CS1010X: Programming Methodology I
Chapter 4: Recursion and Iteration

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 4: Recursion and Iteration

Learning Objectives

By the end of this chapter, students will be able to:

- **Identify and decompose** a recursive function into its base case(s) and recursive step, and **draw** a partial recursion tree that correctly shows the branching and maximum depth for a given input.
- **Formulate and solve** simple linear recurrences, to correctly calculate the time complexity and space complexity of recursive functions.
- **Design** recursive solutions that correctly reconstruct the optimal solution by carrying a partial list of choices (*baggage*) through the recursion.
- **Convert** simple recursive algorithms into equivalent iterative versions using `while` or `for` loops, and **compare** their time and space complexities.
- **Write and reason about** Python `while` and `for` loops with `break` and `continue`, **predict** the values of loop variables (including scope leakage and iterator dominance), and **detect** when a loop will result in an infinite iteration.
- **Define precisely** the asymptotic notations $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$ using inequalities with constants and n_0.
- **Simplify** exact cost functions by **dropping** lower-order terms and constants.
- **Distinguish** between best-case, worst-case and average-case complexity, and **apply** these notions to examples such as linear search (best vs. worst case).
- **Implement** an iterative version of a tree-recursive algorithm (e.g. coin change).

4.1 The Recursive Mental Model

In the previous chapter, we wrote recursive functions like `factorial` and `fib` using *wishful thinking*. We will now make that idea precise. When looking at any recursive function, ask:

1. **Base Case(s).**
   For which smallest inputs can we answer the question *directly*, with no further recursion? Examples: `n == 0`, `amount == 0`, or an empty list.

2. **Recursive Step.**
   How do we reduce the current problem to *smaller* subproblems of the same kind?

From a complexity point of view:

- **Time** is controlled by how many times we run the recursive step (how many nodes in the recursion tree).

- **Space** is controlled by how many *frames* (active calls) can be open at once — i.e. the maximum stack depth.

Later, in subsection 4.1.4, we will also see how we can thread a *partial list of choices* through the recursion in order to reconstruct complete solutions.

4.1.1 Greatest Common Divisor (GCD)

The **greatest common divisor** (GCD) of two integers a and b is the largest positive integer that divides both without a remainder. For example:

$$\gcd(18,24) = 6, \qquad \gcd(15,28) = 1.$$

Euclid's algorithm is based on a simple observation:

$$\gcd(a,b) = \gcd(b, a \bmod b).$$

Intuitively: if we divide a by b, the remainder $r = a \bmod b$ captures whatever is "left over" that b could not divide. Any common divisor of a and b must also divide r.

Thus, our solution is:

```python
def gcd(a, b):
    """Return the greatest common divisor of a and b."""
    # Base Case: if the second number is 0, we are done.
    if b == 0:
        return a
    # Recursive Step (Tail Recursion):
    # Reduce (a, b) -> (b, a % b)
    return gcd(b, a % b)
```

We can obtain the solution by observing that:

- **Base Case:** When `b == 0`, the answer is `a`.
- **Recursive Step:** Move to a smaller pair `(b, a % b)`.

1. Time Complexity: Counting Operations

Let $a_0 \geq b_0 > 0$ be the initial inputs and suppose the algorithm performs k recursive calls:

$$(a_0, b_0) \to (a_1, b_1) \to (a_2, b_2) \to \cdots \to (a_k, b_k),$$

where each step is

$$(a_{i+1}, b_{i+1}) = (b_i,\ a_i \bmod b_i), \qquad i = 0, \ldots, k-1.$$

We always have $0 \leq b_{i+1} < b_i$, so the second argument strictly decreases. We can estimate by trying out small numbers that the number of operations is approximately $\log(\min(a,b))$.

This is actually sufficient for examinations, but we may want to rigorously prove it using mathematics. We can indeed do so. A classical result (Lamé's theorem) makes this precise:

Lamé's Theorem

If we look only at the second argument $b_0, b_1, b_2, \ldots, b_k$, then in the worst case this sequence is dominated by successive Fibonacci numbers: $b_i \geq F_{i+1}, \quad$ for all i.
Since $F_m \geq \varphi^{m-2}$ for $m \geq 2$ (where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio), we obtain

$$b_k \;\geq\; F_{k+1} \;\geq\; \varphi^{k-1}.$$

But $b_k \leq b_0$, so

$$\varphi^{k-1} \leq b_0 \;\Rightarrow\; k-1 \leq \log_\varphi b_0 \;\Rightarrow\; k = O(\log b_0).$$

Thus, the number of steps is proportional to the logarithm of the smaller argument.

Each call to `gcd` does only $O(1)$ work besides the recursive call (one `if` check and one `%`).

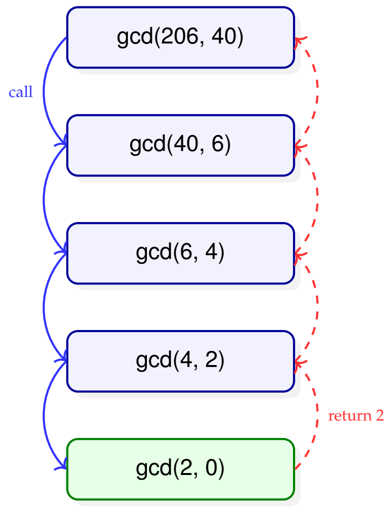
$$\therefore \text{Time} \propto \text{number of calls} = O(\log(\min(a,b))),$$

So, the Time Complexity of `gcd(a, b)` $= O(\log(\min(a,b)))$.

2. Space Complexity: Counting Active Frames

Each active call to `gcd` contributes one frame to the call stack. A frame stores the parameters `a` and `b` and a return address. At the deepest point in the recursion we have exactly one frame for each pair (a_i, b_i) along the sequence above. Since the number of steps is $O(\log n)$, the maximum stack depth is also $O(\log(\min(a,b)))$:

$$\therefore \text{Space Complexity of } \texttt{gcd(a, b)} = O(\log(\min(a,b))).$$



Stack Snapshot for `gcd(206, 40)`

- Each frame stores the current pair `(a, b)`.
- Blue arrows show recursive calls going "down".
- Red dashed arrows show results returning "up".

The chain has length $O(\log n)$, matching our complexity analysis.

Figure 4.1: Execution trace of Euclid's Algorithm.

4.1.2 Towers of Hanoi

We have three pegs (A, B, C) and a tower of n disks on peg A, stacked from largest (bottom) to smallest (top).

Goal: Move the entire tower from peg A to peg C, subject to:

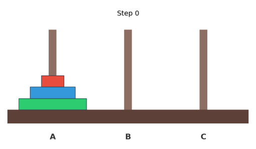
- we may move only one disk at a time; and
- we may never place a larger disk on top of a smaller one.

We note that, by using wishful thinking, assuming we could solve the problem for $n-1$ disks, we can easily solve the problem:

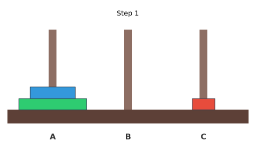
1. Move the top $n-1$ disks from A (source) to B (auxiliary).
2. Move the largest disk from A to C (target).
3. Move the $n-1$ disks from B to C.

Each step is again a smaller instance of "move a tower of height k from one peg to another". Thus, we can solve the problem as follows:

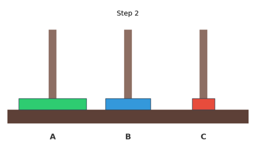
```python
def move_tower(n, source, target, aux):
    """Print the moves to transfer n disks from source to target."""
    # Base Case: only one disk -> move it directly.
    if n == 1:
        print(f"Move disk from {source} to {target}")
        return

    # Step 1: move n-1 disks to auxiliary peg.
    move_tower(n - 1, source, aux, target)
    # Step 2: move the largest disk.
    print(f"Move disk from {source} to {target}")
    # Step 3: move the n-1 disks onto the largest disk.
    move_tower(n - 1, aux, target, source)
```



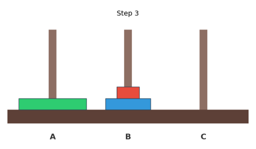
0. Start



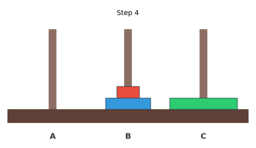
1. $A \rightarrow C$



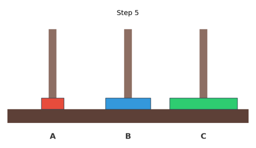
2. $A \rightarrow B$



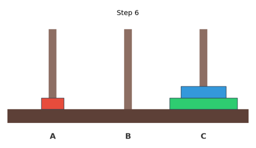
3. $C \rightarrow B$



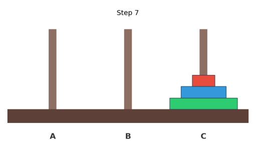
4. $A \rightarrow C$ (big disk)



5. $B \rightarrow A$



6. $B \rightarrow C$



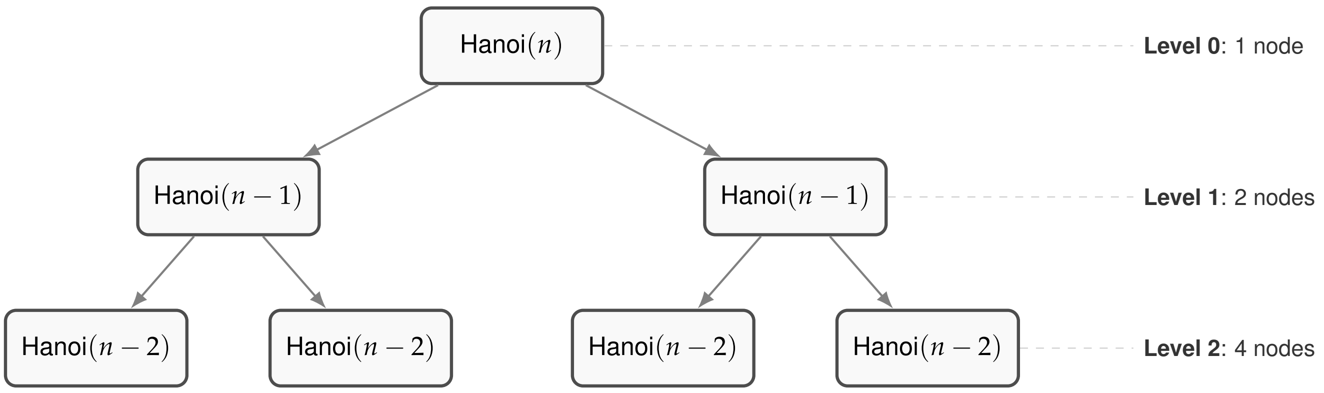
7. $A \rightarrow C$

Figure 4.2: Full execution trace for moving a tower of height 3.

We can obtain the solution by observing that:

- **Base Case:** $n = 1$ is a single legal move.
- **Recursive Step:** Two calls on the smaller problem size $n - 1$.

1. Time Complexity: Counting Operations



Let $T(n)$ be the number of moves produced by `move_tower(n, ...)`.

From the structure of the function:

- first we call `move_tower(n-1, ...)`,
- then we make 1 move,
- then we call `move_tower(n-1, ...)` again.

This yields the recurrence:

$$T(1) = 1, \qquad T(n) = 2T(n-1) + 1 \quad (n \geq 2).$$

Claim. $T(n) = 2^n - 1$ for all $n \geq 1$.

Proof. We prove this claim by induction.

Base case ($n = 1$): $T(1) = 1 = 2^1 - 1$, so $T(1)$ is true.

Inductive step: Let $m \geq 1$. Assume $T(m-1) = 2^{m-1} - 1$. Then, from our observations above,

$$\begin{aligned} T(m) &= 2T(m-1) + 1 \\ &= 2(2^{m-1} - 1) + 1 \\ &= 2^m - 2 + 1 \\ &= 2^m - 1. \end{aligned}$$

Thus, by induction, $T(n) = 2^n - 1$ for all n. Since each step takes $O(1)$ time, then

$$\therefore \text{Time Complexity of } \texttt{move\_tower(n)} = O(2^n).$$

2. Space Complexity: Counting Active Frames

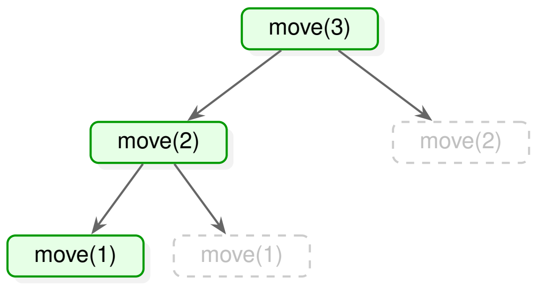
At any moment during execution, the call stack contains one frame for each unfinished call to `move_tower`.

Because the algorithm always recurses on `n-1` before doing any further work, the recursion goes down a single path:

$$n,\ n-1,\ n-2,\ \ldots,\ 1.$$

Only after the base case returns does Python unwind the stack and start exploring the "right" branches. The maximum number of active frames is therefore n:

$$\therefore \text{Space Complexity of Towers of Hanoi} = O(n).$$



Stack Snapshot

At the moment when `move_tower(1, A, C, B)` runs:

- All nodes on the left path are **active** (green).
- Nodes on the right are **future** (gray): they do not consume memory yet.

The depth of this active path is at most n, so stack usage is $O(n)$.

Figure 4.3: Depth-first execution of Towers of Hanoi.

Extra: Master Theorem for Divide-and-Conquer

While Tower of Hanoi reduces problem size by 1 at each step, many algorithms (like Merge Sort) reduce size by a factor. For these, we can use **Master Theorem** to determine the time complexity. Given a recurrence of the form

$$T(n) = a\,T(n/b) + f(n),$$

we compare the work at the root, $f(n)$, against the work at the leaves, $n^{\log_b a}$.

1. **Leaves Dominate:** If $n^{\log_b a} > f(n)$, the complexity is determined by the number of leaves.
$$T(n) = \Theta(n^{\log_b a})$$
2. **Balanced:** If $n^{\log_b a} \approx f(n)$, the cost is evenly distributed. Multiply by the height.
$$T(n) = \Theta(n^{\log_b a} \log n)$$
3. **Root Dominates:** If $f(n) > n^{\log_b a}$, the complexity is determined by the work at the top.
$$T(n) = \Theta(f(n))$$

4.1.3 Counting Change

Suppose we want to know how many different ways there are to make change for a given amount using a set of coin denominations.

Example: How many ways to make 11 cents using coins $\{10, 1\}$?

We use a very common recursion pattern: **partition** the possibilities into two disjoint groups. For a state $(\text{amount}, \text{coins})$:

1. **Use the first coin at least once.**
   Reduce the amount: $(\text{amount} - \text{coins}[0], \text{coins})$.
2. **Skip the first coin entirely.**
   Reduce the list of coins: $(\text{amount}, \text{coins}[1:])$.

Every way of making change either uses the first coin at least once or not at all, so we do not over or under-count. Thus:

$$\text{ways}(\text{amount}, \text{coins}) = \text{ways}(\text{amount} - \text{first}, \text{coins}) + \text{ways}(\text{amount}, \text{rest}).$$

```python
def count_change(amount, coins):
    """Return the number of ways to make 'amount' using 'coins'."""
    # Base Case 1: exact change.
    if amount == 0:
        return 1
    # Base Case 2: overshoot or run out of coins -> no way.
    if amount < 0 or len(coins) == 0:
        return 0

    # Recursive Step:
    # 1) Use the first coin; 2) Skip the first coin.
    return count_change(amount - coins[0], coins) + \
           count_change(amount, coins[1:])
```

We can obtain the solution by observing that:

- **Base cases:**
  - `amount == 0`: we found one valid way.
  - `amount < 0` or no coins left: no valid way.
- **Recursive Step:** Two branches — use the first coin or skip it.

1. Time Complexity: Number of Operations

Every non-base call makes **two** recursive calls:

- one where we *use* the first coin at least once, and
- one where we *skip* the first coin entirely.

This naturally forms a *recursion tree*. Each node is a call `count_change(amount, coins)`, and each internal node has up to two children:

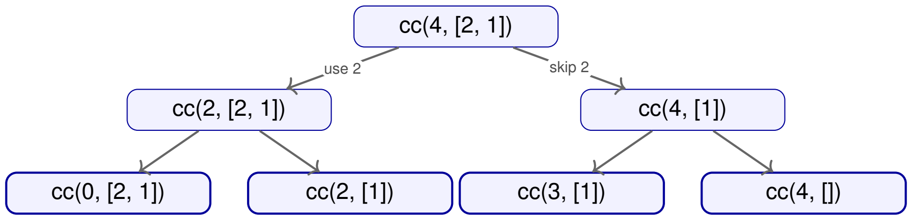


Figure 4.4: Partial recursion tree for `count_change(4, [2, 1])`. Each internal node splits into a "use first coin" branch and a "skip first coin" branch.

Every non-base call to `count_change` makes **two** recursive calls: one for "use the first coin" and one for "skip the first coin". This gives a *binary recursion tree*.

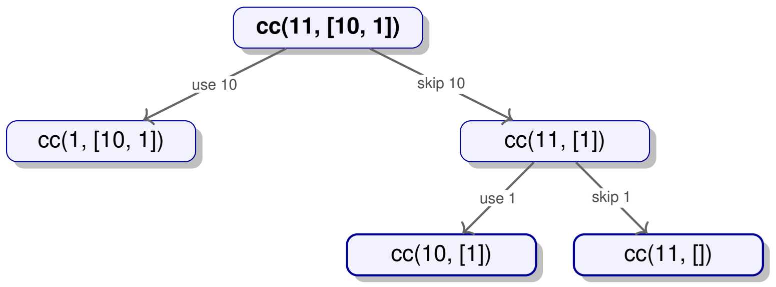


Figure 4.5: Part of the recursion tree for `count_change(11, [10, 1])`.

To express the cost in terms of the initial *amount* and number of *coins*, let:

- a_0 be the initial amount,
- k_0 be the initial number of coin denominations,
- $c_{\min} > 0$ be the smallest coin value.

Each recursive call either:

- reduces the amount by at least $c_{\min}$ ("use first coin"), or
- reduces the number of available coins by 1 ("skip first coin").

Therefore, the height h of the recursion tree is at most

$$h \le \underbrace{\frac{a_0}{c_{\min}}}_{\text{keep using smallest coin}} + \underbrace{k_0}_{\text{keep skipping coins}} .$$

Since this is a binary tree (branching factor at most 2), the total number of nodes, and hence the total number of recursive calls, is bounded by

$$\text{number of nodes} \;\le\; 2^{h+1} - 1 \;=\; O\left(2^{\,h}\right) \;=\; O\left(2^{\frac{a_0}{c_{\min}}+k_0}\right).$$

We can therefore write the number of operations as

$$T(a_0, k_0) = O\left(2^{\frac{a_0}{c_{\min}}+k_0}\right)$$

Since each node only does a constant amount of work, we can say that the time complexity of coin change is $O\left(2^{\frac{a_0}{c_{\min}}+l_0}\right)$ as written.

Thus, this `count_change` algorithm has **exponential time complexity** in the problem size (amount and number of coin denominations).

However, in exams, it is sufficient to notice that the maximum number of nodes of this recursion tree is $2^{a_0+k_0}$, so it is acceptable to write the time complexity is $O(2^{a_0+k_0})$.

Looking Forward: Dynamic Programming

The recursion tree has many repeated subproblems, e.g. `count_change(11, [1])` can be reached via several paths. Later, we will see how **memoization** (caching results) or **bottom-up dynamic programming** can reuse these answers and collapse the exponential recursion tree down to a much smaller polynomial-time algorithm.

2. Space Complexity: Counting Active Frames

In the *counting-only* version above, each call stores just:

- the current `amount`,
- the remaining list `coins`,
- and the return address (where to continue after children return).

Python evaluates recursion in a **depth-first** manner. At any moment, only a single root-to-leaf path of the recursion tree is active; all other branches are either:

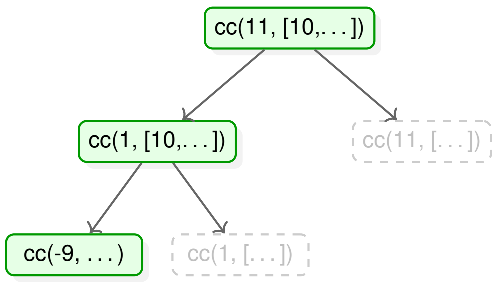
- **future** calls that have not been created yet, or
- **completed** calls whose frames have already been popped.

The worst-case depth of this active path is the same d as above: we can subtract the smallest coin roughly $a/c_{\min}$ times, and we can skip up to k coin types:

$$\text{Max stack depth} \;\le\; \frac{a}{c_{\min}} + k \;=\; O\!\left(\frac{a}{c_{\min}} + k\right).$$

For a fixed set of denominations this is **linear** in the amount:

$$\therefore \text{Space Complexity of naive } \texttt{count\_change} \;=\; O(n), \quad \text{where } n \approx a/c_{\min}.$$



Stack Snapshot

The deepest path occurs when we repeatedly "use" the smallest coin, reducing the amount very slowly.

$$\text{Max Depth} \approx \frac{\text{amount}}{\text{smallest coin}}.$$

Only the **active** frames on this path consume memory; **future** calls do not yet exist.

Figure 4.6: Space complexity of Counting Change: only the longest path in the recursion tree is ever on the stack at once.

Practice: Another Way to Write Coin Change

We can write coin change a little bit differently, especially after we learn for loops in the next section.

```python
def count_change(amount, coins):
    # Base Case 1: exact change -> 0 coins needed.
    if amount == 0:
        return 0
    # Base Case 2: overshoot -> impossible.
    if amount < 0:
        return -1

    best = -1  # "no solution found yet"
    for coin in coins:
        # Recursive step: try using 'coin' once and solve the
            smaller subproblem.
        sub = count_change(amount - coin, coins)
        # If this choice leads to a dead end, skip it.
        if sub == -1:
            continue
        candidate = sub + 1  # one coin (this one) + coins used
            in the subproblem
        # Keep track of the best (minimum) number of coins found
            so far.
        if best == -1 or candidate < best:
            best = candidate
    return best
```

1. Why is this solution also correct?
2. What is the time complexity of this variant of the solution?
3. What is the space complexity of this variant of the solution?

Solution.

1. This solution is correct by analysing the recursion:
   - **Base cases**
     - `amount == 0` $\Rightarrow$ we have made exact change, so we need `0` more coins.
     - `amount < 0` $\Rightarrow$ we overshot, so this path is impossible (`-1`).
   - **Recursive step** Loop over all coin denominations. For each `coin`, we:
     (a) reduce the problem to `count_change(amount - coin, coins)`, and
     (b) if that subproblem is solvable, add 1 to account for the current coin.
2. We define the following variables:
   - a_0 is the initial `amount`,
   - $k_0 = |\texttt{coins}_0|$ is the number of distinct coin denominations,
   - and we assume (for simplicity) that the *smallest* coin has value 1.

   At any non-base call with `amount > 0`, the function tries *every* coin in `coins`.

So in the recursion tree:

- each internal node has at most k_0 children (one for each possible coin choice);
- every move reduces the `amount` by at least 1 (since the smallest coin is 1).

Therefore:

- the **height** of the tree is at most a_0 (we can subtract 1 at most a_0 times before reaching 0 or going negative);
- the **branching factor** is at most k_0 at each level.

In the worst case, the recursion tree is no larger than a full k_0-ary tree of height a_0:

$$\text{Number of nodes} \;\le\; 1 + k_0 + k_0^2 + \cdots + k_0^{a_0} \;=\; \frac{k_0^{a_0+1} - 1}{k_0 - 1} \;=\; O\big(k_0^{a_0}\big),$$

for any fixed $k_0 > 1$.

Each node corresponds to one call to `count_change` and performs only $O(1)$ extra work besides its recursive calls (a loop over k_0 coins with simple arithmetic and comparisons), so the total running time is proportional to the number of nodes in the recursion tree.

Thus, the time complexity of `count_change(a_0, coins)` = $O\big(k_0^{a_0}\big)$, *exponential* in the initial amount a_0, with base equal to the number of coin types k_0.

Remark. If the smallest coin has value $c_{\min} > 0$, then the height is at most $a_0/c_{\min}$ and the bound becomes $O\big(k_0^{a_0/c_{\min}}\big)$; the simplified $O(k_0^{a_0})$ assumes $c_{\min} = 1$. As with previously, the simplified analysis is acceptable.

3. Each recursive call creates one stack frame. What does each frame store?

   - the current `amount` (an integer),
   - a reference to the list `coins` (the same list is reused),
   - a few local variables: `min_cnt`, `coin`, `cnt`.

   All of this is $O(1)$ space *per* call. The key question is: *how many calls can be active at once?*

   As usual, Python evaluates the recursion in a depth-first manner:

   - From a node with `amount = a`, each recursive call goes to `amount = a - coin`.
   - For the deepest path, we can imagine always using the *smallest* coin value, $c_{\min}$.

   So along the worst-case path the amounts look like: $a_0,\ a_0 - c_{\min},\ a_0 - 2c_{\min},\ \ldots$

   So, the maximum stack depth $\le \frac{a_0}{c_{\min}} + 1$.

   Since each frame uses constant space, the total stack space is proportional to this depth:

   $$\therefore \text{Space Complexity of naive } \texttt{count\_change(a\_0, coins)} = O\left(\frac{a_0}{c_{\min}}\right).$$

   Similarly, in examinations, it is acceptable to write space complexity = $O(\text{Amount})$.

4.1.4 Baggage: Reconstructing All Ways

So far, `count_change` only tells us *how many* ways there are. Sometimes we want to reconstruct the actual ways themselves.

To do this, each recursive call needs to remember a *partial list* of coins chosen so far. We will refer to this partial list as the **baggage** of that frame: it is carried down the recursion, extended on one branch, and reused on the other.

```python
def change_ways(amount, coins, chosen=None):
    """Return a list of all ways (each way is a list of coins)."""
    if chosen is None:
        chosen = []

    # Base Case 1: exact change -> one complete way.
    if amount == 0:
        return [chosen]

    # Base Case 2: overshoot or run out of coins -> no way.
    if amount < 0 or len(coins) == 0:
        return []

    first = coins[0]
    rest  = coins[1:]

    # Recursive Step:
    # 1) Use the first coin: extend the baggage with 'first'.
    ways_with_first = change_ways(amount - first, coins,
                                  chosen + [first])

    # 2) Skip the first coin: keep baggage as-is.
    ways_without_first = change_ways(amount, rest, chosen)

    return ways_with_first + ways_without_first
```

Key points about baggage here:

- The parameter `chosen` is the baggage — the partial list of coins chosen along the current path.
- On the "use" branch we pass `chosen + [first]`, so the baggage grows.
- On the "skip" branch we pass `chosen` unchanged.
- When `amount == 0`, the baggage is a *complete* solution, so we wrap it in a list and return `[chosen]`.

The recursion tree and asymptotic time/space complexities are still exponential and linear in depth, respectively. The difference is that each frame now carries a bigger object (a partial list), and the final result is a list containing *all* those lists.

4.2 Introduction to Iteration

In computer science, we have two primary mechanisms for repeating computations: Recursion and Iteration. While they can often solve the same problems, their underlying mechanics differ significantly regarding memory and state management.

We shall intuitively contrast recursion and iteration, before looking at some examples of it, and we will have a more detailed summary at the end.

4.2.1 Recursive Process

A recursive process relies on self-reference. When a function calls itself, the computation cannot complete immediately.

- **Deferred Operations:** The process occurs with pending operations that must be performed *after* the inner calls return.
- **System Management:** The operating system handles the state for us automatically using the **call stack**.

4.2.2 Iterative Process

An iterative process uses looping constructs (like `while` or `for`) to repeat a block of code.

- **No Deferred Operations:** The state is updated explicitly by the programmer in variables. At any point, the program's state is completely contained in the current values of its variables.

Importantly, recursion follows the idea of "top-down" / wishful thinking, while in iteration, we generally follow a "bottom-up" thinking process, where we build our solution from the smallest subproblems.

4.3 Looping Constructs

4.3.1 The While Loop

The `while` loop is the fundamental construct for **indefinite iteration**. It repeatedly executes a block of code as long as a specified boolean condition remains true.

The condition (predicate) is evaluated *before* each iteration. The loop terminates only when the condition evaluates to `False`.

Syntax:

```
while <expression>:
    <body>
```

Warning (Infinite Loops)

The variables involved in the `<expression>` must be updated within the `<body>`. If the logic never causes the expression to become `False`, the program will enter an infinite loop and hang indefinitely.

4.3.2 The For Loop

The `for` loop is used for **definite iteration**. Unlike C or Java for-loops (which are based on counters), Python's for-loop iterates over a sequence (an iterator), assigning the next value to the loop variable at the start of each pass.

Syntax:

```
for <variable> in range([start,] stop[, step]):
    <body>
```

The `range` function generates an immutable sequence of numbers:

Call	Sequence Generated	Note
`range(5)`	0, 1, 2, 3, 4	Default start is 0.
`range(1, 5)`	1, 2, 3, 4	Stops *before* the stop value.
`range(5, 1, -2)`	5, 3	Negative step (decrements).

This should be reminiscent of the `slice()` function, or the string slicing operator.

Loop Variable Behavior: Scoping and Overwriting

Python handles loop variables differently than many other languages. It is crucial to understand two specific behaviors regarding the variable (e.g., i).

1. Scope Leakage

In Python, `for` loops do **not** create their own local scope. The loop variable remains accessible and retains its final value after the loop finishes.

```python
i = 100          # Existing variable
for i in range(3):
    pass         # i becomes 0, then 1, then 2
print(i)         # Output: 2
# The original value (100) is lost/overwritten.
```

2. Iterator Dominance (Internal Modification)

If you modify the loop variable *inside* the loop body, that change does **not** affect the next iteration. At the start of the next loop pass, the `range` object (or iterator) forcefully assigns the next value in the sequence to the variable, overwriting your manual change.

```python
for i in range(3):       # Sequence: 0, 1, 2
    print(f"Start: {i}")
    i = 500              # Attempt to modify i
    print(f"End:   {i}")
# Output Trace:
# Start: 0 -> End: 500
# Start: 1 -> End: 500  (i is reset to 1, ignoring the 500)
# Start: 2 -> End: 500  (i is reset to 2)
```

C-Style Loops

This contrasts with C-style loops where modifying the counter inside the loop affects the iteration logic.

```c
// In C, the loop is driven by the condition check on 'i'.
// Modifying 'i' inside the body DIRECTLY changes the flow.
for (int i = 0; i < 3; i++) {
    printf("Start: %d\n", i);
    i = 500;  // We manually change the counter to 500
    printf("End:   %d\n", i);
}

/* OUTPUT:
Start: 0
End:   500
(The loop terminates immediately because 500 increments to 501,
and 501 < 3 is False. It does NOT reset to 1 like Python.)
*/
```

4.3.3 Control Flow: Break and Continue

We can alter the linear flow of loops using these keywords:

- **break:** Immediately terminates the loop. Execution jumps to the first line *after* the loop body.
- **continue:** Skips the remainder of the *current iteration* and jumps back to the top (condition check for `while`, next item for `for`).

Using `break`

```python
for i in range(5):
    if i == 3:
        break  # Stops entire loop
    print(i)

# Output:
# 0
# 1
# 2
# (Loop terminates at 3)
```

Using `continue`

```python
for i in range(5):
    if i == 3:
        continue # Skips just this one
    print(i)

# Output:
# 0
# 1
# 2
# (3 is skipped)
# 4
```

Infinite Loops

When using `continue` inside a `while` loop, you must ensure the iterator update happens *before* the continue statement. Failing to do so creates an infinite loop because the state never changes.

Incorrect (Infinite Loop)

```python
i = 0
while i < 5:
    if i == 3:
        # DANGER: i is never incremented!
        continue
    print(i)
    i += 1
```

Correct Approach

```python
i = 0
while i < 5:
    i += 1       # Update state first
    if i == 3:
        continue # Safe to skip
    print(i)
```

4.4 Examples of Iteration

4.4.1 Factorial

Let us consider this using the factorial function in contrast to its recursive solution:

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

Unlike recursion, which breaks the problem down ($n \to n-1$), iteration builds the solution up by maintaining a running product. We can approach this in two directions:

1. **Counting Down:** Start with n, multiply by $n-1$, down to 1.
2. **Counting Up:** Start with 1, multiply by 2, up to n.

```python
def factorial(n):
    # assume n >= 1
    ans = 1
    m = 2
    while m <= n:
        ans = ans * m   # Update the running product
        m = m + 1       # Increment the counter explicitly
    return ans
```

To understand iteration, we trace the values of variables at each step. Notice there is no "stack" growing; only the values in the slots `m` and `ans` change.

n	m	ans
5	2	1
5	3	2
5	4	6
5	5	24
5	6	120

Table 4.1: Trace for `factorial(5)`

We can also rewrite the factorial function more concisely using a `for` loop. This removes the need to manually manage the counter variable.

```python
def factorial(n):
    # assume n >= 1
    ans = 1
    # Count down from n to 2 (stop value 1 is exclusive)
    for i in range(n, 1, -1):
        ans = ans * i
    return ans
```

Trace for `factorial(5)`: The loop variable `i` takes values 5, 4, 3, 2. The result `ans` updates: $1 \to 5 \to 20 \to 60 \to 120$.

1. Time Complexity: Number of Operations

For the iterative factorial, the loop runs n times, performing a constant amount of multiplication work each time.

$$\text{Time Complexity} = O(n) \quad \text{(Linearly proportional to } n\text{)}$$

2. Space Complexity: Active Stack Frames

This is the major advantage of iteration over recursion.

- **Recursive:** $O(n)$ space. It builds n stack frames (deferred operations) waiting for the base case.
- **Iterative:** $O(1)$ space (Constant).

Reasoning: There are **no deferred operations**. The memory usage does not grow with n. We only require storage for a fixed set of variables (`ans`, `i`, `n`) regardless of whether $n = 5$ or $n = 5000$.

4.4.2 Fibonacci

The standard recursive Fibonacci algorithm is inefficient ($O(2^n)$) because it re-calculates the same values (e.g. $fib(3)$) multiple times. The iterative solution calculates strictly forward and stores only the necessary history.

To compute $f(n)$, we only need $f(n-1)$ and $f(n-2)$. We do not need the entire history.

```python
def fib(n):
    if n < 2: return n

    f_minus1 = 1  # Represents f(i-1)
    f_minus2 = 0  # Represents f(i-2)

    # Iterate from 2 to n
    for i in range(2, n + 1):
        f_n = f_minus1 + f_minus2  # Compute current

        # Shift the window for the next iteration
        f_minus2 = f_minus1
        f_minus1 = f_n

    return f_n
```

- **Time Complexity:** $O(n)$**.** The loop runs $n - 1$ times. $O(1)$ work is done per iteration.
- **Space Complexity:** $O(1)$**.**
  - Unlike recursion, there are **no deferred operations**.
  - We do not need a stack to remember history.
  - We only strictly need storage for three variables (`f_minus1`, `f_minus2`, `f_n`) regardless of whether $n = 10$ or $n = 10,000$.

4.4.3 Greatest Common Divisor (GCD)

The Euclidean algorithm is naturally iterative. The recursive definition $\gcd(a, b) = \gcd(b, a\%b)$ translates directly into a `while` loop where we update the pair (a, b) until the remainder is zero.

```python
def gcd(a, b):
    # assume a and b are non-negative
    while b != 0:
        a, b = b, a % b  # Simultaneous update
    return a
```

The Euclidean algorithm is a classic example of **indefinite iteration**. We do not know in advance how many steps it will take for the remainder to reach zero.

- **The While Loop:** Naturally expresses "repeat *until* condition."
- **The For Loop:** Designed for "repeat N times." To use it here, we must provide a "worst-case" upper bound (e.g. $b + 1$) and manually `break` early.

Natural (While Loop)

```python
def gcd(a, b):
    # Repeat until done
    while b != 0:
        a, b = b, a % b
    return a
```

Forced (For Loop)

```python
def gcd(a, b):
    # Guess max iterations (b
        +1)
    for i in range(b + 1):
        a, b = b, a % b
        if b == 0:
            break # Manual
                exit
    return a
```

Using a `for` loop here is semantically misleading. It implies we intend to iterate exactly $b + 1$ times, whereas in reality, the loop will almost always terminate much earlier via `break`.

4.5 Iterating Tree Recursion: The Explicit Stack

Simple loops work well for linear problems (Factorial, GCD, Fibonacci). However, problems like **Tower of Hanoi** or **Coin Change** naturally form a tree structure.

- **Branching:** These algorithms branch into multiple subproblems at every step (e.g. Hanoi branches twice).
- **Challenge:** A simple linear loop cannot easily track multiple active branches.

To solve these iteratively, the programmer must **manually implement a stack data structure** to simulate the system call stack. In Python, we can use a `list` to achieve this goal (we will learn about lists in future chapters).

4.5.1 Tower of Hanoi

We simulate the recursion by pushing "jobs" onto a list. Since a stack is LIFO (Last-In, First-Out), we must push jobs in the **reverse order** of desired execution.

```python
def hanoi_iterative(n, source, aux, target):
    # Stack stores tuples: (number_of_disks, src, a, dst)
    # Initial job: Move n disks from source to target
    stack = [(n, source, aux, target)]

    while stack:
        # Pop the current job
        n, s, a, t = stack.pop()

        if n == 1:
            # Base Case: Directly move the single disk
            print(f"Move disk from {s} to {t}")
        else:
            # Recursive Step: Push 3 sub-problems to stack.
            # We push in REVERSE order of execution so they pop
                correctly.

            # 3. Later, move n-1 from Aux -> Target
            stack.append((n - 1, a, s, t))

            # 2. Then, move disk n from Source -> Target
            # We represent this as a "base case" of moving 1 disk
            stack.append((1, s, a, t))

            # 1. First, move n-1 from Source -> Aux
            stack.append((n - 1, s, t, a))
```

Notice that in this case, we do not get an improvement in terms of the space complexity. We have simply used a manual stack instead of the call stack.

4.5.2 Coin Change

For Coin Change, we use a stack to explore the "state space". The state consists of the current remaining `amount` and the index of the `coin` we are considering.

```python
def count_change_iterative(amount, coins):
    # Stack stores state: (current_amount, coin_index)
    stack = [(amount, 0)]
    ways = 0

    while stack:
        cur_amount, idx = stack.pop()

        # Base Case 1: Success
        if cur_amount == 0:
            ways += 1
            continue

        # Base Case 2: Failure (Overshot or no coins left)
        if cur_amount < 0 or idx >= len(coins):
            continue

        # Recursive Steps: Push children to stack
        # Branch 2: Skip the current coin
        stack.append((cur_amount, idx + 1))

        # Branch 1: Use the current coin
        stack.append((cur_amount - coins[idx], idx))

    return ways
```

Notice that, once again, we do not get an improvement in terms of the space complexity. We have simply used a manual stack instead of the call stack.

Looking Forward: Dynamic Programming

While this iterative version prevents a `RecursionError` (stack overflow), it is **not** faster than the recursive version.
Both are still **exponential time** ($O(k^n)$) because they blindly re-calculate the same overlapping subproblems (e.g. reaching amount 5 via multiple different paths).
In the Dynamic Programming section later on, we will learn to use **memoization** (caching) to store these results, reducing the complexity to Polynomial Time.

Breadth-First Search and Depth-First Search

The iterative coin change algorithm above is an example of **Depth-First Search (DFS)**. In any iterative DFS, we replace the system's call stack with a programmer-defined **stack** (Last-In, First-Out). This forces the algorithm to dive deep into the most recently discovered path before backtracking.

```python
def dfs_iterative(start_node):
    # 1. Initialize Stack
    stack = [start_node]
    visited = set() # To prevent cycles (if graph, not tree)
    while stack:
        # 2. Pop the most recent node (LIFO)
        current = stack.pop()
        if current in visited: continue
        visited.add(current)
        # 3. Process current node
        if is_goal(current):
            return "Found"
        # 4. Push children to stack
        # Note: We push all neighbors. Because it's a stack,
        # the LAST neighbor pushed is the FIRST one visited.
        for neighbor in get_neighbors(current):
            stack.append(neighbor)
```

In coin change, our "neighbors" were the two recursive branches: *skip coin* and *use coin*.

Similarly, we could also get a correct answer by doing **breadth-first search**, which uses a queue (First-In, First-Out). The algorithm changes from diving deep to exploring **level-by-level**. It processes all nodes at depth d before moving to depth $d+1$.

```python
from collections import deque # Generally not allowed in exams
def bfs_iterative(start_node):
    # 1. Initialize Queue (FIFO)
    queue = deque([start_node])
    visited = set()
    while queue:
        # 2. Pop the OLDEST node (FIFO - popleft)
        current = queue.popleft()
        if current in visited: continue
        visited.add(current)
        if is_goal(current): return "Found"
        # 3. Add neighbors to back of queue
        for neighbor in get_neighbors(current):
            queue.append(neighbor)
```

4.6 Summary: Iteration vs. Recursion

While both iteration and recursion are mechanisms to repeat computations, they differ fundamentally in how they manage the **state** of the program and how the computer executes them.

4.6.1 The Fundamental Difference: State Management

The core distinction lies in whether operations are **deferred** (left pending) or completed immediately.

1. **Recursive Process (Deferred Operations):**
    - When a function calls itself, the current computation pauses.
    - The system must "remember" where to return and what to do with the result.
    - This creates a chain of **deferred/pending** operations.
    - **Mechanism:** The operating system handles this automatically using the **call stack**. We rely on the system for convenience.
2. **Iterative Process (No Deferred Operations):**
    - The state is updated explicitly by the programmer using variables (counters, accumulators).
    - There are **no deferred operations**. At any point in the loop, the program's progress is fully captured by the current values of the variables.
    - **Mechanism:** Uses standard language constructs (`while`, `for`) without consuming additional stack memory.

4.6.2 Trade-offs: Efficiency vs. Complexity

Choosing between recursion and iteration involves a trade-off between human conceptual simplicity and machine efficiency.

	Recursion	**Iteration**
Pros	Conceptually easier to obtain solutions, especially for hierarchical data (trees, graphs).	It can often be more efficient in terms of time and space.
Cons	Can be expensive. Consumes memory for stack frames ($O(n)$ space) and time for function call overhead.	Can be difficult to implement for complex problems (like Tower of Hanoi). Requires the programmer to **manually handle the stack**.

4.6.3 When to use which?

Given that iteration is sometimes more efficient, you might be tempted to avoid recursion. However, for this course, we follow this guideline:

"Don't hesitate to write recursive processes."

- If a problem is naturally recursive (like traversing a tree), write the recursive solution. It is cleaner, easier to debug, and easier to understand.
- **Leave the optimization to the interpreter** (or compiler). Modern computers are fast enough that the overhead of recursion is often negligible compared to the cost of developer time and potential bugs in complex iterative code.
- Only switch to iteration if you hit a specific performance bottleneck (e.g. `RecursionError` or memory limits).

Unless otherwise stated, time complexity will not affect the grading of your solution in this course.

Looking Forward: Other CS Courses?

Of course, the focus of CS1010(X) is on correctness, while introducing you to basic ideas of efficiency.
So, where does this all fit in?

- CS2030/S will focus on *maintainability and readability* of your code, so that other programmers are able to understand your code.
- CS2040/S/C will focus on *efficiency*. Since writing code with $O(2^n)$ time complexity is often infeasible, CS2040/S/C explores ways to make our code more efficient.
- CS3230 and CS4234 further focus on *efficiency*. What are some advanced techniques used to speed up our code, especially in performance-critical systems?

4.7 Order of Growth (Complexity Analysis)

When designing software, we must distinguish between two levels of abstraction:

- **Algorithm:** The abstract "recipe" for solving a problem (e.g. "repeatedly pick the smallest card and move it to the front").
- **Program:** A concrete implementation of that recipe in a particular language, running on a particular machine.

A single algorithm can have many implementations. If we compare them using *wall-clock time* (seconds), the result depends on:

- CPU speed (phone vs. supercomputer),
- compiler optimisations,
- current system load, cache behaviour, etc.

To reason about algorithms in a machine-independent way, we study how their *resource usage* grows as the input size n becomes large. This is the idea of **Order of Growth**.

Formally, for each algorithm we associate a function

$$T(n) = \text{number of primitive steps for an input of size } n,$$

and then study how $T(n)$ behaves as $n \to \infty$.

4.7.1 Resources: Time and Space

We focus on two main resources:

1. **Time Complexity** The number of basic operations (comparisons, assignments, arithmetic) an algorithm performs as a function of input size n.
2. **Space Complexity** The amount of memory (RAM / stack) used during the execution, also as a function of n.

In practice, there is often a **time–space trade-off**. For example, we can:

- spend extra *time* recomputing answers from scratch each time, or
- spend extra *space* storing answers in a table (memoization) so that future queries are $O(1)$.

4.7.2 Asymptotic Notation

We now introduce the standard notations used to describe asymptotic growth.

Throughout, let $f(n)$ and $g(n)$ be non-negative functions defined on sufficiently large integers.

1. Big-O: Asymptotic Upper Bound

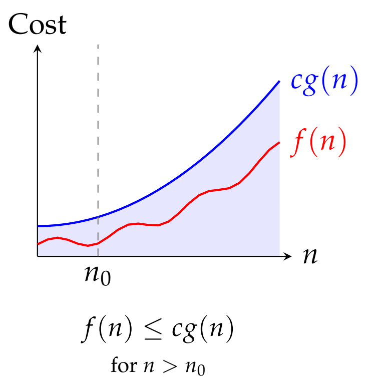
We say
$$f(n) = O(g(n))$$
if there exist constants $c > 0$ and $n_0 \geq 1$ such that
$$0 \leq f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$
Intuitively, for large n, $f(n)$ never grows faster than a constant multiple of $g(n)$. Big-O is an *asymptotic upper bound* (worst-case ceiling).

2. Big-Omega: Asymptotic Lower Bound

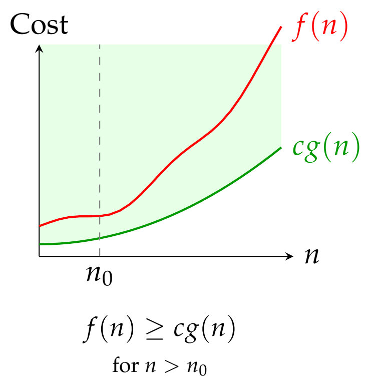
We say
$$f(n) = \Omega(g(n))$$
if there exist constants $c > 0$ and $n_0 \geq 1$ such that
$$0 \leq c \cdot g(n) \leq f(n) \quad \text{for all } n \geq n_0.$$
Intuitively, for large n, $f(n)$ never grows slower than a constant multiple of $g(n)$. Big-Omega is an *asymptotic lower bound* (best-case floor).

3. Big-Theta: Tight Bound

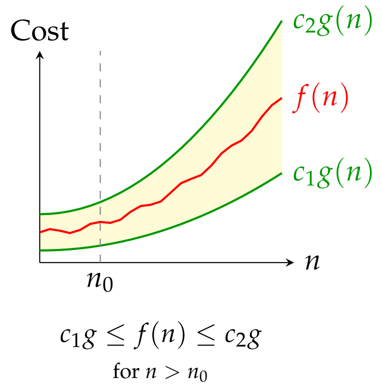
We say
$$f(n) = \Theta(g(n))$$
if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 1$ such that
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0.$$
In other words, $f(n)$ is bounded *both* above and below by constant multiples of $g(n)$. Big-Theta is a *tight asymptotic bound*: f and g grow at the same rate up to constant factors.



(a) $O(g(n))$: Upper Bound



(b) $\Omega(g(n))$: Lower Bound



(c) $\Theta(g(n))$: Tight Bound

In this course, when we say "the complexity is $O(n^2)$", we often implicitly mean a tight bound, i.e. $T(n) = \Theta(n^2)$, unless otherwise stated.

4.7.3 Simplification Rules

In principle, we could write down exact cost functions:

$$T(n) = 3n^2 + 10n + 500.$$

Asymptotically, only the fastest-growing term matters.

1. **Drop lower-order terms.** As $n \to \infty$, the fastest-growing term dominates:

$$3n^2 + 10n + 500 \sim 3n^2.$$

2. **Drop constant factors.** Constants reflect machine speed and implementation details:

$$3n^2 \rightsquigarrow n^2 \quad \Rightarrow \quad T(n) = O(n^2).$$

Formally:

$$3n^2 + 10n + 500 = \Theta(n^2).$$

Logarithms and Bases

Logarithms with different bases differ only by a constant factor. For any bases $a, b > 1$:

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

The term $\frac{1}{\log_b a}$ is just a constant, so in asymptotic notation:

$$O(\log_2 n) = O(\log_{10} n) = O(\ln n).$$

For this reason we simply write $O(\log n)$ and ignore the base, unless we are doing very fine-grained analysis.

4.7.4 Common Orders of Growth

We classify algorithms by how their cost grows with input size n:

Notation	Name	Typical Example
$O(1)$	Constant	Array index access
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Simple loop
$O(n \log n)$	Linearithmic	Merge sort
$O(n^2)$	Quadratic	Nested loops
$O(2^n)$	Exponential	Power Set (Subsets)
$O(n!)$	Factorial	Permutations (Traveling Salesperson - naive)
$O(n^n)$	Super-Exp.	All mappings between two sets

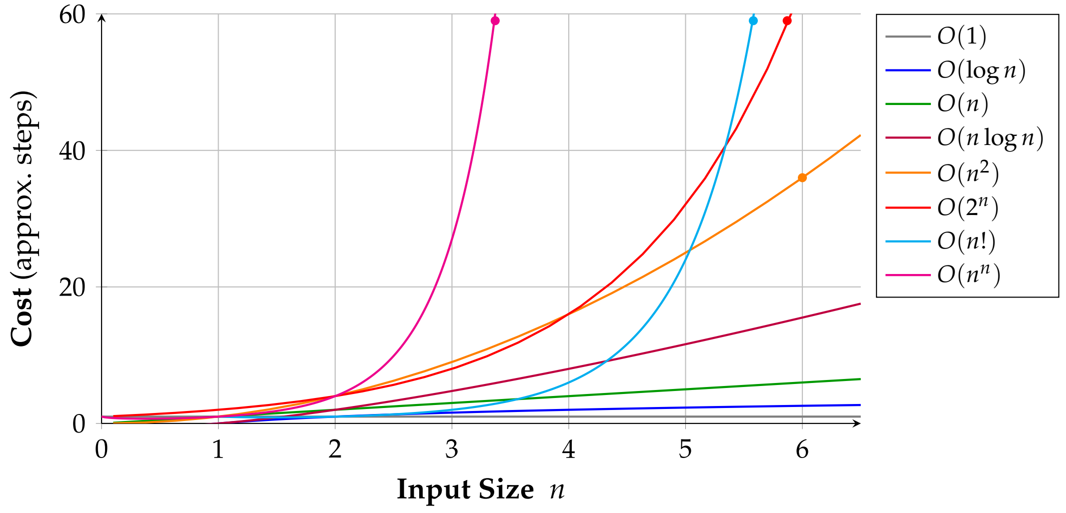


Figure 4.8: Comparison of common asymptotic growth rates. Notice how quickly exponential and quadratic costs grow compared to linear or logarithmic costs, even for small inputs.

Practical Limits: The "Cubic Wall"

Even though $O(n^2)$ and $O(n^3)$ are *polynomial* and thus considered "tractable" in theory, they can be too slow in practice for large n.

- $n = 10^5$: a linear-time algorithm ($\approx 10^5$ ops) is trivial.
- $n^2 = 10^{10}$: might take minutes to hours.
- $n^3 = 10^{15}$: would take **years** on a single core.

This is why cubic algorithms (e.g. naive matrix multiplication) quickly become impractical without clever optimisations or hardware support.

Galactic Algorithms

Sometimes, an algorithm has a mathematically superior asymptotic bound but with an enormous hidden constant factor. For example, suppose we had two functions doing the same computation:

$$T_1(n) = 10^{100} \cdot n \quad \text{vs.} \quad T_2(n) = n^2.$$

Asymptotically, $T_1(n) = O(n)$ is better than $T_2(n) = O(n^2)$, but T_1 is faster only when

$$10^{100} \cdot n < n^2 \quad \Rightarrow \quad n > 10^{100},$$

far larger than any realistic input size (> number of atoms in the observable universe). Algorithms that are asymptotically great but useless for all practical input sizes are called **Galactic Algorithms**. Thus, when we use Big-O to compare algorithms, we implicitly assume constant factors are "reasonable".

4.7.5 Worked Examples of Complexity

We now analyse some small pieces of code to see these ideas in action.

Example 1: Nested Loops

```python
total = 0
for i in range(n):          # Runs n times
    for j in range(i):      # Runs 0, 1, 2, ..., n-1 times
        total += 1
```

The inner statement `total += 1` runs

$$0 + 1 + 2 + \cdots + (n-1) = \sum_{i=0}^{n-1} i$$

times. Using the arithmetic series formula:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2).$$

Thus the whole block has

$$T(n) = \Theta(n^2).$$

Example 2: More Nested Loops?

This pattern appears whenever we combine a linear outer loop with an inner loop that grows or shrinks exponentially.

```python
for i in range(n):          # Outer loop: runs n times
    j = 1
    while j < n:            # Inner loop: j doubles each time
        # some O(1) work
        j = j * 2
```

- The outer loop clearly runs n times.
- In the inner loop, j takes values $1, 2, 4, 8, \ldots, 2^k$ until $2^k \geq n$, so $k \approx \log_2 n$.

Hence the total work is approximately

$$n \times \log_2 n \quad \Rightarrow \quad T(n) = \Theta(n \log n).$$

4.7.6 Case Study: Exponentiation

We compare two ways to compute a^n. Let us treat n as the input size.

Naive Iteration: $\Theta(n)$ Multiplications

```python
def power(a, n):
    res = 1
    for i in range(n):
        res = res * a
    return res
```

We perform exactly n multiplications, so the time complexity is

$$T(n) = \Theta(n),$$

and space is $\Theta(1)$ (a few variables).

Binary Exponentiation: $\Theta(\log n)$ Multiplications

```python
def fast_power(a, n):
    res = 1
    while n > 0:
        if n % 2 == 1:
            res = res * a    # multiply
        a = a * a            # square base
        n = n // 2           # half exponent
    return res
```

Binary exponentiation relies on the following recurrence relation, splitting the problem based on the parity of n:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ is odd} \end{cases}$$

Key Intuition:

- If n is **even**, we can compute $a^{n/2}$ once and square it.
- If n is **odd**, we convert it to an even case by extracting one factor of a: $a^n = a^{n-1} \cdot a$.
- Since the exponent n is halved at every even step, the time complexity is logarithmic, $O(\log n)$.

Time complexity Analysis

Let $T(n)$ be the time for `fast_power(a, n)`. Each loop iteration:

- does $O(1)$ work (a few multiplications, modulus, division), and
- replaces n with $\lfloor n/2 \rfloor$.

So

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1).$$

We notice that the loop is called approximately $\log_2 n$ times. So,

$$T(n) = \Theta(\log n).$$

Rigorous Derivation

We substitute the recurrence into itself repeatedly:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c \\ &= \left(T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + c\right) + c \qquad = T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 2c \\ &= \left(T\left(\left\lfloor \frac{n}{8} \right\rfloor\right) + c\right) + 2c \qquad = T\left(\left\lfloor \frac{n}{8} \right\rfloor\right) + 3c \\ &\vdots \\ &= T\left(\left\lfloor \frac{n}{2^k} \right\rfloor\right) + k \cdot c \end{aligned}$$

The recursion terminates when the problem size reaches the base case, i.e. when:

$$\left\lfloor \frac{n}{2^k} \right\rfloor = 1.$$

$$\text{This implies} \quad 1 \le \frac{n}{2^k} < 2 \implies 2^k \le n < 2^{k+1}.$$

Taking the base-2 logarithm:

$$k \le \log_2 n < k + 1 \implies k = \lfloor \log_2 n \rfloor.$$

3. Final Substitution Substituting $k = \lfloor \log_2 n \rfloor$ back into our expanded equation:

$$\begin{aligned} T(n) &= T(1) + c \cdot \lfloor \log_2 n \rfloor \\ &= d + c \cdot \lfloor \log_2 n \rfloor. \end{aligned}$$

Since c and d are constants, and $\lfloor \log_2 n \rfloor \le \log_2 n$:

$$T(n) \approx c \log_2 n \implies T(n) = \Theta(\log n).$$

The space complexity is still $O(1)$: we keep only constant many variables (`a`, `n`, `res`), regardless of the size of n.

4.7.7 Best, Worst, and Average Case

For some algorithms, running time varies significantly depending on the input of the same size n. We distinguish:

- **Worst-case complexity** (usually denoted with Big-O): The maximum running time over all inputs of size n. *Example:* Linear search in the worst case scans all n items.
- **Best-case complexity** (often denoted with Big-Omega): The minimum running time over all inputs of size n. *Example:* In linear search, if the item is at index 0, we succeed in $O(1)$ time.
- **Average-case complexity**: The expected running time assuming a probability distribution over inputs. *Example:* Quicksort has worst-case $O(n^2)$ but average-case $O(n \log n)$ under mild assumptions on pivots.

In CS1010X, unless explicitly stated, we will usually talk about **worst-case** time complexity when we write $O(\cdot)$.

Looking Forward: Computer Science

Computer Science, as explored in CS1010X, is largely about **data structures** and **algorithms**.
We have learnt tools to analyse the complexity of algorithms, so we will move on to understand how to write better programs:

- **Data Structures** are ways of organizing data in our programs to facilitate computations (execution of algorithms). Some data structures we will look at include strings, sets, tuples, lists and dictionaries.
- But before we delve into data structure abstractions, we will look at higher-order functional abstractions.