Inefficient Recursive Function

Direct Coding of function from an inductive definition may be very inefficient.

Fibonacci Sequence : *An Example*

Consider the following Fibonacci[a] Sequence.

n	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
$fib(n)$	0	1	1	2	3	5	8	13	21	34	55	$\cdots$

[a] Leonardo Pisano Fibonacci (1170 - 1250 (?), Pisa)

Inductive Definition

The inductive definition of the n^{th} term of the sequence is

$$fib_n = \begin{cases} n, & if\ 0 \leq n < 2, \\ fib_{n-1} + fib_{n-2}, & if\ n \geq 2. \end{cases}$$

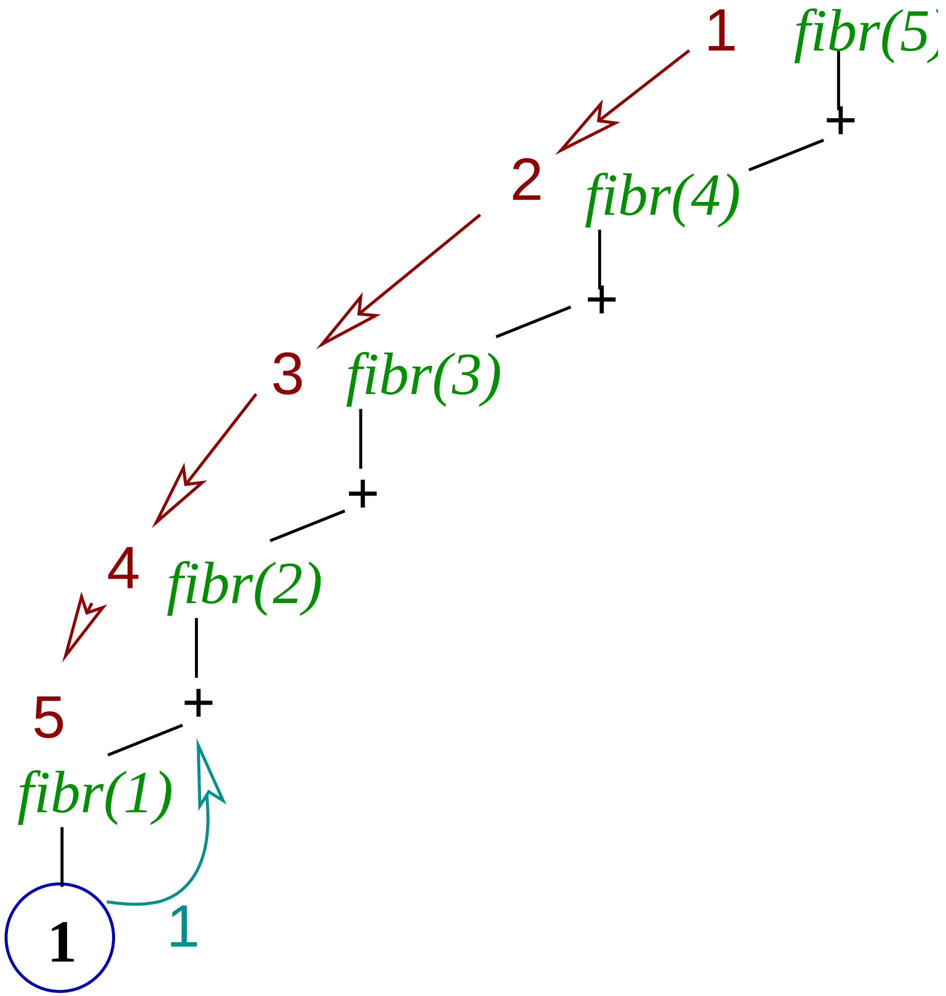
C Function

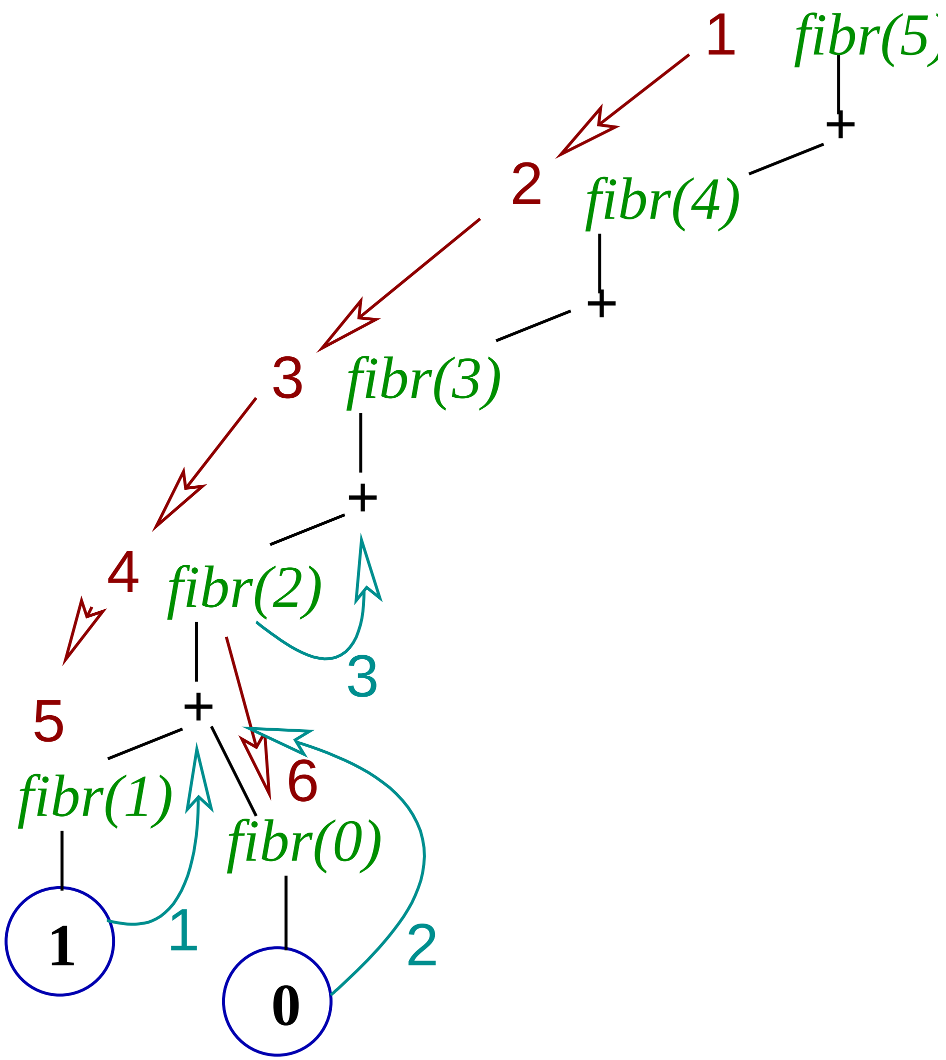
The definition can be directly coded as a C function.

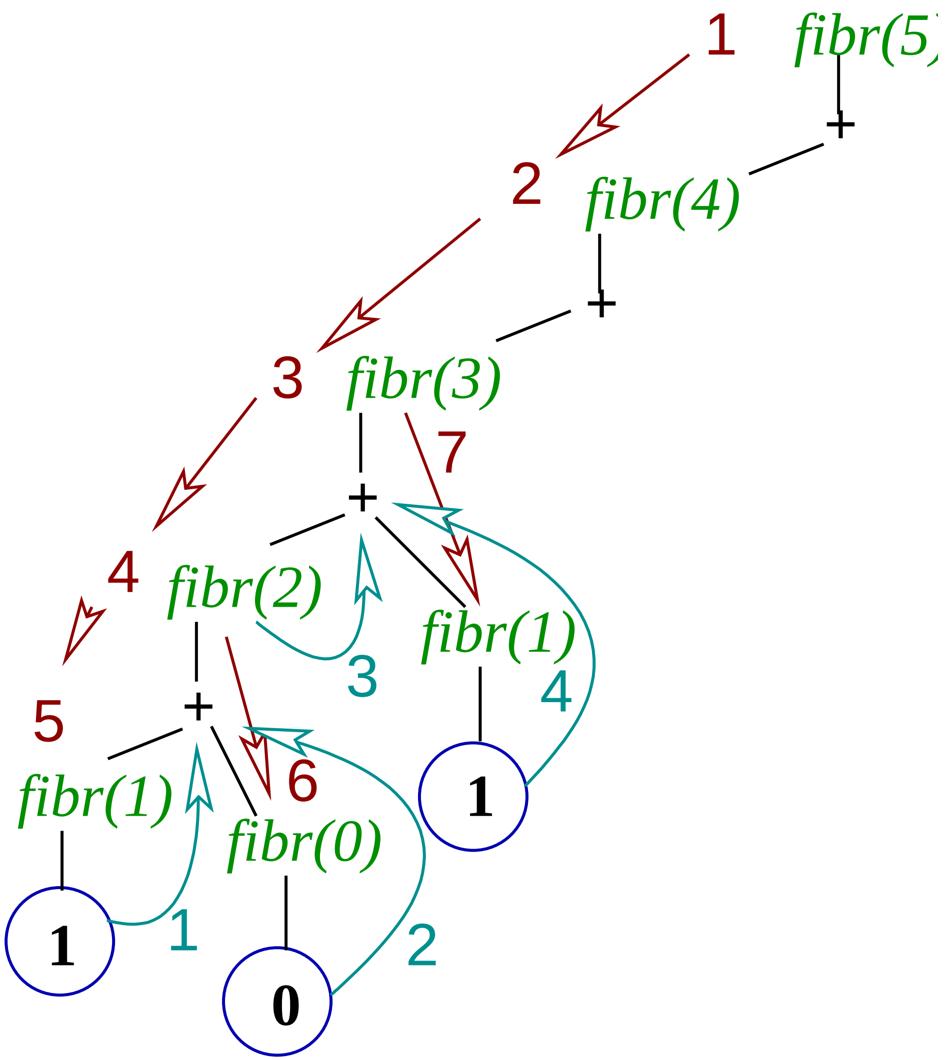
```
int fibr(int n){ // fibonacciFR1.c
    if(n < 2) return n ;
    return fibr(n-1) + fibr(n-2) ;
}
```

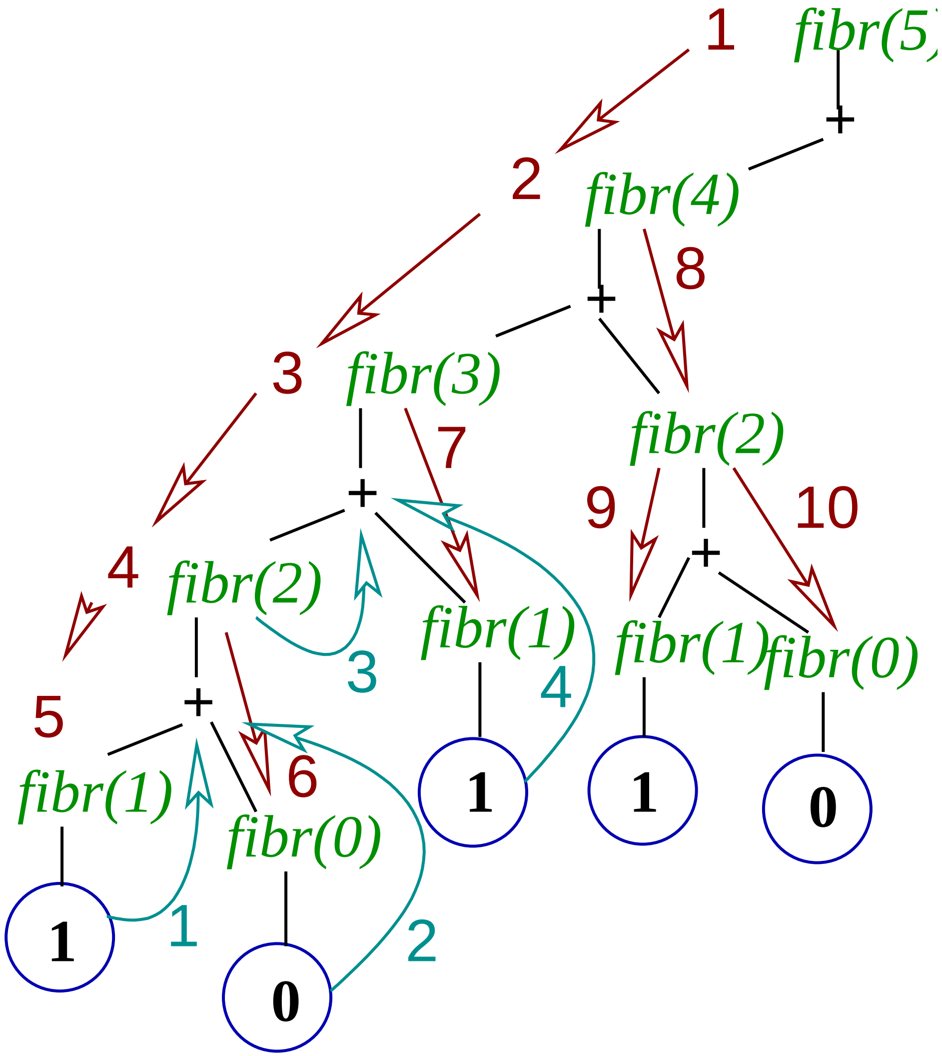
The Call Tree: $n = 5$

The call sequence for $n = 5$ is as follows.

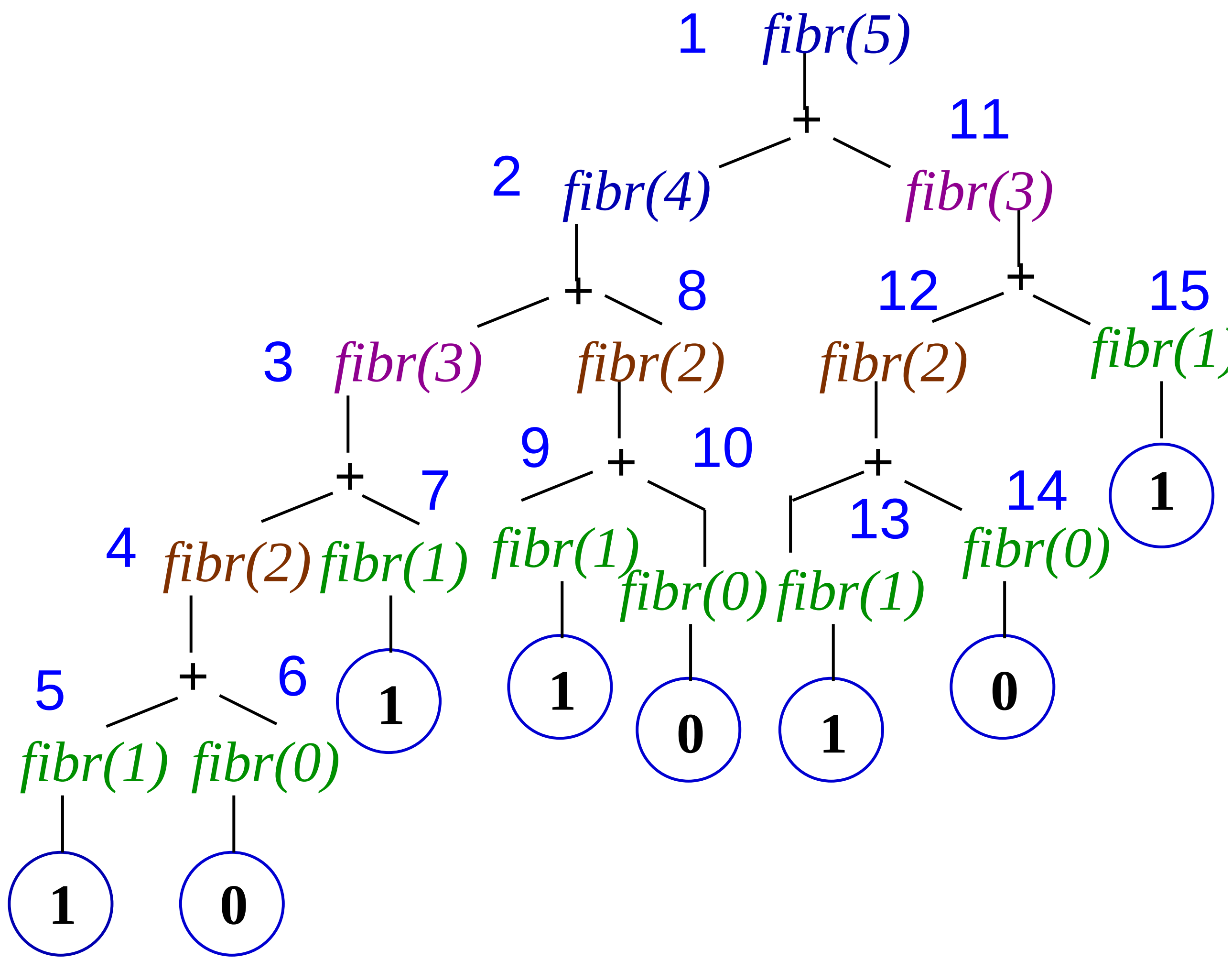

1
fibr(5)
+
2
fibr(4)
+
3
fibr(3)
+
4
fibr(2)
+
5
fibr(1)
1
1


1
fibr(5)
+
2
fibr(4)
+
3
fibr(3)
+
4
fibr(2)
+
5
fibr(1)
6
fibr(0)
3
1
1
0
2


1
fibr(5)
+
2
fibr(4)
+
3
fibr(3)
7
+
4
fibr(2)
fibr(1)
3
4
5
+
6
fibr(1)
fibr(0)
1
1
1
0
2


1
fibr(5)
+
2
fibr(4)
+
8
3
fibr(3)
+
7
fibr(2)
9
10
+
4
fibr(2)
+
3
fibr(1)
4
fibr(1)
fibr(0)
5
+
6
fibr(1)
fibr(0)
1
1
0
1
1
0
2

Call Tree


1 fibr(5)
+
2 fibr(4)
11 fibr(3)
+
3 fibr(3)
8 fibr(2)
12 fibr(2)
15 fibr(1)
+
4 fibr(2)
7 fibr(1)
9 fibr(1)
10 fibr(0)
13 fibr(1)
14 fibr(0)
5 fibr(1)
6 fibr(0)
1
0
1
1
0
1
0
1

Note

Fifteen calls are made and seven additions are performed. This could have been done by only four additions in a iterative program.

n	0	1	2	3	4	5
$fibr(n)$	0	1	1	2	3	5
op			+	+	+	+

Note

The main problem is the re-computation of the same result again and again. To compute the value of the 5^{th} Fibonacci number, the function computes the 3^{rd} Fibonacci number twice, the 2^{nd} Fibonacci number three times etc.

Note

The number of additions to compute the n^{th} Fibonacci number in this function is given in the following table.

n	0	1	2	3	4	5	6	$\cdots$
fib_n	0	1	1	2	3	5	8	$\cdots$
add_n	0	0	1	2	4	7	12	$\cdots$

Note

$$\mathrm{add}_n = \begin{cases} 0 & \text{if } n = 0, 1, \\ \mathrm{add}_{n-1} + \mathrm{add}_{n-2} + 1 & \\ = \mathrm{fib}_{n+1} - 1 & \text{if } n > 1 \end{cases}$$

Note

If the function is called with n as parameter, there may be $n+1$ activation records (stack frames) present on the stack. Compared to this there are only constant number of variables in the iterative program.

A nonRecursive C Function

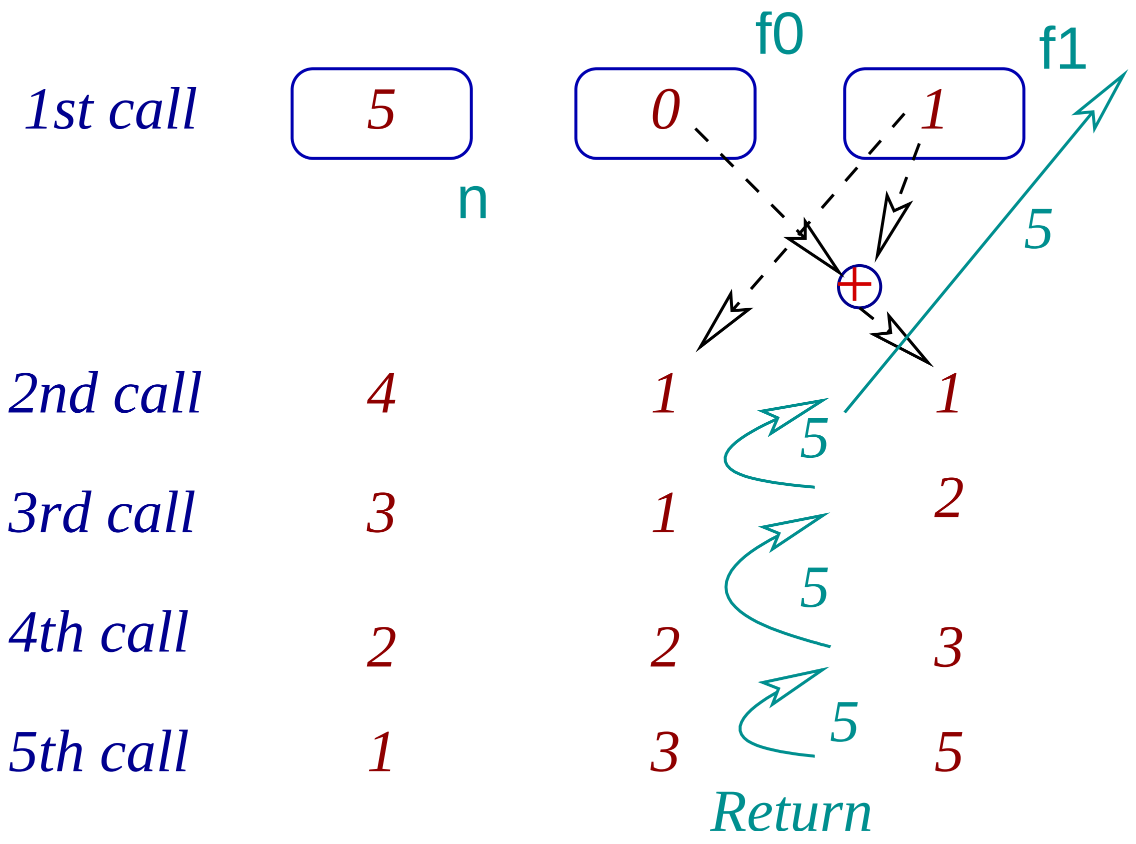
```
int fib(int n){ // fibonacciF.c
    int f0=0, f1=1, i;

    if(n < 2) return n ;
    for(i=2; i<=n; ++i) f1 += f0, f0 = f1 - f0;
    return f1 ;
}
```

An Efficient Recursive Function

We can write a recursive C function that will compute like the iterative program. This function has three parameters and is called as `fib(n, 0, 1)`, where 0 and 1 are base values corresponding to fib(0) and fib(1).

Efficient Recursive Function

```
int fib(int n, int f0, int f1) {
    if(n == 0) return f0 ;
    if(n == 1) return f1 ;
    return fib(n-1, f1, f1+f0);
}
```



1st call
5
0
1
f0
f1
n
5
2nd call
4
1
1
5
3rd call
3
1
2
5
4th call
2
2
3
5
5th call
1
3
5
Return

Program

```
#include <stdio.h>
int fib(int, int, int) ;
int main() // fibonacciFR2.c
{
    int n ;

    printf("Enter a non-ve integer: ") ;
    scanf("%d", &n) ;
    printf("fib(%d)=%d\n",n,fib(n,0,1));
```

```
    return 0;
}
int fib(int n, int f0, int f1) {
    if(n == 0) return f0 ;
    if(n == 1) return f1 ;
    return fib(n-1, f1, f1+f0);
}
```

Static Variable

- A `static` variable name is local to the function. It is not directly visible from out side.
- But unlike an automatic variable, it does not evaporate when the control comes out of the function. It remains dormant with its current value frozen.

Static Variable

- If the function is invoked again, the static variable is available with its last updated value.
- It is not initialized every time the function is called.
- It does not have a new binding at every call. It is not allocated on the stack.

An Efficient Recursive Function

We can write a recursive C function with a dynamics similar to the previous one using static variables[a]. This function takes one parameter `fib(n)`.

[a] This function is not thread safe in a multi threading environment.

```
int fib(int n) {
    static int f0=0, f1=1;

    if(n == 0) return f0 ;
    if(n == 1) { // why this step?
       int temp = f1 ;
       f0 = 0, f1 = 1;
       return temp ;
    }
    f1 += f0, f0 = f1 - f0;
    return fib(n-1);
} // fibonacciFR3.c
```

Static Initialized

Call	Function	n	fib0	fib1
1st Call	fibRecIter(5)	*5*	*0*	*1*
2nd Call	fibRecIter(4)	*4*	*1*	*1*
3rd Call	fibRecIter(3)	*3*	*1*	*2*
4th Call	fibRecIter(2)	*2*	*2*	*3*
5th Call	fibRecIter(1)	*1*	*3*	*5*

```
#include <stdio.h>
int fib(int) ;
int main() // fibonacciFR3.c
{
    int n ;

    printf("Enter a non-ve integer: ") ;
    scanf("%d", &n) ;
    printf("fib(%d) = %d\n", n, fib(n)) ;
    return 0;
}
int fib(int n) {
    static int f0=0, f1=1;
```

```
    if(n == 0) return f0 ;
    if(n == 1) { // why this step?
       int temp = f1 ;
       f0 = 0, f1 = 1;
       return temp ;
    }
    f1 += f0, f0 = f1 - f0;
    return fib(n-1);
}
```

Global Variable

Similar function can be written using global variable. But we strongly discourage it.

$\binom{n}{r}$

Consider the following inductive definition of the number of choices of r distinct objects from a collection of n distinct objects,

$$\binom{n}{r} = \begin{cases} 1, & \text{if } n = r \text{ or } r = 0, \\ \binom{n-1}{r} + \binom{n-1}{r-1}, & \text{if } 0 < r < n. \end{cases}$$

Note

Verify that a direct encoding of this definition to a C function is very inefficient. Use the concept of Pascal's triangle and an 1-D array of type `int` to compute $\binom{n}{r}$ efficiently.

Pascal's Triangle for $\binom{n}{r}$

$r \rightarrow$	0	1	2	3	4	5	6	7	$\cdots$
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	**4** $\searrow$	**6** $\downarrow$	4	1				
5	1	5	**10**	10	5	1			
6	1	6	15	20	15	6	1		
$n \uparrow$				$\cdots$					

Note

- One row of the Pascal's Triangle can be stored in a 1-D array of positive integers.
- $\binom{n+1}{r}$ for all r, $0 \leq r \leq n+1$, can be computed from $\binom{n}{r}$ for all r, $0 \leq r \leq n$.
- The same array can be reused.

Computation: An Example

$\binom{5}{r}$:

1	5	10	10	5	1	...

$\Downarrow$

$\binom{6}{r}$:

1	6	15	20	15	6	1