

# Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, .....

In mathematical terms, the sequence  $F_n$  of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \text{ and } F_1 = 1.$$

Write a function `int fib(int n)` that returns  $F_n$ . For example, if  $n = 0$ , then `fib()` should return 0. If  $n = 1$ , then it should return 1. For  $n > 1$ , it should return  $F_{n-1} + F_{n-2}$

Following are different methods to get the  $n$ th Fibonacci number.

## Method 1 ( Use recursion )

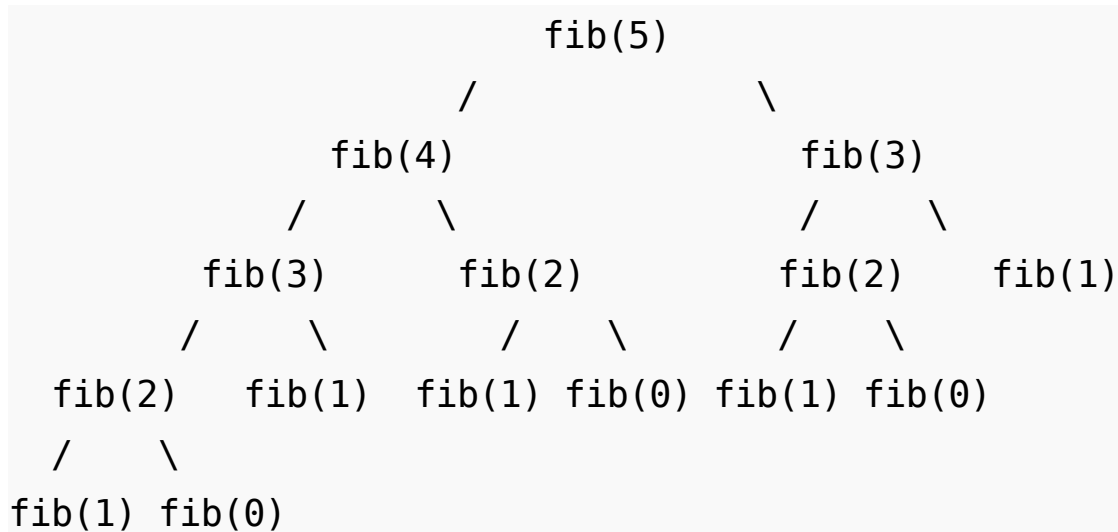
A simple method that is a direct recursive implementation mathematical recurrence relation given above.

```
#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

*Time Complexity:*  $T(n) = T(n-1) + T(n-2)$  which is exponential.

We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.



*Extra Space:*  $O(n)$  if we consider the function call stack size, otherwise  $O(1)$ .

## Method 2 ( Use Dynamic Programming )

We can avoid the repeated work done in the method 1 by storing the Fibonacci numbers calculated so far.

```
#include<stdio.h>
```

```
int fib(int n)
{
    /* Declare an array to store Fibonacci numbers. */
    int f[n+1];
    int i;

    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers in the series
           and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}
```

```
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

*Time Complexity:*  $O(n)$

*Extra Space:*  $O(n)$

### Method 3 ( Space Optimized Method 2 )

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

```
#include<stdio.h>
int fib(int n)
{
    int a = 0, b = 1, c, i;
    if( n == 0)
        return a;
    for (i = 2; i <= n; i++)
    {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

*Time Complexity:*  $O(n)$

*Extra Space:*  $O(1)$

## Method 4 ( Using power of the matrix $\begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}$ )

This another  $O(n)$  which relies on the fact that if we  $n$  times multiply the matrix  $M = \begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}$  to itself (in other words calculate  $\text{power}(M, n)$ ), then we get the  $(n+1)$ th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

```
#include <stdio.h>
```

```
/* Helper function that multiplies 2 matrices F and M o
puts the multiplication result back to F[][] */
```

```
void multiply(int F[2][2], int M[2][2]);
```

```
/* Helper function that calculates F[][] raise to the po
result in F[][]
```

```
Note that this function is desinged only for fib() and
power function */
```

```
void power(int F[2][2], int n);
```

```
int fib(int n)
```

```
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
```

```
    return F[0][0];
```

```
}
```

```
void multiply(int F[2][2], int M[2][2])
```

```
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];
```

```
F[0][0] = x;
```

```
F[0][1] = y;
```

```
F[1][0] = z;
```

```
F[1][1] = w;
```

```


}

void power(int F[2][2], int n)
{
    int i;
    int M[2][2] = {{1,1},{1,0}};

    // n - 1 times multiply the matrix to {{1,0},{0,1}}
    for (i = 2; i <= n; i++)
        multiply(F, M);
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```



*Time Complexity:*  $O(n)$

*Extra Space:*  $O(1)$

### Method 5 ( Optimized Method 4 )

The method 4 can be optimized to work in  $O(\log n)$  time complexity. We can do recursive multiplication to get  $\text{power}(M, n)$  in the previous method (Similar to the optimization done in [this post](#))

```

#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

/* function that returns nth Fibonacci number */
int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
}

```

```

    return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
    if( n == 0 || n == 1)
        return;
    int M[2][2] = {{1,1},{1,0}};

    power(F, n/2);
    multiply(F, M);

    if (n%2 != 0)
        multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
    int x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z =  F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w =  F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(9));
    getchar();
    return 0;
}

```

**Time Complexity:  $O(\text{Log}n)$**

**Extra Space:**  $O(\text{Log}n)$  if we consider the function call stack size, otherwise  $O(1)$ .