1 Time: Counting Operations

We assume that basic operations (addition, multiplication, returning) take a constant amount of time, k. If a process requires N steps, the total time is roughly $N \times k$. Since k depends on the hardware, we ignore it. We say the time is **proportional to** N.

$$\text{Time} \propto N \implies O(N).$$

For example, computing the factorial of N should take N operations to compute $N!$. We say that `factorial(n)` takes $O(n)$ time, where n is the input to the function.

Therefore, suppose I were to compute 5! and 10!. We know that the time is proportional to N, so computing 10! should take approximately twice as long as 5!.

3.5.2 Space: Counting Active Frames

Space complexity measures the **maximum** amount of memory used at any one instant.

$$\text{Space} \approx (\text{Max Depth of Stack}) \times (\text{Size of 1 Frame}).$$

Since frame size is approximately constant, space is **proportional to the maximum depth of recursion**.

We shall analyse the idea of recursion depth in the next section.

3.6 Types of Recursion

3.6.1 Linear Recursion (Factorial)

Consider the computation of $n!$ using the recursive program we wrote before:

$$n! = n \times (n-1)!.$$

```python
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

The pending operations build up. We cannot calculate `4 * ...` until we get the result from `factorial(3)`.

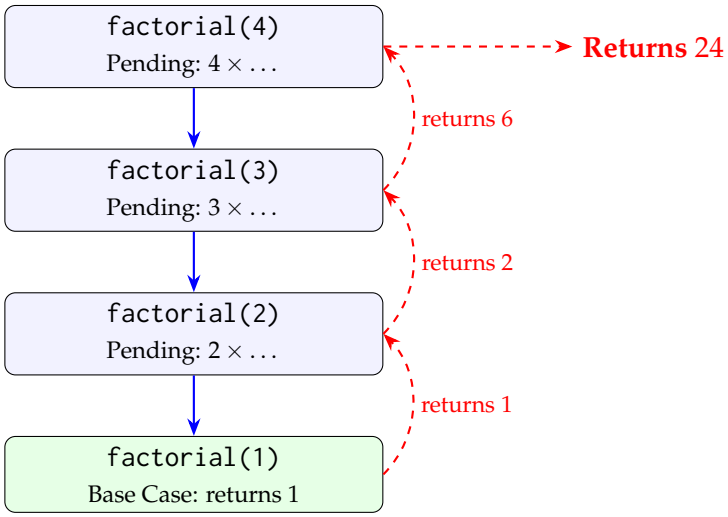


Figure 3.2: Linear recursion: the stack grows linearly with n.

Based on the ideas in section 3.5, we can analyze the cost of running `factorial(n)`.

1. Time Complexity: Counting Operations

Recall that Time $\propto$ Number of Steps. In this recursive function, the "step" is the function call itself. Each call performs a constant amount of work (checking `if n == 1` and performing one multiplication). To calculate `factorial(n)`, the computer must sequentially trigger:

$$\text{factorial}(n) \to \text{factorial}(n-1) \to \cdots \to \text{factorial}(1)$$

- We make exactly n function calls to reach the base case.
- Since each call takes constant time k, Total Time $\approx n \times k$.

$$\therefore \text{Time Complexity} = O(n) \quad \text{(Linear)}$$

2. Space Complexity: Counting Active Frames

Recall that Space $\approx$ Maximum Depth of Stack. The crucial observation in linear recursion is that the frames **accumulate**.

As shown in Figure 1, `factorial(4)` cannot disappear from memory until `factorial(3)` returns. `factorial(3)` cannot disappear until `factorial(2)` returns.

- The system memory usage peaks when we hit the **Base Case**.
- At that exact moment, we have frames for $n, n-1, \ldots, 1$ all open simultaneously.
- Maximum Depth $= n$.

$$\therefore \text{Space Complexity} = O(n) \quad \text{(Linear)}$$

Deep Dive: Tail Recursion Optimization (TCO)

In our standard `factorial` function, the recursive call is *not* the last step.

$$\text{return } n \times \text{factorial}(n-1)$$

The multiplication happens *after* the return. This forces the computer to keep the current stack frame open to "remember" n.
A function is **Tail Recursive** if the recursive call is the absolute final action. To achieve this, we usually pass the running total down as an *accumulator*:

```
def fact_tail(n, acc=1):
    if n == 1:
        return acc
    # The call is the very last step. No pending math.
    return fact_tail(n - 1, acc * n)
```

The Optimization (C++, Scheme, Haskell)
Smart compilers realize that since there is no work left to do in the current frame, they do not need to preserve it. They can perform a **Tail Call Optimization (TCO)**.
Instead of allocating a new frame, they simply update the function arguments and jump back to the start of the function. This effectively converts the recursion into a `while` loop under the hood. As a result, the space complexity drops from $O(n)$ to $O(1)$.

However, **Python does NOT support TCO.** Even if you write your code in the perfect tail-recursive style shown above, the Python interpreter will still create a new stack frame for every single call.
Why? Guido van Rossum (Python's creator) argues that TCO obscures debugging. If an error occurs, TCO destroys the stack trace, making it impossible to see the path the program took to get there.

Therefore, in Python, even if you write your program in a tail-recursive style, recursion limit errors (stack overflow) are still a real risk.

3.6.2 Tree Recursion (Fibonacci)

Consider the Fibonacci sequence, an extremely famous sequence in Mathematics defined by:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1. \end{cases}$$

This process branches. Notice the redundancy: `fib(2)` and `fib(3)` are computed multiple times.

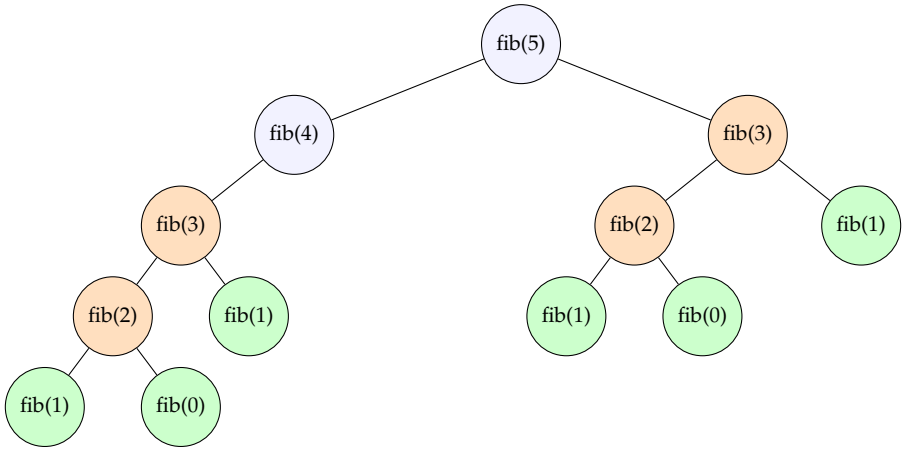


Figure 3.3: Call tree for `fib(5)`. Base cases are in green; repeated subproblems are highlighted in orange (**Note:** Not all nodes are drawn out due to space constraints).

Looking Forward: Repeated Computations

We will eventually learn methods to avoid redundant and repeated computations, using a technique known as Dynamic Programming.

We shall now implement the Fibonacci function. Note that the base cases and the recursive step have already been provided for us:

```
1 def fib(n):
2     # Base Cases: n=0 or n=1
3     if n <= 1:
4         return n
5     # Recursive Step
6     else:
7         return fib(n - 1) + fib(n - 2)
```

Based on the ideas in section 3.5, we can analyze the cost of running `fib(n)`.

1. Time Complexity: Counting Operations

Recall that since every node in the tree above must be completed for us to get our final answer, Time $\propto$ Number of Nodes in the Recursion Tree.

Unlike linear recursion, this function branches. Every call to `fib(n)` spawns **two** more calls: `fib(n-1)` and `fib(n-2)`. This creates a tree structure where the number of operations grows exponentially.

- At depth 0, we have 1 node.
- At depth 1, we have 2 nodes.
- At depth 2, we have 4 nodes.
- At depth k, we have roughly 2^k nodes.

Since the tree height is n, the total work is roughly proportional to 2^n.

$$\therefore \text{Time Complexity} = O(2^n) \quad \text{(Exponential)}$$

Exam Note: In an examination, it is acceptable to approximate this by observing that the function makes two recursive calls at every step. Since the branching factor is 2 and the depth is n, the complexity is $O(2^n)$.

Advanced Derivation: The Golden Ratio ($O(\varphi^n)$)

While $O(2^n)$ is a safe upper bound, the *tight* bound is actually stricter. Let $N(n)$ be the number of operations.

$$N(n) = N(n-1) + N(n-2) + 1$$

This is a linear homogeneous recurrence relation with constant coefficients. The characteristic equation is:

$$r^2 - r - 1 = 0$$

Solving for r using the quadratic formula gives the Golden Ratio:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Thus, the number of operations grows according to powers of the golden ratio:

$$\text{Exact Time Complexity} = O(\varphi^n) \approx O(1.618^n)$$

2. Space Complexity: Counting Active Frames

Recall that Space $\approx$ Maximum Depth of Stack.

This is often a point of confusion. Even though the tree contains $O(2^n)$ total nodes, the computer does **not** hold all nodes in memory simultaneously. Python (and most languages) uses **Depth-First Evaluation**:

1. The program goes as deep as possible down the left branch first (`n` down to `1`).
2. Once a base case returns, that stack frame is **popped** (freed).
3. Only then is the space reused for the right branch.

At any moment, the maximum number of active frames is simply the height of the tree, n.

$$\therefore \text{Space Complexity} = O(n) \quad \text{(Linear)}$$

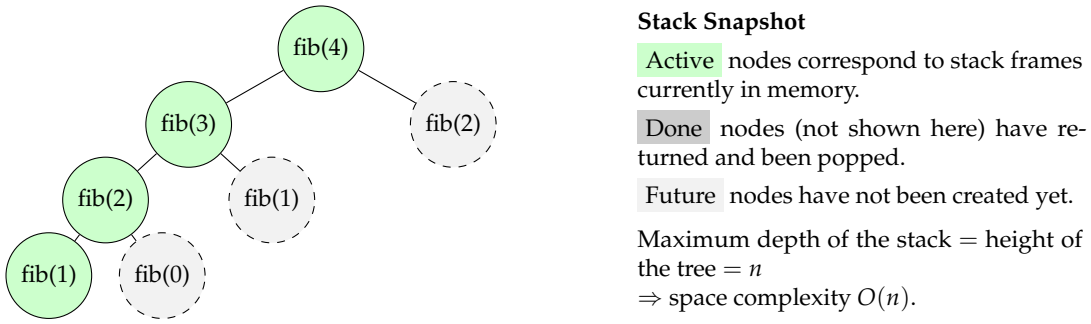


Figure 3.4: Depth-first evaluation of `fib(4)`: one branch at a time.

3.6.3 Mutual Recursion

Mutual recursion is a cyclic dependency: function `A` calls function `B`, and function `B` calls function `A`.

```python
def is_even(n: int) -> bool:
    if n == 0:
        return True
    return is_odd(n - 1)   # Calls B

def is_odd(n: int) -> bool:
    if n == 0:
        return False
    return is_even(n - 1)  # Calls A
```