

Cuaderno de problemas  
Fundamentos de algorítmia.

Recursivos

Prof. Isabel Pita

12 de noviembre de 2020

# Índice

<b>1. Jugando con los dígitos.</b>	<b>3</b>
1.1. Objetivos del problema . . . . .	4
1.2. Ideas generales. . . . .	4
1.3. Coste de la solución . . . . .	5
1.4. Modificaciones al problema. . . . .	5
1.5. Implementaciones. . . . .	5
<b>2. Transformar de decimal a binario.</b>	<b>8</b>
2.1. Objetivos del problema . . . . .	9
2.2. Ideas generales. . . . .	9
2.3. Algunas cuestiones sobre implementación. . . . .	9
2.4. Coste de la solución. . . . .	10
2.5. Modificaciones al problema. . . . .	10
2.6. Implementaciones. . . . .	10
<b>3. Cambio climático.</b>	<b>12</b>
3.1. Objetivos del problema. . . . .	13
3.2. Ideas generales. . . . .	13
3.3. Algunas cuestiones sobre implementación. . . . .	13
3.4. Ideas detalladas. . . . .	13
3.5. Coste de la solución. . . . .	13
3.6. Modificaciones al problema. . . . .	14
3.7. Implementación. . . . .	14
<b>4. Fibonacci.</b>	<b>15</b>
4.1. Objetivos del problema. . . . .	16
4.2. Ideas generales. . . . .	16
4.3. Algunas cuestiones sobre implementación. . . . .	16
4.4. Ideas detalladas. . . . .	16
4.5. Errores frecuentes. . . . .	16
4.6. Coste de la solución. . . . .	17
4.7. Implementación. . . . .	17
<b>5. Números combinatorios.</b>	<b>20</b>
5.1. Objetivos del problema. . . . .	21
5.2. Ideas generales. . . . .	21
5.3. Algunas cuestiones sobre implementación. . . . .	22
5.4. Ideas detalladas. . . . .	22
5.5. Errores frecuentes. . . . .	22
5.6. Coste de la solución. . . . .	22
5.7. Implementación. . . . .	22
<b>6. Formas de sumar.</b>	<b>25</b>
6.1. Objetivos del problema. . . . .	26
6.2. Ideas generales. . . . .	26

## 1. Jugando con los dígitos.

### Jugando con los dígitos de un número

Vamos a modificar los dígitos de un número de la siguiente forma. Si el dígito es impar le restamos uno y si el dígito es par le sumamos uno.

*Requisitos de implementación.*

Resolver el problema utilizando primero una función recursiva no final que reciba un número entero y devuelva el número transformado según se indicó en el problema.

A continuación resolverlo utilizando una función recursiva final.

Probar ambas soluciones en el juez automático.

#### Entrada

La entrada comienza con el número de casos de prueba. A continuación cada caso se escribe en una línea y consiste en un número  $0 \leq N \leq 1.000.000.000$ .

#### Salida

Para cada caso de prueba se muestra en una línea el número obtenido sustituyendo cada dígito por su transformado.

#### Entrada de ejemplo

```
7
45637
555
2
90
81
0
11
```

#### Salida de ejemplo

```
54726
444
3
81
90
1
0
```

## 1.1. Objetivos del problema

- Practicar el tratamiento recursivo de números enteros positivos. Obtener los dígitos de un número.
- Practicar la recursión lineal, final y no final.

## 1.2. Ideas generales.

- El dígito de menor peso (dígito más a la derecha) de un número se obtiene como el resto de dividir el número entre 10. Para ello se utiliza el operador módulo.

```
int k = n % 10;
```

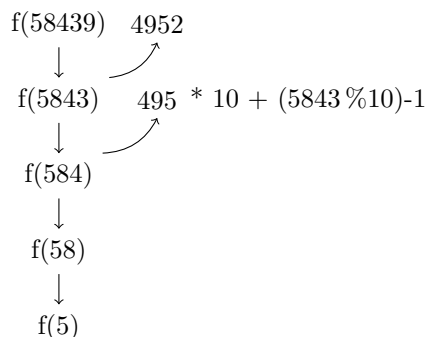
- Para eliminar de un número su dígito de menor peso lo dividimos entre 10.

```
int k = n / 10;
```

- Podemos realizar una implementación final, o no final del algoritmo. Empezamos explicando la implementación no final por resultar más sencilla ya que los dígitos de la solución deben quedar en el mismo orden que están en el número inicial. Si en el problema se pidiera invertir los dígitos del número, la solución final sería más sencilla.

- Implementación no final:

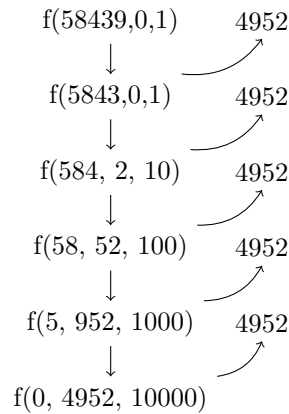
- Caso base. Conocemos la solución al problema cuando el número tiene un único dígito. En este caso se debe devolver el dígito transformado.
- Caso recursivo.
  - Llamada a la función recursiva (función sucesor). Dado un número  $n$ , las llamadas sucesivas deben realizarse con números más pequeños hasta alcanzar el caso base. Por lo tanto hacemos la llamada con el número sin su dígito de menor peso (el número dividido entre 10).
  - Función de combinación. La función recursiva devuelve el número dividido entre 10 ya transformado. Si este número lo multiplicamos por 10 y le sumamos el transformado del último dígito obtenemos el transformado del número.
  - Por ejemplo, la función que debe transformar el número 5843 llama a la función con el valor 584. Esta llamada devuelve el valor 495. Multiplicando este valor por 10 y sumándole el transformado del último dígito obtenemos el resultado de esta función.



- Implementación final:

- Para resolver el problema con un algoritmo final debemos utilizar los parámetros de entrada de la función recursiva para ir construyendo la solución. En este problema necesitamos dos parámetros, en uno llevamos el número transformado que vamos construyendo y en el otro la potencia de 10 por la que debemos multiplicar el número.
- Caso base. Cuando el número es cero. El número transformado debe estar calculado en el segundo parámetro. Se devuelve su valor.

- Caso recursivo. Se calculan los valores con los que debe realizarse la llamada a la función. El número con el que se realiza la llamada es el valor dividido por 10, al número transformado le añadimos el último dígito del número sumándole o restándole 1 según sea par o impar, por último multiplicamos la potencia que nos dan por 10.



### 1.3. Coste de la solución

- En la implementación no final, el caso base tiene coste constante ya que se trata de una instrucción de retorno. El caso recursivo tiene una llamada recursiva, en la que el parámetro se divide entre 10 y una operación de coste constante. La recurrencia se define como

$$T(n) = \begin{cases} c_1 & n == 0 \\ c_1 & n == 1 \\ T(n/10) + c_2 & n > 1 \end{cases}$$

Por lo tanto el coste está en el orden  $\mathcal{O}(\log_{10} n)$ , siendo  $n$  el parámetro de entrada.

El despliegue de la recurrencia puede consultarse ...

La recurrencia en el caso final es la misma.

### 1.4. Modificaciones al problema.

- Calcular el número de dígitos de un número.
- Calcular el inverso de un número.

### 1.5. Implementaciones.

- Implementación lineal no final:

```
#include <iostream>
#include <fstream>
#include <string>

// Solucion recursiva lineal no final

// Funcion utilizando expresion condicional
int transformadoNoFinal1 (int n) {
    if (n < 10) return ((n%2==1)?(n%10-1):(n%10+1));
    else {
        return transformadoNoFinal1(n/10) * 10 + ((n%2==1)?(n%10-1):(n%10+1));
    }
}

// Solucion utilizando instrucciones condicionales
int transformadoNoFinal2 (int n) {
```

```

    if (n < 10) {
        if (n%2 == 0) return (n%10+1);
        else return n%10-1;
    }
    else {
        int digito;
        if (n%2 == 0) digito = n%10+1;
        else digito = n%10-1;
        return transformadoNoFinal2(n/10) * 10 + digito;
    }
}

void resuelveCaso() {
    int a;
    std::cin >> a;
    std::cout << transformadoNoFinal2(a) << ' ';
    std::cout << transformadoFinal2(a,0,1) << '\n';
}

int main() {
    #ifndef DOMJUDGE
        std::ifstream in("datos1.txt");
        auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
    #endif

    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i) resuelveCaso();

    #ifndef DOMJUDGE // para dejar todo como estaba al principio
        std::cin.rdbuf(cinbuf);
        system("PAUSE");
    #endif
    return 0;
}

```

- Implementación lineal final (menos eficiente que la implementación lineal no final):

```

// Solucion recursiva lineal final

// Funcion utilizando expresion condicional
int transformadoFinal1 (int n, int nuevo, int pot) {
    if (n < 10) return ((n%2==1)?(n%10-1):(n%10+1))*pot + nuevo;
    else {
        return transformadoFinal1(n/10,((n%2==1)?(n%10-1):(n%10+1))*pot + nuevo,10*pot);
    }
}

// Solucion utilizando instrucciones condicionales
int transformadoFinal2 (int n, int nuevo, int pot) {
    int digito;
    if (n%2 == 0) digito = n%10+1;
    else digito = n%10-1;

    if (n < 10) {
        return digito*pot + nuevo;
    }
    else {
        return transformadoFinal2(n/10,digito*pot + nuevo,10*pot);
    }
}

```

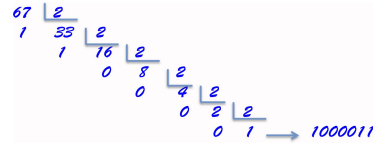
```
void resuelveCaso() {  
    int n;  
    std::cin >> n;  
    std::cout << decimalBinario(n,"") << '\n';  
}
```

## 2. Transformar de decimal a binario.

### Transformar un número decimal a binario

Con la revolución digital, el sistema de numeración en base dos o sistema de numeración binario adquiere una gran relevancia al ser el sistema utilizado por los ordenadores y las redes de comunicación. Todo informático necesita en algún momento obtener la representación binaria de un número.

En este problema desarrollaremos un conversor de números expresados en base 10 a números expresados en base 2.



*Requisitos de implementación.*

El conversor debe realizarse con una función que dado un número entero devuelva en una cadena de caracteres de tipo `std::string` su representación en binario.

Se utilizará una función `resuelveCaso` para leer el dato de entrada, llamar al conversor y escribir la cadena de salida.

#### Entrada

La entrada comienza con una línea en que se indica el número de casos de prueba. Cada caso consiste en un número entero positivo  $n$  ( $0 \leq N \leq 2^{31} - 1$ ).

#### Salida

Para cada caso de prueba se muestra en una línea la representación en binario del número.

#### Entrada de ejemplo

```
6
3
8
1
24
156
345
```

#### Salida de ejemplo

```
11
1000
1
11000
10011100
101011001
```

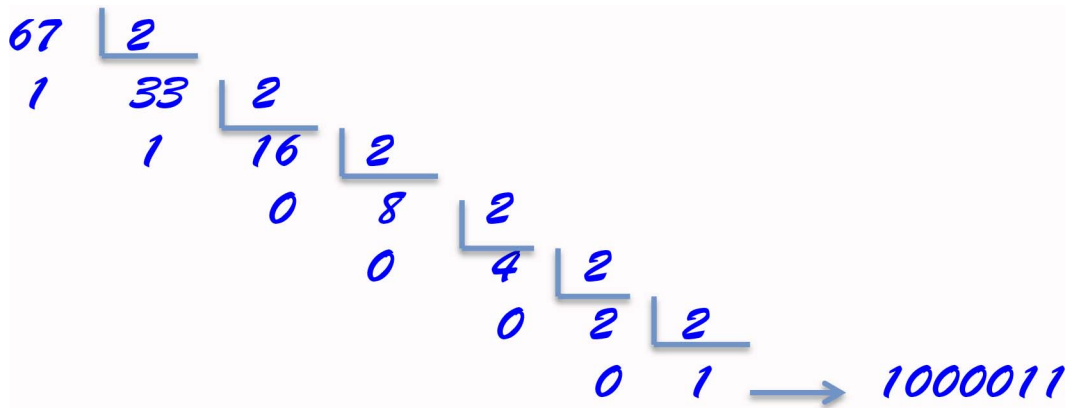


## 2.1. Objetivos del problema

- Practicar la recursión lineal, final y no final.
- Recordar el tipo `std::string`.

## 2.2. Ideas generales.

- Para transformar un número decimal a binario debemos dividirlo entre 2 hasta obtener cociente 1. A continuación formaremos el número en binario concatenando el último cociente con los restos de todas las divisiones realizadas en orden inverso a aquel en que se realizaron.



- Para resolver el problema de forma recursiva, podemos obtener la representación binaria de un número  $n$ , concatenando por la derecha el resto de dividir el número entre 2, al número binario obtenido al transformar  $n/2$  a una cadena binaria.

$$\text{bin}(n) = \begin{cases} "0" & n == 0 \\ "1" & n == 1 \\ \text{bin}(n/2) + \text{caracterNumerico}(n \% 2) & n > 1 \end{cases}$$

- El problema se puede resolver con recursión lineal final o con recursión lineal no final.
- Si seguimos el proceso descrito anteriormente obtenemos un algoritmo recursivo no final. Debemos realizar la llamada recursiva y con el resultado obtenido construir la nueva solución concatenando un nuevo dígito por la derecha.
- Otra posibilidad es ir obteniendo el número binario según se van realizando las llamadas recursivas. Para ello tenemos que concatenar los restos que vamos obteniendo al dividir, por el lado izquierdo de la cadena, para que el último cociente sea el que tenga más peso en el número binario. Para almacenar la cadena que vamos construyendo se utiliza un parámetro por valor.

## 2.3. Algunas cuestiones sobre implementación.

- Si se utiliza el operador `+` (concatenación) definido para el tipo `std::string`, se evita utilizar variables auxiliares ya que el valor de retorno se construye sin necesidad de almacenarlo.
  - Si necesitamos concatenar el carácter por la parte derecha de la cadena, contamos con el prototipo: `std::string operator+ (const std::string& lhs, char rhs);`.
  - Si necesitamos concatenar el carácter por la parte izquierda de la cadena, utilizamos la sobrecarga del operador: `std::string operator+ (char lhs, const std::string& rhs);`.
  - Para obtener el carácter numérico correspondiente a un dígito  $x$ , utilizaremos el casting `char('0'+x)`. Por ejemplo dado el dígito 3, al sumarle el código ASCII del carácter '0' obtenemos el código ASCII del carácter '3'. Este valor del código ASCII lo transformamos en un carácter utilizando un casting.

- En la implementación lineal final, se utiliza un parámetro de tipo `std::string` por valor. Esto significa que se realiza una copia de la cadena de caracteres en cada llamada recursiva. Si esta cadena es larga la solución no es eficiente ni en tiempo ni en espacio.

## 2.4. Coste de la solución.

- En la implementación no final, el caso base tiene coste constante ya que se trata de una instrucción de retorno. El caso recursivo tiene una llamada recursiva, en la que el parámetro se divide por la mitad y una operación de concatenación con coste amortizado constante ya que se añade un carácter a la cadena por la parte derecha. La recurrencia se define como

$$T(n) = \begin{cases} c_1 & n == 0 \\ c_1 & n == 1 \\ T(n/2) + c_2 & n > 1 \end{cases}$$

Por lo tanto el coste está en el orden  $\mathcal{O}(\log_2(n))$ , siendo  $n$  el parámetro de entrada.

El despliegue de la recurrencia puede consultarse ...

- En la implementación final, el caso base tiene coste logarítmico respecto al dato inicial de entrada, ya que se realiza una concatenación por la parte izquierda de la cadena (recordar que una cadena de caracteres se almacena en un vector, y para insertar un elemento al comienzo del vector es necesario desplazar todos los elementos a la derecha, y la cadena tiene longitud igual al número de veces que el dato de entrada se puede dividir entre 2). El caso recursivo tiene una llamada recursiva, en la que el parámetro se divide por la mitad y una operación de concatenación con coste logarítmico respecto al dato de entrada ya que se añade un carácter a la cadena por la parte izquierda. La recurrencia se define como

$$T(n) = \begin{cases} \log_2(n_i) & n == 0 \\ \log_2(n_i) & n == 1 \\ T(n/2) + \log_2(n_i) & n > 1 \end{cases}$$

Siendo  $n_i$  el valor de la llamada inicial a la función recursiva.

Por lo tanto el coste está en el orden  $\mathcal{O}(\log_2^2(n))$ , siendo  $n$  el parámetro de entrada.

El despliegue de la recurrencia puede consultarse ...

## 2.5. Modificaciones al problema.

- Calcular el número de dígitos de un número.
- Calcular el inverso de un número.

## 2.6. Implementaciones.

- Implementación lineal no final:

```
#include <iostream>
#include <fstream>
#include <string>

// Solucion recursiva lineal no final
std::string decimalBinario (int n) {
    if (n == 0) return "0";
    else if (n == 1) return "1";
    else return decimalBinario(n/2) + char('0'+n%2);
}

void resuelveCaso() {
    int n;
    std::cin >> n;
```

```

        std::cout << decimalBinario(n) << '\n';
    }

    int main() {
        #ifndef DOMJUDGE
            std::ifstream in("datos1.txt");
            auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
        #endif

        int numCasos;
        std::cin >> numCasos;
        for (int i = 0; i < numCasos; ++i) resuelveCaso();

        #ifndef DOMJUDGE // para dejar todo como estaba al principio
            std::cin.rdbuf(cinbuf);
            system("PAUSE");
        #endif
        return 0;
    }
}

```

- Implementación lineal final (menos eficiente que la implementación lineal no final):

```

// Solucion recursiva lineal final
std::string decimalBinario (int n, std::string bin) {
    if (n = 0) return '0' + bin;
    else if (n = 1) return '1' + bin;
    else return decimalBinario(n/2, char('0'+n%2)+bin);
}

void resuelveCaso() {
    int n;
    std::cin >> n;
    std::cout << decimalBinario(n,"") << '\n';
}


```







### 3. Cambio climatico.

## Cambio climático

El cambio climático se ha convertido en una preocupación mundial. Cada vez es mas frecuente oír comentar que en Madrid nunca nevaba en mayo, o que este verano ha sido el más caluroso desde hace 20 años.

Mi abuelo no está tan seguro. El opina que simplemente nos olvidamos de lo que pasó de un año para otro y que las temperaturas no varían tanto como dicen. Para poder demostrar su teoría ha conseguido la temperatura media de cada día del año durante varios años y ahora va a estudiar el primer y último día de cada uno de ellos en que se alcanzó una cierta temperatura. Con ello espera poder realizar una bonita gráfica en la que se vea que todos los años son semejantes.



vie 14		sáb 15			
12-18 h	viernes 14	00-06 h	06-12 h	12-18 h	18-24 h
					
29°C	24°C	19°C	27°C	27°C	22°C

### Entrada

La entrada comienza con el número de casos de prueba. Cada caso consta de dos líneas, en la primera se muestra el número de temperaturas medias tomadas durante un año y el valor de la temperatura que se quiere comprobar que se alcanza. En la línea siguiente se muestra el valor de las temperaturas.

El número de medidas tomadas es un valor  $0 \leq N \leq 1000000$ . Cada temperatura es un número entero  $-50 \leq t \leq 50$ .

### Salida

Para cada caso de prueba se muestra en una línea el número de días que han transcurrido hasta que se alcanza la temperatura pedida desde el comienzo del año por primera y última vez. La primera medida se alcanza en cero días. Si no se llega a alcanzar la temperatura pedida se escribirá el número total de días.

### Entrada de ejemplo

```
4
4 5
5 3 5 2
0 5

4 1
-8 0 -6 1
7 20
6 7 8 10 12 3 1
```

### Salida de ejemplo

```
0 2
0 0
3 3
7 7
```

### 3.1. Objetivos del problema.

- Practicar el recorrido recursivo de vectores de izquierda a derecha y de derecha a izquierda.
- Recordar la diferencia entre realizar una búsqueda en un vector ordenado y en un vector no ordenado.

### 3.2. Ideas generales.

- Para obtener el primer valor con el que se alcanza la temperatura pedida se puede implementar una función recursiva final que compruebe el valor de la izquierda del vector y si este no es igual o mayor que el valor pedido busque el valor en la parte derecha del vector de forma recursiva.
- Para obtener el último valor implementaremos otra función, también recursiva final. En este caso comenzaremos comprobando el último valor del vector y si este no es igual o mayor que el valor pedido buscaremos el valor en la parte izquierda del vector.

### 3.3. Algunas cuestiones sobre implementación.

- Para evitar hacer copia del vector, utilizaremos un parámetro `const&`. De esta forma el vector es el mismo en todas las llamadas recursivas.
- Para indicar que parte del vector se está considerando en la llamada, utilizamos un parámetro por valor de tipo entero. Este parámetro puede representar la posición en que comienza el vector o la posición en que finaliza. El otro extremo se considera fijo.
- Si quisiéramos considerar una parte interna del vector, en que ninguno de sus extremos coincidiese con el comienzo o final del vector en alguna de las llamadas recursivas, tendríamos que utilizar dos parámetros, uno para indicar la posición de comienzo y otro para indicar la posición del final. Este tipo de recursión se practicará en el tema de divide y vencerás.

### 3.4. Ideas detalladas.

- Si la llamada recursiva se realiza con la parte derecha del vector, el caso base se alcanza cuando el vector es vacío, o lo que es lo mismo cuando el punto de comienzo del vector que estamos considerando (el parámetro que hemos definido) coincide con el final del vector. En este caso el punto de comienzo del vector debe empezar en la primera posición del vector e ir aumentando.
- Si la llamada recursiva se realiza con la parte izquierda del vector, el caso base se alcanza, como en el caso anterior, cuando el vector es vacío. Pero en este caso, el vector es vacío cuando su comienzo es menor que cero. El punto de comienzo por lo tanto debe empezar en el último elemento del vector e ir disminuyendo.

### 3.5. Coste de la solución.

- La implementación de la función `sequentialSearchIz` tiene complejidad del orden de  $\mathcal{O}(n)$  siendo  $n$  el número de elementos del vector.

El caso base tiene coste constante ya que se trata de una instrucción de retorno. El caso recursivo tiene una llamada recursiva, en la que el parámetro se incrementa en una unidad y una operación constante para calcular el valor de los parámetros. La recurrencia se define como

$$T(n) = \begin{cases} c_1 & n == 0 \\ T(n-1) + c_2 & n > 0 \end{cases}$$

Observad que en la implementación el parámetro `n` es la posición en que comienza el vector que estamos considerando, mientras que en la recurrencia  $n$  es el número de elementos que tiene el vector considerado.

El despliegue de la recurrencia puede consultarse ....

- La implementación de la función `sequentialSearchDr` tiene también coste lineal respecto al número de elementos del vector. La recurrencia es igual que la planteada en la otra función.
- Si el vector estuviese ordenado deberíamos emplear el algoritmo de búsqueda binaria que se puede implementar con un coste  $\mathcal{O}(\log(n))$ . Este algoritmo emplea la técnica de divide y vencerás.

### 3.6. Modificaciones al problema.

- Buscar un elemento concreto.
- Buscar el valor máximo o el mínimo.
- Recorrer el vector completo realizando alguna operación sobre los valores. Por ejemplo, sumar todos los elementos de un vector.

### 3.7. Implementación.

```
// Solucion recursiva haciendo la llamada recursiva con la parte derecha del vector
int sequentialSearchIz (std::vector<int> const& v, int n, int x) {
    if (n = v.size()) return n; // Caso base. Vector vacio
    else {
        if (v[n] ≥ x) return n; // Caso base. Encontrado elemento
        else return sequentialSearchIz(v,n+1,x);
    }
}

// Solucion recursiva haciendo la llamada recursiva con la parte izquierda del vector
int sequentialSearchDr (std::vector<int> const& v, int n, int x) {
    if (n < 0) return (int)v.size(); // Caso base. Vector vacio
    else {
        if (v[n] ≥ x) return n; // Caso base. Encontrado el elemento
        else return sequentialSearchDr(v,n-1,x);
    }
}

void resuelveCaso() {
    int numElems, x;
    std::cin >> numElems >> x;
    std::vector<int> v(numElems);
    for (int &n : v) std::cin >> n;
    std::cout << sequentialSearchIz(v,0,x) << ' ' << sequentialSearchDr(v,v.size()-1,x) << '\n';
}

int main() {
    // Para la entrada por fichero.
    #ifndef DOMJUDGE
        std::ifstream in("datos1.txt");
        auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
    #endif

    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i) {
        resuelveCaso();
    }
    // Para restablecer entrada. Comentar para acepta el reto
    #ifndef DOMJUDGE // para dejar todo como estaba al principio
        std::cin.rdbuf(cinbuf);
        //system("PAUSE");
    #endif
    return 0;
}
```

## 4. Fibonacci.

### La sucesión de Fibonacci.

La sucesión de Fibonacci fue descrita por Leonardo da Vinci en el siglo XIII. Es famosa por sus numerosas aplicaciones, y porque aparece en muchas formaciones de la naturaleza.

La sucesión puede definirse de forma recursiva como sigue:

$$fib(n) = \begin{cases} 0 & n == 0 \\ 1 & n == 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

0 1 1 2 3 5 8 13 21  
34 55 89 144 233 377  
610 987 1597 2584 4181 6765  
10946 17711 28657 46368 75025  
121393 196418 317811 514229 832040 1346269  
2178309 3584578 5702887 9227465 14930352 2415117 39075169 63490321  
FIBONACCI FIBONACCI FIBONACCI FIBONACCI FIBONACCI

*Requisitos de implementación.*

Realizar una implementación con recursión múltiple y comprobar que el tiempo de ejecución es inaceptable para valores de entrada mayores que 50.

Realizar una implementación con recursión simple utilizando parámetros acumuladores. Estudiar el coste de esta solución.

#### Entrada

La entrada consta de una serie de casos. Cada caso se escribe en una línea y consiste en un número  $N$  ( $0 \leq N \leq 89$ ). El final de los casos se indica con el valor -1.

#### Salida

Para cada valor de entrada mostrar en una línea el elemento de la sucesión de Fibonacci en la posición  $N$ . La posición del primer elemento de la sucesión es cero.

#### Entrada de ejemplo

```
0
1
3
6
-1
```

#### Salida de ejemplo

```
0
1
2
8
```

#### 4.1. Objetivos del problema.

- Practicar la recursión múltiple.
- Observar el tiempo de ejecución de algoritmos exponenciales cuando crece el valor de los datos de entrada.
- Practicar la mejora de la eficiencia de algoritmos por medio de parámetros auxiliares.
- Observar la diferencia entre una implementación recursiva final y una implementación recursiva no final.

#### 4.2. Ideas generales.

- El algoritmo se puede implementar de forma recursiva aplicando la definición de la sucesión.
- La implementación con recursión múltiple tiene coste exponencial respecto al valor de entrada, por lo que su tiempo de ejecución es inaceptable para valores grandes del dato de entrada.
- Para conseguir coste lineal se puede realizar una implementación recursiva final añadiendo dos parámetros en los que se construye el valor de la sucesión o una implementación recursiva no final, en la que se devuelven los valores de la sucesión para los sucesivos valores de entrada como resultado de la función.

#### 4.3. Algunas cuestiones sobre implementación.

- La sucesión de Fibonacci crece muy rápidamente. En el enunciado se dice que los límites del valor de entrada son  $0 \leq n \leq 89$ . El valor 89 de la sucesión es 1779979416004714189. Para poder almacenar este valor debe utilizarse el tipo `long long int`.
- Puede utilizarse un alias del tipo con la instrucción: `using lli = long long int`.
- Deben declararse del tipo `lli` únicamente los parámetros y variables que almacenan valores de la sucesión y el tipo de retorno de la función. El valor de entrada debe declararse de tipo `int`, ya que pertenece al intervalo  $[0 \dots 89]$ .

#### 4.4. Ideas detalladas.

- Si se quiere realizar una función recursiva final se añaden dos parámetros por valor, en los que se construye la solución al realizar las llamadas.
  - En los casos base se devuelve el valor de estos parámetros, que debe tener calculada la solución ya que al ser una recursión final, el valor que se devuelve en el caso base será el valor final de la recursión.
  - Puede añadirse otro parámetro en que se lleve el valor de la sucesión que estamos construyendo, alcanzándose el caso base cuando este parámetro iguale el valor de la posición que se pide.
  - Si no se utiliza este parámetro, se disminuirá el valor de entrada llegando al caso base cuando este valor sea 0 o 1. Esto se puede hacer porque el valor de entrada sólo se utiliza para llevar cuenta del número de llamadas recursivas que quedan por realizar.
- Si se quiere realizar una función recursiva no final se devolverá como resultado de la función para un dato de entrada  $n$ , los valores de la sucesión para  $n$  y para  $n - 1$ . Ambos valores se agrupan en un tipo `struct`. Los casos base se alcanzan cuando  $n == 0$  o  $n == 1$ . En el caso recursivo después de realizar la llamada recursiva se utilizan los valores devueltos por la función para calcular los valores de retorno.

#### 4.5. Errores frecuentes.

- Olvidarse de declarar de tipo `long long int` algún parámetro, variable, o valor de retorno que almacene valores de la sucesión. Este error se detecta al ejecutar el programa para los valores mas altos de entrada.



## 4.6. Coste de la solución.

El coste de las diferentes implementaciones realizadas es diferente. Se indica el coste de cada implementación concreta.

- El coste de la solución que utiliza dos llamadas recursivas es exponencial respecto al valor de entrada. La recurrencia que define el coste es:

$$T(n) = \begin{cases} c_1 & n == 0 \\ c_1 & n == 1 \\ T(n-1) + T(n-2) + c_2 & n > 1 \end{cases}$$

siendo  $n$  el valor de entrada del algoritmo.

Para simplificar el cálculo de la recurrencia calculamos una cota superior del coste suponiendo que las dos llamadas recursivas se realizan con el mismo valor:

$$T(n) = \begin{cases} c_1 & n == 0 \\ c_1 & n == 1 \\ 2T(n-1) + c_2 & n > 1 \end{cases}$$

Se hace el despliegue de la recurrencia (ver Sección ....) se obtiene que el coste pertenece a  $\mathcal{O}(2^n)$ .

- El coste de las soluciones que utilizan parámetros acumuladores es lineal respecto al valor de entrada. La recurrencia que define el coste en los tres casos es:

$$T(n) = \begin{cases} c_1 & n == 0 \\ c_1 & n == 1 \\ T(n-1) + c_2 & n > 1 \end{cases}$$

donde  $n$  es

- En la primera solución  $n = n_p - x_p$  siendo  $n_p$  y  $x_p$  los valores de los correspondientes parámetros de entrada.
- En la segunda y tercera solución coincide con el parámetro de entrada  $n_p$ .

Explicación de la recurrencia:

- El coste en el caso base es constante porque se trata de instrucciones de asignación o retorno.
- En el caso recursivo se realiza una llamada a una función con el parámetro que define la recurrencia variando en una unidad. El coste de las instrucciones que acompañan a la llamada recursiva es constante ya que son instrucciones de asignación. Además el coste de calcular los parámetros de la función es constante.

Haciendo el despliegue de la recurrencia (ver Sección ....) se obtiene que el coste pertenece a  $\mathcal{O}(n)$ .

## 4.7. Implementación.

- Solución lineal final utilizando tres parámetros auxiliares:

```
using lli = long long int;
lli resolver(int n, int x, lli n1, lli n2)
{
    if (n == x) return n2;
    else return resolver(n, x+1, n1+n2, n1);
}

bool resuelveCaso() {
    int elem;
    std::cin >> elem;
    if (elem == -1) return false;
```

```

        std::cout << resolver(elem, 0, 1, 0) << '\n';
        return true;
    }

    int main() {

        #ifndef DOMJUDGE
            std::ifstream in("datos.txt");
            auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos
            #endif

            while (resuelveCaso())
                ;

            #ifndef DOMJUDGE // para dejar todo como estaba al principio
                std::cin.rdbuf(cinbuf);
                //system("PAUSE");
            #endif

            return 0;
    }

```

- Solución lineal final con dos parámetros auxiliares:

```

#include <iostream>
#include <fstream>

using lli = long long int;
lli resolver(int n, lli n1, lli n2)
{
    if (n == 0) return n2;
    else if (n == 1) return n1;
    else return resolver(n-1, n1+n2, n1);
}

bool resuelveCaso() {
    int elem;
    std::cin >> elem;
    if (elem == -1) return false;
    std::cout << resolver(elem, 1, 0) << '\n';
    return true;
}

```

- Solución lineal no final con dos parámetros por referencia:

```

using lli = long long int;
struct tsolucion {
    lli n1, n2;
};

tsolucion resolver(int n)
{
    tsolucion sol;
    if (n == 0) {sol.n1 = 0; sol.n2 = 0;}
    else if (n == 1) {sol.n1 = 1; sol.n2 = 0;}
    else {
        sol = resolver(n-1);
        lli aux = sol.n2;
        sol.n2 = sol.n1;
        sol.n1 = aux + sol.n1;
    };
    return sol;
}

```

```
}

bool resuelveCaso() {
    int elem;
    std::cin >> elem;
    if (elem == -1) return false;
    tsolucion sol = resolver(elem);
    std::cout << sol.n1 << '\n';
    return true;
}
```

## 5. Números combinatorios.

### Números combinatorios

Calcular el número combinatorio  $\binom{a}{b}$   
Sabiendo que

$$\binom{a}{b} = \begin{cases} 1 & b == 0 \\ 1 & a == b \\ \binom{a-1}{b-1} + \binom{a-1}{b} & b > 0 \wedge a \neq b \end{cases}$$

*Requisitos de implementación.*

Para mejorar el coste se generalizará la función recursiva con un nuevo parámetro de salida. El nuevo parámetro es una matriz de  $a$  filas y  $b$  columnas inicializada en la función `resuelveCaso` con el valor -1. En el elemento  $i,j$  de la matriz se guarda el número combinatorio  $\binom{i}{j}$ . Antes de realizar una llamada recursiva se comprueba si el valor que se quiere calcular ya está guardado en la matriz. En caso afirmativo se utiliza este valor, en caso negativo se realiza la llamada recursiva. Al terminar una llamada recursiva se debe actualizar el valor en la matriz para que pueda ser utilizado por otras llamadas.

#### Entrada

La entrada consiste en una serie de casos de prueba. Cada caso son dos números enteros positivos tales que:  $0 \leq b \leq a$ .

#### Salida

Para cada caso de prueba se escribe en una línea el número combinatorio  $\binom{a}{b}$ .

#### Entrada de ejemplo

```
5 2
5 3
5 4
7 4
```

#### Salida de ejemplo

```
10
10
5
35
```

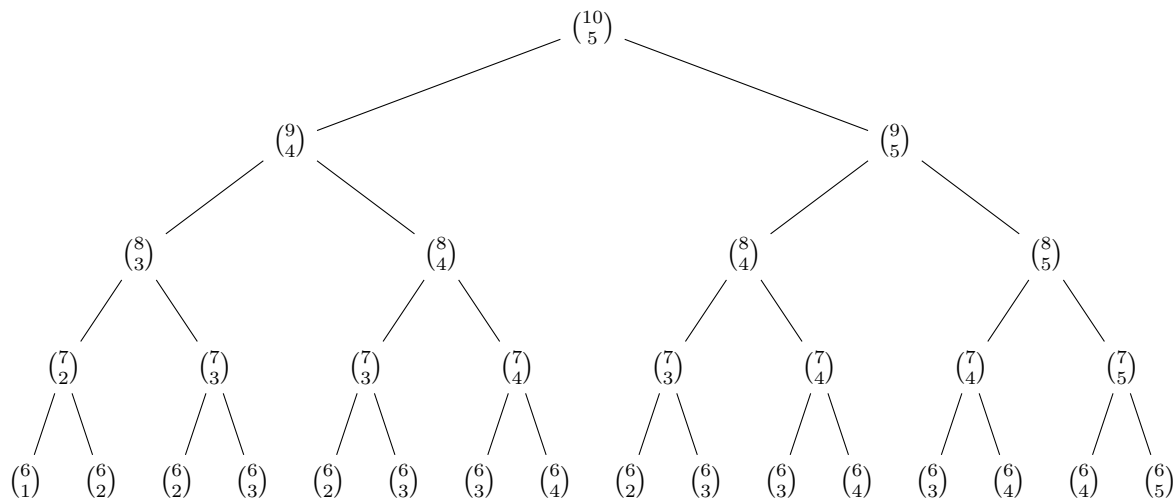
**Autor:** Isabel Pita.

## 5.1. Objetivos del problema.

- Practicar la definición recursiva de problemas.
- Mejorar la eficiencia de una solución utilizando parámetros auxiliares que nos permitan consultar valores calculados previamente.
- Manejo de matrices.

## 5.2. Ideas generales.

- Veamos en primer lugar como se obtiene la definición recursiva dada en el enunciado del problema.
  - El número combinatorio  $\binom{a}{b}$  representa el número de formas de elegir  $b$  elementos de un conjunto  $A$  de  $a$  elementos.
  - Un elemento  $k$  del conjunto  $A$ , puede formar parte de los elementos elegidos o no formar parte.
  - Si el elemento  $k$  forma parte de los elementos elegidos, nos quedará por elegir  $b - 1$  elementos del conjunto  $A - k$ . El número de formas de elegir  $b - 1$  elementos del conjunto  $A - k$  viene dado por el número combinatorio  $\binom{a-1}{b-1}$ .
  - Si el elemento  $k$  no forma parte de los elementos elegidos, nos quedarán por elegir los  $b$  elementos del conjunto  $A - k$ . El número de formas de elegir  $b$  elementos del conjunto  $A - k$  viene dado por el número combinatorio  $\binom{a-1}{b}$ .
  - El número de formas de elegir  $b$  elementos del conjunto  $A$  es la suma de los dos números anteriores.
  - Toda definición recursiva debe tener un caso base. En nuestro problema, sabemos que:
    - Sólo hay una forma de elegir 0 elementos de un conjunto.
    - Sólo hay una forma de elegir todos los elementos de un conjunto.
- Para implementar el problema, realizar dos llamadas recursivas para calcular  $\binom{a-1}{b-1}$  y  $\binom{a-1}{b}$  no es factible debido al coste de la solución obtenida. El problema está en que se realizan muchas llamadas recursivas iguales, las cuales se resuelven completas cada vez que se ejecutan. La siguiente figura muestra las llamadas recursivas que se van produciendo:



- Para evitar hacer llamadas recursivas que ya se han ejecutado:
  - Almacenamos el resultado de la ejecución en una variable. En este problema se utilizará una matriz de dimensión  $a \times b$ .
  - La matriz se inicializa al valor  $-1$ .

- Antes de hacer una llamada recursiva,  $\binom{i}{j}$ , se comprueba en la matriz si el valor de la posición  $\langle i, j \rangle$  es  $-1$ .
- Si lo es, se realiza la llamada y se almacena el valor calculado en la matriz.
- En otro caso se utiliza el valor de la matriz, ya que es el valor calculado previamente.

### 5.3. Algunas cuestiones sobre implementación.

- Definir un tipo matriz como:

```
using tMatriz = std::vector<std::vector<int>>>;
```

- Declarar una variable de tipo tMatriz como:

```
tMatriz matriz(n+1, std::vector<int>(m+1, -1));
```

Observad que para calcular el número combinatorio  $\binom{a}{b}$  se almacenan los valores del rango  $[0..a] \times [0..b]$  y por lo tanto la matriz se declara de dimensión  $(a+1) \times (b+1)$ .

### 5.4. Ideas detalladas.

- Se puede utilizar la propiedad de los números combinatorios  $\binom{a}{b} = \binom{a}{a-b}$  para elegir como valor de  $b$  el mínimo entre  $b$  y  $a-b$ . Esto no cambia el orden de complejidad del algoritmo pero disminuye la constante multiplicativa.

### 5.5. Errores frecuentes.

- No actualizar la matriz cuando se calcula un valor.

### 5.6. Coste de la solución.

Cada número combinatorio se calcula como máximo una vez. Por lo tanto una cota superior del algoritmo es  $\mathcal{O}(a \times b)$ , siendo  $a$  y  $b$  los valores de entrada al algoritmo.

Observad que si se aplica la definición directamente haciendo todas las llamadas recursivas aunque ya hayan sido calculadas, la recurrencia que nos proporciona el coste del algoritmo es:

$$T(a, b) = \begin{cases} c_1 & b == 0 \vee a == b \\ T(a-1, b-1) + T(a-1, b) + c_2 & b \neq 0 \wedge a \neq b \end{cases}$$

Podemos aproximar esta recurrencia con:

$$T(a, b) = \begin{cases} c_1 & b == 0 \vee a == b \\ 2T(a-1, b) + c_2 & b \neq 0 \wedge a \neq b \end{cases}$$

y consultando su despliegue obtenemos un coste  $\mathcal{O}(2^{a-b})$ , siendo  $a$  y  $b$  los valores de entrada al algoritmo.

### 5.7. Implementación.

```
using tMatriz = std::vector<std::vector<int>>>;

int resolver(int n, int m, tMatriz & matriz) {
    if (m == 0 || m == n) return matriz[n][m] = 1;
    else if (m == 1 || m == n-1) return matriz[n][m] = n;
    else {
        if (matriz[n-1][m-1] == -1) matriz[n-1][m-1] = resolver(n-1, m-1, matriz);
        if (matriz[n-1][m] == -1) matriz[n-1][m] = resolver(n-1, m, matriz);
        return matriz[n][m] = matriz[n-1][m-1] + matriz[n-1][m];
    }
}
```

```

    }
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuraci'on, y escribiendo la respuesta
bool resuelveCaso() {
    int n,m;
    std::cin >> n >> m;
    if (!std::cin) return false;
    tMatriz matriz(n+1,std::vector<int>(m+1,-1));
    std::cout << resolver(n,m,matriz) << '\n';
    return true;
}

```





## 6. Formas de sumar.

### Calcula el número de formas de obtener un valor con unos ciertos números realizando solo sumas

Nos dan un vector de números enteros positivos todos ellos diferentes y un valor entero  $S$ . Nos piden el número de formas distintas que hay de obtener la cantidad  $S$  utilizando únicamente sumas y los valores del vector. Sólo se puede utilizar cada valor una vez.

*Requisitos de implementación.*

Definir la estrategia recursiva que se va a utilizar.

Utilizar una matriz como parámetro acumulador para evitar repetir llamadas recursivas.

#### Entrada

La entrada consiste en una serie de casos de prueba. Cada caso se define en dos líneas. En la primera aparece el número de elementos del vector y la cantidad  $S$  que se quiere obtener. En la segunda línea aparecen los valores del vector separados por un blanco.

#### Salida

Para cada caso de prueba se escribe en una línea el número de formas de obtener la cantidad  $S$ .

#### Entrada de ejemplo

```
3 4
1 2 3
4 3
2 1 3 4
5 6
4 2 3 5 1
5 10
4 2 6 3 5
```

#### Salida de ejemplo

```
1
2
3
2
```

**Autor:** Isabel Pita.

### 6.1. Objetivos del problema.

- Practicar la definición recursiva de problemas.
- Practicar la implementación de funciones recursivas con historia.

### 6.2. Ideas generales.

- En este problema únicamente indicaremos la forma de obtener la definición recursiva.
- Si queremos sumar una cantidad  $N$  utilizando  $i$  valores, podemos sumar  $N$  utilizando únicamente  $i - 1$  valores, o podemos considerar que utilizamos uno de los valores  $k$  en la suma y obtener  $N - k$  con los restantes  $i - 1$  valores.
- Consideraremos casos base, cuando la suma que queremos calcular es cero y cuando no nos quedan valores para realizar la suma.
- Definimos la función: *suma(N,i) = número de formas de sumar N, utilizando i valores.*

La definición recursiva es:

$$suma(N, i) = \begin{cases} 1 & N == 0 \\ 0 & i = 0 \wedge N > 0 \\ suma(N, i-1) + suma(N-v[i]) & N > 0 \wedge i > 0 \end{cases}$$

- Este problema se implementa utilizando la misma idea que el ejercicio anterior. La definición recursiva del problema da lugar a llamadas con valores repetidos. Para mejorar el coste se utilizará una matriz en la que se almacenarán los valores que se van calculando.

.tex