GNG1106
Fundamentals of Engineering Computation

Instructor: **Hitham Jleed**

University of Ottawa

Fall 2023 ~

In-Class Exercise:

Recursion

- In C, a function is allowed to call itself. This self-calling behaviour of a function is called recursion or recursive function call.
- A computation problem can be solved using recursion if the problem can be reduced to a smaller problem or decomposed into several smaller problems of the same kind.

Example

Consider the problem of computing

$$S(n) = 1 + 2 + \ldots + n.$$

Since

$$S(n) = S(n-1) + n,$$

computing $S(n)$ can be solved by solving the smaller problem of computing $S(n-1)$ and then adding n to the answer. This problem can then be solved using recursion (see code).

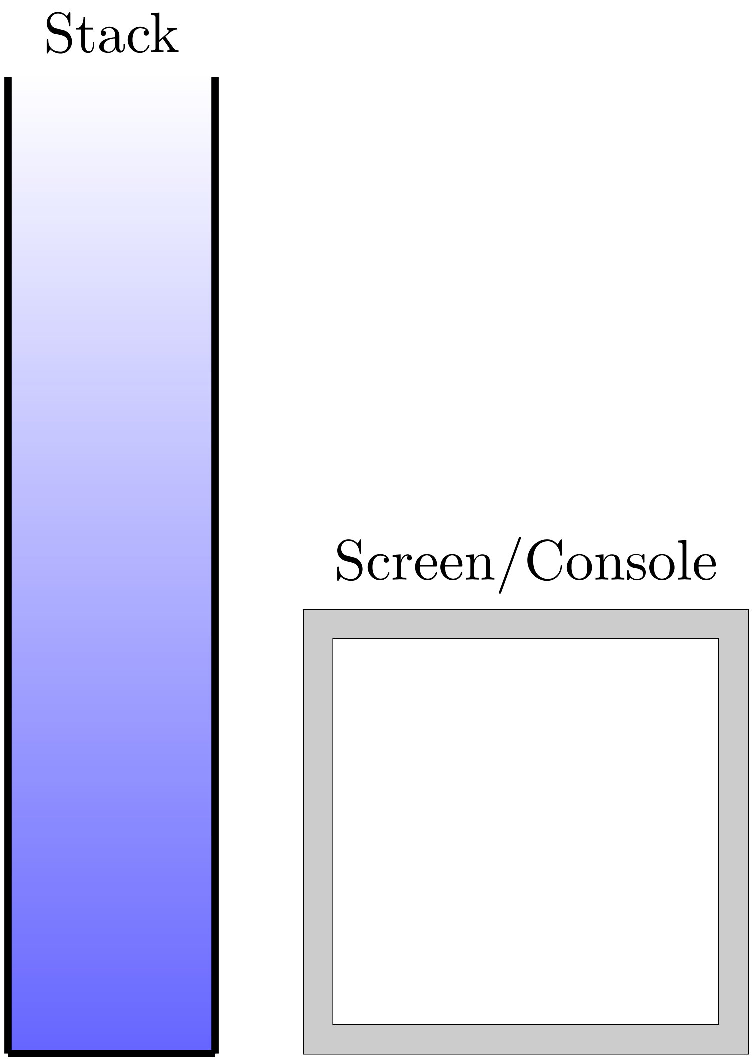
A recursive function always involves two branches:

- “Exit branch”: If the input value corresponds to the smallest problem, or the “base problem”, the solution to the problem is directly computed.
- “Recursion branch”: if the input value does not corresponds to the base problem, decompose the problem into smaller problems and express its solution in terms of the solution(s) of the smaller problem(s).

Recursion in Programming Model

Code Segment

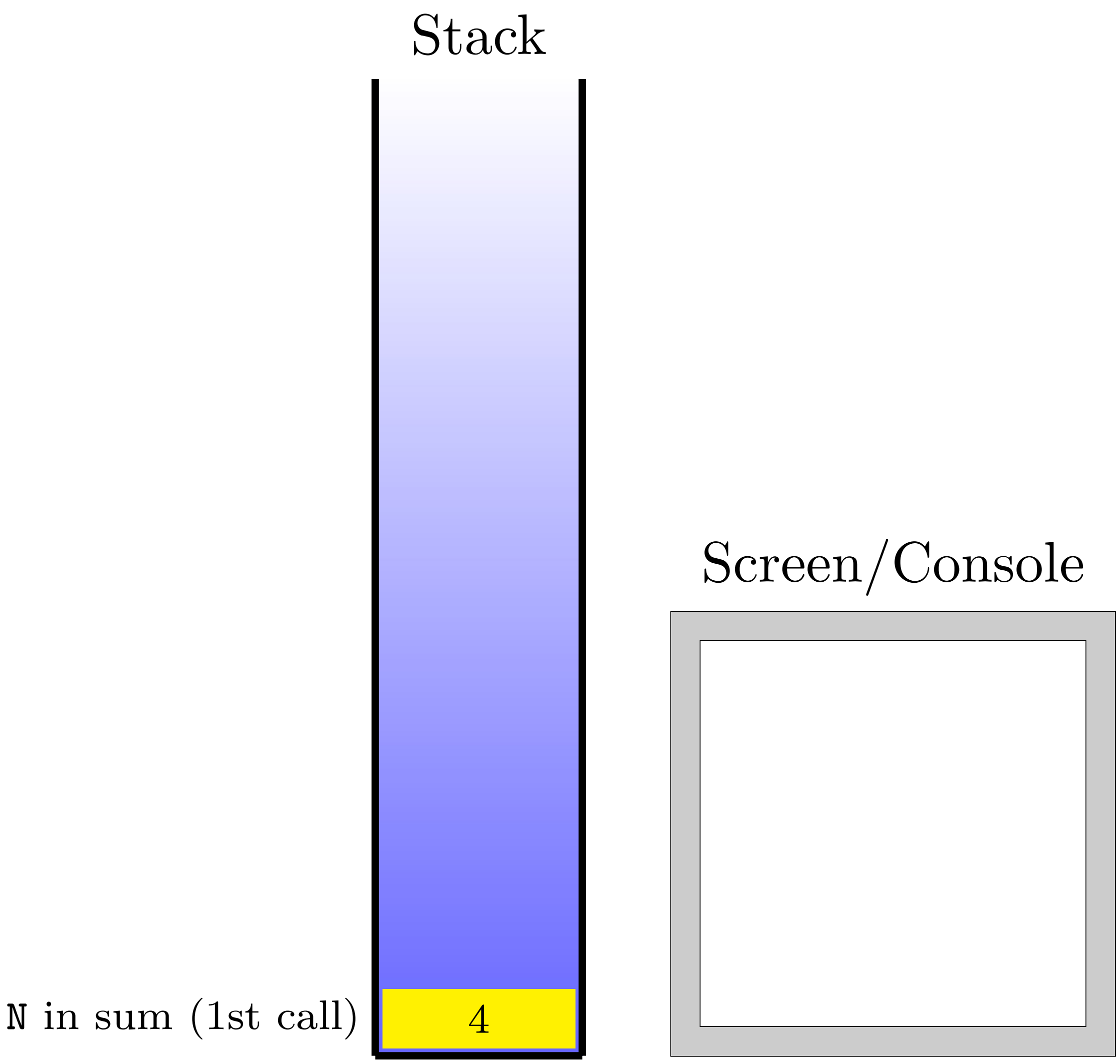
```
#include <stdio.h>
int sum(int N)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
    sum(4));
 return 0;
}
```



Step 1

Code Segment

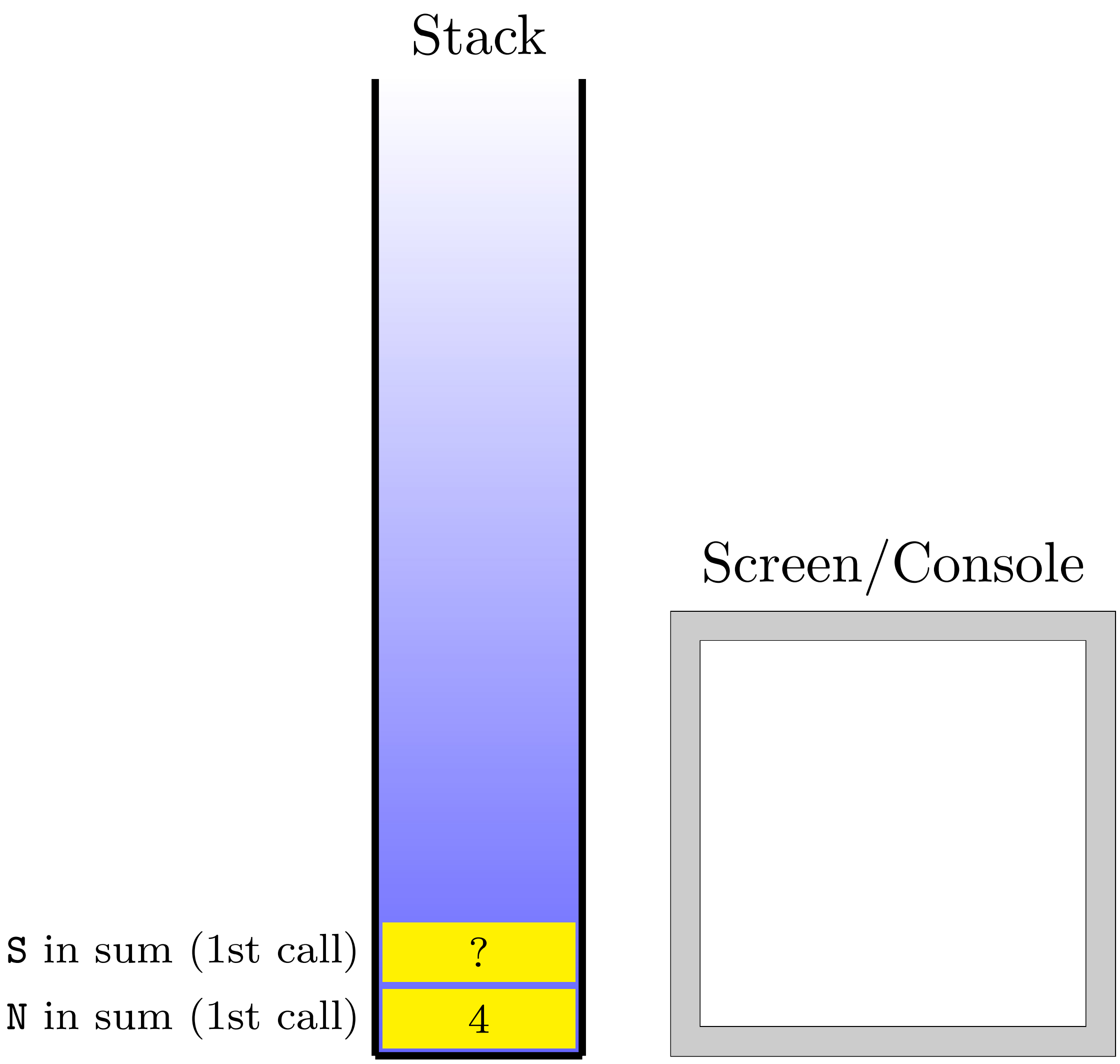
```
#include <stdio.h>
int sum(int N)(1)
{
  int S;
  if (N>1)
    S = sum(N-1)+N;
  else
    S=1;
  return S;
}

int main()
{
  printf("The sum is %d\n",
      sum(4));
  return 0;
}
```



Step 2

Code Segment

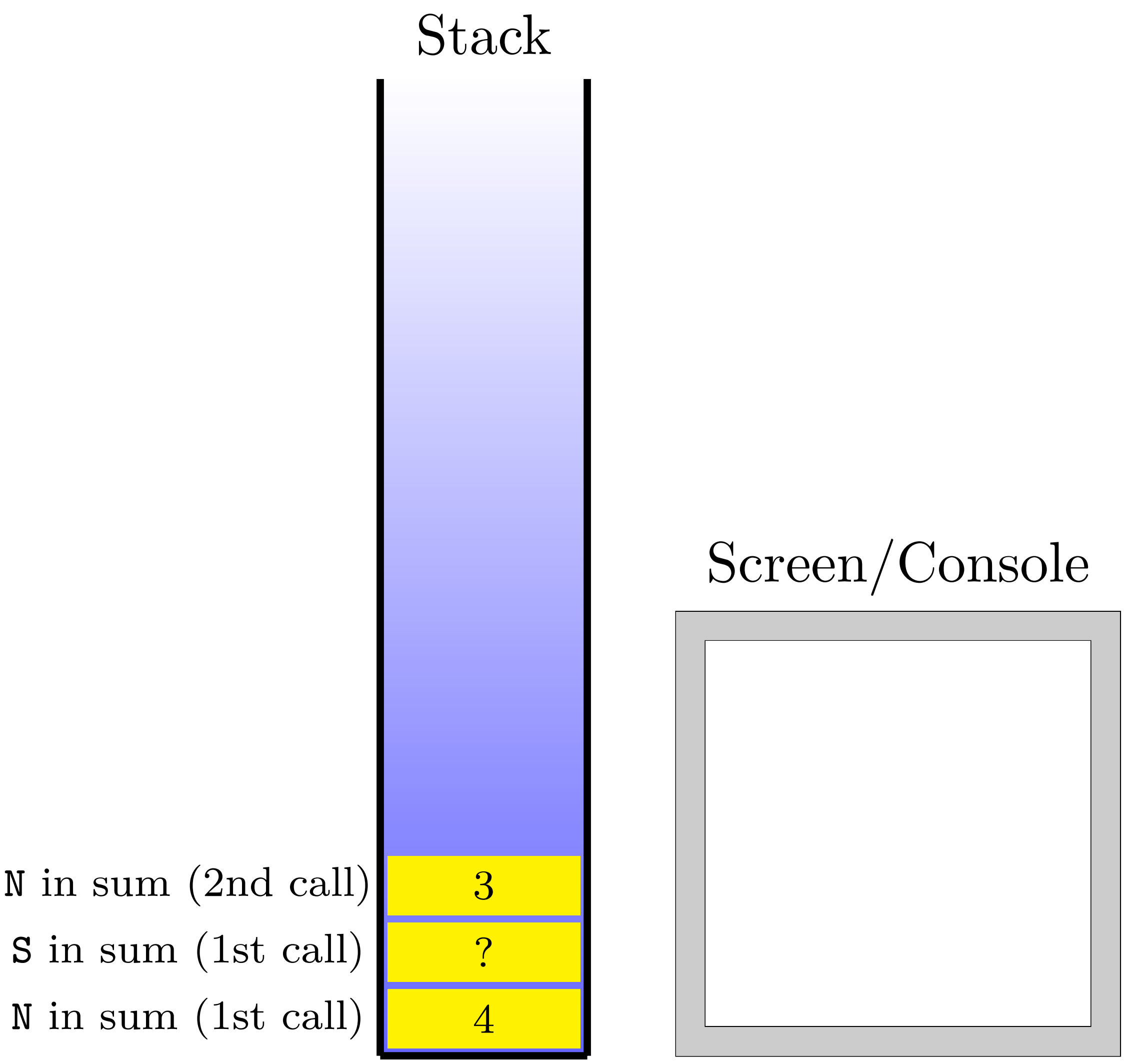
```
#include <stdio.h>
int sum(int N)(1)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
     sum(4));
 return 0;
}
```



Step 3

Code Segment

```
#include <stdio.h>
int sum(int N)(1)(2)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
      sum(4));
 return 0;
}
```



Stack

N in sum (2nd call) 3

S in sum (1st call) ?

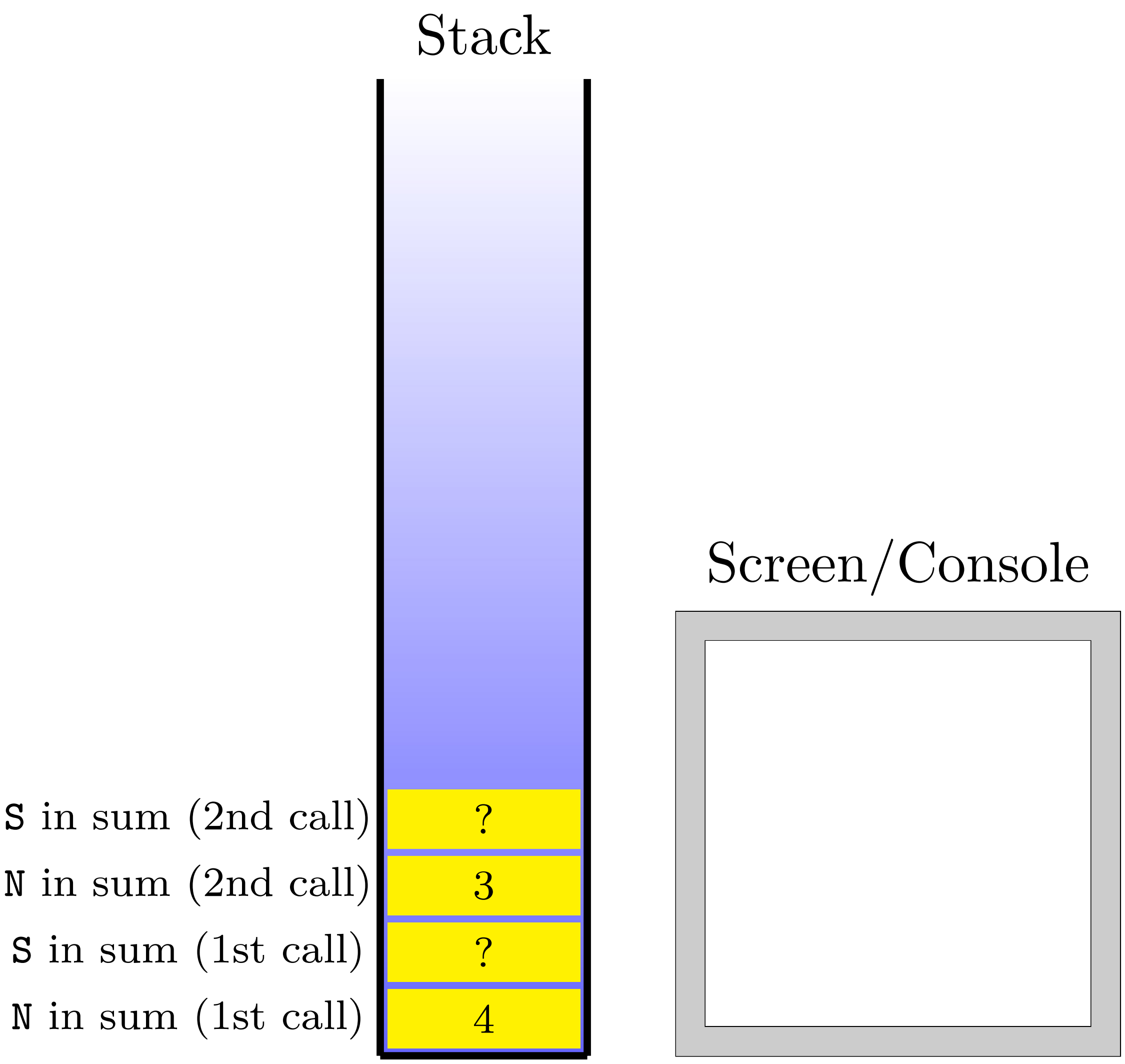
N in sum (1st call) 4

Screen/Console

Step 4

Code Segment

```
#include <stdio.h>
int sum(int N)(1)(2)
{
 int S;
 if (N>1)
    S = sum(N-1)+N;
 else
    S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
     sum(4));
 return 0;
}
```



Stack

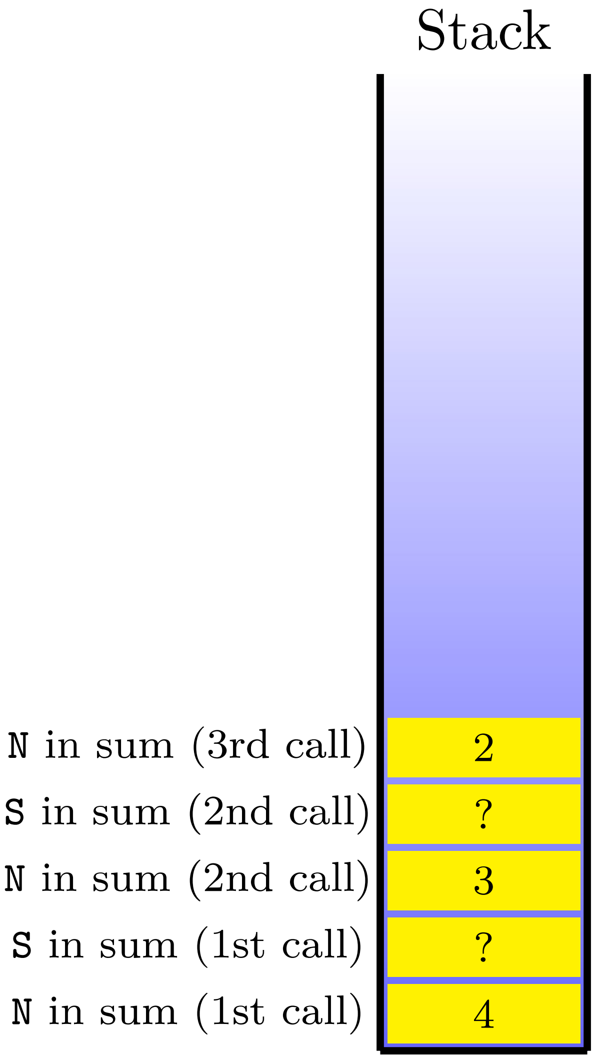
S in sum (2nd call)	?
N in sum (2nd call)	3
S in sum (1st call)	?
N in sum (1st call)	4

Screen/Console

Step 5

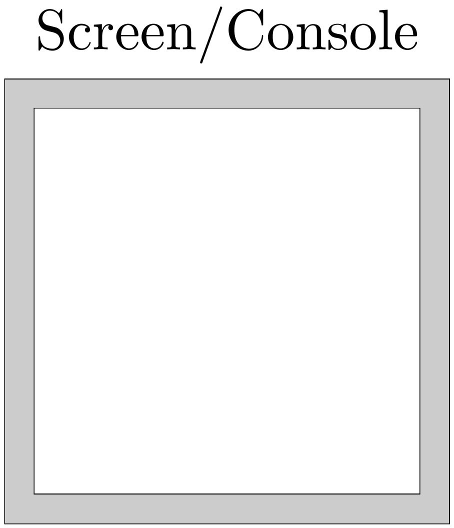
Code Segment

```c
#include <stdio.h>
int sum(int N)(1)(2)(3)
{
  int S;
  if (N>1)
    S = sum(N-1)+N;
  else
    S=1;
  return S;
}

int main()
{
  printf("The sum is %d\n",
      sum(4));
  return 0;
}
```



Stack

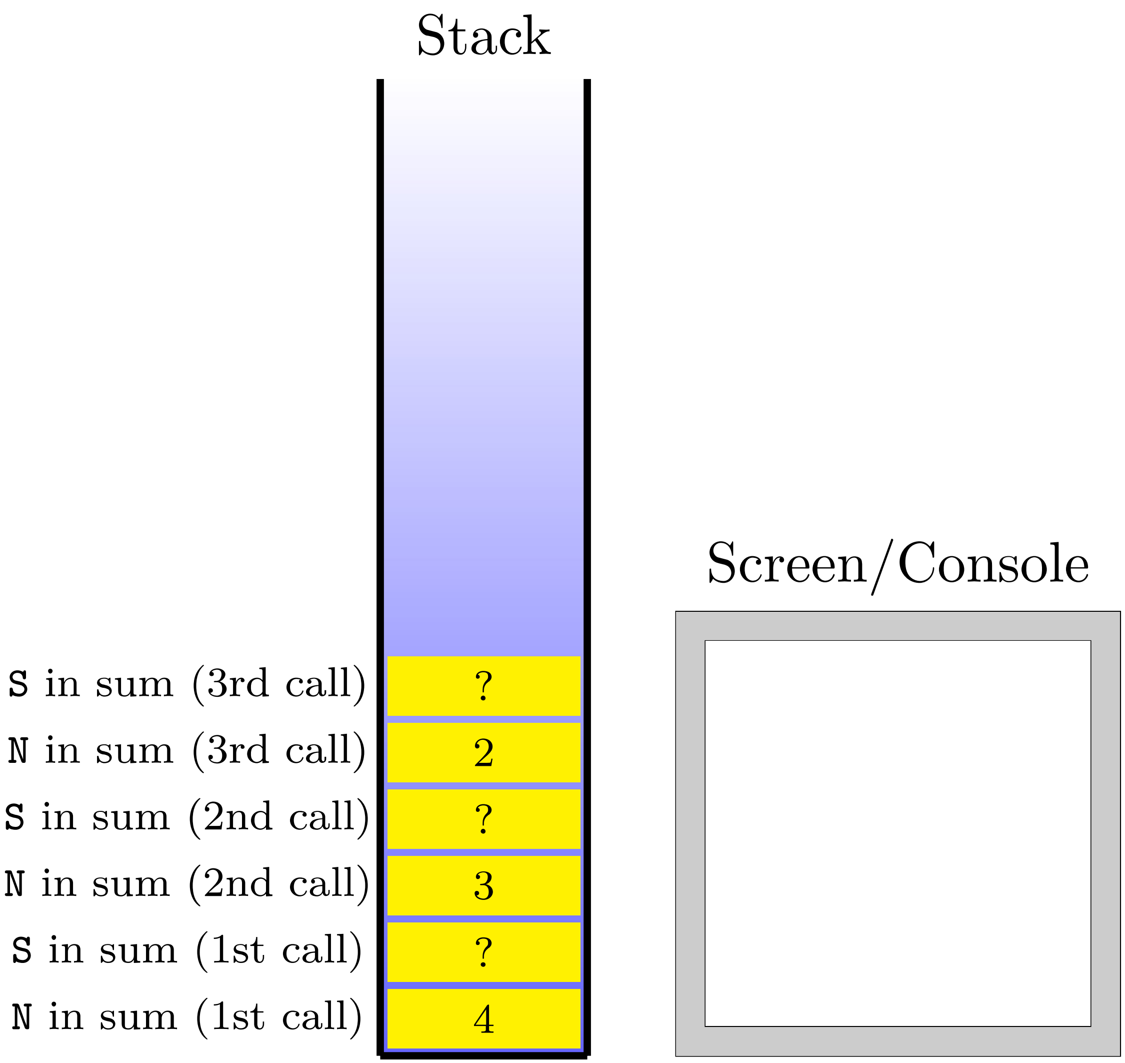
N in sum (3rd call)	2
S in sum (2nd call)	?
N in sum (2nd call)	3
S in sum (1st call)	?
N in sum (1st call)	4



Step 6

Code Segment

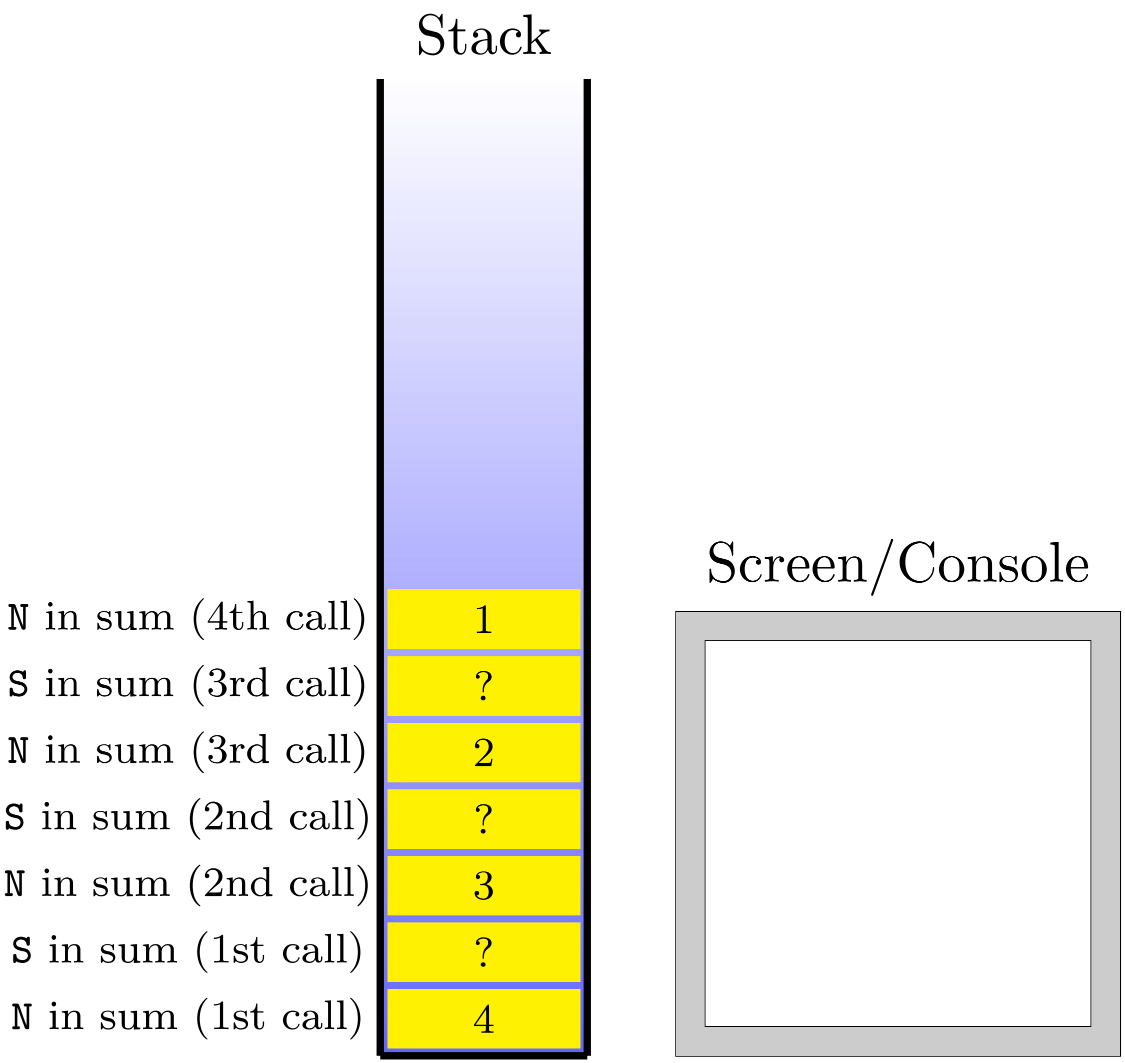
```
#include <stdio.h>
int sum(int N)(1)(2)(3)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
       sum(4));
 return 0;
}
```



Step 7

Code Segment

```
#include <stdio.h>
int sum(int N)(1)(2)(3)(4)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
      sum(4));
 return 0;
}
```



Stack

N in sum (4th call)	1
S in sum (3rd call)	?
N in sum (3rd call)	2
S in sum (2nd call)	?
N in sum (2nd call)	3
S in sum (1st call)	?
N in sum (1st call)	4

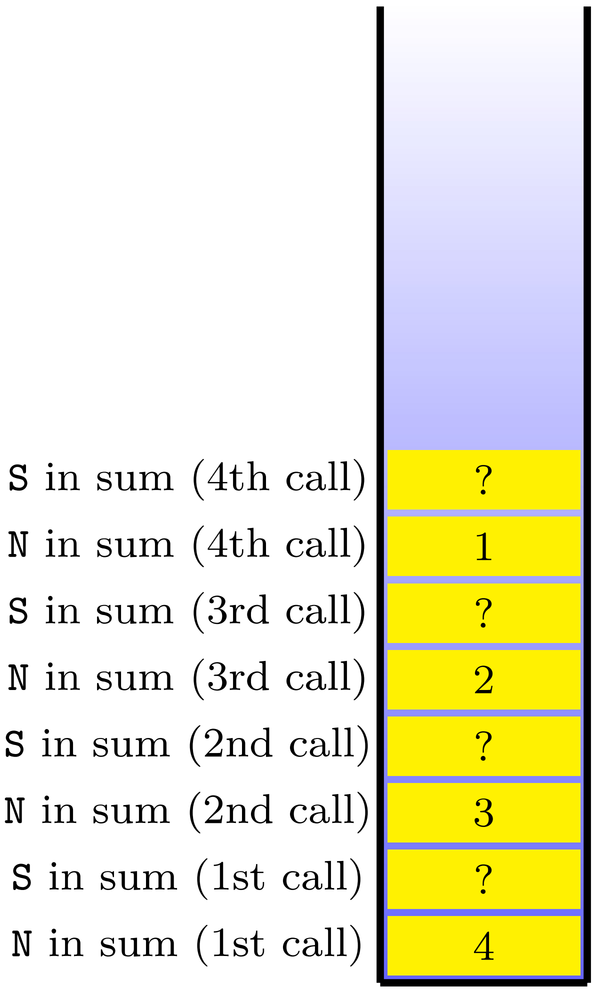
Screen/Console

Step 8

Code Segment

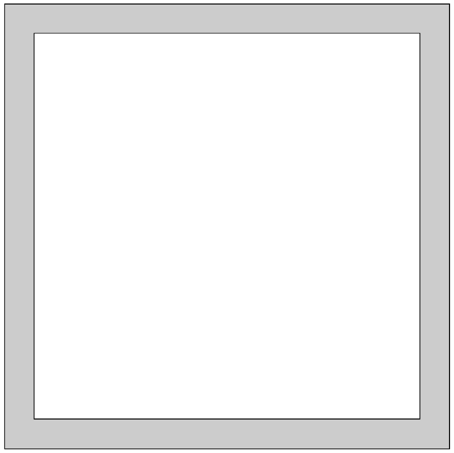
```
#include <stdio.h>
int sum(int N)(1)(2)(3)(4)
{
  int S;
  if (N>1)
    S = sum(N-1)+N;
  else
    S=1;
  return S;
}

int main()
{
  printf("The sum is %d\n",
      sum(4));
  return 0;
}
```

Stack



S in sum (4th call)	?
N in sum (4th call)	1
S in sum (3rd call)	?
N in sum (3rd call)	2
S in sum (2nd call)	?
N in sum (2nd call)	3
S in sum (1st call)	?
N in sum (1st call)	4

Screen/Console

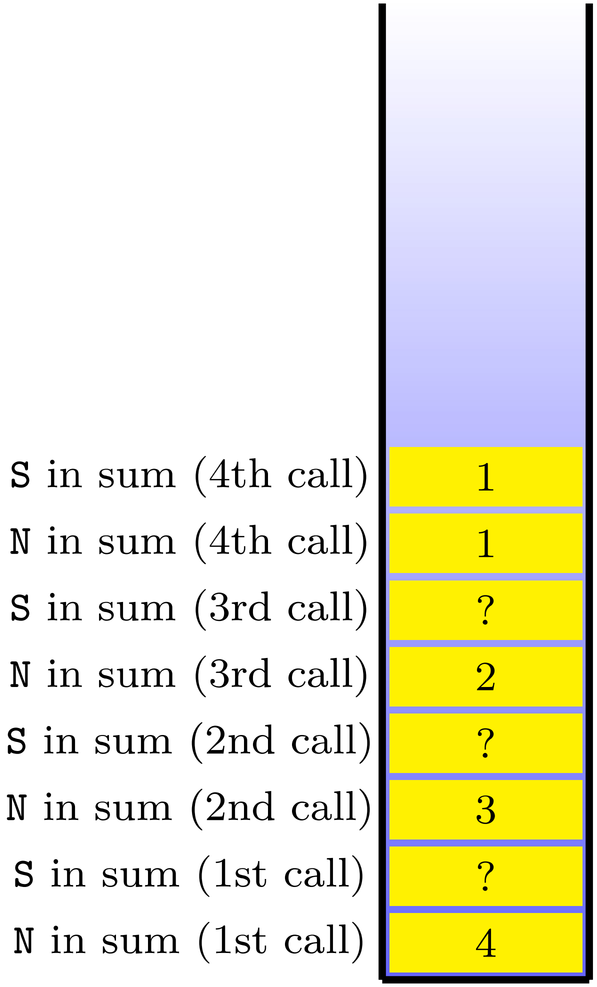


Step 9

Code Segment

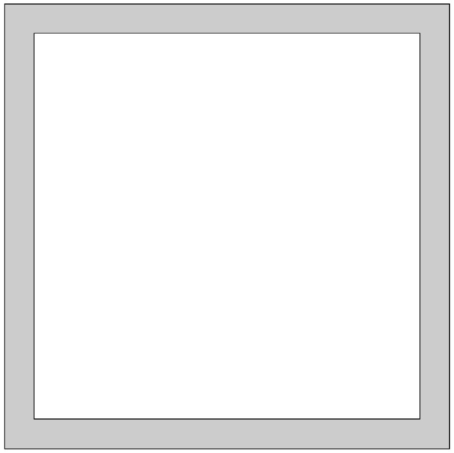
```
#include <stdio.h>
int sum(int N)(1)(2)(3)(4)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
     sum(4));
 return 0;
}
```

Stack



S in sum (4th call)	1
N in sum (4th call)	1
S in sum (3rd call)	?
N in sum (3rd call)	2
S in sum (2nd call)	?
N in sum (2nd call)	3
S in sum (1st call)	?
N in sum (1st call)	4

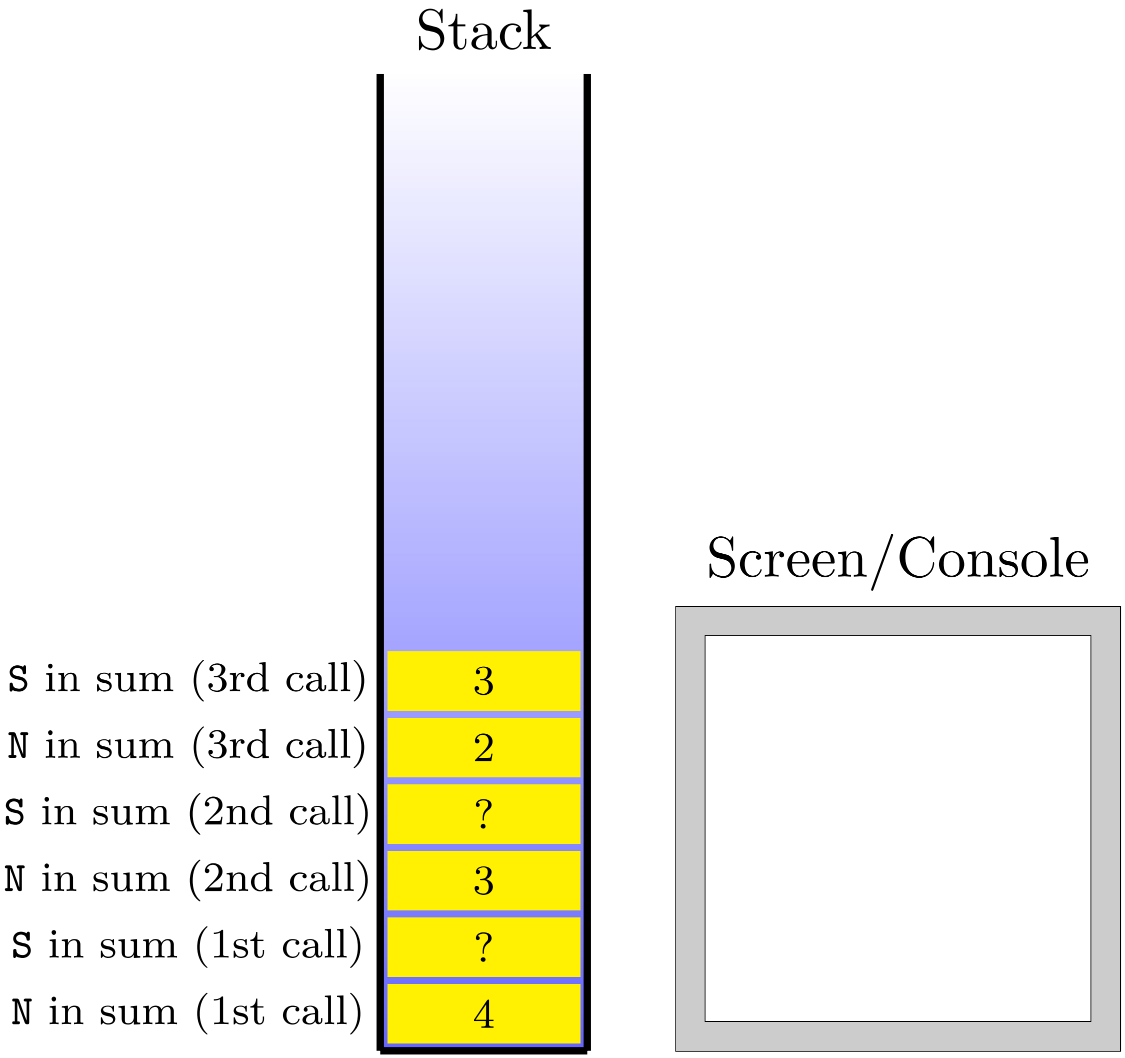
Screen/Console



Step 10

Code Segment

```
#include <stdio.h>
int sum(int N)(1)(2)(3)
{
 int S;
 if (N>1)
   S = sum(N-1)+N;
 else
   S=1;
 return S;
}

int main()
{
 printf("The sum is %d\n",
      sum(4));
 return 0;
}
```



Stack

S in sum (3rd call) 3

N in sum (3rd call) 2

S in sum (2nd call) ?

N in sum (2nd call) 3

S in sum (1st call) ?

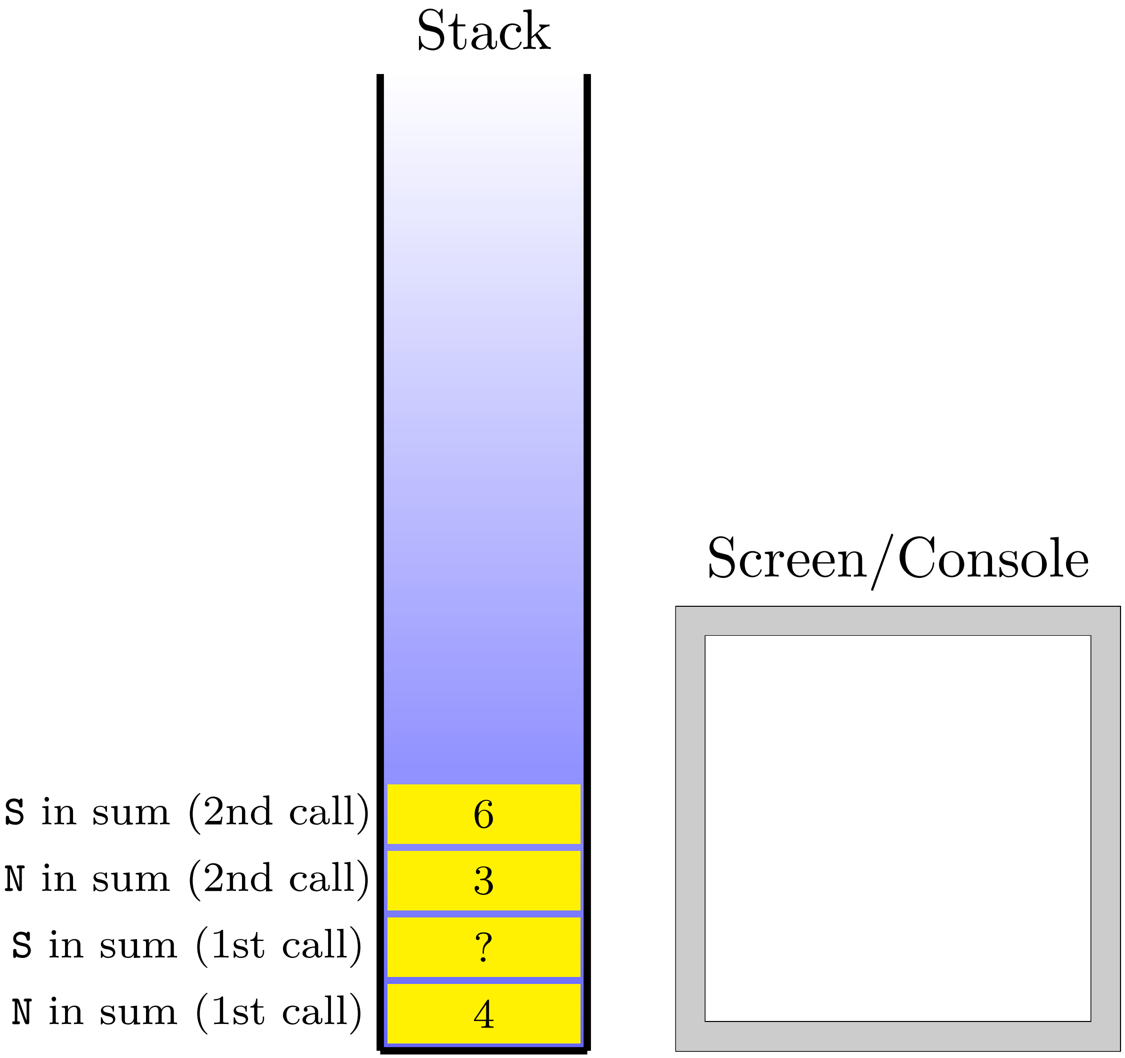
N in sum (1st call) 4

Screen/Console

Step 11

Code Segment

```
#include <stdio.h>
int sum(int N)(1)(2)
{
  int S;
  if (N>1)
     S = sum(N-1)+N;
  else
     S=1;
  return S;
}

int main()
{
  printf("The sum is %d\n",
       sum(4));
  return 0;
}
```



Stack

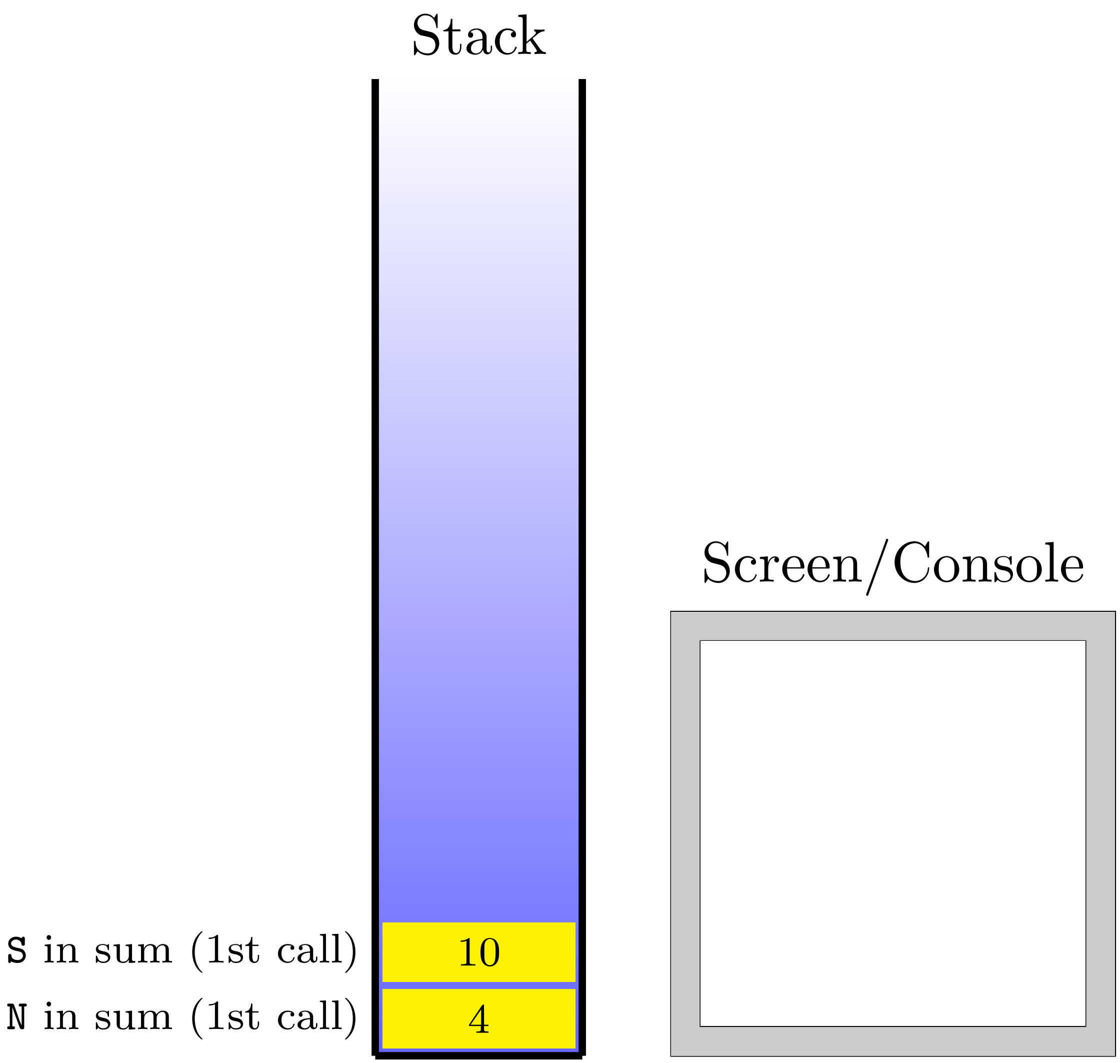
S in sum (2nd call)	6
N in sum (2nd call)	3
S in sum (1st call)	?
N in sum (1st call)	4

Screen/Console

Step 12

Code Segment

```
#include <stdio.h>
int sum(int N)(1)
{
  int S;
  if (N>1)
    S = sum(N-1)+N;
  else
    S=1;
  return S;
}

int main()
{
  printf("The sum is %d\n",
     sum(4));
  return 0;
}
```



Stack

S in sum (1st call) 10

N in sum (1st call) 4

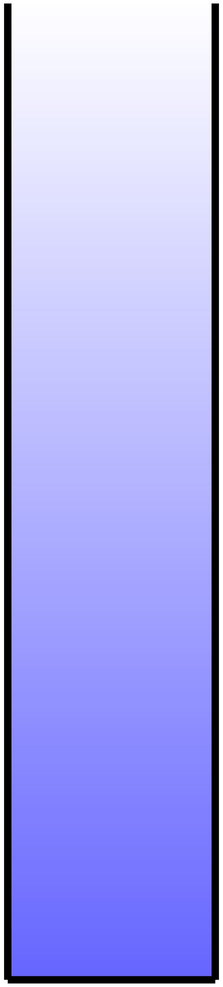
Screen/Console

Step 13

Code Segment

```
#include <stdio.h>
int sum(int N)
{
  int S;
  if (N>1)
     S = sum(N-1)+N;
  else
     S=1;
  return S;
}

int main()
{
  printf("The sum is %d\n",
        sum(4));
  return 0;
}
```

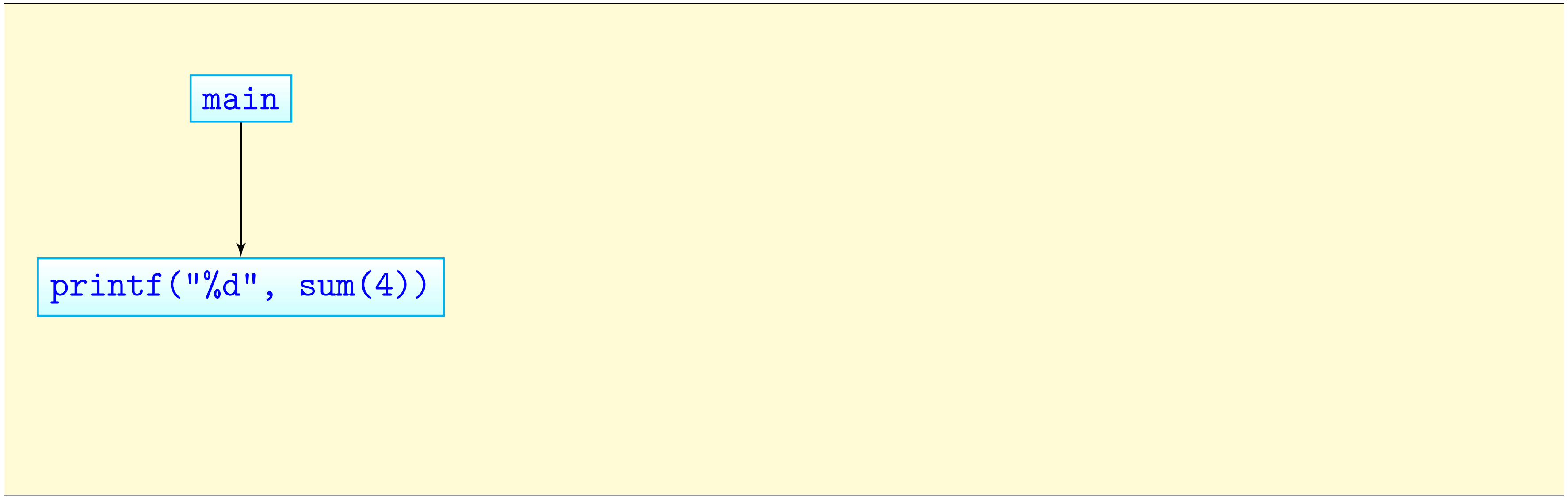
Stack



Screen/Console

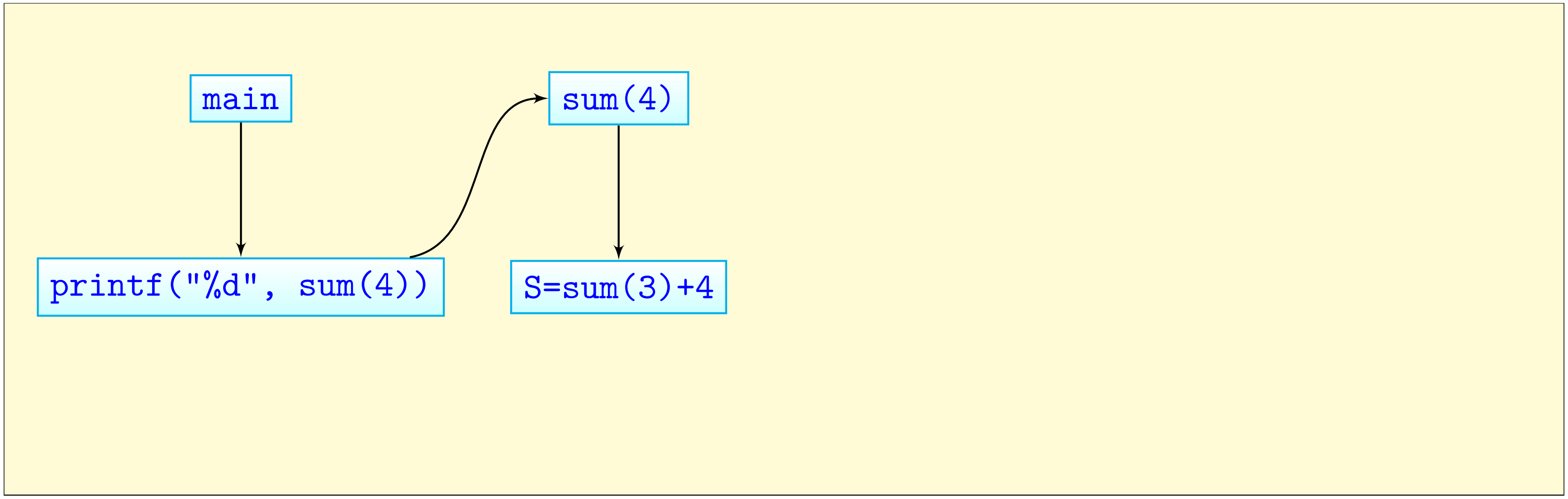
```
The sum is 10
```

Execution Flow: 1

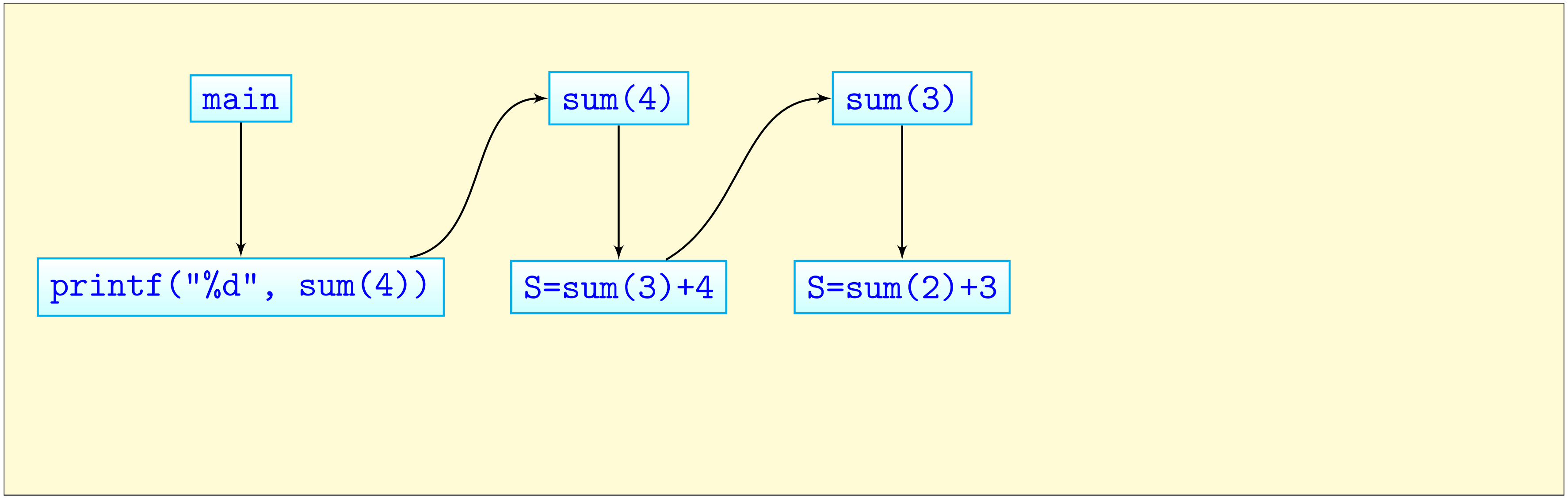


```
main
printf("%d", sum(4))
```

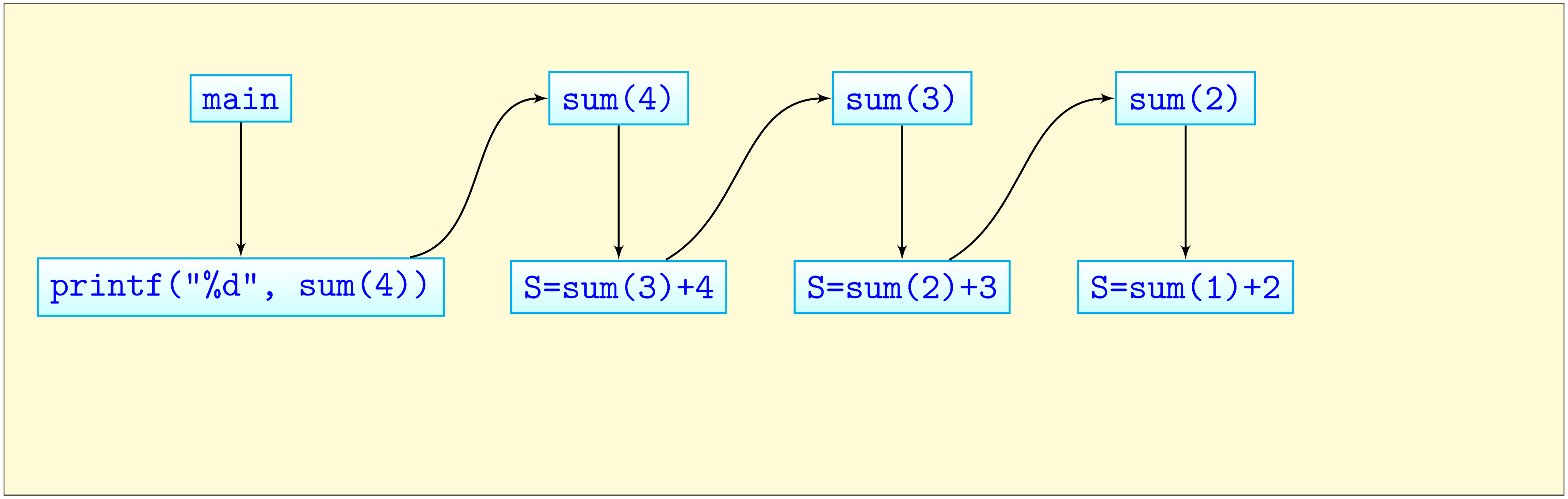
Execution Flow: 2



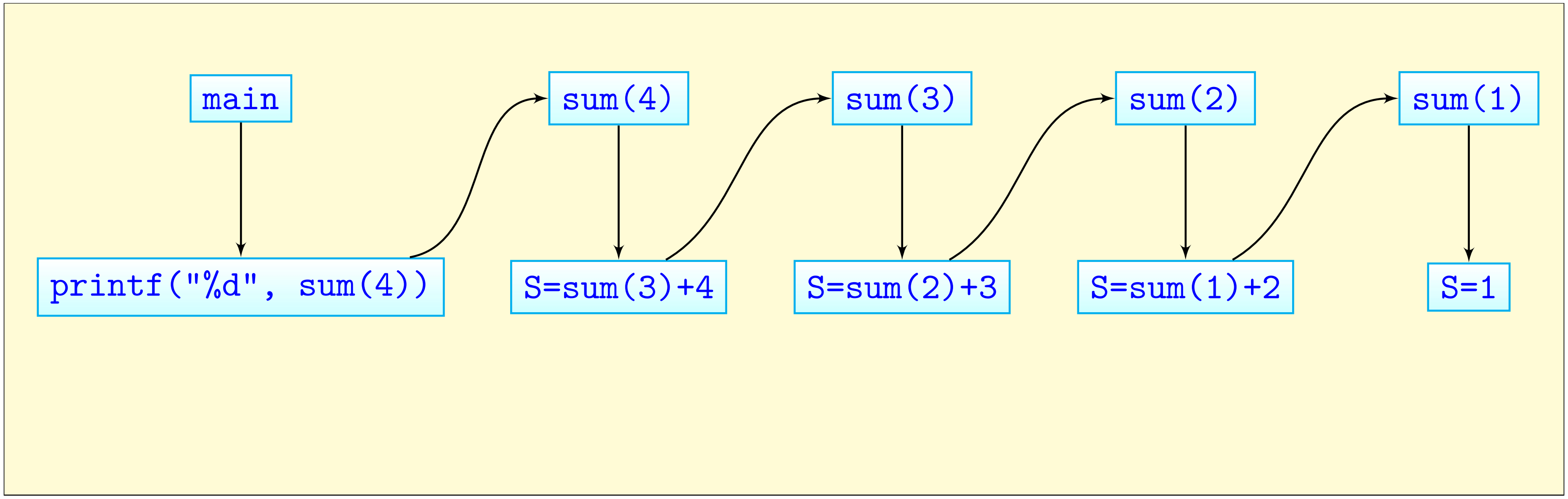
Execution Flow: 3



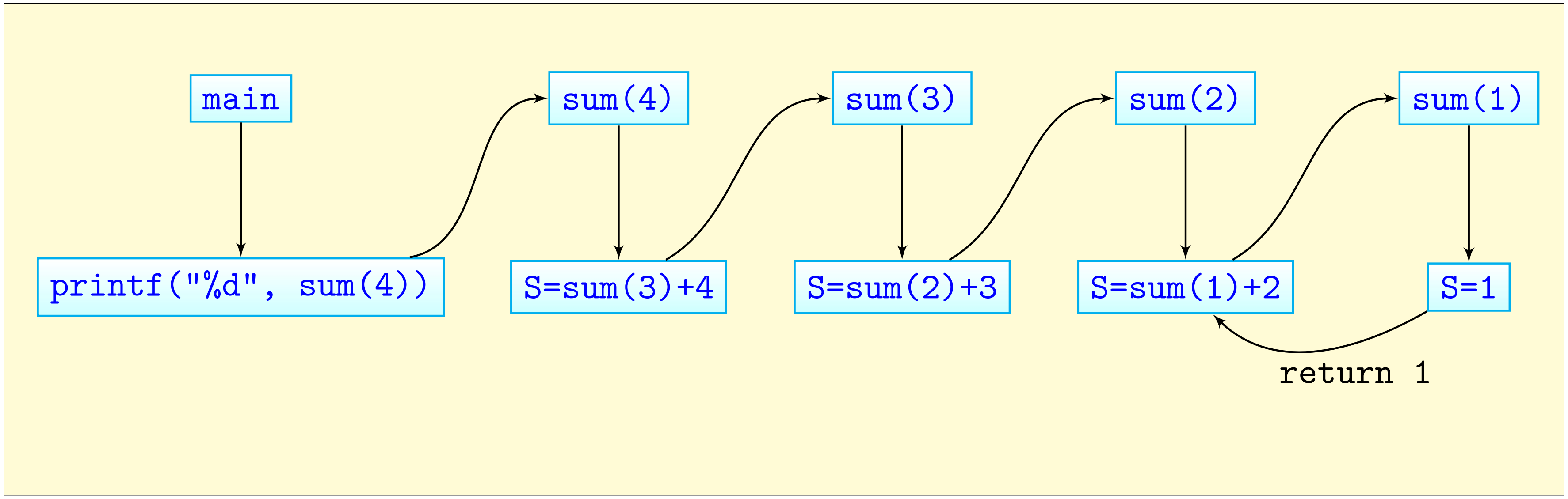
Execution Flow: 4



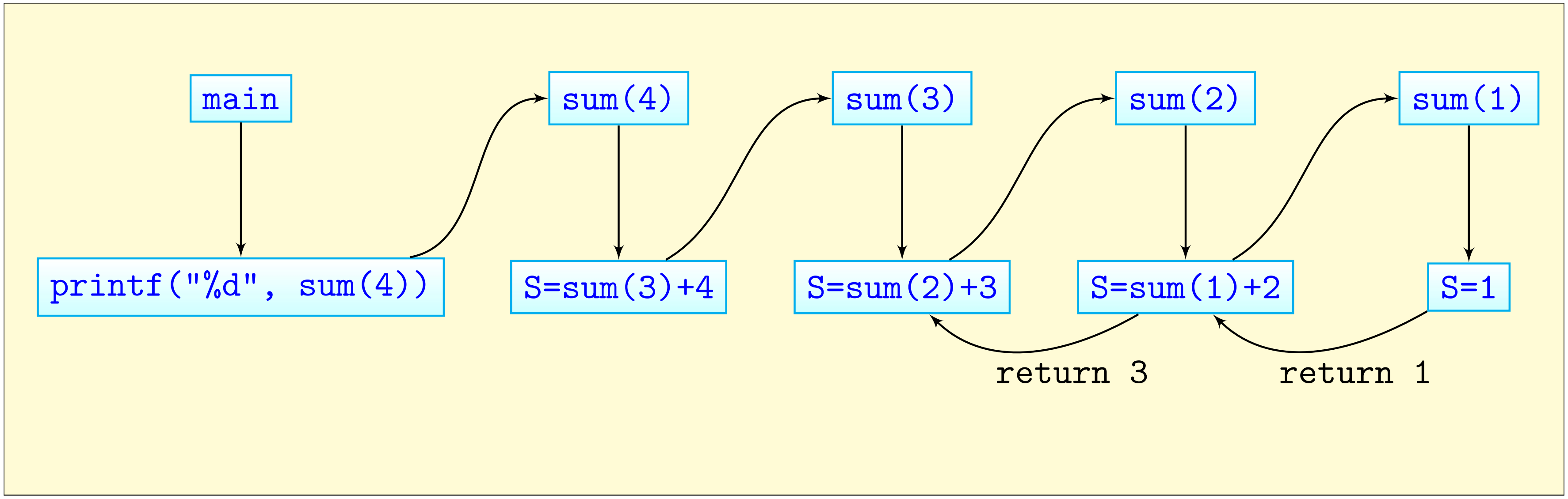
Execution Flow: 5



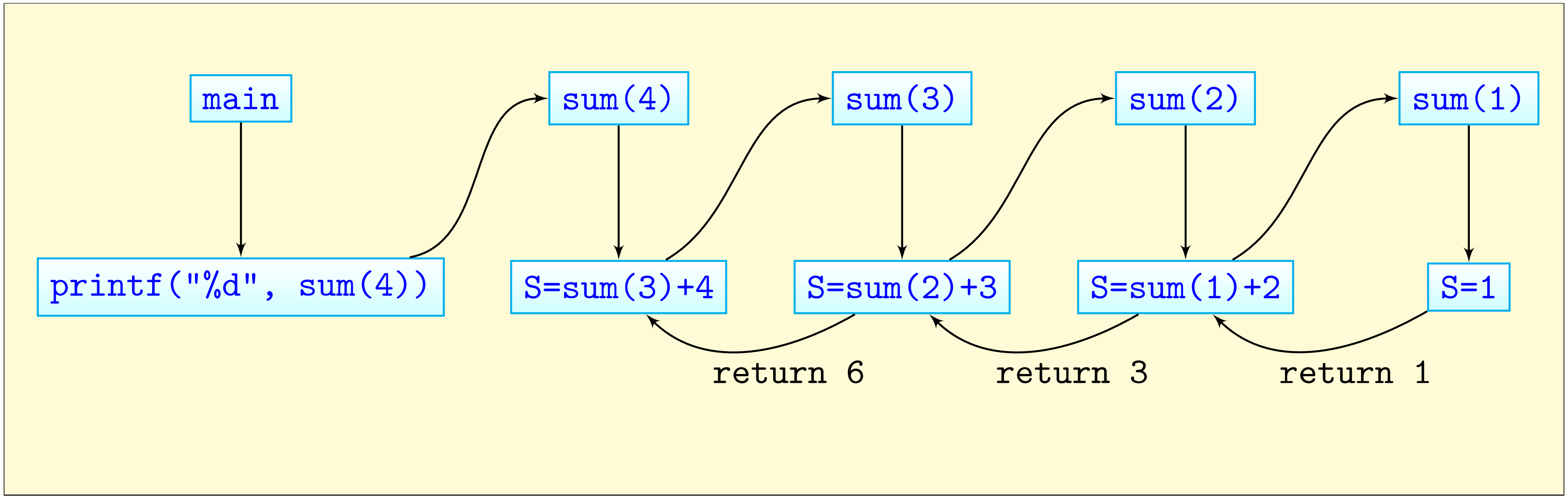
Execution Flow: 6



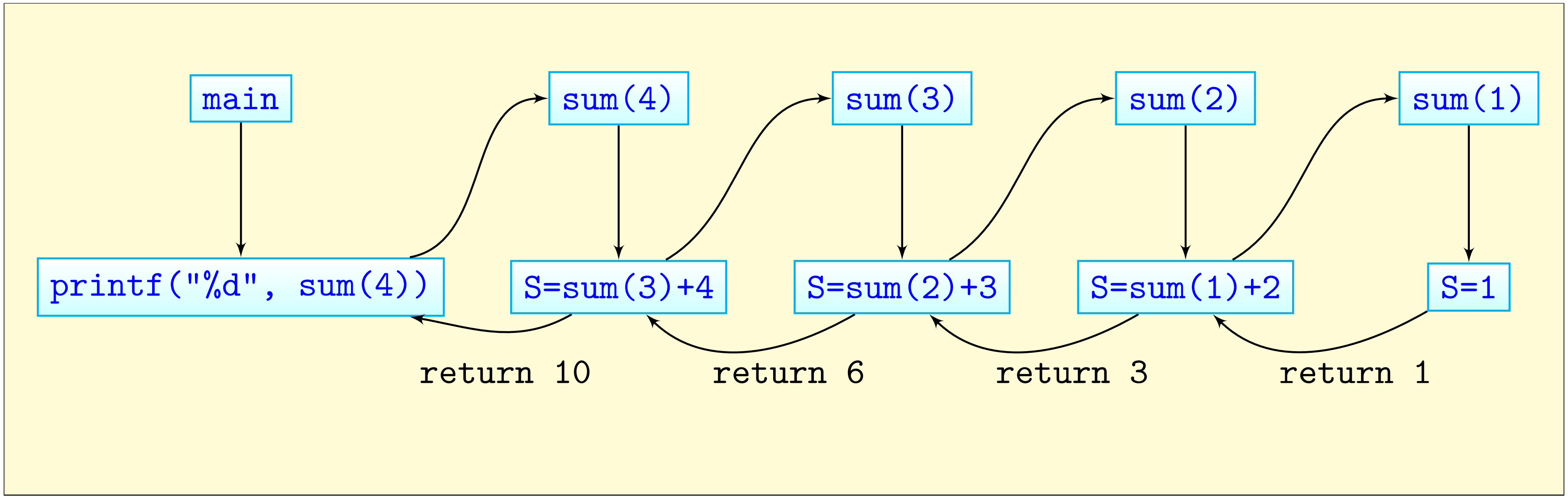
Execution Flow: 7



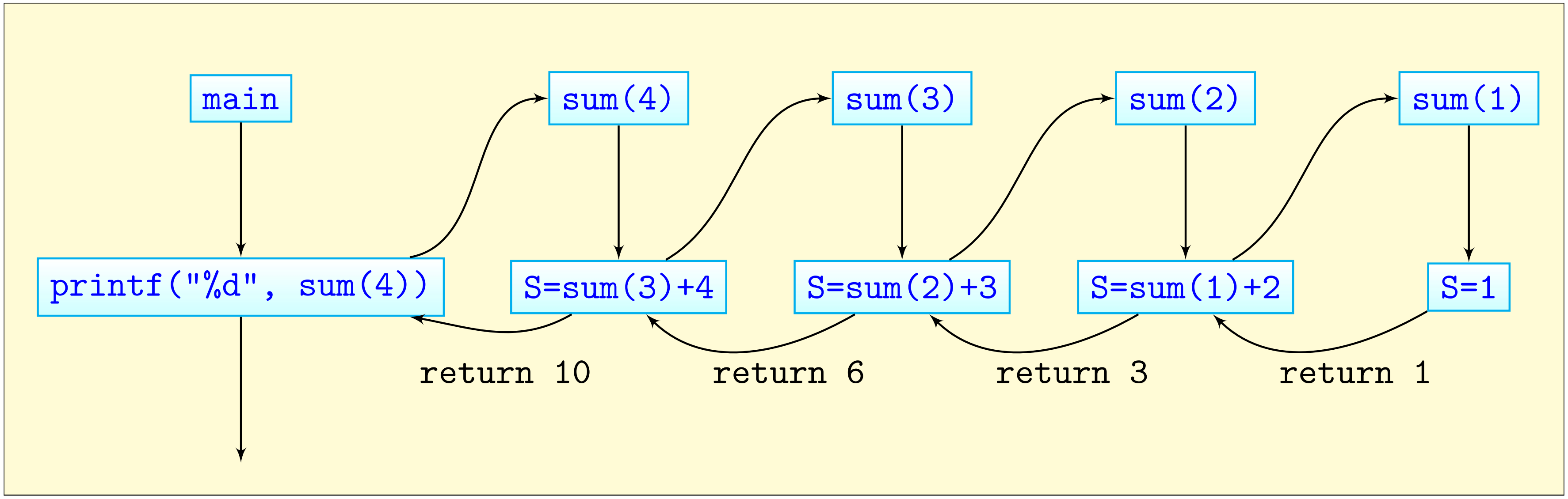
Execution Flow: 8



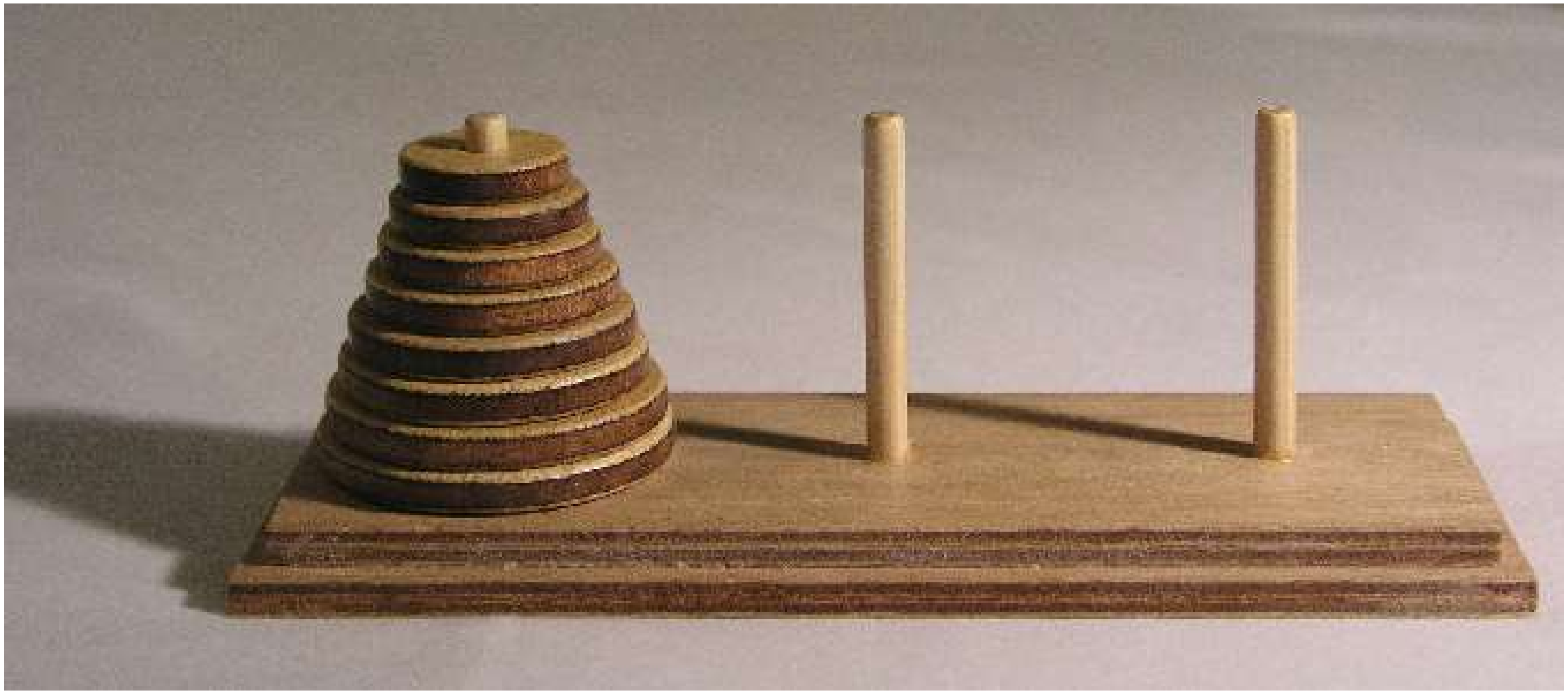
Execution Flow: 9



Execution Flow: 10



Hanoi Tower Problem



We would like to solve the Hanoi Tower Problem with n disks.

Check `https://en.wikipedia.org/wiki/Tower_of_Hanoi`
for the description of the problem
and an animated picture showing the solution

- Suppose that
  - There are n disks, numbered from 1 to n (small to big)
  - The three poles are numbered by 1, 2, and 3 (left to right)
  - The Hanoi problem with 4 disks can then be formulated as "follow the game rule to move disks 1 to 4 from pole 1 to pole 3, using pole 2 as buffer"
- We will create 4 variables:
  - `n`: number of disks
  - `sourcePole`: the ID of the pole on which the disks originally stay
  - `targetPole`: the ID of the pole on which the disks are finally piled
  - `bufferPole`: the ID of the remaining pole
- We will solve the Hanoi tower problem by designing a function with these 4 input variables.
- The function prints the moves as the solution to the problem.

```
void hanoi(int n, int sourcePole, int targetPole, int bufferPole)
{
    ... // TO IMPLEMENT
}
int main()
{
   hanoi(4, 1, 3, 2);
   //move disks 1 to 4 from pole 1 to pole 3, using pole 2 as
buffer
   return 0;
}
```

- Exit branch ($n = 1$):
    - Move disk 1 from sourcePole to targetPole
- Recursion branch ($n > 1$): The problem can be decomposed into the following 3 steps.
    1. Move disks 1 to $n - 1$ from `sourcePole` to `bufferPole`, using `targetPole` as buffer $\Rightarrow$ smaller problem!
    2. Move disk n from `sourcePole` to `targetPole`
    3. Move disks 1 to $n - 1$ from `bufferPole` to `targetPole`, using `sourcePole` as buffer $\Rightarrow$ smaller problem!

Code it Up!

- Advantage of Recursion:
    - Elegant
    - It often provides simple solutions to otherwise very difficult problems
- Disadvantage of Recursion:
    - When the recursive function uses a significant amount of memory (by allocating its local variables), deep recursion may lead to excessive memory consumption.
    - Each call of the recursive function needs to copy its input to a local variable, which costs additional time (relative to solutions without recursion).