

Quiz 1: code before and after a recursive call

```
void trace(int n)
{
    if (n == 0) return;
    printf("enter %d\n", n);
    trace(n - 1);
    printf("leave %d\n", n);
}
```

Write the exact output of trace(3).

Does leave 3 print before or after leave 1?

Quiz 1: code before and after a recursive call

```
void trace(int n)
{
    if (n == 0) return;
    printf("enter %d\n", n);
    trace(n - 1);
    printf("leave %d\n", n);
}
```

Write the exact output of trace(3).

Does leave 3 print before or after leave 1?

Answer

enter 3, enter 2, enter 1, leave 1, leave 2, leave 3, each on a new line.

Why the second print happens on the return path

Phase	What executes
Going down	Each non-base call prints enter n , then calls <code>trace(n - 1)</code>
Base case	<code>trace(0)</code> returns immediately and prints nothing
Coming back	Each waiting call resumes at <code>printf("leave ...")</code>

A return finishes only the current call

After `trace(0)` returns, `trace(1)` still has work to do.

After `trace(1)` returns, `trace(2)` still has work to do.

Statements after a recursive call execute when that call returns.

Quiz 2: a base case must be reachable

```
int fact(int n)
{
    if (n == 1) return 1;
    return n * fact(n - 1);
}
```

For `fact(0)`, write the first four argument values.

What goes wrong, and how would you fix it for nonnegative inputs?

Quiz 2: a base case must be reachable

```
int fact(int n)
{
    if (n == 1) return 1;
    return n * fact(n - 1);
}
```

For `fact(0)`, write the first four argument values.

What goes wrong, and how would you fix it for nonnegative inputs?

Answer

Arguments: 0, -1, -2, -3. The call never reaches `n == 1`. Use `n == 0` as the base case and require `n >= 0`. Continued calls typically exhaust the stack.

Binomial coefficients: choosing a group

$\binom{n}{k}$ counts the ways to choose k items from n distinct items.

Choose two letters from $\{A, B, C, D\}$:

$$AB, AC, AD, BC, BD, CD \Rightarrow \binom{4}{2} = 6.$$

- Order within the chosen group does not matter: AB and BA are the same choice.
- The factorial formula is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
- Pascal's triangle gives a recurrence that avoids computing factorials.

Pascal's triangle: rows and positions

n = 0					1										
n = 1					1		1								
n = 2					1		2		1						
n = 3					1		3		3		1				
n = 4					1		4		6		4		1		
n = 5					1		5		10		10		5		1

- Row numbers begin at $n = 0$.
- Positions in row n run from $k = 0$ to $k = n$.
- Entry k is $\binom{n}{k}$.
- Every row starts and ends with 1.

Every interior entry is the sum of its two parents in the previous row.
Example: the highlighted 6 is $3 + 3$.

Why Pascal's identity works

Choose a group of k students from n students. Focus on one student, A .

Groups containing A

Groups without A

Choose the other $k-1$ students from the remaining $n-1$

$\binom{n-1}{k-1}$ groups

Choose all k students from the remaining $n-1$

$\binom{n-1}{k}$ groups

$$\boxed{\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}} \quad (0 < k < n).$$

The two cases cover every group, with no overlap.

At the edges, $\binom{n}{0} = \binom{n}{n} = 1$.

A recursive function for a binomial coefficient

```
int binom(int n, int k)
{
    if (k == 0 || k == n) return 1;
    int left = binom(n - 1, k - 1);
    int right = binom(n - 1, k);
    return left + right;
}
```

Inputs: $0 \leq k \leq n$, with small n .

- The two edges are base cases.
- Both calls move to row $n - 1$.
- Add the two returned answers.

`binom(4, 2)` returns 6.

Two recursive calls: one completes before the next

For `binom(3, 1)`, follow the two assignment statements in order.

Step	Work performed
------	----------------

- | | |
|---|--|
| 1 | <code>binom(2, 0)</code> reaches a base case and returns 1 |
| 2 | The call <code>binom(3, 1)</code> stores <code>left = 1</code> |
| 3 | <code>binom(2, 1)</code> calls <code>binom(1, 0)</code> and <code>binom(1, 1)</code> |
| 4 | These return 1 each, so <code>binom(2, 1)</code> returns 2 |
| 5 | The original call stores <code>right = 2</code> , then returns $1 + 2 = 3$ |
-

The first recursive call finishes completely before the second starts.

Printing Pascal's triangle

```
int main(void)
{
    int last = 5;
    for (int n = 0; n <= last; n++) {
        for (int k = 0; k <= n; k++)
            printf("%4d", binom(n, k));
        printf("\n");
    }
    return 0;
}
```

Include `<stdio.h>` and define `binom` before `main`.

Output

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10  5  1
```

`n` selects the row.

`k` selects the entry within it.

`%4d` uses a field of width 4.

Quiz 3: rows, values, and loop bounds

1. Write row $n = 6$ of Pascal's triangle.
2. What is `binom(5, 2)`? Name the two values added to obtain it.
3. The outer loop runs from $n = 0$ to $n \leq 5$. How many rows print?

Quiz 3: rows, values, and loop bounds

1. Write row $n = 6$ of Pascal's triangle.
2. What is `binom(5, 2)`? Name the two values added to obtain it.
3. The outer loop runs from $n = 0$ to $n \leq 5$. How many rows print?

Answer

1. 1 6 15 20 15 6 1
2. $10 = \binom{4}{1} + \binom{4}{2} = 4 + 6$
3. Six rows, numbered 0 through 5.

Pascal's triangle also gives expansion coefficients

The entries in row n are the coefficients of $(1 + x)^n$.

$$(1 + x)^4 = 1 + 4x + 6x^2 + 4x^3 + 1x^4.$$

row 4 : 1 4 6 4 1

Why is the coefficient of x^2 equal to 6?

In $(1 + x)(1 + x)(1 + x)(1 + x)$, choose x from two factors and 1 from the others.
There are $\binom{4}{2} = 6$ ways to make that choice.

Virahanka numbers: counting brick towers

We have bricks of heights 1 and 2. Let V_n count towers of total height n .

Write a tower as its brick heights, from bottom to top.

Height	Possible towers	Count
1	(1)	$V_1 = 1$
2	(1, 1), (2)	$V_2 = 2$
3	(1, 1, 1), (1, 2), (2, 1)	$V_3 = 3$

Order matters: (1, 2) and (2, 1) are different towers.

Virahanka recurrence: look at the bottom brick

For a tower of height $n \geq 2$, there are two possible bottom bricks.

Bottom brick has height 1

Bottom brick has height 2

Height still to fill: $n - 1$

Height still to fill: $n - 2$

Number of completions: V_{n-1}

Number of completions: V_{n-2}

$$V_n = V_{n-1} + V_{n-2}.$$

Every tower belongs to exactly one case. Add the counts.

Example: $V_4 = V_3 + V_2 = 3 + 2 = 5$.

Virahanka base cases and Fibonacci numbers

- $V_0 = 1$: one way to fill height zero, using no bricks.
- $V_1 = 1$: one tower, consisting of one height-1 brick.

n	0	1	2	3	4	5	6
V_n	1	1	2	3	5	8	13
F_n	0	1	1	2	3	5	8

With the usual Fibonacci convention $F_0 = 0$, $F_1 = 1$, we have

$$V_n = F_{n+1}.$$

The recurrence is the same, but the starting values differ.

Virahanka numbers in C

```
int virahanka(int n)
{
    if (n == 0 || n == 1) return 1;
    int a = virahanka(n - 1);
    int b = virahanka(n - 2);
    return a + b;
}
```

Example: virahanka(3)

$$V_3 = V_2 + V_1 = (V_1 + V_0) + V_1 = 3.$$

- Each call reduces n.
- The base cases handle 0 and 1.
- The two calls count the two possible bottom bricks.

Quiz 4: count and classify the towers

1. List all towers of height 4, from bottom to top.
2. How many begin with a height-1 brick? With a height-2 brick?
3. What would go wrong if we used $V_0 = 0$?

Quiz 4: count and classify the towers

1. List all towers of height 4, from bottom to top.
2. How many begin with a height-1 brick? With a height-2 brick?
3. What would go wrong if we used $V_0 = 0$?

Answer

$(1, 1, 1, 1)$, $(1, 1, 2)$, $(1, 2, 1)$, $(2, 1, 1)$, $(2, 2)$.

Three begin with 1; two begin with 2.

With $V_0 = 0$, the recurrence gives $V_2 = 1$ and fails to count the single height-2 brick.