CS 112

Introduction to Data Structures

WEEK 10

Recursion and Sorting

Eric Araújo

Calvin University · Fall 2026

This Week

1. What recursion is
2. Base case and recursive case
3. The run-time stack **KEY**
4. Winding and unwinding
5. Towers of Hanoi
6. Selection and insertion sort
7. Why merge sort recurses

A Function That Calls Itself

The sum of an array is the first item plus *the sum of the rest*.

```
int sum(int a[], int n) {
    if (n == 0) { return 0; }            // trivial case
    return a[0] + sum(a + 1, n - 1);     // the rest
}
```

> The definition of the answer contains a smaller version of the same question. That's recursion, and it's why the code reads almost exactly like the sentence above it.

Designing a Recursive Solution

- **1. Find the trivial case.** Which input can you answer with no work at all? That's the *basis*.
- **2. Find the step.** How does the answer for *n* relate to the answer for something smaller?
- **3. Check it shrinks.** Every call must move toward the basis, or it never stops.

> Miss the basis and you get infinite recursion. Miss the shrinking and you get infinite recursion. Both end the same way: **stack overflow**.

Factorial

```
// n! = n * (n-1)!      and      0! = 1! = 1

unsigned factorial(unsigned n) {
    if (n <= 1) {
        return 1;                       // basis
    }
    return n * factorial(n - 1);        // recursive step
}
```

Basis: 0! and 1! are 1, no work needed.

Step: n! is n times a *smaller* factorial.

TALK TO YOUR NEIGHBOR · TTYN

What happens if the basis is removed?

```
unsigned factorial(unsigned n) {
    return n * factorial(n - 1);
}
```

A. It returns 0

B. It recurses forever until the stack overflows

C. The compiler rejects it

D. It still works for positive n

The Run-Time Stack

Every call gets a **frame**: its parameters, its locals, and where to return to.

top →	**factorial(1)**	returns 1
factorial(2)		waiting on 1
factorial(3)		waiting on 2
factorial(4)		waiting on 3
main()		waiting on 4

> It is literally a stack, the same LIFO structure from last week. The most recent call is the first to finish.

Winding and Unwinding

Winding: calls go down

```
factorial(4)
 factorial(3)
  factorial(2)
   factorial(1) → 1
```

Unwinding: answers come back

```
   1
  2 * 1  = 2
 3 * 2   = 6
4 * 6    = 24
```

> Nothing is multiplied on the way *down*. All the arithmetic happens on the way back *up*: which is why every frame has to stay on the stack until the basis is reached.

TALK TO YOUR NEIGHBOR · TTYN

What is the time complexity of `factorial(n)`?

A. O(1)

B. O(lg n)

C. O(n)

D. $O(n^2)$

Recursion Isn't Free

Iterative

```
unsigned f(unsigned n) {
  unsigned r = 1;
  for (unsigned i = 2;
       i <= n; ++i) {
    r *= i;
  }
  return r;
}
```

O(n) time, **O(1) space**

Recursive

```
unsigned f(unsigned n) {
  if (n <= 1) return 1;
  return n * f(n - 1);
}
```

O(n) time, **O(n) space**

> Every call costs a frame and a jump. For a plain loop like this, iteration wins. Reach for recursion when the *problem itself* is recursive.

When Recursion Earns Its Keep

Printing a singly-linked list **backwards**. Iteratively that means walking the list once per item, $O(n^2)$, because you can't step back.

```
void Node::printReverse(ostream &out) const {
    if (myNext != nullptr) {
        myNext->printReverse(out);   // rest of the list FIRST
    }
    out << myItem << " ";           // then me
}
```

> One pass, O(n). The run-time stack remembers the way back for you; that's the data structure you'd otherwise have to build by hand.

Towers of Hanoi

Move a stack of disks from A to B using C. Never put a larger disk on a smaller one.

A		start
B	**(empty)**	goal
C	**(empty)**	spare

> Iteratively this is a nightmare. Recursively it's three lines, because the problem is recursive all the way down.

Hanoi, Recursively

```
void move(int n, char src, char dest, char aux) {
    if (n == 1) {
        cout << "move disk from " << src << " to " << dest;
        return;
    }
    move(n - 1, src, aux, dest);    // get the top n-1 out of the way
    move(1, src, dest, aux);        // move the big one
    move(n - 1, aux, dest, src);    // bring the n-1 back on top
}
```

> Notice we never say *how* to move n−1 disks. We assume it works and use it, the recursive leap of faith, justified by the basis.

TALK TO YOUR NEIGHBOR · TTYN

How many moves does `move(n, ...)` make?

A. n

B. n^2

C. $2^n - 1$

D. $n \lg n$

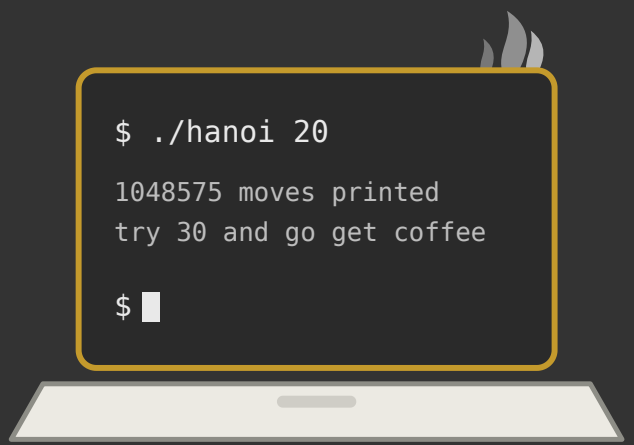
Sixty-Four Disks

The legend says monks are moving 64 disks, and the world ends when they finish.

- $2^{64} - 1 \approx 1.8 \times 10^{19}$ moves
- A supercomputer printing 1,000,000 moves per second…
- …still needs about **580,000 years**

> Elegant code and fast code are different things. An exponential algorithm is unusable at almost any interesting size, no faster machine will save you.

Demo



Tracing recursion in the debugger.

(base case reached. patience not.)

Why This Matters: Sorting

Sorting those same million items:

Algorithm	Complexity	Rough operations
Bubble sort	$O(n^2)$	10^{12}, hours
Merge / quick sort	O(n lg n)	2×10^7, under a second

> Same computer, same data, same answer. The only difference is the shape of the growth curve, and it's the difference between a coffee break and a lunch break you never come back from.

Selection Sort

```
for (int i = 0; i < n - 1; ++i) {
    int smallest = i;
    for (int j = i + 1; j < n; ++j) {
        if (a[j] < a[smallest]) { smallest = j; }
    }
    swap(a[i], a[smallest]);
}
```

Find the smallest of what's left, swap it into place, repeat.

> Nested loops over n → **O(n²)**, always. It does the same work on sorted input as on random input, but it makes at most n−1 swaps.

Insertion Sort

```
for (int i = 1; i < n; ++i) {
    Item key = a[i];
    int j = i - 1;
    while (j >= 0 && a[j] > key) {
        a[j + 1] = a[j];      // shift right
        --j;
    }
    a[j + 1] = key;           // drop it in
}
```

Take the next item, slide it back until it's in place, the way you sort a hand of cards.

> Worst case $O(n^2)$, but on **already-sorted** input the inner loop never runs: $O(n)$.

TALK TO YOUR NEIGHBOR · TTYN

Your data is almost sorted already. Which of the two is better?

A. Selection sort; it always does the same work

B. Insertion sort; it approaches O(n) when little is out of place

C. Identical, both are $O(n^2)$

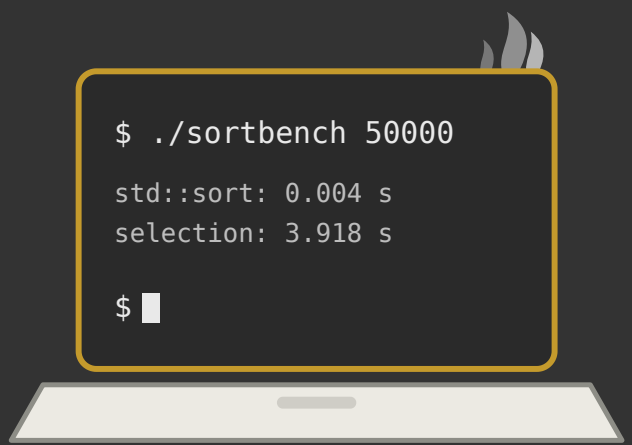
D. Neither works on partly sorted data

Comparing the Two

	Selection sort	Insertion sort
Best case	$O(n^2)$	**O(n)**
Worst case	$O(n^2)$	$O(n^2)$
Swaps / writes	**O(n)**	$O(n^2)$
Stable?	no	**yes**

> Neither is what you'd ship, `std::sort` is O(n lg n). But every fast sort is built from ideas you can now read, and both of these are the right choice for a small enough n.

Demo



Racing selection sort against std::sort.

(selection sort is having a long afternoon)

Week 10 Recap

- A recursive function needs a **basis** and a step that **shrinks**
- Every call takes a frame on the **run-time stack**
- Work happens on the way back up: that's the unwinding phase
- Iteration is usually faster; recursion is for recursive problems
- Towers of Hanoi is $O(2^n)$, elegant and completely impractical

Next: a12 · a13 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**