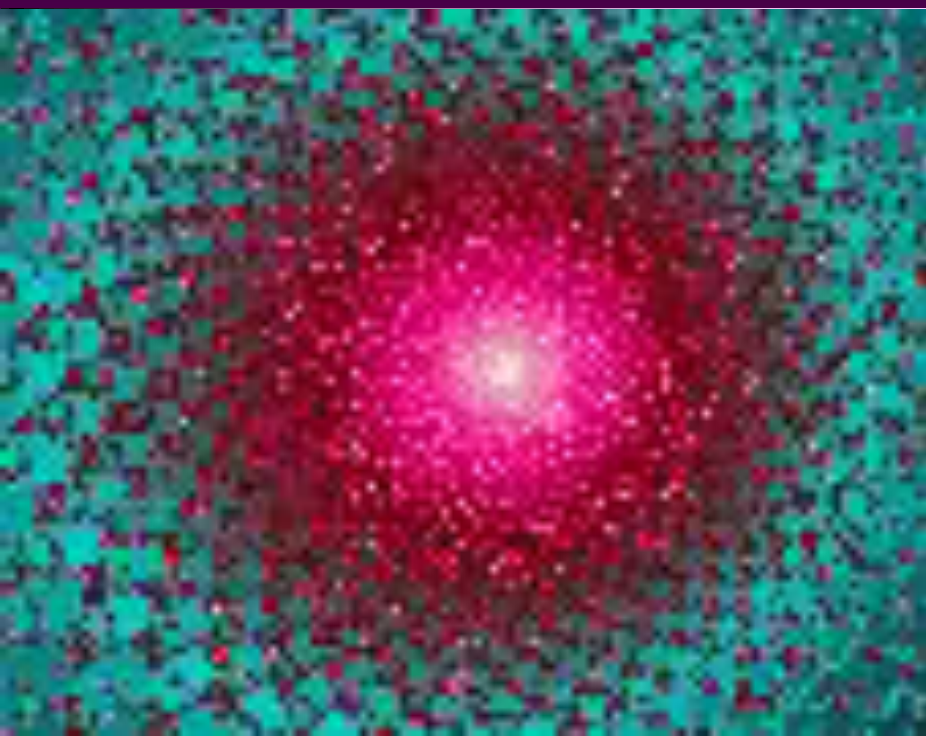


# Recursion

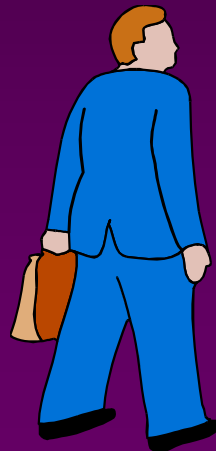


# Ex. 1: The Handshake Problem

There are  $n$  people in a room. If each person shakes hands once with every other person. What is the total number  $h(n)$  of handshakes?

$$h(n) = h(n-1) + n-1$$

$$h(4) = h(3) + 3 \quad h(3) = h(2) + 2 \quad h(2) = 1$$



$$h(n): \text{Sum of integer from } 1 \text{ to } n-1 = n(n-1) / 2$$

# Recursion

- ✉ In some problems, it may be natural to define the problem in terms of the problem itself.
- ✉ Recursion is useful for problems that can be represented by a **simpler version** of the same problem.
- ✉ Example: the factorial function

$$6! = 6 * 5 * 4 * 3 * 2 * 1$$

We could write:

$$6! = 6 * 5!$$

## Example 2: factorial function

In general, we can express the factorial function as follows:

$$n! = n * (n-1)!$$

Is this correct? Well... almost.

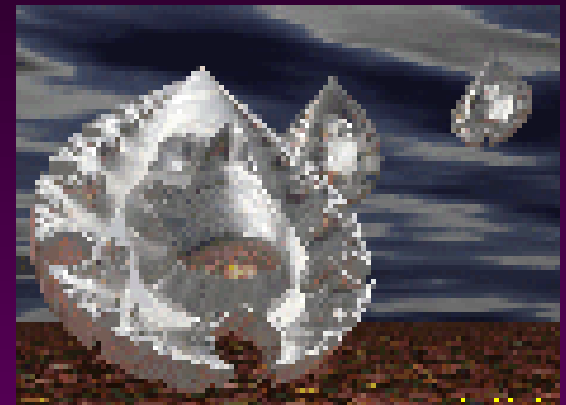
The factorial function is only defined for *positive* integers. So we should be a bit more precise:

$$\begin{array}{ll} n! = 1 & \text{(if } n \text{ is equal to 1)} \\ n! = n * (n-1)! & \text{(if } n \text{ is larger than 1)} \end{array}$$

# factorial function

The C++ equivalent of this definition:

```
int fac(int numb){  
    if (numb<=1)  
        return 1;  
    else  
        return numb * fac(numb-1);  
}
```



***recursion* means that a function calls itself**



# factorial function

☒ Assume the number typed is 3, that is, numb=3.

fac(3) :

3 <= 1 ?

No.

fac(3) = 3 \* fac(2)

fac(2) :

2 <= 1 ?

No.

fac(2) = 2 \* fac(1)

fac(1) :

1 <= 1 ?

Yes.

return 1

fac(2) = 2 \* 1 = 2

return fac(2)

fac(3) = 3 \* 2 = 6

return fac(3)

fac(3) has the value 6

```
int fac(int numb) {
    if(numb<=1)
        return 1;
    else
        return numb * fac(numb-1);
}
```

# factorial function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

## Recursive solution

```
int fac(int numb) {  
    if (numb <= 1)  
        return 1;  
    else  
        return numb * fac (numb-1) ;  
}
```

## Iterative solution

```
int fac(int numb) {  
    int product=1;  
    while (numb > 1) {  
        product *= numb;  
        numb--;  
    }  
    return product;  
}
```

# Recursion

We have to pay a price for recursion:

- calling a function **consumes more time and memory** than adjusting a loop counter.
- high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)



# Recursion

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)  
...
```



Oops!

```
int result = 1;  
while(result >0) {  
...  
result++;  
}
```



Oops!

# Recursion

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){  
    return numb * fac(numb-1);  
}
```

**Oops!**  
**No termination condition**

Or:

```
int fac(int numb){  
    if (numb<=1)  
        return 1;  
    else  
        return numb * fac(numb+1);  
}
```

**Oops!**

# Recursion

We must always make sure that the recursion *bottoms out*:

- ✉ A recursive function must contain **at least one non-recursive branch**.
- ✉ The recursive calls must eventually lead to a non-recursive branch.

# Recursion

- ✉ Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- ✉ The smallest example of the same task has a non-recursive solution.

Example: The factorial function

$$n! = n * (n-1)! \text{ and } 1! = 1$$

# How many pairs of rabbits can be produced from a single pair in a year's time?

## ✉ Assumptions:

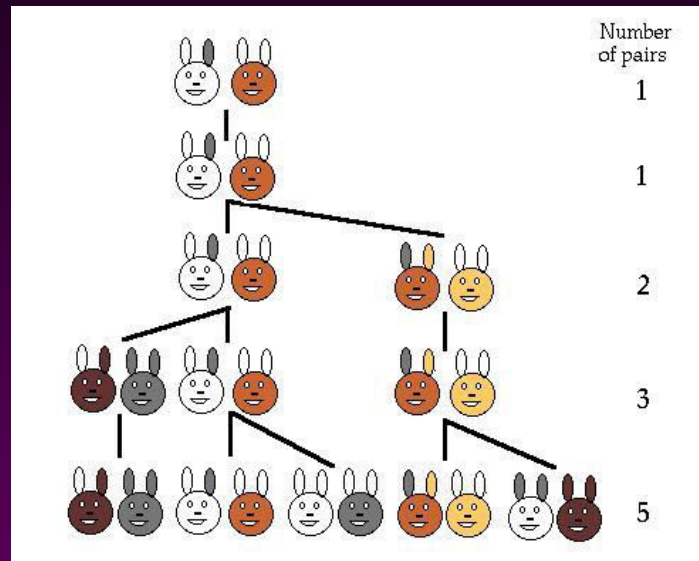
- Each pair of rabbits produces a new pair of offspring every month;
- each new pair becomes fertile at the age of one month;
- none of the rabbits dies in that year.

## ✉ Example:

- After 1 month there will be 2 pairs of rabbits;
- after 2 months, there will be 3 pairs;
- after 3 months, there will be 5 pairs (since the following month the original pair and the pair born during the first month will both produce a new pair and there will be 5 in all).



# Population Growth in Nature



- ✉ Leonardo Pisano (Leonardo Fibonacci = Leonardo, son of Bonaccio) proposed the sequence in 1202 in *The Book of the Abacus*.
- ✉ Fibonacci numbers are believed to model nature to a certain extent, such as Kepler's observation of leaves and flowers in 1611.

# Direct Computation Method

## ✉ Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

where each number is the sum of the preceding two.

## ✉ Recursive definition:

$$\blacksquare F(0) = 0;$$

$$\blacksquare F(1) = 1;$$

$$\blacksquare F(\text{number}) = F(\text{number}-1) + F(\text{number}-2);$$





Back



Forward



Reload



Home



Search



Netscape



Print



Security



Stop



Bookmarks



Location:

<http://www.ee.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibnat.html>

What's Related

# Fibonacci Numbers and Nature








This page has been split into TWO PARTS.

This, the **first**, looks at the Fibonacci numbers and why they appear in various "family trees" and patterns of spirals of leaves and seeds.

The second page then examines why the golden section is used by nature in some detail, including animations of growing plants.

## Contents of this Page

The  line means there is a Things to do investigation at the end of the section.

- [Fibonacci's Rabbits..](#)  
    ..and Dudeney's Cows
- [Honeybees, Fibonacci numbers and Family Trees](#) 
- [Fibonacci Numbers and the golden number](#) 
- [The Fibonacci Rectangles and Shell Spirals](#)
- [Fibonacci numbers and branching plants](#)
- [Petals on flowers](#)
- [Seed heads](#)
- [Pine cones](#) 
- [Leaf arrangements](#) 
- [Fibonacci Fingers?](#)
- [A quote from Coxeter on Phyllotaxis](#)
- [References](#)
- [Other WWW links on Phyllotaxis, the Fibonacci Numbers and Nature](#)

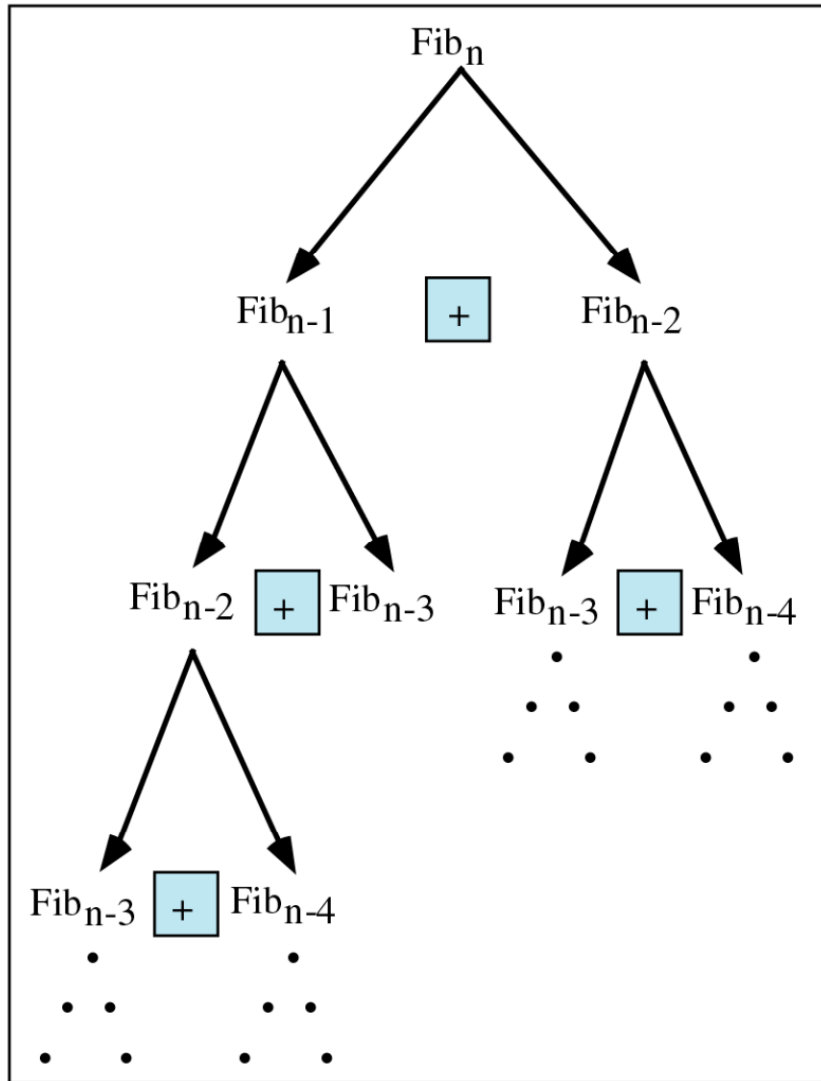


# Example 3: Fibonacci numbers

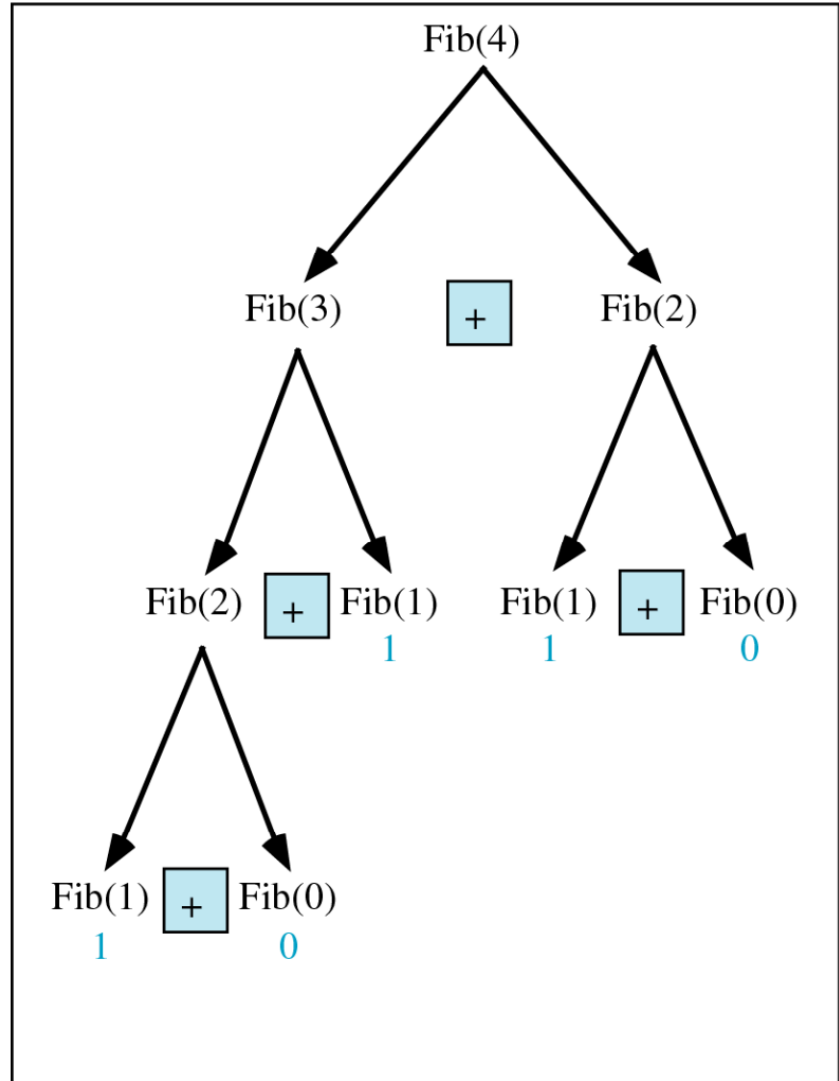
```
//Calculate Fibonacci numbers using recursive function.  
//A very inefficient way, but illustrates recursion well
```

```
int fib(int number)  
{  
    if (number == 0) return 0;  
    if (number == 1) return 1;  
    return (fib(number-1) + fib(number-2));  
}
```

```
int main(){           // driver function  
    int inp_number;  
    cout << "Please enter an integer: ";  
    cin >> inp_number;  
    cout << "The Fibonacci number for "<< inp_number  
           << " is "<< fib(inp_number)<<endl;  
    return 0;  
}
```



(a)  $\text{Fib}(n)$



(b)  $\text{Fib}(4)$

# Trace a Fibonacci Number

✉ Assume the input number is 4, that is, num=4:

**fib(4) :**

4 == 0 ? No;    4 == 1? No.

fib(4) = fib(3) + fib(2)

**fib(3) :**

3 == 0 ? No; 3 == 1? No.

fib(3) = fib(2) + fib(1)

**fib(2) :**

2 == 0? No; 2==1? No.

fib(2) = fib(1)+fib(0)

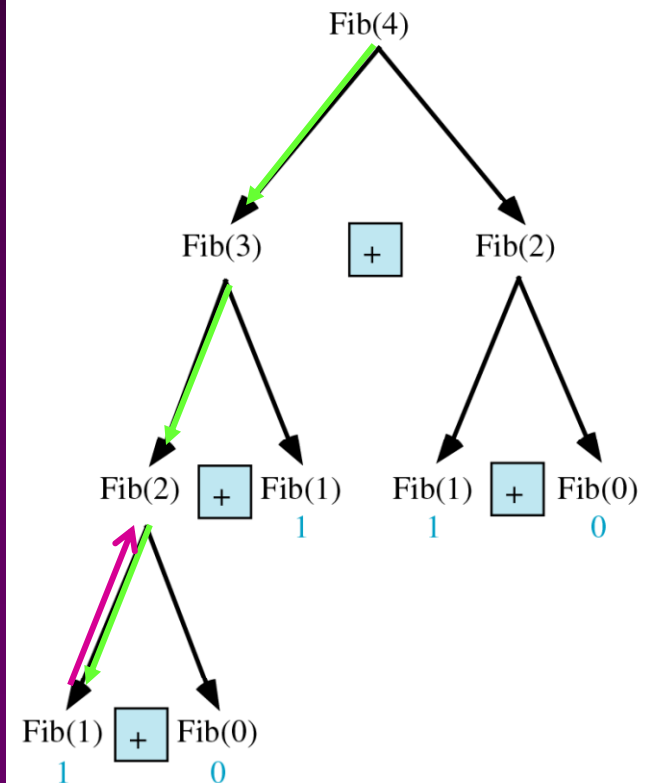
**fib(1) :**

1== 0 ? No; 1 == 1? Yes.

fib(1) = 1;

return fib(1);

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
        (fib(num-1)+fib(num-2)) ;
}
```



# Trace a Fibonacci Number

```

fib(0):
    0 == 0 ? Yes.
    fib(0) = 0;
    return fib(0);

```

```

fib(2) = 1 + 0 = 1;
return fib(2);

```

```

fib(3) = 1 + fib(1)

```

```

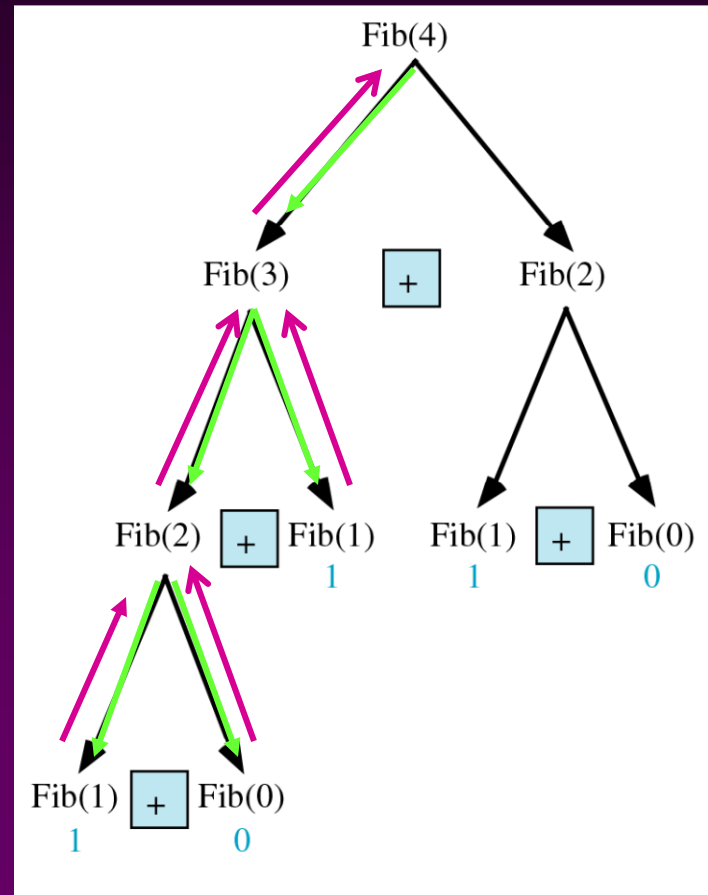
fib(1):
    1 == 0 ? No; 1 == 1? Yes
    fib(1) = 1;
    return fib(1);

```

```

fib(3) = 1 + 1 = 2;
return fib(3)

```



# Trace a Fibonacci Number

**fib(2) :**

2 == 0 ? No; 2 == 1? No.

fib(2) = fib(1) + fib(0)

**fib(1) :**

1 == 0 ? No; 1 == 1? Yes.

fib(1) = 1;

return fib(1);

**fib(0) :**

0 == 0 ? Yes.

fib(0) = 0;

return fib(0);

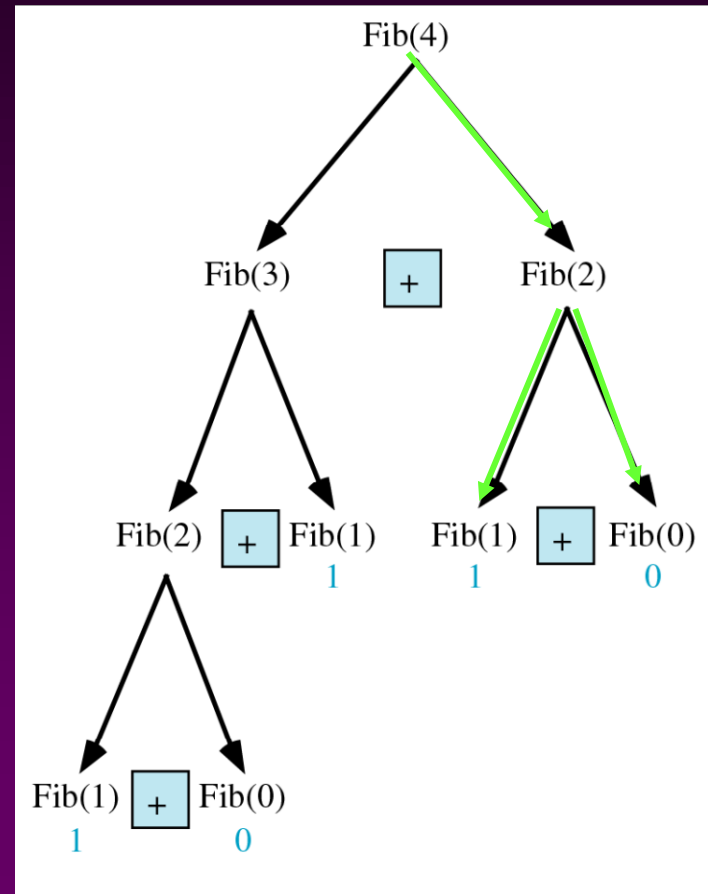
fib(2) = 1 + 0 = 1;

return fib(2);

fib(4) = fib(3) + fib(2)

= 2 + 1 = 3;

return fib(4);



# Example 4: Fibonacci number w/o recursion

```
//Calculate Fibonacci numbers iteratively  
//much more efficient than recursive solution
```

```
int fib(int n)  
{  
    int f[100];  
    f[0] = 0; f[1] = 1;  
    for (int i=2; i<= n; i++)  
        f[i] = f[i-1] + f[i-2];  
    return f[n];  
}
```

# Fibonacci Numbers

- ✉ Fibonacci numbers can also be represented by the following formula.

$$F_n = \frac{1}{\sqrt{5}} \left( \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right)$$

# Example 5: Binary Search

## ■ Search for an element in an array

- 📁 Sequential search

- 📁 Binary search

[see applet](#)

## ■ Binary search

- 📁 Compare the search element with the middle element of the array

- 📁 If not equal, then apply binary search to half of the array (if not empty) where the search element would be.



# Binary Search with Recursion

```
// Searches an ordered array of integers using recursion
int bsearchr(const int data[], // input: array
             int first,       // input: lower bound
             int last,        // input: upper bound
             int value        // input: value to find
) // output: index if found, otherwise return -1

{ //cout << "bsearch(data, "<<first<< ", last "<< ", "<<value << "); "<<endl;
  int middle = (first + last) / 2;
  if (data[middle] == value)
    return middle;
  else if (first >= last)
    return -1;
  else if (value < data[middle])
    return bsearchr(data, first, middle-1, value);
  else
    return bsearchr(data, middle+1, last, value);
}
```

# Binary Search

```
int main() {  
    const int array_size = 8;  
    int list[array_size]={1, 2, 3, 5, 7, 10, 14, 17};  
    int search_value;  
  
    cout << "Enter search value: ";  
    cin >> search_value;  
    cout << bsearchr(list,0,array_size-1,search_value)  
        << endl;  
  
    return 0;  
}
```

# Binary Search w/o recursion

```
// Searches an ordered array of integers
int bsearch(const int data[], // input: array
            int size,        // input: array size
            int value         // input: value to find
            ){               // output: if found, return
                             // index; otherwise, return -1

    int first, last, upper;
    first = 0;
    last = size - 1;
    while (true) {
        middle = (first + last) / 2;
        if (data[middle] == value)
            return middle;
        else if (first >= last)
            return -1;
        else if (value < data[middle])
            last = middle - 1;
        else
            first = middle + 1;
    }
}
```

# Recursion General Form

✉ How to write recursively?

```
int recur_fn(parameters) {  
    if(stopping condition)  
        return stopping value;  
    // other stopping conditions if needed  
    return function of recur_fn(revised parameters)  
}
```



# Example 6: exponential fun

☒ How to write `exp(int numb, int power)` recursively?

```
int exp(int numb, int power) {  
    if(power == 0)  
        return 1;  
    return numb * exp(numb, power - 1);  
}
```

# Example 6: number of zero

✉ Write a recursive function that counts the number of zero digits in an integer

✉ `zeros(10200)` returns 3.

```
int zeros(int numb) {
    if(numb==0)                // 1 digit (zero/non-zero) :
        return 1;             // bottom out.
    else if(numb < 10 && numb > -10)
        return 0;
    else                       // > 1 digits: recursion
        return zeros(numb/10) + zeros(numb%10);
}
```

```
zeros(10200)
zeros(1020)      + zeros(0)
zeros(102)        + zeros(0) + zeros(0)
zeros(10)         + zeros(2) + zeros(0) + zeros(0)
zeros(1) + zeros(0) + zeros(2) + zeros(0) + zeros(0)
```

# Problem Solving Using Recursion

Let us consider a simple problem of printing a message for  $n$  times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for  $n-1$  times. The second problem is the same as the original problem with a smaller size. The base case for the problem is  $n==0$ . You can solve this problem using recursion as follows:

```
void nPrintln(char * message, int times)
{
    if (times >= 1) {
        cout << message << endl;
        nPrintln(message, times - 1);
    } // The base case is n == 0
}
```

# Think Recursively

Many of the problems can be solved easily using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
bool isPalindrome(const char * const s)
{
    if (strlen(s) <= 1) // Base case
        return true;
    else if (s[0] != s[strlen(s) - 1]) // Base case
        return false;
    else
        return isPalindrome(substring(s, 1, strlen(s) - 2));
}
```



# Recursive Helper Methods

The preceding recursive isPalindrome method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a helper method:

```
bool isPalindrome(const char * const s, int low, int high)
{
    if (high <= low) // Base case
        return true;
    else if (s[low] != s[high]) // Base case
        return false;
    else
        return isPalindrome(s, low + 1, high - 1);
}

bool isPalindrome(const char * const s)
{
    return isPalindrome(s, 0, strlen(s) - 1);
}
```

# Example 7: Towers of Hanoi



[see applet](#)

- Only one disc could be moved at a time
- A larger disc must never be stacked above a smaller one
- One and only one extra needle could be used for intermediate storage of discs

# Towers of Hanoi

```
void hanoi(int from, int to, int num)
{
    int temp = 6 - from - to; //find the temporary
                              //storage column

    if (num == 1){
        cout << "move disc 1 from " << from
              << " to " << to << endl;
    }
    else {
        hanoi(from, temp, num - 1);
        cout << "move disc " << num << " from " << from
              << " to " << to << endl;
        hanoi(temp, to, num - 1);
    }
}
```

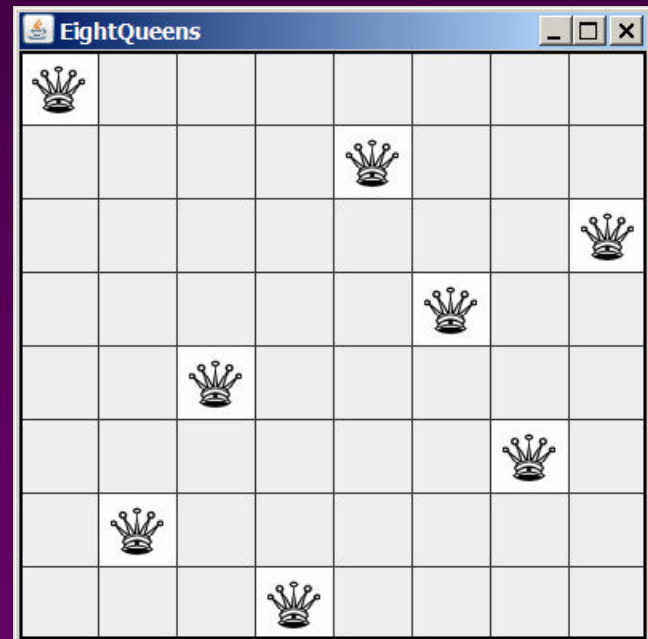
# Towers of Hanoi

```
int main() {  
    int num_disc;    //number of discs  
  
    cout << "Please enter a positive number (0 to quit)";  
    cin >> num_disc;  
  
    while (num_disc > 0){  
        hanoi(1, 3, num_disc);  
        cout << "Please enter a positive number ";  
        cin >> num_disc;  
    }  
    return 0;  
}
```

# Eight Queens

Place eight queens on the chessboard such that no queen attacks any other one.

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3



```
bool empty(int t[], int row, int col) {
    for( int j = 0; j < row; j++) {
        if (t[j] == col)           //same column
            return false;
        if (abs(t[j] - col) == (row - j)) //on cross
line            return false;
    }
    return true;
}

bool queens(int t[], int row, int col) {
    if (row == SIZE) // found one answer
        return true;
    for (col = 0; col < SIZE; col++)
    {
        t[row] = col;
        if (empty(t, row, col) && queens(t, row + 1, 0))
            return true;
    }
    return false;
}
```

```

void print(int t[]){
    // print solution
    for(int i = 0; i < SIZE; i++) {
        for(int j = 0; j < SIZE; j++) {
            if (j == t[i])
                cout << " Q ";
            else
                cout << " _ ";
        }
        cout << endl;
    }
}

int main() {
    int t[SIZE]={0};
    for (int i= 0; i <SIZE; i++){
        t[0] = i; //on first row, Queen on different
        column
        cout << endl << endl <<"i is: " << i << endl;
        if (queens(t, 1,0))
            print(t);
    }
}

```