Recursion

What is Recursion?

- Definition
  - A repetitive process in which **an algorithm calls itself**

```c
#include <stdio.h>

void Recursive(int n)
{
    // Base case: termination condition!
    if (n == 0) return;

    printf("Recursive call: %d\n", n);

    Recursive(n - 1);
}
```

Example: Summing from 1 to *n*

- Iterative vs. Recursive programming

```
int sum(int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = sum + i;
    return sum;
}
```

$$S(n) = \sum_{i=0}^{n} i$$

```
int rsum(int n)
{

}
```

$$S(n) = \begin{cases} 0 & if\ n = 0 \\ S(n-1) + n & otherwise \end{cases}$$

Designing Recursive Programming

- Two parts
  - **Base case**: Solve the smallest problem directly.
  - **Recursive case**: Simplify the problem into smaller ones and calculate a recurrence relation.

$$\sum_{i=0}^{n} i = n + (n-1) + \cdots + 1 + 0$$

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$

$$S(n) = \begin{cases} 0 & if\ n = 0 \\ S(n-1) + n & otherwise \end{cases}$$

```
int S(int n)
{
    if (n == 0) return 0;
    else return S(n - 1) + n;
}
```

Function Call/Return

- Is it a correct code?
  - **Stack overflow**: Eventually halt when runs out of (stack) memory.

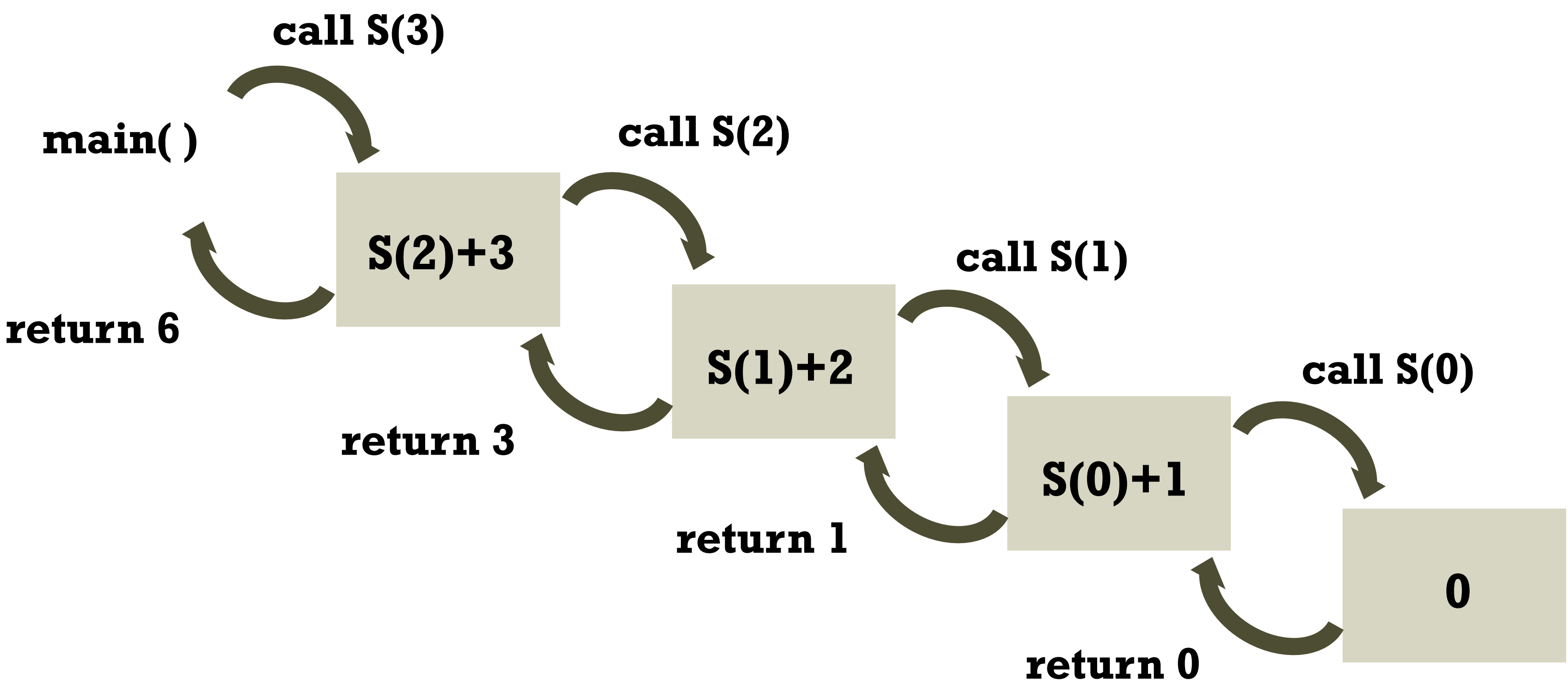
```
void Recursive(int n)
{
    printf("Recursive call: %d\n", n);
    Recursive(n - 1);
}
```

```
void Recursive(int n)
{
    // Base case: termination condition!
    if (n == 0) return;
    else
    {
        printf("Recursive call: %d\n", n);
        Recursive(n - 1);
    }
}
```

Example: Summing from 1 to *n*

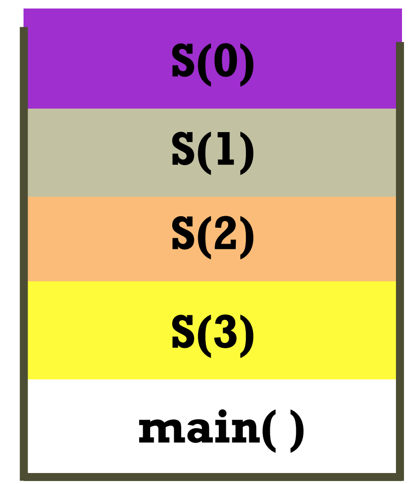
- How does recursive programming work?
  - How many calls/returns happen?

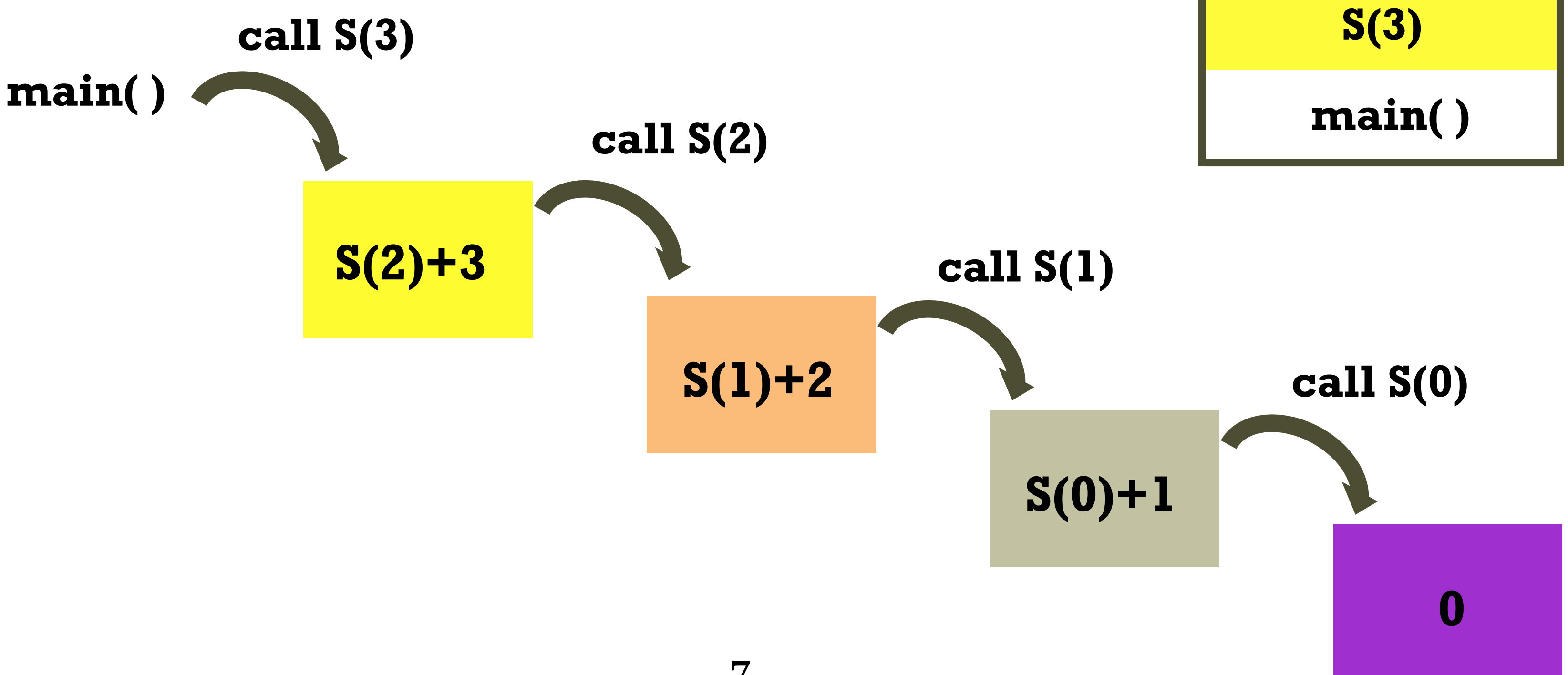
```
int S(int n)
{
    if (n == 0) return 0;
    else return S(n - 1) + n;
}
```



Function Call/Return

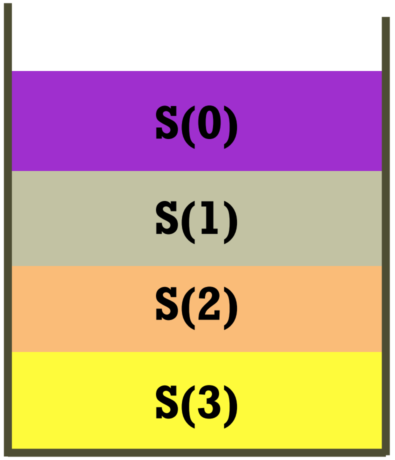
- How is stack memory changed?
  - When a function calls, it is sequentially stored in stack memory.
    - The returned address is kept into system stack.
    - All local variables are newly allocated into system stack.

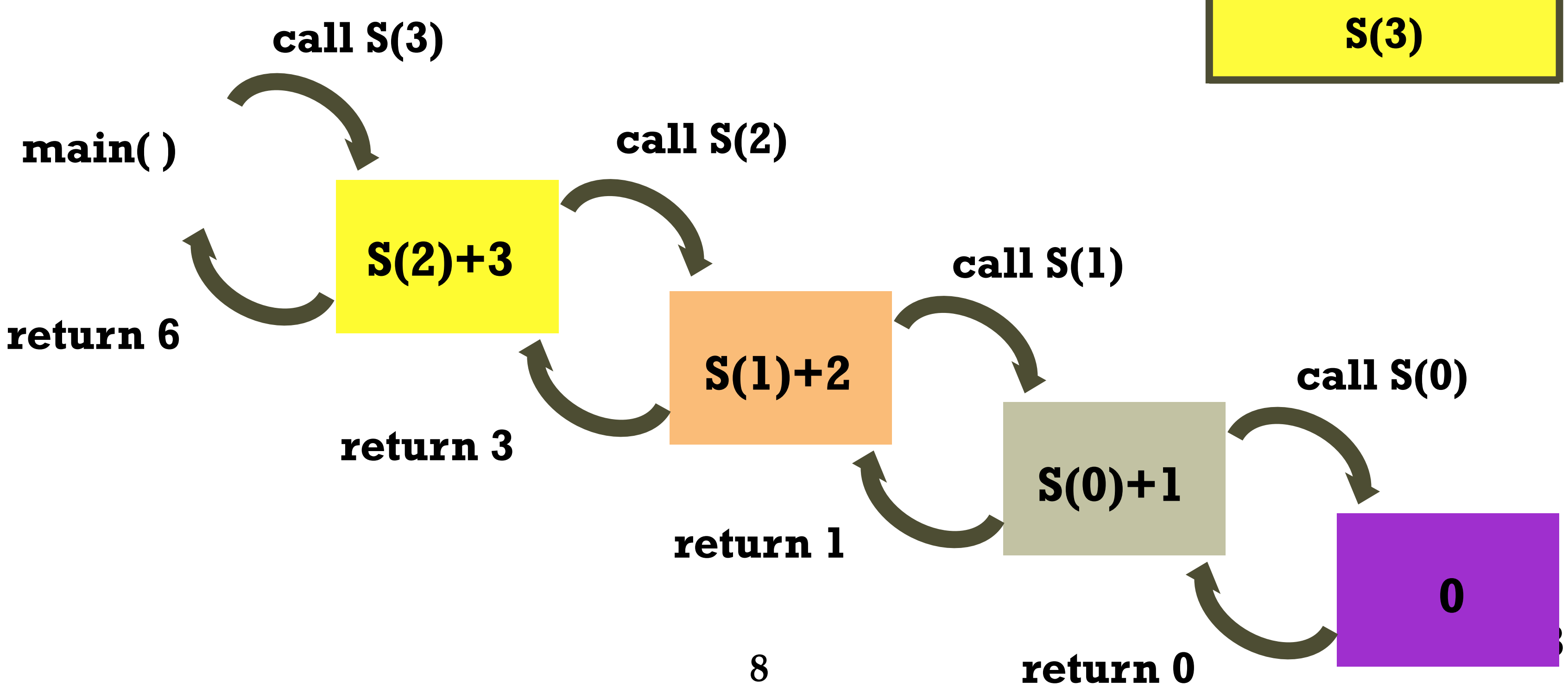




Function Call/Return

- How is stack memory changed?
  - When a function is returned, it is removed from stack memory.
    - All local variables are removed.
    - Return the address kept in the stack.





Recursion vs. Iteration

- Iteration
  - Terminate when a condition is proven to be false(usually).
  - Each iteration does **NOT require** extra memory.
  - Some iterative solutions may not be as obvious as recursive solutions.

- Recursion
  - Terminate when a base case is reached.
  - Each recursive call **requires extra space** on stack memory.
  - Some solutions are **easier** to formulate recursively.
    - Some of them are ALMOST impossible with iteration.

Summing Multiples of Three

- Calculate the sum of all multiples of three from 0 to n.

```
int sum(int n)
{
    int sum = 0;
    for (int i = 0; i <= n; i+=3)
        sum = sum + i;
    return sum;
}
```

```
int rsum(int n)
{
    if (n == 0)
        return 0;
    else if (n % 3 != 0)
        return rsum(n - n % 3);
    else
        return rsum(n - 3) + n;
}
```

Finding Maximum Number

- Search the maximum number in array.

```
int findMax(int* arr, int n)
{
	int max = arr[0];
	for (int i = 1; i < n; i++)
	{
		if (arr[i] > max)
			max = arr[i];
	}
	return max;
}
```

Finding Maximum Number

- Search the maximum number in array.

$$\max(v_1, v_2, \dots, v_n) = \begin{cases} v_1 & \textit{if } n = 1 \\ \max(v_k, \max(v_1, \dots, v_{k-1})) & \textit{otherwise} \end{cases}$$

```
int rfindMax(int* arr, int n)
{
	if (n == 1)
		return arr[0];
	else
	{
		int max = rfindMax(arr, n - 1);
		if (max < arr[n - 1])
			return arr[n - 1];
		else
			return max;
	}
}
```

Printing Reverse String

- Print a string in a reverse manner.
  - `rprint("abc")` → `cba`, `rprint("2a1bc")` → `cb1a2`

```
void rprint(char* s, int n)
{
    for (int i = n - 1; i >= 0; i--)
        printf("%c", s[i]);
}
```

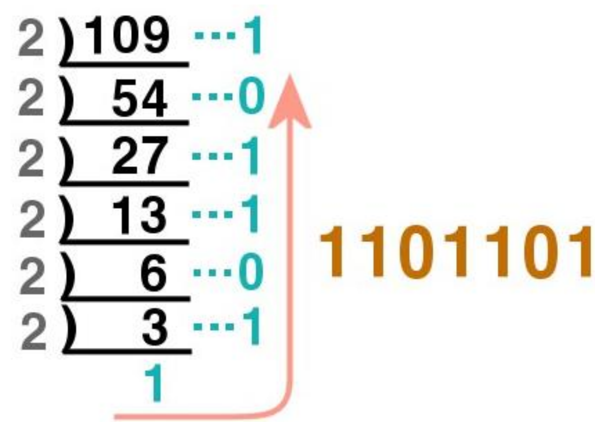
↓

```
void rrprint(char* s, int n)
{
    if (n == 0) return;
    else {
        printf("%c", s[n - 1]);
        return rrprint(s, n - 1);
    }
}
```

Printing Binary Number

- Print a binary number using recursion.
  - Note: Input a positive integer only.
  - `binary(1)` → `1`, `binary(3)` → `11`
  - `binary(10)` → `1010`, `binary(109)` → `1101101`

```c
void binary(int n)
{
    if (n == 0) return;
    else {
        binary(n / 2);
        printf("%d", n % 2);
    }
}
```



Calculating the Power of X

- Iterative vs. recursive programming

```
int power(int x, int n)
{
    int pow = 1;
    for (int i = 0; i < n; i++)
        pow *= x;

    return pow;
}
```

```
int rpower(int x, int n)
{
    if (n == 0) return 1;
    else return x * rpower(x, n - 1);
}
```

Calculating the Power of X

- How to implement recursion more efficiently?
  - $x^{10} = (x^5)^2 = \left((x^2)^2 \times x\right)^2$

```
int rpower(int x, int n)
{
    if (n == 0) return 1;
    else return x * rpower(x, n - 1);
}
```

⬇

```
int rpower2(int x, int n)
{
    if (n == 0) return 1;
    else if (n % 2 == 0) {
        int m = rpower2(x, n / 2);
        return m * m;
    }
    else return x * rpower2(x, n - 1);
}
```

Calculating Fibonacci Numbers

- Every number is the sum of two preceding ones.
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

```
int fibo(int n) {
	if (n == 1 || n == 2) return 1;
	else {
		int prev = 1, cur = 1, next = 1;
		for (int i = 3; i <= n; i++) {
			prev = cur, cur = next;
			next = prev + cur;
		}
		return next;
	}
}
```

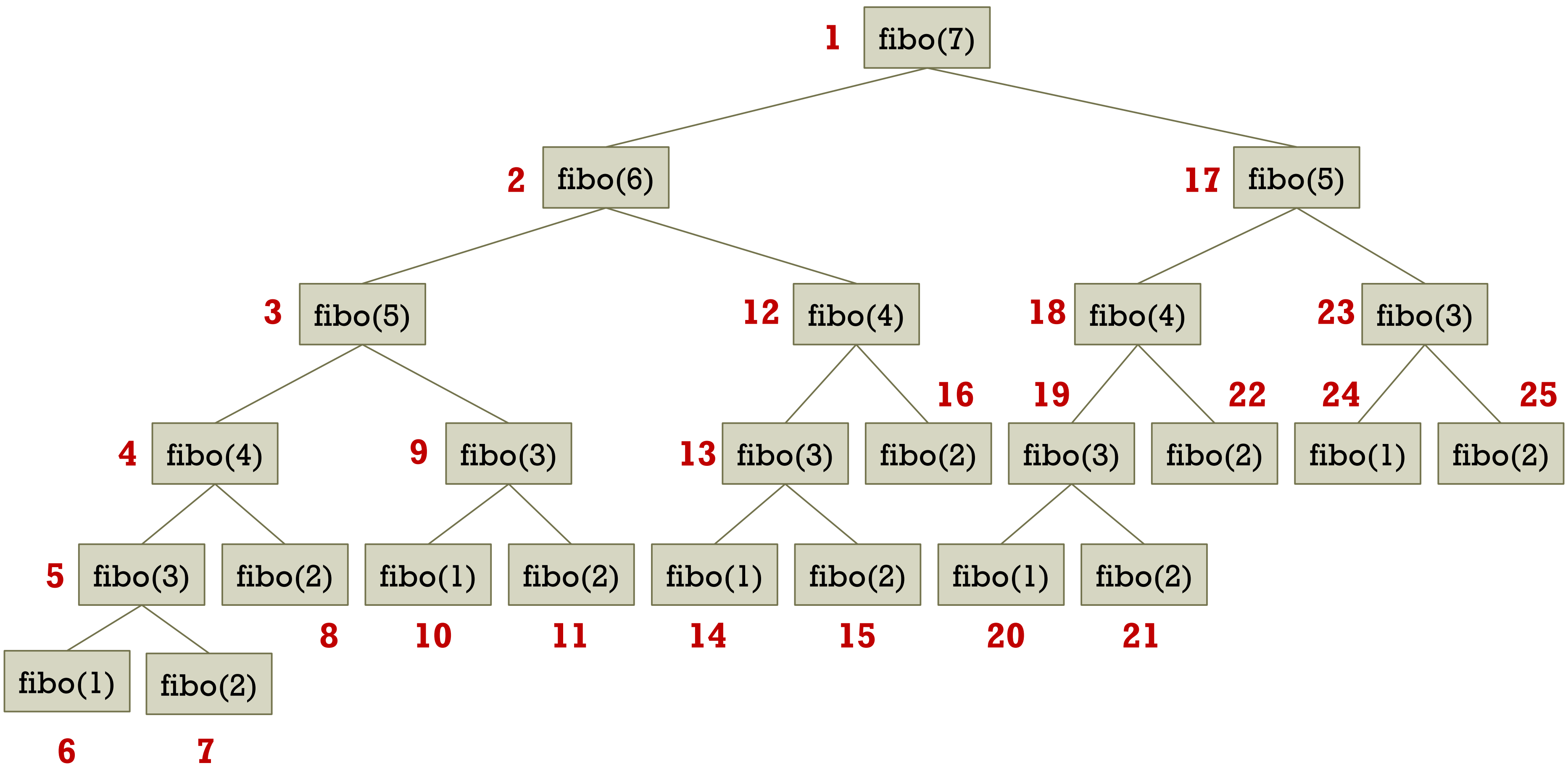
$$F_n = \begin{cases} F_1 = 1 & if\ n = 1 \\ F_2 = 1 & if\ n = 2 \\ F_{n-1} + F_{n-2} & otherwise \end{cases}$$

⬇

```
int rfibo(int n)
{
	if (n == 1 || n == 2) return 1;
	else
		return rfibo(n - 1) + rfibo(n - 2);
}
```

Recursion for Fibonacci Numbers

- How many calls happen?



Binary Search using Recursion

- Compare the median value in search space to the target value.
  - Can eliminate half of the search space at each step.
  - It will eventually be left with a search space consisting of a single element.

```
int bsearch(int arr[], int low, int high, int target)
{
    if (low > high) return -1;
    else {
        int mid = (low + high) / 2;
        if (target == arr[mid])
            return (mid);
        else if (target < arr[mid])
            bsearch(arr, low, mid - 1, target);
        else
            bsearch(arr, mid + 1, high, target);
    }
}
```

Time Complexity for Recursion

- How to calculate time complexity for recursion
  - **T(n)**: the maximum amount of time taken on input of size *n*

```
int S(int n)
{
    if (n == 0) return 0;
    else return S(n - 1) + n;
}
```

- Formulate a recurrence relation with sub-problems.

$$T(n) = T(n-1) + c$$

$$T(n-1) = T(n-2) + c$$

……

Time Complexity for Recursion

- Time complexity for binary search

$$T(n) = T\left(\frac{n}{2}\right) + 1 = T\left(\frac{n}{4}\right) + 2 = T\left(\left\lfloor\frac{n}{2^k}\right\rfloor\right) + k, \text{where } \left\lfloor\frac{n}{2^k}\right\rfloor = 1$$

$$\Longrightarrow T(n) = 1 + \lfloor \log_2 n \rfloor = O(\log n)$$

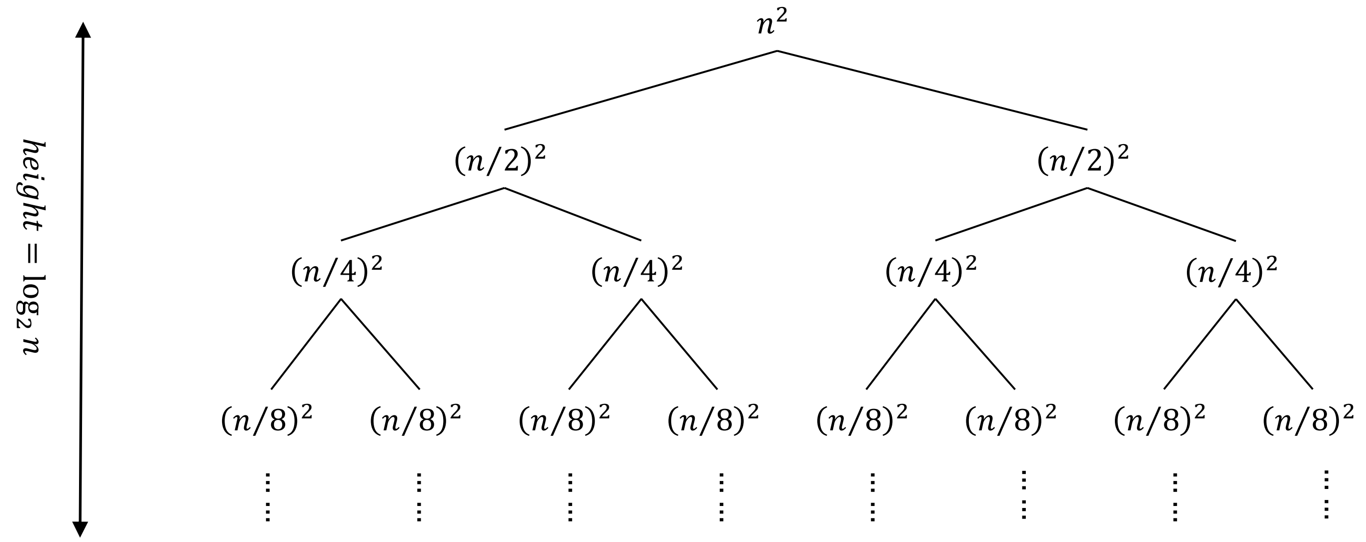
```
int bsearch(int arr[], int low, int high, int target)
{
    if (low > high) return -1;
    else {
        int mid = (low + high) / 2;
        if (target == arr[mid])
            return (mid);
        else if (target < arr[mid])
            bsearch(arr, low, mid - 1, target);
        else
            bsearch(arr, mid + 1, high, target);
    }
}
```

Recursion Tree

- Visualizing how recurrences are iterated

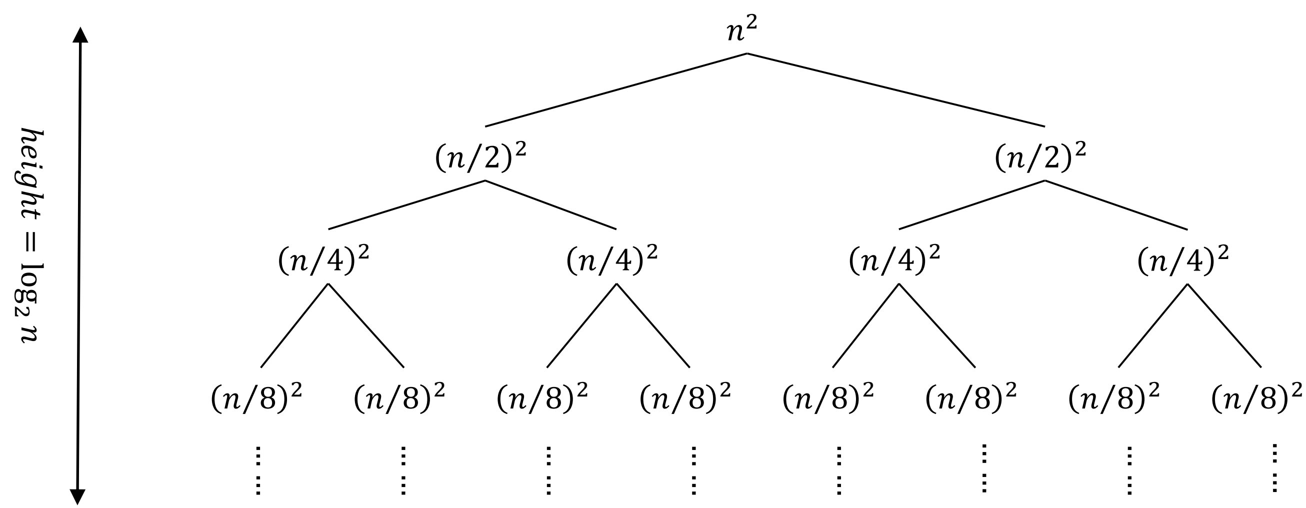
$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

  - The recursion tree for this recurrence has the following form:



Recursion Tree

- In the recursion tree,
  - The depth of the tree does not really matter.
  - The amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.

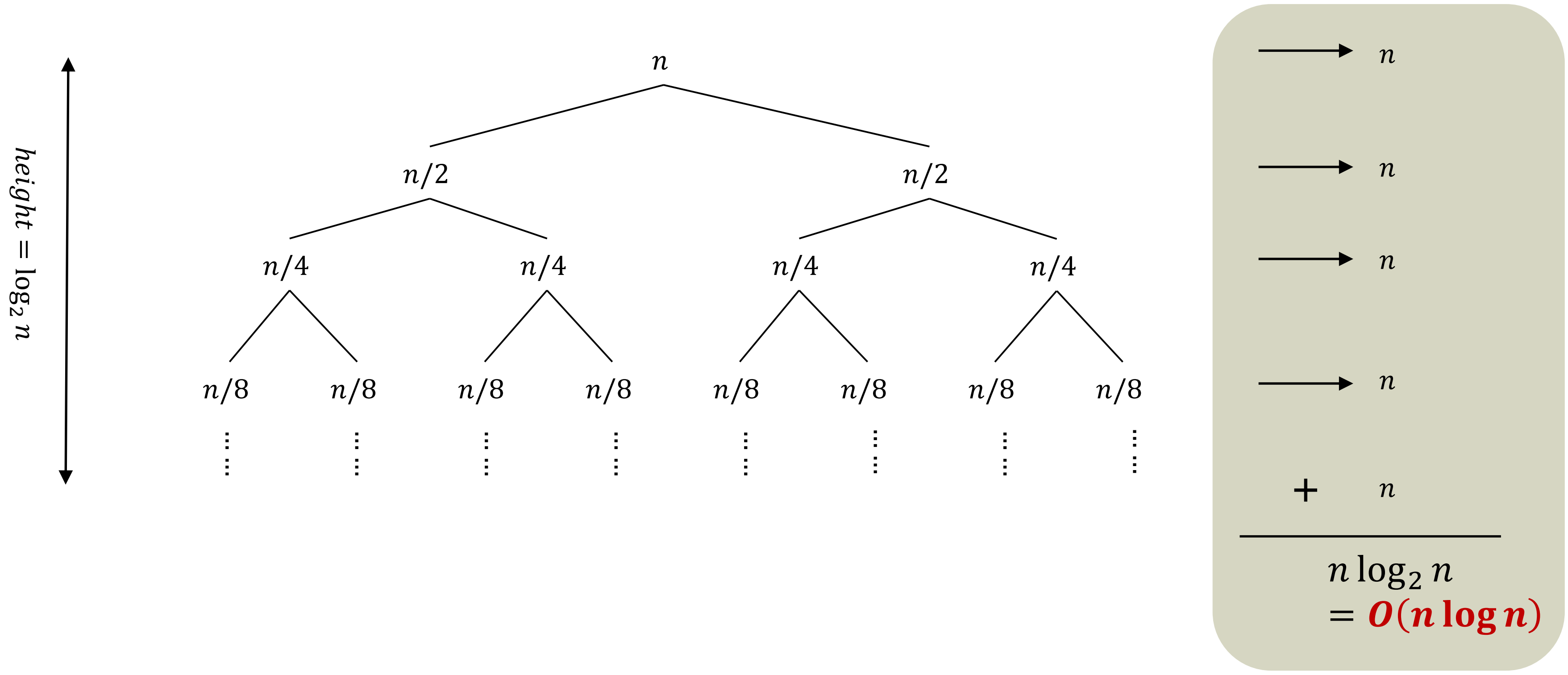


$$
\begin{aligned}
T(n) &= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots + \frac{n^2}{2^{\lfloor \log_2 n \rfloor}} \\
&= n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \frac{1}{2^{\lfloor \log_2 n \rfloor}}\right) = O(n^2)
\end{aligned}
$$

Example: Recursion Tree

- Concern the following recurrence relation form:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



Tail Recursion

- It is a subroutine call performed as the final action of a function
  - **Tail-call optimization**: The current function is no longer needed and can thus be replaced by the tail-call function.

```
int rsum(int n)
{
    if (n == 0)
        return 0;
    else
        return rsum(n - 1) + n;
}
```

```
rsum(4)
rsum(3) + 4
(rsum(2) + 3) + 4
((rsum(1) + 2) + 3) + 4
(((rsum(0) + 1) + 2) + 3) + 4
```

⬇

```
int rsum2(int n, int sum)
{
    if (n == 0)
        return sum;
    else
        return rsum2(n - 1, sum + n);
}
```

```
rsum2(4, 0)
rsum2(3, 4)
rsum2(2, 7)
rsum2(1, 9)
rsum2(0, 10)
return 10
```

Example: Fibonacci Numbers

- Key advantage of tail recursion
  - Tail-call optimization can save both space and time, because it only needs the address of the tail-call function.

```
int fibo(int n)
{
    if (n == 1 || n == 2) return 1;
    else
        return fibo(n - 1) + fibo(n - 2);
}
```

⬇

```
int rfiboTail(int n, int prev, int cur)
{
    if (n == 1 || n == 2) return cur;
    else
    return rfiboTail(n - 1, cur, prev + cur);
}
```

Example: Finding Maximum Number

- How to implement a tail-recursive version

```
int rfindMax(int* arr, int n)
{
    if (n == 1)
        return arr[0];
    else
    {
        int max = rfindMax(arr, n - 1);
        if (max < arr[n - 1])
            return arr[n - 1];
        else
            return max;
    }
}
```

```
int rfindMaxTail(int* arr, int n, int max)
{
    if (n == 1) return max;
    else {
        if (max < arr[n - 1])
            max = arr[n - 1];

        return rfindMaxTail(arr, n - 1, max);
    }
}
```