

UNIT 6

Recursion

(4 Hours) — Data Structures and Algorithms

Recursion is a powerful problem-solving technique in which a function solves a problem by calling itself on smaller instances of the same problem. This unit covers the principle behind recursion, how it compares with iteration, two classic recursive examples — the Tower of Hanoi and the Fibonacci series — practical applications of recursion, and its use in searching a binary search tree.

6.1 Introduction to Recursion

A function is said to be recursive if it calls itself, either directly or indirectly (through another function), to solve a problem. Recursion is based on the mathematical idea of induction: a large problem is expressed in terms of one or more smaller instances of the same problem, together with one or more base cases that can be solved directly, without further recursion.

```
int factorial(int n) {  
    if (n == 0) /* base case */  
        return 1;  
    else /* recursive case */  
        return n * factorial(n - 1);  
}
```

Here, $\text{factorial}(n)$ is defined in terms of $\text{factorial}(n-1)$, a smaller version of the same problem, and the recursion terminates at the base case $\text{factorial}(0) = 1$. Every correct recursive function must have both an ingredient that reduces the problem and a way to stop.

6.1.1 Essential Components of a Recursive Function

- Base case — the simplest instance of the problem, solved directly without further recursive calls. Without a base case the recursion never terminates.
- Recursive case — the part of the function where it calls itself with a smaller or simpler input, moving the problem closer to the base case.
- Progress towards the base case — each recursive call must work on an input that is “smaller” in some well-defined sense (e.g. $n-1$, $n/2$, a shorter list), guaranteeing that the base case is eventually reached.

Note: A recursive function with no base case, or one whose recursive calls never approach the base case, causes infinite recursion, which eventually crashes the program with a stack overflow error.

6.2 Principle of Recursion

Every function call in a program — recursive or not — is managed using the system call stack. When a function is called, an activation record (stack frame) containing its local variables, parameters, and return address is pushed onto the stack. When the function returns, its frame is popped off and control resumes in the caller. Recursion is nothing but repeated function calls that exploit this same mechanism.

The execution of a recursive function therefore has two clearly separated phases:

- Winding (calling) phase — the function keeps calling itself with smaller arguments; a new stack frame is pushed for every call, until the base case is reached.
- Unwinding (returning) phase — once the base case returns a value, each pending call resumes, computes its own result using the value it just received, and returns it to its own caller; frames are popped one by one until the original call finally returns.

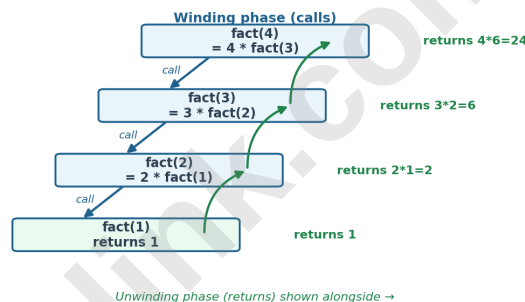


Fig 6.1 — Call stack behaviour of `factorial(4)`: the winding phase builds up calls, the unwinding phase computes and returns results.

Example 6.1 — Tracing `factorial(4)`

Trace the calls and returns of `factorial(4)` using the definition above.

Solution

`factorial(4)` calls `factorial(3)` calls `factorial(2)` calls `factorial(1)` calls `factorial(0)`.

`factorial(0)` returns 1 (base case reached — winding phase ends).

`factorial(1)` returns $1 * 1 = 1$.

`factorial(2)` returns $2 * 1 = 2$.

`factorial(3)` returns $3 * 2 = 6$.

`factorial(4)` returns $4 * 6 = 24$. — Final answer: 24

6.2.1 Space Requirement of Recursion

Because every pending call keeps its own stack frame alive until it returns, a recursive function that makes n nested calls before reaching its base case uses $O(n)$ additional memory on the call stack, even though it may perform no explicit data structure of its own. This is an important cost to consider when choosing recursion over iteration.

6.3 Recursion vs. Iteration

Any problem that can be solved recursively can also, in principle, be solved iteratively using a loop, and vice versa. The two techniques differ mainly in how they manage repeated execution and state.

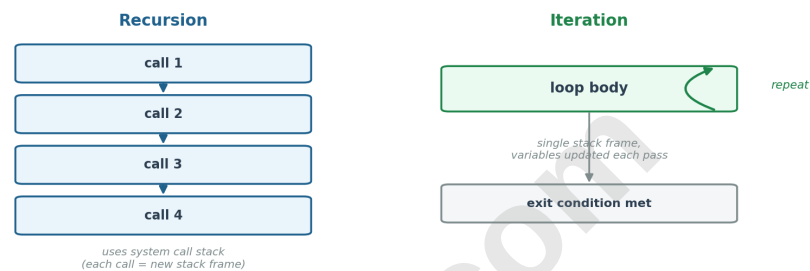


Fig 6.2 — Recursion relies on the call stack (one frame per call); iteration reuses a single stack frame across passes of a loop.

Aspect	Recursion	Iteration
Mechanism	Function calls itself	Loop (for / while) repeats a block
Termination	Base case	Loop condition becomes false
Memory usage	$O(n)$ extra stack space for n nested calls	$O(1)$ extra space (usually)
Speed	Slightly slower — function-call overhead	Generally faster — no call overhead
Code clarity	Often shorter, closer to mathematical definition	Can be longer, more explicit control flow
Best suited for	Problems naturally defined in terms of themselves (trees, divide-and-conquer)	Simple repetitive tasks with a clear loop counter
Risk	Stack overflow if recursion is too deep or has no base case	Infinite loop if the condition never becomes false

Example 6.2 — Factorial: recursive vs. iterative

Recursive: as shown in Section 6.1 — concise, but uses $O(n)$ stack space.

Iterative version

```
int factorial(int n) {  
    int result = 1;  
    for (int i = 2; i <= n; i++)  
        result *= i;  
    return result;  
}
```

This uses only $O(1)$ extra space, since a single variable is updated each pass instead of stacking calls.

Note: As a rule of thumb: prefer iteration when a simple loop solves the problem cleanly; prefer recursion when the problem has a naturally recursive structure (trees, divide-and-conquer, backtracking) where an iterative solution would need an explicit stack anyway.

bcalink.com

6.4 Recursion Examples

6.4.1 Tower of Hanoi (TOH)

The Tower of Hanoi is a classic puzzle consisting of three pegs (A = source, B = auxiliary, C = destination) and n disks of different sizes stacked on the source peg in decreasing order of size (largest at the bottom). The goal is to move the entire stack to the destination peg, obeying two rules:

- Only one disk may be moved at a time, and only the topmost disk of a peg may be moved.
- A larger disk may never be placed on top of a smaller disk.



Fig 6.3 — Tower of Hanoi with 3 disks: initial state (left) and final state (right).

The recursive idea: to move n disks from A to C using B as auxiliary, (1) move the top $n-1$ disks from A to B using C as auxiliary, (2) move the single largest disk directly from A to C, and (3) move the $n-1$ disks from B to C using A as auxiliary. Each of steps (1) and (3) is the same problem on a smaller number of disks, which is exactly what makes the solution recursive.

```
void TOH(int n, char from, char aux, char to) {
    if (n == 0) return; /* base case: nothing to move */
    TOH(n - 1, from, to, aux); /* move n-1 disks out of the way */
    printf("Move disk %d from %c to %c\n", n, from, to);
    TOH(n - 1, aux, from, to); /* move n-1 disks onto the target */
}
/* Call as: TOH(3, 'A', 'B', 'C'); */
```

Example 6.3 — Number of moves for TOH

How many moves does TOH(n) make, and how many for $n = 3$?

Solution

Let $M(n)$ be the number of moves. From the recursive definition:

$$M(n) = M(n-1) + 1 + M(n-1) = 2 \cdot M(n-1) + 1, \text{ with } M(0) = 0.$$

Solving this recurrence gives $M(n) = 2^n - 1$.

For $n = 3$: $M(3) = 2^3 - 1 = 7$ moves.

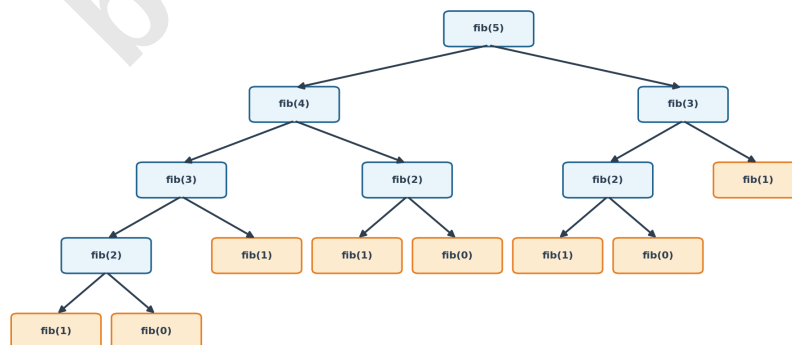
Note: Because $M(n) = 2^n - 1$ grows exponentially, the Tower of Hanoi is often used to illustrate exponential time complexity: even $n = 30$ disks would require over a billion moves.

6.4.2 Fibonacci Series

The Fibonacci series is defined recursively as: $\text{fib}(0) = 0$, $\text{fib}(1) = 1$, and $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ for $n \geq 2$. Each term (after the first two) is the sum of the two preceding terms, giving the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
int fib(int n) {  
    if (n == 0) return 0; /* base case 1 */  
    if (n == 1) return 1; /* base case 2 */  
    return fib(n - 1) + fib(n - 2); /* recursive case */  
}
```

Unlike factorial or TOH, $\text{fib}(n)$ makes two recursive calls instead of one. This produces a branching recursion tree, and — critically — many identical subproblems are solved over and over again.



Overlapping subproblems, e.g. $\text{fib}(3)$ and $\text{fib}(2)$, are recomputed many times — highlighted nodes are base cases.

Fig 6.4 — Recursion tree for $\text{fib}(5)$. Notice $\text{fib}(3)$ is computed twice and $\text{fib}(2)$ three times — these are overlapping subproblems.

Version	Time Complexity	Space Complexity	Remarks
Plain recursion (above)	$O(2^n)$ — exponential	$O(n)$ — call stack depth	Simple but re-solves the same subproblems repeatedly

Version	Time Complexity	Space Complexity	Remarks
Recursion with memoisation	$O(n)$	$O(n)$	Cache each fib(k) the first time it is computed
Iterative (bottom-up)	$O(n)$	$O(1)$	Keep only the last two values; no recursion needed

Note: The exponential blow-up of plain recursive Fibonacci is a textbook example of why overlapping subproblems matter, and it motivates techniques such as memoisation and dynamic programming.

bcalink.com

6.5 Applications of Recursion

Recursion is not just an academic exercise — it is the natural way to express solutions to many important problems, particularly those with a self-similar or hierarchical structure.

Application	Why recursion fits naturally
Factorial / power / GCD computation	Defined mathematically in terms of a smaller instance of themselves
Tower of Hanoi	The n-disk problem reduces cleanly to two (n-1)-disk sub-problems
Fibonacci series	Each term is defined using the two preceding terms
Tree traversals (inorder, preorder, postorder)	A tree is itself made of smaller sub-trees (left and right)
Binary search / binary search tree operations	Each step searches a smaller half of the remaining data
Divide-and-conquer algorithms (Merge Sort, Quick Sort)	The problem is split into smaller sub-problems solved the same way
Backtracking (maze solving, N-Queens, Sudoku)	Explores choices by recursively trying and undoing decisions
Parsing expressions / nested structures (brackets, JSON, HTML)	Nested structures are naturally self-referential

Example 6.4 — Sum of digits (a simple recursive utility)

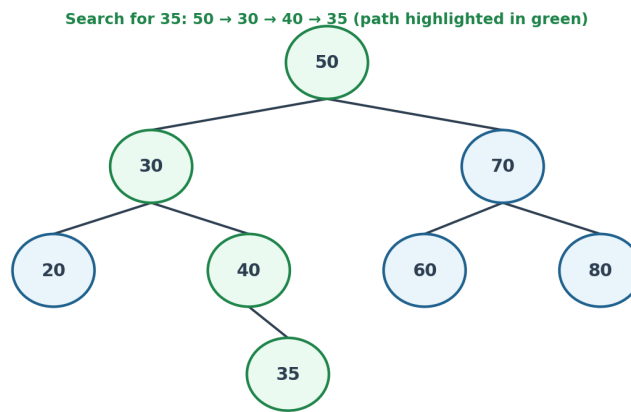
Write a recursive function to compute the sum of the digits of a non-negative integer n.

Solution

```
int digitSum(int n) {  
    if (n == 0) return 0; /* base case */  
    return (n % 10) + digitSum(n / 10); /* last digit + rest */  
}
```

6.6 Search Tree

A search tree — most commonly a Binary Search Tree (BST) — is a binary tree that stores data so it can be searched efficiently using recursion. For every node, all values in its left subtree are smaller than the node's value, and all values in its right subtree are larger.



At each node: if key < node → go left; if key > node → go right; if equal → found

Fig 6.5 — A binary search tree; searching for 35 recursively compares against 50, then 30, then 40, then 35.

Searching a BST is a textbook example of recursion because the same comparison logic applies at every level — the search for a key in a tree reduces to the same search on a smaller sub-tree (the left or right child):

```

struct TNode { int key; struct TNode *left, *right; };
typedef struct TNode *TNODEPTR;
TNODEPTR search(TNODEPTR root, int key) {
    if (root == NULL || root->key == key)
        return root; /* base case: found, or not present */
    if (key < root->key)
        return search(root->left, key); /* recurse left */
    else
        return search(root->right, key); /* recurse right */
}
  
```

Example 6.5 — Tracing search(root, 35) on Fig 6.5

Trace the recursive calls made while searching for 35 in the tree shown above.

Solution

search(50, 35): 35 < 50 ⇒ recurse left → search(30, 35)

search(30, 35): 35 > 30 ⇒ recurse right → search(40, 35)

search(40, 35): 35 < 40 ⇒ recurse left → search(35, 35)

search(35, 35): key == root->key ⇒ base case — node found, return it.

Because each recursive call moves to a child that (on average) covers roughly half the remaining nodes, searching a reasonably balanced BST with n nodes takes only $O(\log n)$ time — far faster than a linear scan of a list, and a direct benefit of exploiting the tree's naturally recursive structure.

Note: If the tree is skewed (e.g. every node has only a right child), it degenerates into a structure similar to a linked list, and search time worsens to $O(n)$. Balanced search trees (AVL, Red-Black) are designed to avoid this.

Key Takeaways

Unit 6 — Key Takeaways

- ✓ A recursive function solves a problem by calling itself on a smaller instance, and must always have a base case plus progress towards it.
- ✓ Recursion executes in a winding phase (calls build up on the stack) and an unwinding phase (results are computed and returned as calls pop off).
- ✓ Compared to iteration, recursion is often more concise and natural for self-similar problems, but costs $O(n)$ extra stack space for n nested calls.
- ✓ Tower of Hanoi needs $2^n - 1$ moves for n disks and is a natural, elegant recursive solution; Fibonacci's plain recursive version wastes time recomputing overlapping subproblems ($O(2^n)$).
- ✓ Recursion is the natural fit for tree traversal, divide-and-conquer algorithms, backtracking, and parsing nested structures.
- ✓ Searching a binary search tree is recursive by nature — at each node, the search reduces to the same search on the left or right sub-tree, giving $O(\log n)$ time on a balanced tree.

— End of Unit 6 —