UNIT 3 · STACKS AND RECURSION

12. Recursion

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

Agenda

1. Base case and recursive case
2. How recursion uses the call stack, traced frame by frame
3. Recursion vs. iteration — a real timed comparison on Fibonacci
4. Why deep recursion can exhaust the stack
5. Types of recursion, and its advantages and limitations

Base Case and Recursive Case

Base case
The simplest version, answered directly — no further recursive call

Recursive case
Calls itself on a smaller problem, then combines that result

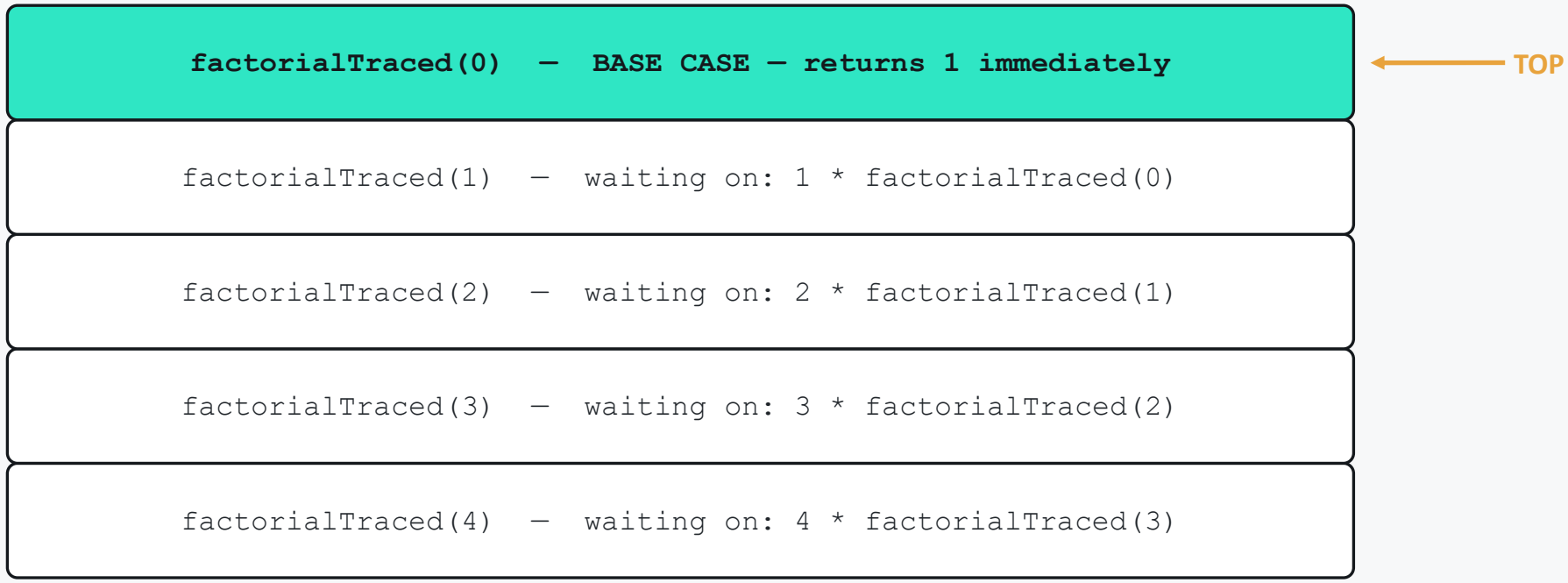
factorial.cpp

```cpp
unsigned long factorial(int n) {
    if (n == 0) {
        return 1;              // base case
    }
    return n * factorial(n - 1);
    // recursive case: smaller problem, then combine
}
```

Without a reachable base case, the function calls itself forever — the single most common bug in recursive code.

The Call Stack, Frame by Frame

Every call pushes a stack frame; a recursive call is just a function pushing another copy of itself — stacked exactly like Lecture 10's LinkedStack.



Unwinding, base case up: 0→1 (1×1=1) → 2 (2×1=2) → 3 (3×2=6) → 4 (4×6=24) — each frame resumes, multiplies, and pops.

Watching It Happen: A Traced Call

recursion_trace.cpp

```
int factorialTraced(int n, int depth=0) {
  string indent(depth*2, ' ');
  cout << indent << "-> f(" << n
       << ") called" << endl;

  if (n == 0) {
    cout << indent << "<- base case";
    return 1;
  }

  int result = n *
    factorialTraced(n-1, depth+1);
  cout << indent << "<- returning "
       << result << endl;
  return result;
}
```

output for factorialTraced(4)

```
-> factorialTraced(4) called
  -> factorialTraced(3) called
    -> factorialTraced(2) called
      -> factorialTraced(1) called
        -> factorialTraced(0) called
        <- base case, returning 1
      <- factorialTraced(1) returning 1
    <- factorialTraced(2) returning 2
  <- factorialTraced(3) returning 6
<- factorialTraced(4) returning 24
Result: 24
```

Calls go down (indenting deeper) until the base case; results come back up in reverse order — the call stack's LIFO behavior, made visible.

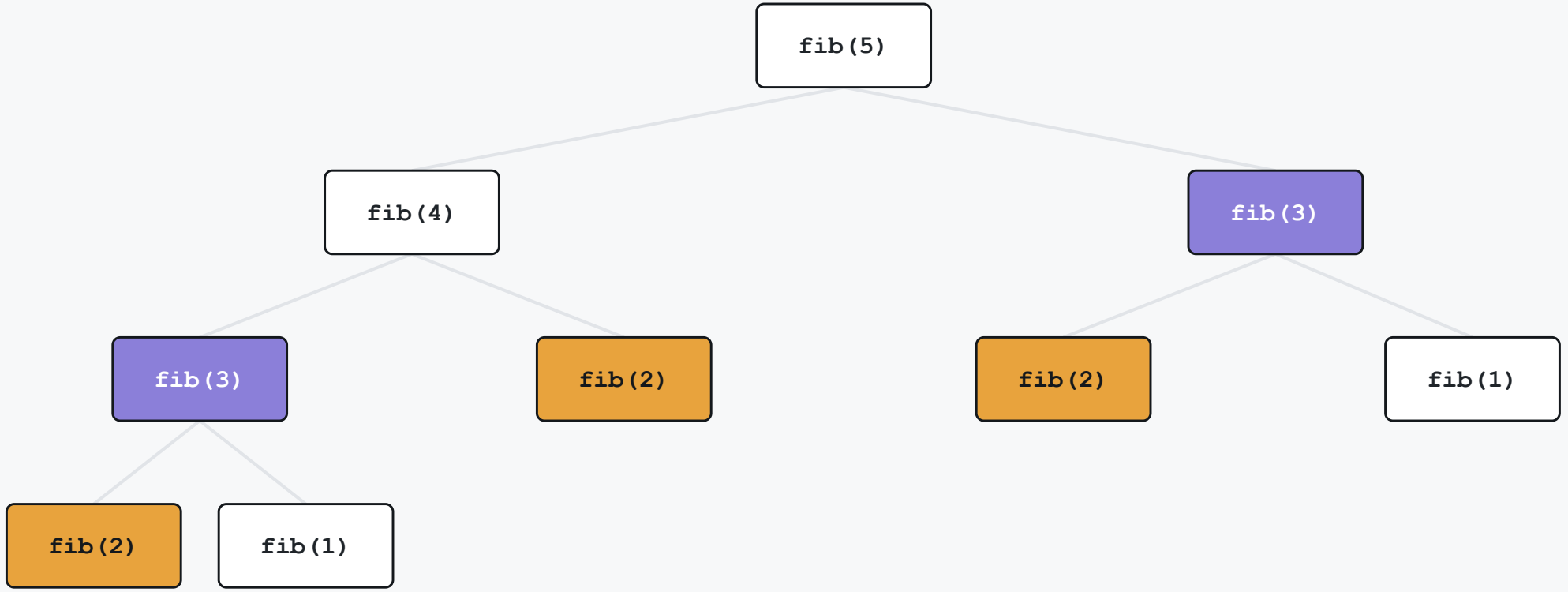
Recursion vs. Iteration

	Recursion	Iteration
Memory	O(n) call-stack space for n nested calls	O(1) extra space, just loop variables
Readability	Often closer to the mathematical definition (trees are naturally recursive)	Can be less obvious for inherently recursive problems
Risk	Stack overflow if the base case is unreachable or input is very large	Infinite loop if the condition is wrong — no stack risk

Anything recursion can do, a loop can also do — the two are computationally equivalent. The choice is about which expresses the problem more naturally.

Fibonacci: Exponential Repeated Work

fib(3) appears twice and fib(2) appears three times in just this small slice down to fib(5) — each identical call redoes its entire subtree from scratch.



The call count roughly doubles with every increase in n — $O(2^n)$ total calls, genuinely exponential, unlike factorial's O(n).

Same Answer, Wildly Different Cost

fib_recursive_vs_iterative.cpp

```
long fibRecursive(int n) {
    if (n <= 1) return n;
    return fibRecursive(n-1)
         + fibRecursive(n-2);
}

long fibIterative(int n) {
    long prev=0, curr=1;
    for (int i=2; i<=n; i++) {
        long next = prev+curr;
        prev = curr; curr = next;
    }
    return curr;
}
```

100x+

fibRecursive(32) is at least 100x slower than fibIterative(32) on the same machine

2178309

both compute the exact same fib(32) — correctness isn't the issue, cost is

The program prints a bucketed verdict (“at least 100x slower”), not a raw millisecond count — the exact ratio depends on the machine, but exponential-beats-linear by orders of magnitude does not.

Units 5 and 6 (trees and graphs) fix exactly this kind of repeated-subproblem waste with memoization — caching each subproblem's answer the first time it's computed.

Why Deep Recursion Can Exhaust the Stack

deep_recursion.cpp

```
int recurseTo(int target, int current = 1) {
    if (current >= target) return current;
    return recurseTo(target, current + 1);
}
```

1,000
reached successfully

5,000
reached successfully

20,000
reached successfully

~30–40k
typical crash threshold, MinGW/Windows

A typical call stack is only a few megabytes (set by the OS at start), nothing like the gigabytes of heap new can draw from. Recurse deep enough and it runs out: a stack overflow that typically crashes the program immediately, with no exception to catch.

The fix is never “recurse more carefully” — it's converting to an equivalent iterative loop, or explicitly managing your own heap-based stack.

Types of Recursion

Direct recursion

A function calls itself directly, as factorial does.

Indirect recursion

Function A calls function B, which calls A again.

Tail recursion

The recursive call is the very last thing the function does, with no pending work afterward — some compilers can optimize this into a loop.

factorial(int n) is not tail-recursive (pending n means the frame is still needed after the call returns); a rewrite with an accumulator parameter would be — but C++ never guarantees the optimization happens.*

Advantages and Limitations

Advantages

- Naturally expresses problems that are themselves defined recursively — tree traversal, graph traversal, and divide-and-conquer sorting all read far more clearly as recursion than hand-rolled loops.
- Often shorter and closer to a mathematical proof than the equivalent iterative code.

Limitations

- Each call consumes stack memory — deep recursion on very large inputs can overflow the stack, a risk plain loops don't share.
- Can be slower than iteration due to function-call overhead, unless the compiler optimizes it away.

Key Takeaways

- Every correct recursive function needs a base case (stops the recursion) and a recursive case (calls itself on a smaller problem).
- Recursive calls use the call stack exactly like any other function call — LIFO, each paused call waiting for the one below it to return.
- Recursion and iteration are computationally equivalent, but not equally fast: naive recursive Fibonacci is $O(2^n)$ from massively overlapping repeated work, while the iterative version is $O(n)$.
- A missing or unreachable base case causes a stack overflow — the most common bug in recursive code; even a correct base case doesn't save you if the recursion is simply too deep for the available stack space.
- Tail recursion can in principle be optimized into a loop at zero extra stack cost, but C++ never guarantees this, so it shouldn't replace an explicit iterative rewrite when stack depth is a genuine concern.