Don't Fear Recursion: 20 Practice Problems

A companion to Don't Fear Recursion. Build its two habits: **trust the function** when you write it, **focus on one step** when you trace it.

Rules: no loops and no global variables inside the recursive function. Whatever a step needs travels in its parameters or comes back in its return value. That's the whole point of the exercise.

Try before you read on. Every problem is followed by its solution, so cover it up and write your own first. Solutions show only the recursive function; reading input and printing output is up to you. The code is C, but the ideas carry over to any language.

Contents

On the way down, on the way up

1 Print an array in reverse
2 Print from both ends
3 Print a series
4 Reverse a line without an array

Trust the smaller answer

5 Factorial
6 Fibonacci
7 Sum of a series
8 Maximum of an array
9 Greatest common divisor
10 String functions
11 Second maximum

Carry it in the parameters

12 Prime test
13 LCM, the long way
14 Reverse an integer
15 Remove odd numbers in place
16 Linear search

Shrink from both sides

17 Binary search
18 Palindrome sentence

More than one call

19 Fibonacci call orders
20 Tower of Hanoi

Originally posted on my old blog, I, ME AND MYSELF !!!, on December 8, 2009, with the answers in separate documents. Rebuilt with the solutions inline, explained, and traced. Read the full article at zobayer.net/posts/dont-fear-recursion.

On the way down, on the way up

Work before the recursive call runs in calling order; work after it runs in reverse.

1 Print an array in reverse

Print the elements of an array, last one first.

INPUT

```
69 87 45 21 47
```

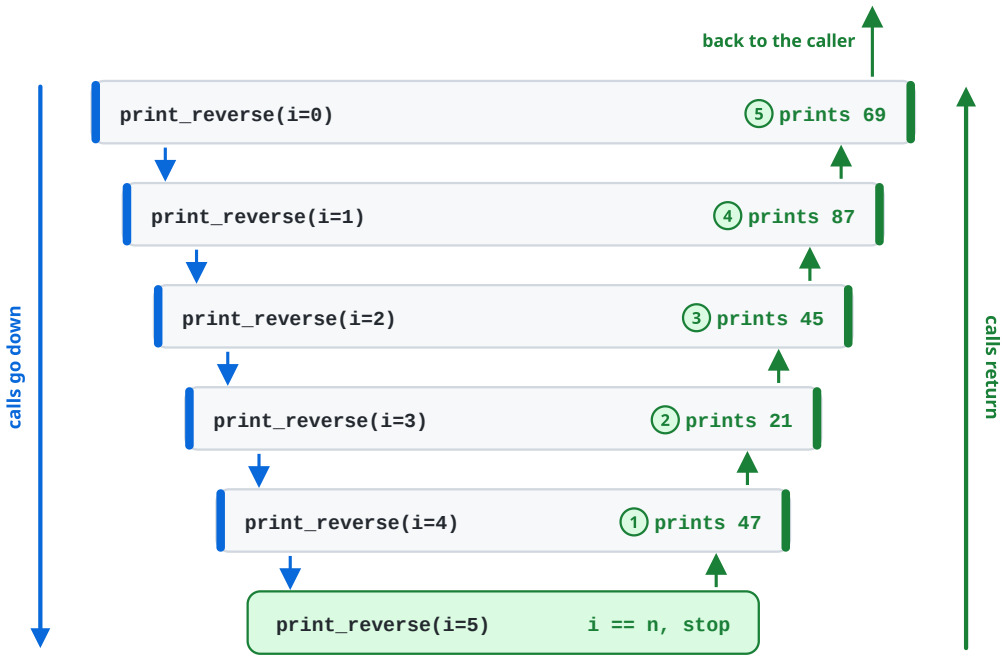
OUTPUT

```
47 21 45 87 69
```

SOLUTION

```c
// call: print_reverse(a, 0, n)
void print_reverse(const int *a, int i, int n) {
    if (i == n) return;
    print_reverse(a, i + 1, n);
    printf("%d ", a[i]);
}
```

How it runs. The `printf` comes after the call, so nothing is printed on the way down. The calls run past the end of the array, and each paused call prints its element on the way back up, the last one first. Move the `printf` above the call and the array prints in order.



output: **47 21 45 87 69**

2 Print from both ends

Print the first and last elements, then the second and second-to-last, and so on, until they meet in the middle.

INPUT

```
1 5 7 8 9
```

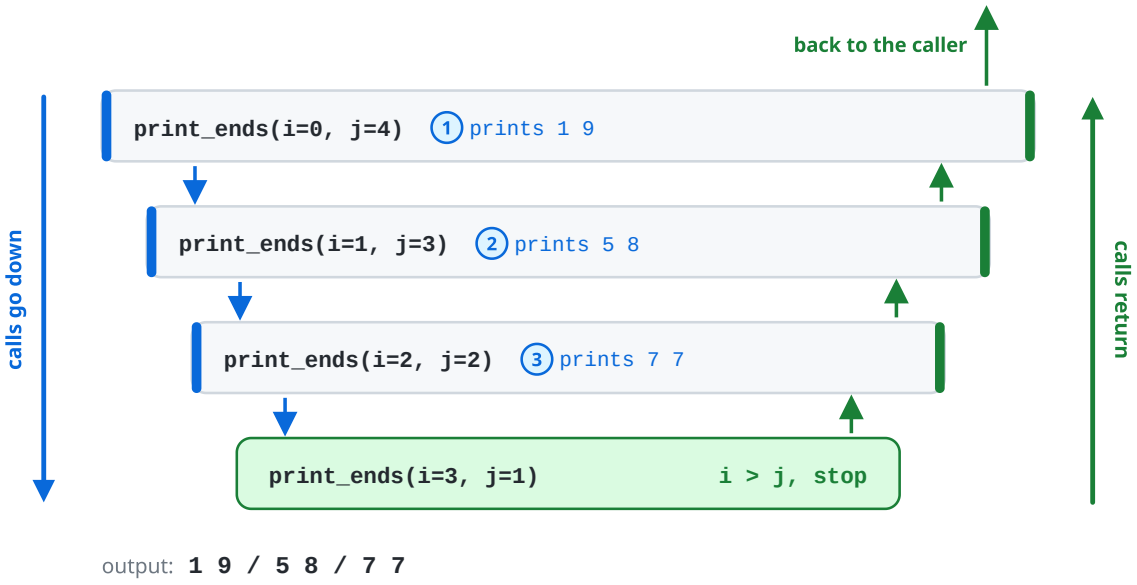
OUTPUT

```
1 9
5 8
7 7
```

SOLUTION

```c
// call: print_ends(a, 0, n - 1)
void print_ends(const int *a, int i, int j) {
    if (i > j) return;
    printf("%d %d\n", a[i], a[j]);
    print_ends(a, i + 1, j - 1);
}
```

How it runs. Each pair is printed before the call, on the way down. Every call moves both ends one step inward, and the calls stop once `i` passes `j`. Swap the last two lines to print from the middle outwards.



output: **1 9 / 5 8 / 7 7**

3 Print a series

Given *n*, print $1 + x + x^2 + \ldots + x^{n-1}$.

INPUT

```
5
```

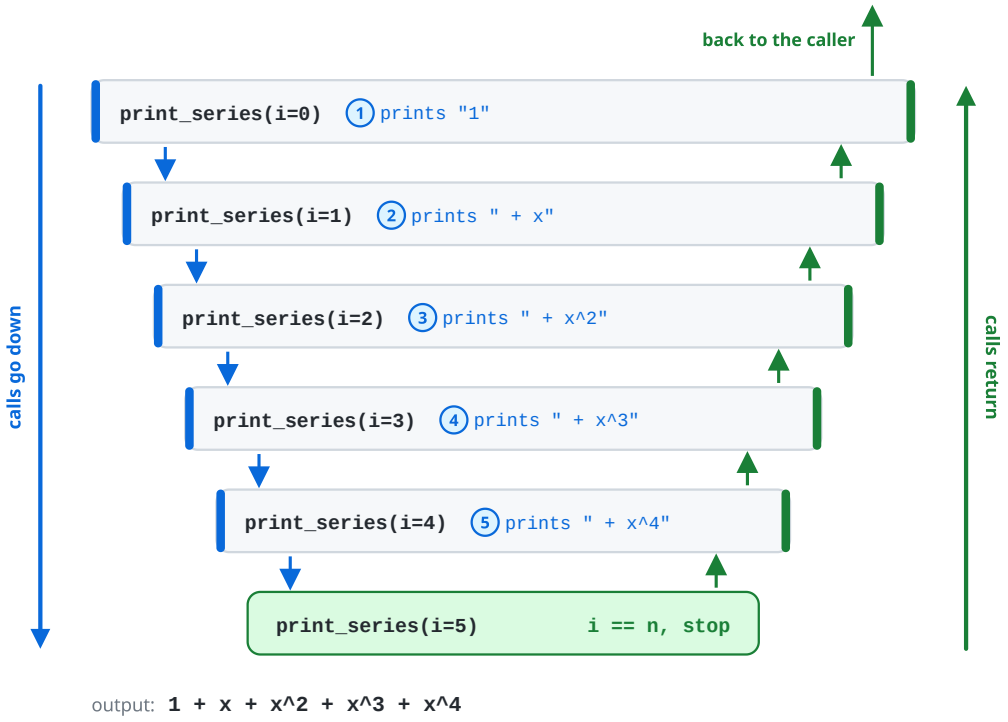
OUTPUT

```
1 + x + x^2 + x^3 + x^4
```

SOLUTION

```c
// call: print_series(0, n)
void print_series(int i, int n) {
    if (i == n) return;
    if (i == 0) printf("1");
    else if (i == 1) printf(" + x");
    else printf(" + x^%d", i);
    print_series(i + 1, n);
}
```

How it runs. Every term is printed before the call, so the series comes out in calling order, and the returns carry nothing back. Try printing it highest power first.



output: `1 + x + x^2 + x^3 + x^4`

4 Reverse a line without an array

Read a line and print it backwards. No arrays and no strings, only `getchar()` and `putchar()`.

INPUT

```
helloo
```

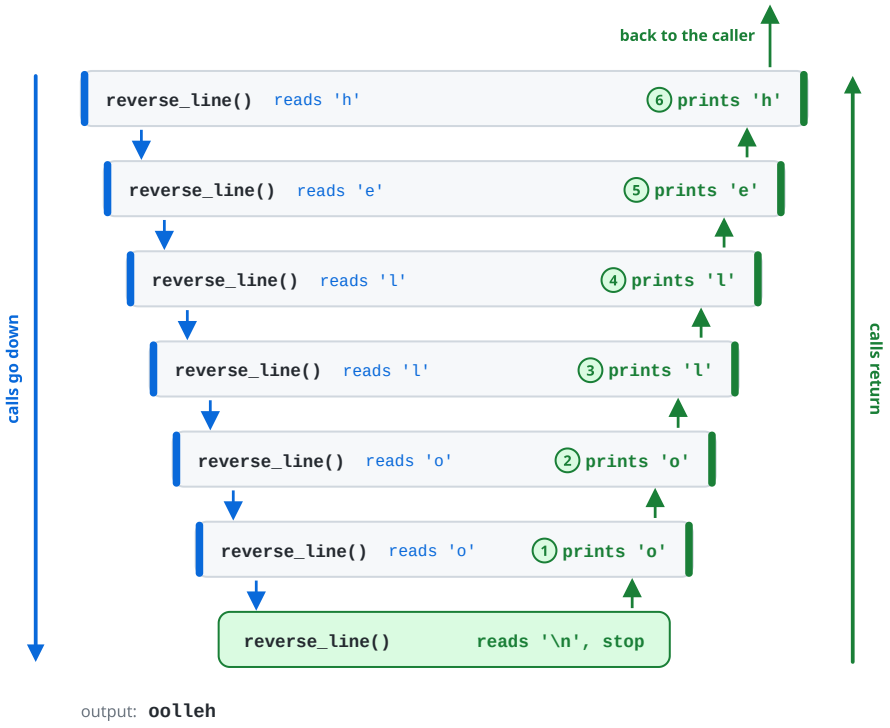
OUTPUT

```
oolleh
```

SOLUTION

```c
void reverse_line(void) {
    int c = getchar();
    if (c == '\n' || c == EOF) return;
    reverse_line();
    putchar(c);
}
```

How it runs. Each call reads one character into its own `c`, then calls itself before printing. Nothing is printed until the newline stops the calls. Then every paused call prints its character on the way back up, last one first: the call stack is the array.



output: **oolleh**

Trust the smaller answer

Promise what the function returns, answer the case you already know, and build everything else from a smaller answer.

5 Factorial

Compute *n*!.

INPUT

```
5
```

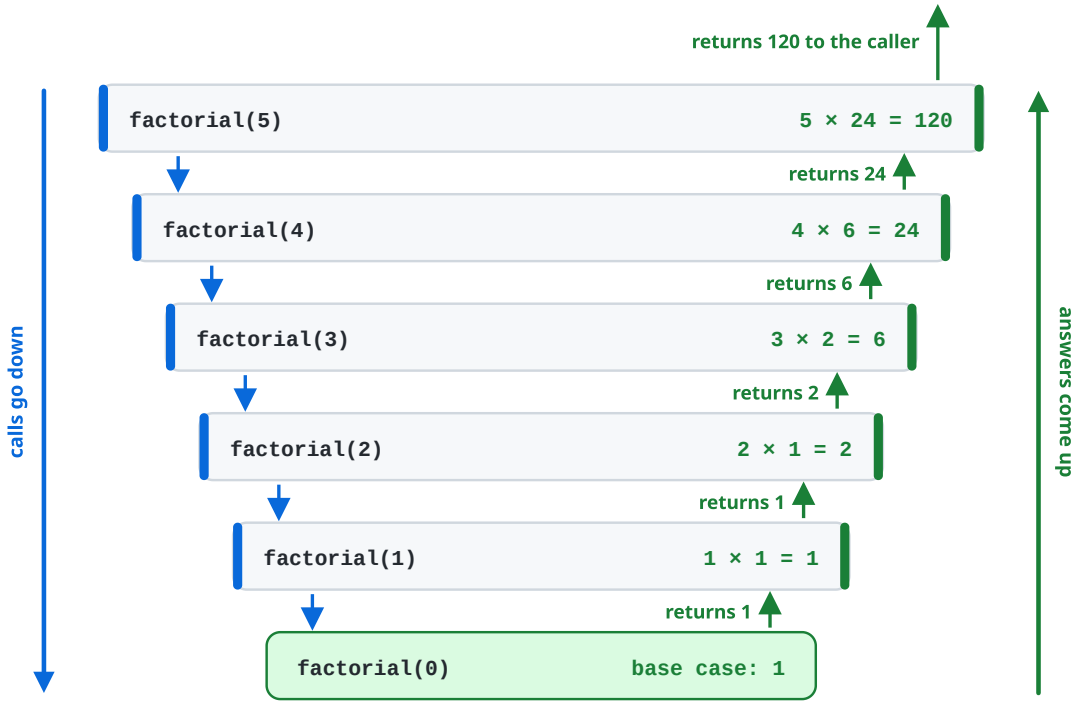
OUTPUT

```
120
```

SOLUTION

```
long long factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

How it runs. `factorial(5)` trusts `factorial(4)` to return 24, then multiplies it by 5. On the way up, every call does exactly one multiplication. `long long` holds up to 20!.



6 Fibonacci

Compute the *n*'th Fibonacci number, where the first two are both 1.

INPUT

6

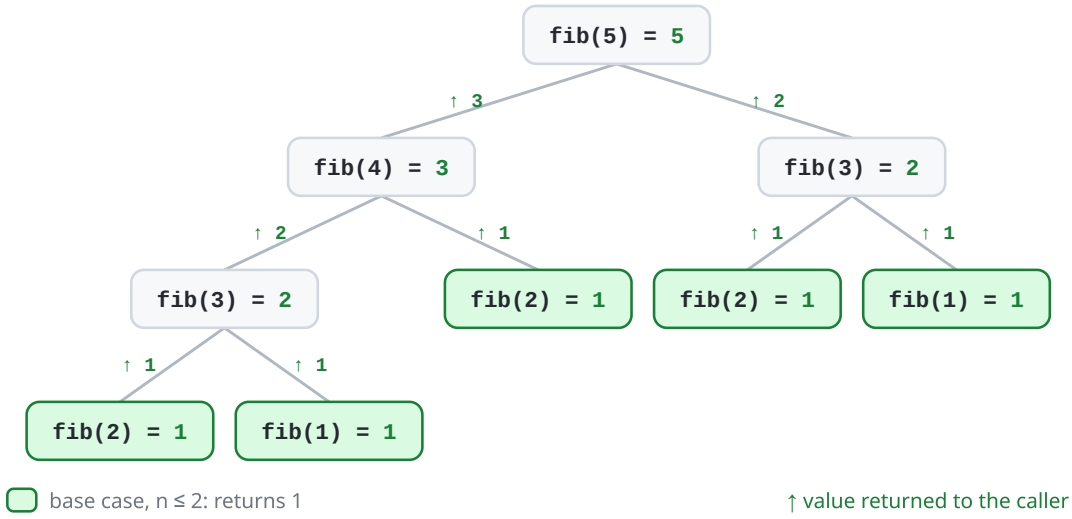
OUTPUT

8

SOLUTION

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

How it runs. Each call adds the answers of its two calls. The call tree for `fib(5)` shows the cost: `fib(3)` is computed twice and `fib(2)` three times, and the waste grows exponentially with *n*. Correct, but slow for large *n*.



7 Sum of a series

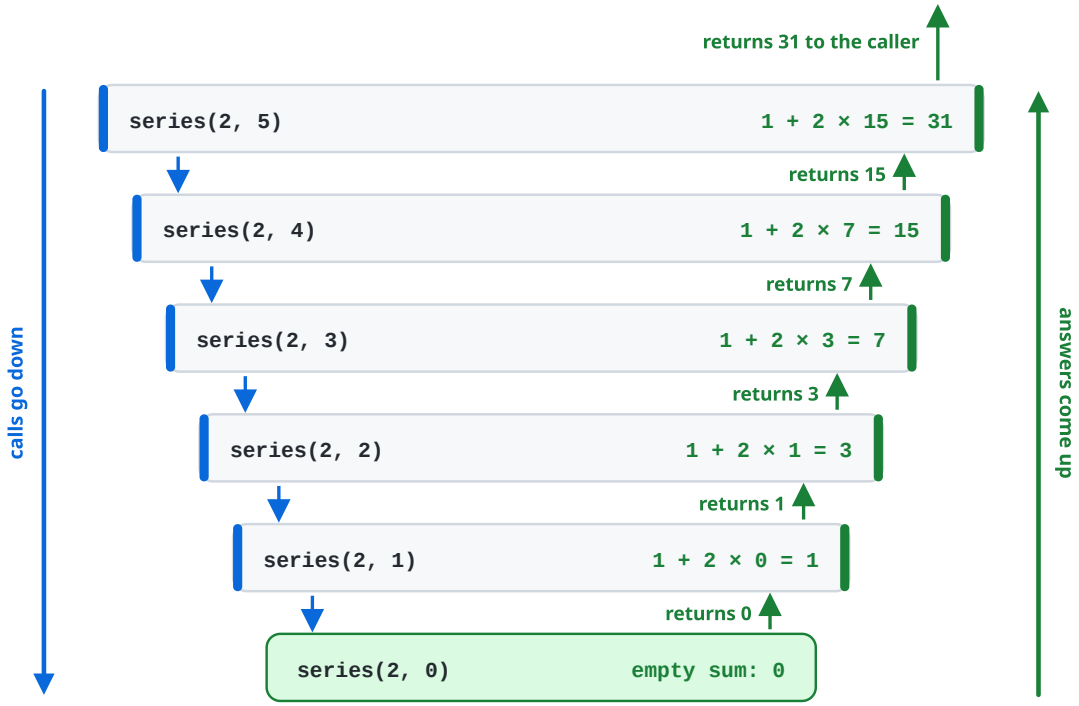
Given x and n, compute $1 + x + x^2 + \ldots + x^{n-1}$.

INPUT	OUTPUT
2 5	31

SOLUTION

```
// promise: series(x, n) returns 1 + x + ... + x^(n-1)
long long series(long long x, int n) {
    if (n == 0) return 0;
    return 1 + x * series(x, n - 1);
}
```

How it runs. Factor out an x: $1 + x(1 + x + \ldots + x^{n-2})$. The bracket is the same problem for $n - 1$, so each call trusts the call below for it, multiplies by x and adds 1. The empty sum, $n = 0$, is the base case. No `pow()` needed.



8 Maximum of an array

Find the largest element of an array.

INPUT

```
7 4 9 6 2
```

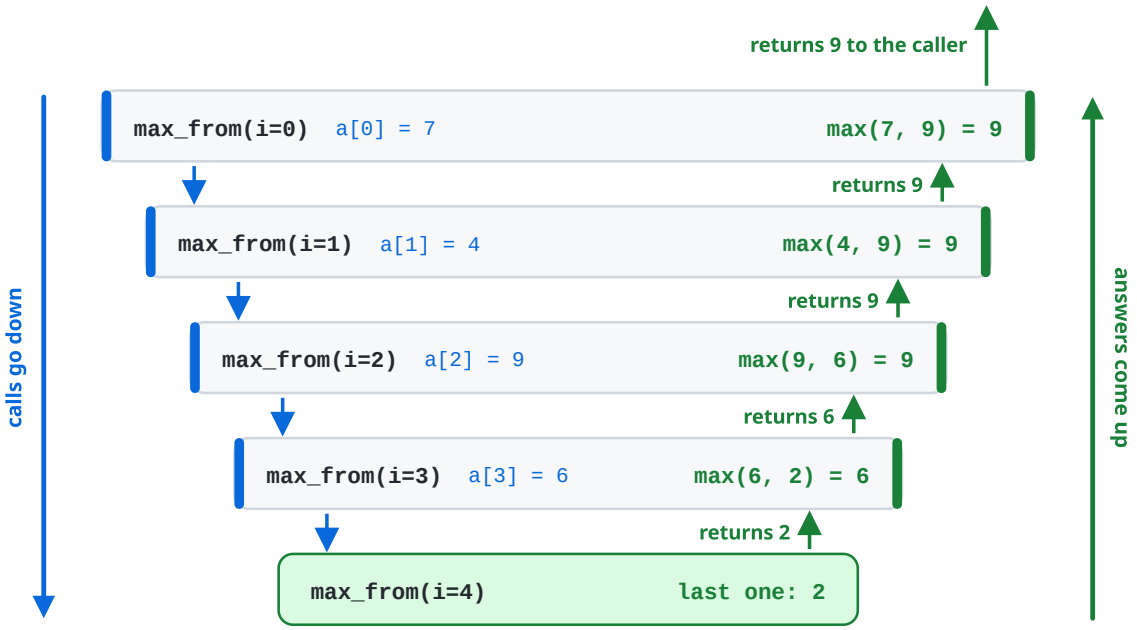
OUTPUT

```
9
```

SOLUTION

```c
// promise: max_from(a, i, n) returns the largest of a[i..n-1]
int max_from(const int *a, int i, int n) {
    if (i == n - 1) return a[i];
    int rest = max_from(a, i + 1, n);
    return a[i] > rest ? a[i] : rest;
}
```

How it runs. Each call compares its own element with the largest of the rest, which the call below returns. Try the minimum next.



9 Greatest common divisor

Find the greatest common divisor of two non-negative integers.

INPUT

```
25 8895
```

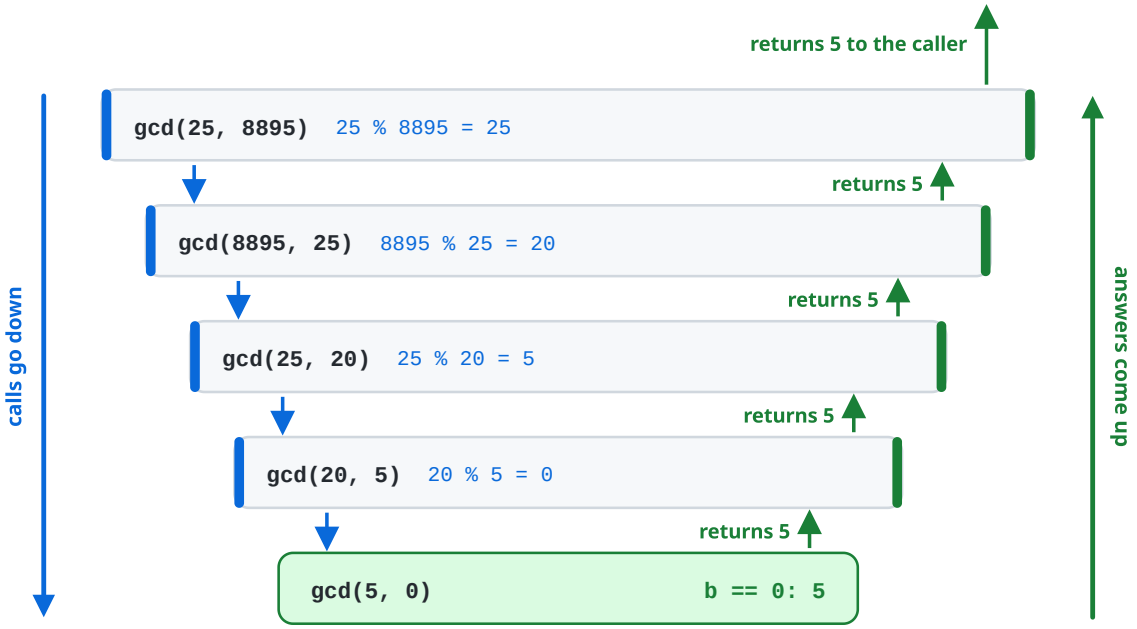
OUTPUT

```
5
```

SOLUTION

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

How it runs. Euclid's algorithm: gcd(a, b) = gcd(b, a mod b). Every call just hands new arguments to the next one, and the answer travels back up unchanged. The depth is only O(log min(a, b)), the safe kind.



10 String functions

Write your own `strlen`, `strcpy`, `strcat` and `strcmp`. Test them on two strings: the length of the first, a copy of it, the second appended to the copy, and a comparison of the two.

INPUT

```
abc
abd
```

OUTPUT

```
length:  3
copy:    abc
append:  abcabd
compare: negative
```

SOLUTION

```c
int my_strlen(const char *s) {
    if (*s == '\0') return 0;
    return 1 + my_strlen(s + 1);
}

void my_strcpy(char *dst, const char *src) {
    *dst = *src;
    if (*src != '\0') my_strcpy(dst + 1, src + 1);
}

void my_strcat(char *dst, const char *src) {
    if (*dst == '\0') my_strcpy(dst, src);
    else my_strcat(dst + 1, src);
}

int my_strcmp(const char *a, const char *b) {
    if (*a != *b || *a == '\0') return (unsigned char)*a - (unsigned char)*b;
    return my_strcmp(a + 1, b + 1);
}
```

How it runs. `my_strlen` counts the first character and trusts the call on the rest of the string, `s + 1`. The other three work the same way: handle the first character, then recurse on the rest. The library `strcpy` and `strcat` also return `dst`, which is left out here.



11 Second maximum

Find the second largest element of an array. Hint: the second maximum of the rest isn't enough on its own. Promise more.

INPUT

```
5 8 7 9 3
```

OUTPUT

```
8
```

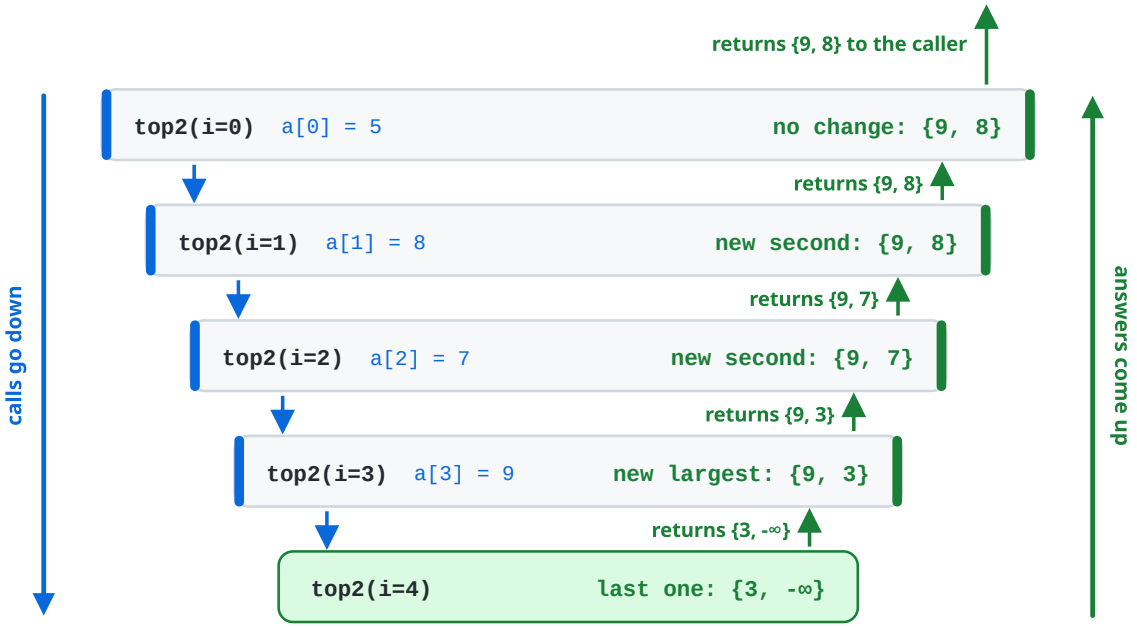
SOLUTION

```c
#include <limits.h>

typedef struct { int first, second; } Top2;

// promise: top2(a, i, n) returns the two largest of a[i..n-1]
Top2 top2(const int *a, int i, int n) {
    if (i == n - 1) return (Top2){ a[i], INT_MIN };
    Top2 t = top2(a, i + 1, n);
    if (a[i] > t.first) {
        t.second = t.first;
        t.first = a[i];
    } else if (a[i] > t.second) {
        t.second = a[i];
    }
    return t;
}
```

How it runs. Each call gets the top two of the rest from the call below and fits its own element in: as the new largest, the new second largest, or not at all. A C function can return a struct, which is how one call hands back two answers without globals. Call it as `top2(a, 0, n).second`.



Carry it in the parameters

Sometimes there's no smaller answer to trust, only progress to make. Then the parameters carry the state of the work, like a loop counter.

12 Prime test

Decide whether each number is prime.

INPUT

```
49
999983
1
```

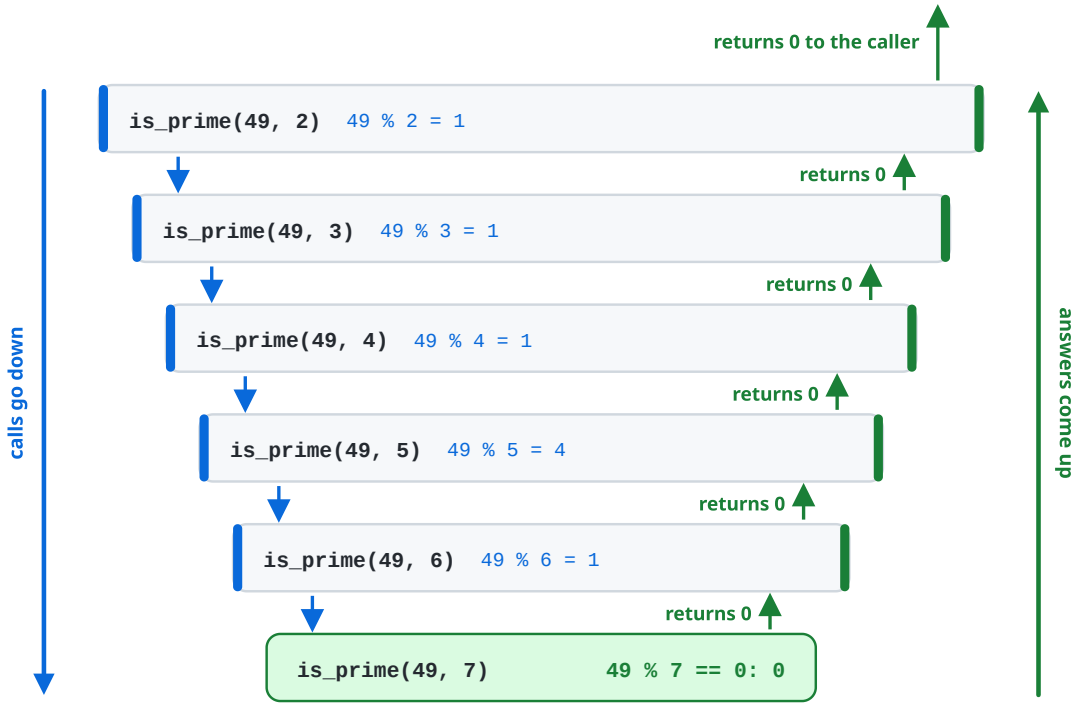
OUTPUT

```
not prime
prime
not prime
```

SOLUTION

```c
// call: is_prime(n, 2)
int is_prime(int n, int d) {
    if (n < 2) return 0;
    if ((long long)d * d > n) return 1;    // no divisor up to sqrt(n)
    if (n % d == 0) return 0;
    return is_prime(n, d + 1);
}
```

How it runs. `d` is the loop counter, passed along instead of incremented: each call tests one divisor and passes `d + 1` on. For 49, the calls stop at 7, which divides it, and 0 is passed back up.



13 LCM, the long way

Find the least common multiple of two numbers, without using lcm(a, b) = a × b / gcd(a, b).

INPUT

```
23 488
```

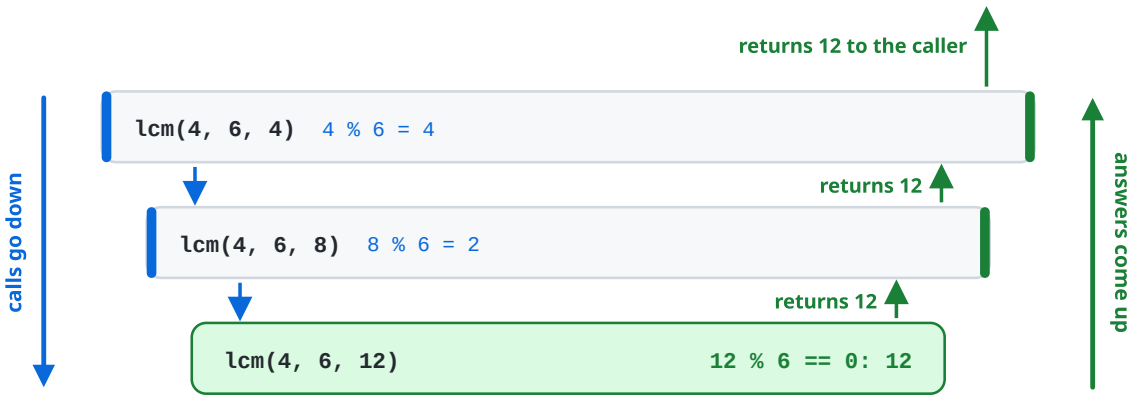
OUTPUT

```
11224
```

SOLUTION

```
// call: lcm(a, b, a); m walks through the multiples of a
long long lcm(long long a, long long b, long long m) {
    if (m % b == 0) return m;
    return lcm(a, b, m + a);
}
```

How it runs. `m` walks through the multiples of `a`. The first one divisible by `b` is the answer, and it is passed back up unchanged. Shown for 4 and 6; 23 and 488 would take 488 calls. That's a loop in disguise: compare it with problem 9.



14 Reverse an integer

Reverse the digits of a number, returning an `int`.

INPUT

```
123405
```

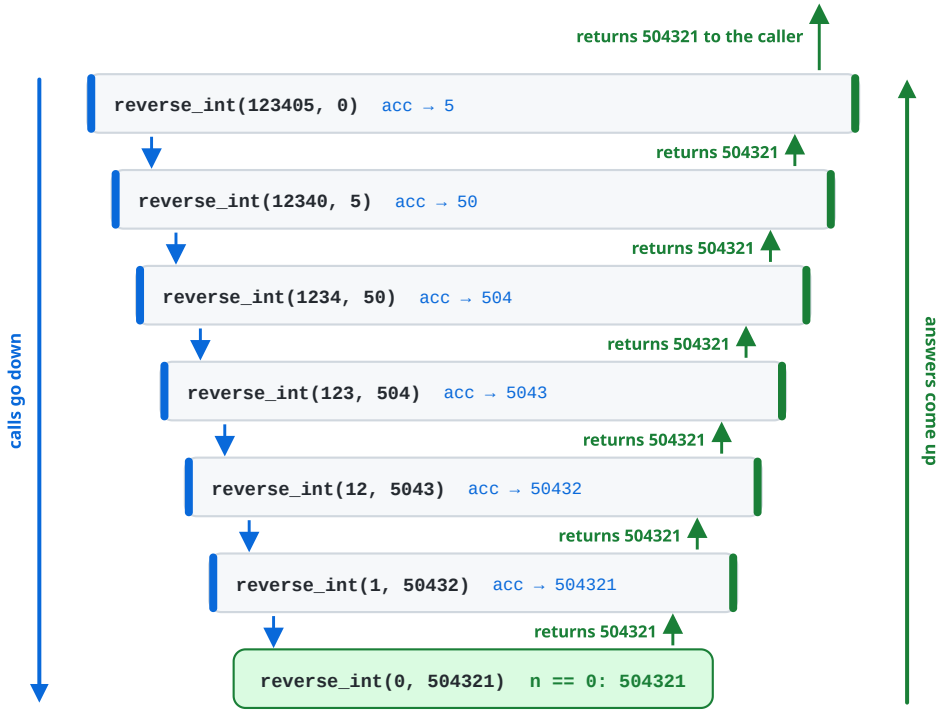
OUTPUT

```
504321
```

SOLUTION

```c
// call: reverse_int(n, 0); acc holds the digits reversed so far
int reverse_int(int n, int acc) {
    if (n == 0) return acc;
    return reverse_int(n / 10, acc * 10 + n % 10);
}
```

How it runs. The answer is built on the way down: each call moves the last digit of `n` onto the accumulator `acc`. When `n` runs out, `acc` holds the answer, which is passed back up unchanged.



15 Remove odd numbers in place

Remove the odd numbers from an array, with no second array and no printing inside the function. Print the result afterwards.

INPUT

```
1 54 88 6 55 7
```

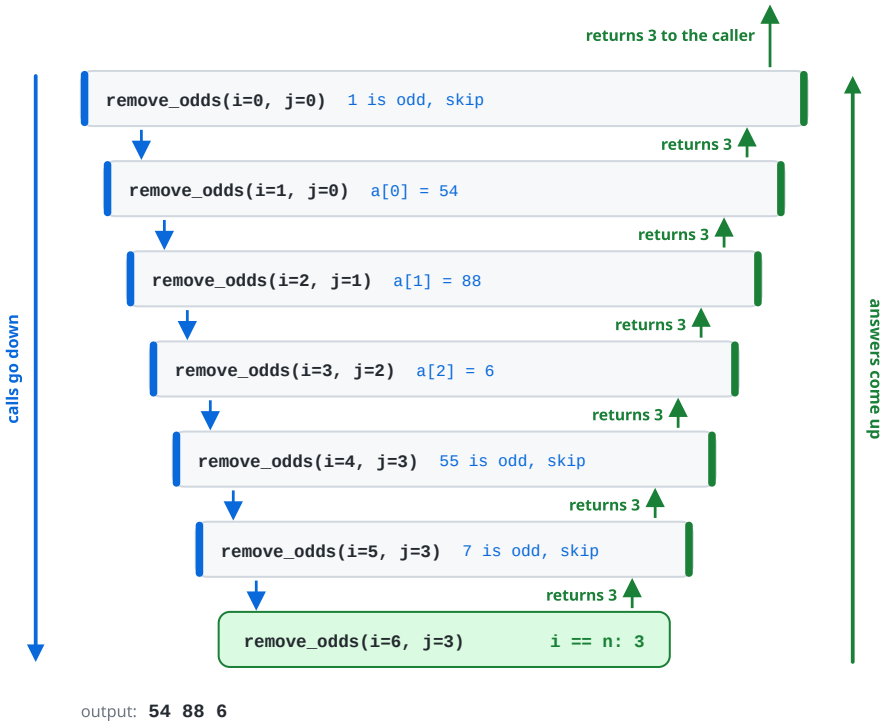
OUTPUT

```
54 88 6
```

SOLUTION

```c
// call: n = remove_odds(a, 0, 0, n); returns the new length
int remove_odds(int *a, int i, int j, int n) {
    if (i == n) return j;
    if (a[i] % 2 == 0) a[j++] = a[i];
    return remove_odds(a, i + 1, j, n);
}
```

How it runs. `i` reads every element, while `j` only moves when an even number is copied. `j` never passes `i`, so nothing is overwritten before it is read. When `i` reaches the end, `j` is the new length.



output: **54 88 6**

16 Linear search

The first line is an array, the second the values to look for. For each value, print its index, or -1 if it isn't there.

INPUT

```
2 9 4 7 6
5 9
```

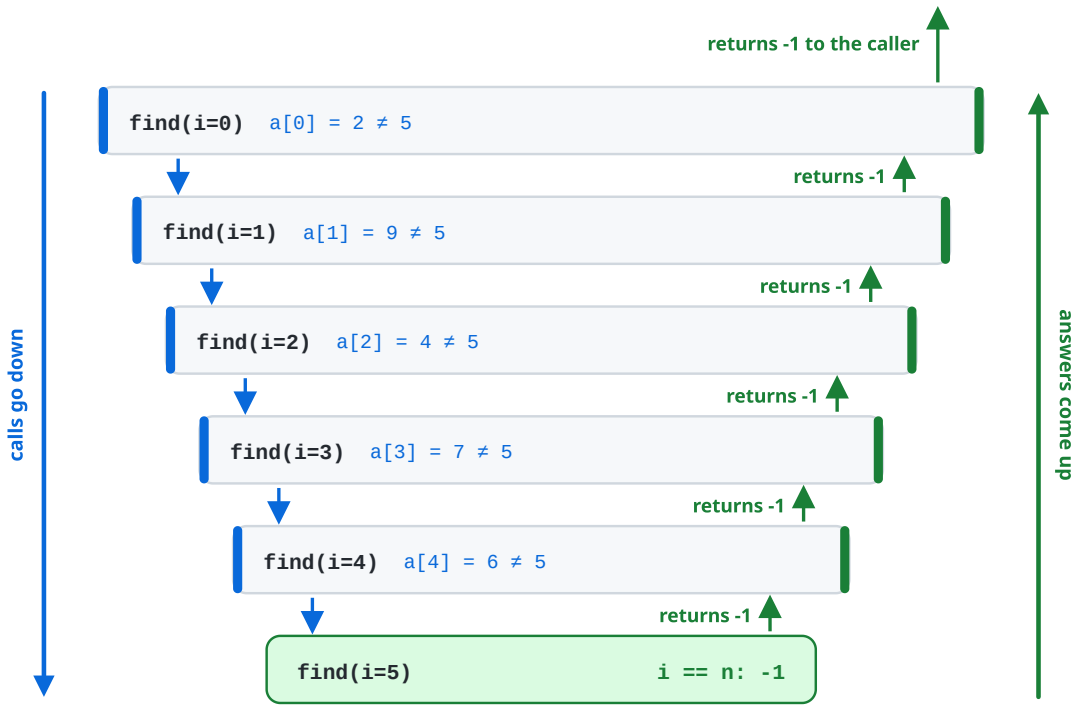
OUTPUT

```
-1
1
```

SOLUTION

```c
// call: find(a, 0, n, v)
int find(const int *a, int i, int n, int v) {
    if (i == n) return -1;
    if (a[i] == v) return i;
    return find(a, i + 1, n, v);
}
```

How it runs. Each call checks one element and moves on. Looking for 5, nothing matches, the calls run off the end of the array, and -1 comes back up. Try searching from the end next.



Shrink from both sides

17 Binary search

Same as problem 16, but the array is sorted.

INPUT

```
1 2 3 4 5
3 -5
```

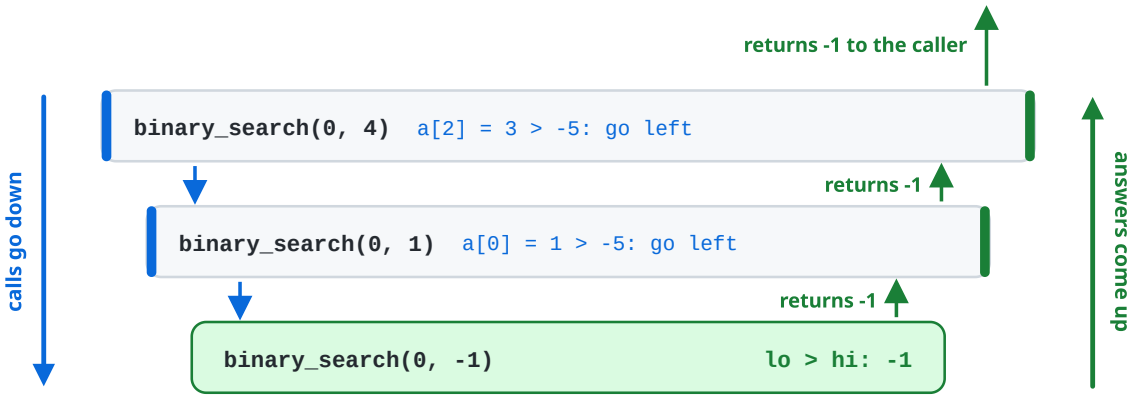
OUTPUT

```
2
-1
```

SOLUTION

```c
// call: binary_search(a, 0, n - 1, v)
int binary_search(const int *a, int lo, int hi, int v) {
    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (a[mid] < v) return binary_search(a, mid + 1, hi, v);
    if (a[mid] > v) return binary_search(a, lo, mid - 1, v);
    return mid;
}
```

How it runs. Each call checks the middle of its range and drops the half that can't contain `v`, so the depth is O(log n). Looking for -5, the range is empty after two calls. `lo + (hi - lo) / 2` avoids the overflow that `(lo + hi) / 2` can hit on huge arrays.



18 Palindrome sentence

Decide whether each line is a palindrome, counting only letters and digits.

INPUT

```
madam, i'm adam
hulala
```

OUTPUT

```
palindrome
not palindrome
```

SOLUTION

```c
#include <ctype.h>

// call: is_palindrome(s, 0, (int)strlen(s) - 1)
int is_palindrome(const char *s, int i, int j) {
    if (i >= j) return 1;
    if (!isalnum((unsigned char)s[i])) return is_palindrome(s, i + 1, j);
    if (!isalnum((unsigned char)s[j])) return is_palindrome(s, i, j - 1);
    if (s[i] != s[j]) return 0;
    return is_palindrome(s, i + 1, j - 1);
}
```

How it runs. Each call compares the outer pair and moves inward, skipping anything that isn't a letter or a digit. Shown for the shorter "pop, pop". Try ignoring case next, so "Madam, I'm Adam" passes too.



More than one call

19 Fibonacci call orders

`fib(n)` calls `fib(n - 1)`, then `fib(n - 2)`, and stops at $n \le 2$. Print the n of every call in *preorder* (before its calls), *inorder* (between them) and *postorder* (after them).

INPUT

```
5
```

OUTPUT

```
Preorder:  5 4 3 2 1 2 3 2 1
Inorder:   2 3 1 4 2 5 2 3 1
Postorder: 2 1 3 2 4 2 1 3 5
```

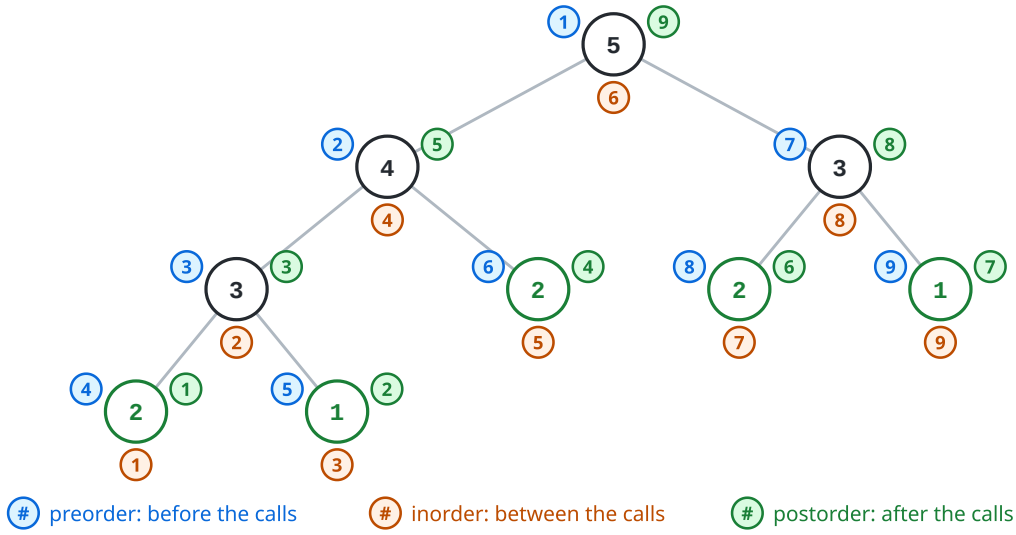
SOLUTION

```c
void preorder(int n) {
    if (n <= 2) { printf(" %d", n); return; }
    printf(" %d", n);
    preorder(n - 1);
    preorder(n - 2);
}

void inorder(int n) {
    if (n <= 2) { printf(" %d", n); return; }
    inorder(n - 1);
    printf(" %d", n);
    inorder(n - 2);
}

void postorder(int n) {
    if (n <= 2) { printf(" %d", n); return; }
    postorder(n - 1);
    postorder(n - 2);
    printf(" %d", n);
}
```

How it runs. Every call prints its n exactly once; only the moment changes, before, between or after the calls. That's the same rule as problems 1 to 4. Follow one colour's numbers in the tree and you read off that order.



20 Tower of Hanoi

Three pegs `a`, `b` and `c`, and *n* disks on `a`, largest at the bottom. Move them all to `c`, one disk at a time, never placing a larger disk on a smaller one. Print each move, such as `a -> c`.

INPUT

```
3
```

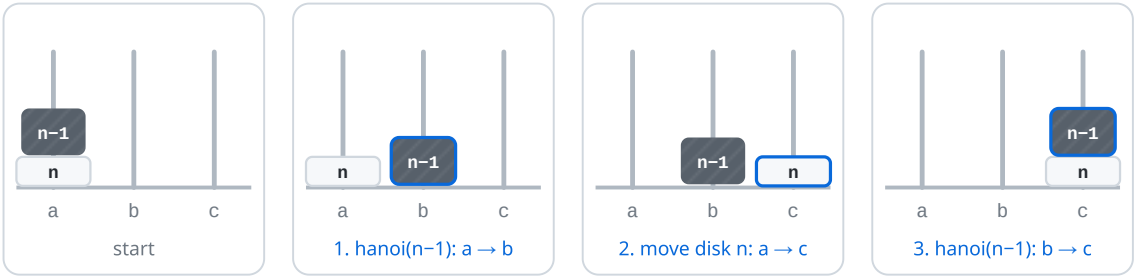
OUTPUT

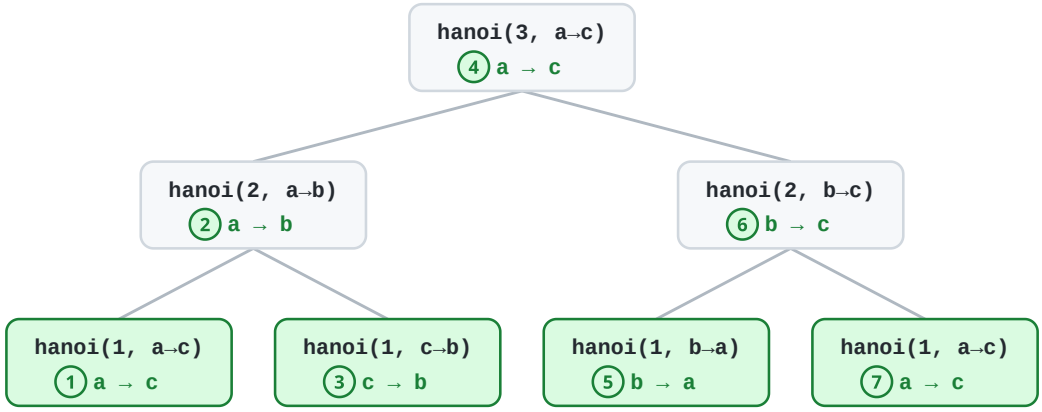
```
a -> c
a -> b
c -> b
a -> c
b -> a
b -> c
a -> c
```

SOLUTION

```c
// call: hanoi(n, 'a', 'b', 'c')
void hanoi(int n, char from, char via, char to) {
    if (n == 0) return;
    hanoi(n - 1, from, to, via);
    printf("%c -> %c\n", from, to);
    hanoi(n - 1, via, from, to);
}
```

How it runs. Trust `hanoi(n - 1)` to move a stack of $n - 1$ disks between any two pegs. Then n disks take three steps: move the top $n - 1$ out of the way, move the largest disk, and move the $n - 1$ back on top of it. Each call prints its move between its two calls, so the moves come out in inorder, just like problem 19. That's $2^n - 1$ moves; tracing them by hand is a mess, trusting the function takes three lines.





Badges number the moves in the order they are printed. Calls with n = 0 are not drawn.