Apuntes OIFem II Nivel 1
Funciones

Funciones

Una función es un bloque de código que solo se ejecuta cuando se llama. Las funciones se utilizan para realizar determinadas acciones y son importantes para reutilizar el código: solo es definido una vez y se utiliza muchas veces. Las funciones permiten descomponer un programa en diversos subprogramas, por lo cual el código es más leíble y estructurado, y es más fácil de corregir y mejorar.

La función main() es una función pre-definida en C++ que se utiliza para ejecutar el código. Se pueden crear otras funciones para llevar a cabo diferentes acciones. Las funciones en C++ siempre deben definirse antes de la función main(). Sino, no se compilará porque habrá un error.

Nota: En C++ existen funciones definidas que se pueden utilizar añadiendo paquetes. Por ejemplo, con “#include <cmath>” se pueden utilizar las funciones pre-definidas sqrt() o pow(), que calculan la raíz cuadrada de un número y la potencia n-ésima de un número, respectivamente. Véase Librería <cmath>.

Código-función

```
#include <iostream>
using namespace std;

int max(int a, int b) {
    if (a > b) return a;
    return b;
}

int main() {
    int x, y;
    cin >> x >> y;
    cout << max2(x, y) << '\n';
}
```

Estructura de una función

Para definir una función, debemos indicar el tipo de valor que deseamos que devuelva la función, como int, string, double, etc. La palabra “return” se debe usar para devolver el valor deseado. La palabra “void” indica que la función no debe devolver ningún valor.

Después de definir el tipo de función, se escribe su nombre, y se enumeran los parámetros formales, es decir, la información que se pasa a la función. Los **parámetros formales** actúan como variables dentro

de la función. De hecho, cuando se invoca una función se deben pasar los valores necesarios que recibirán los parámetros formales. Estos parámetros que se pasan a la función se llaman **parámetros reales o argumentos**. Al invocar una función, el valor de los parámetros reales o argumentos se transmite a los parámetros formales.

Nota: También podemos utilizar el valor de un parámetro predeterminado, asignándolo mediante el símbolo "=".

Nota: Debemos tener en cuenta que la llamada a la función debe tener el mismo número de argumentos que parámetros formales, y los argumentos se deben pasar en el mismo orden.

Código-función-2

```
#include <iostream>
using namespace std;

// Devuelve el factorial de un número natural n.
int factorial(int n) {
    int fac = 1;
    for (int i = 1; i <= n; ++i) {
        fac = fac*i;
    }
    return fac;
}

int main() {
    int x;
    cin >> x;
    cout << factorial(x) << '\n';
}
```

Sobrecarga de funciones

La **sobrecarga de funciones** se denomina cuando el mismo nombre de una función se refiere a diferentes funciones. Para saber la función que se invoca, es necesario que el número y/o los tipos de parámetros sean diferentes.

Código-sobrecarga

```
#include <iostream>
using namespace std;

int max(int a, int b) {
    if (a > b) return a;
    return b;
}

int max(int a, int b, int c) {
    return max(a, max(b, c));
}

int main() {
    int x, y, z;
```

```
    cin >> x >> y >> z;
    cout << max(x, y) << '\n';
    cout << max(x, y, z) << '\n';
}
```

Pasar por referencia

En los ejemplos anteriores, hemos pasado el valor de los argumentos a los parámetros formales. También se puede pasar una referencia a una función, lo cual es útil cuando necesitamos cambiar el valor de los argumentos. En otras palabras, si cambiamos el valor de la referencia, este habrá sido cambiado directamente en la variable del main().

Código-pasar por referencia

```
#include <iostream>
using namespace std;

void intercambiar(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int primero, segundo;
  cin >> primero >> segundo;
  cout << "Antes de intercambiar: " << primero << ' ' << segundo << "\n";
  intercambiar(primero, segundo); //Llamar a la funcion intercambiar, que cambiará
  ↪ los valores de primero y segundo
  cout << "Despues de intercmabiar: " << primero << ' ' << segundo << "\n";
}
```

Ejemplo práctico: Número de dígitos iterativo

Escribe una función iterativa que devuelva el número de dígitos de un número n.

INTERFICIE: La función iterativa del programa debe tener la siguiente estructura:

`int number_of_digits (int n);`

PRECONDICIÓN: Se asegura que $n \geq 0$.

OBSERVACIÓN: Solo se necesita enviar la función requerida, el programa principal será ignorado.

Link al problema: Número de dígitos iterativos

Solución

Como los números están en base decimal, cuando vamos dividiendo por 10, encontramos las cifras del número, es decir, el número de dígitos.

```cpp
#include <iostream>
using namespace std;

int number_of_digits(int n) {
  int contador = 1;
  while (n >= 10) {
    n = n/10;
    contador++;
  }
  return contador;
}

int main() {
  int x;
  cin >> x;
  cout << number_of_digits(x) << '\n';
}
```

Ejemplo práctico: Iterative greatest common divisor

Escribe una función iterativa que calcule el máximo común divisor de dos números naturales a y b usando la versión rápida del Algoritmo de Euclides.

INTERFICIE: La función iterativa del programa debe tener la siguiente estructura:

```cpp
int gcd(int a, int b);
```

PRECONDICIÓN:

Ni a ni b son números negativos, y al menos uno de ellos es estrictamente mayor que cero.

OBSERVACIÓN:

Solo se necesita enviar la función requerida, el programa principal será ignorado.

Link al problema: Iterative greatest common divisor

Solución

El Algoritmo de Euclides es el método que utilizamos para calcular el máximo común divisor entre dos números. Si tenemos a y b, entonces gcd(a,b) = gcd(b, r), donde r es el residuo de dividir a entre b: $a = b \cdot q + r$, con $r \geq 0$.

Para más información sobre el Algoritmo de Euclides para encontrar el máximo común divisor de dos números consulta: Algoritmo de Euclides.

```cpp
#include <iostream>
using namespace std;

int gcd(int a, int b) {
    while (b > 0) {
        int r = a%b;
        a = b;
```

```
        b = r;
    }
    return a;
}

int main()
{
    int a, b;
    cin >> a >> b;
    cout << gcd(a,b) << '\n';
}
```

Ejemplo práctico: Sort three

Escribe una función que ordene a, b, c en orden no-decreciente. Por ejemplo, si $a = 7, b = -3, c = 1$, los valores después de llamar a la función serán $a = -3, b = 1, c = 7$.

INTERFICIE: La función iterativa del programa debe tener la siguiente estructura:

`void sort3(int& a, int& b, int& c);`

OBSERVACIÓN:

Solo se necesita enviar la función requerida, el programa principal será ignorado.

Link al problema: Sort three

Solución

Para ordenar los tres números a, b, c debemos usar el paso por referencia de las variables, para poder cambiarlas dentro de las funciones `swap2` y `sort3`.

Para intercambiar el valor de dos variables en la función `swap2`, debemos crear una nueva variable auxiliar que tenga el valor de una de las variables.

Para ordenar las tres variables a, b, c en orden creciente miramos intercambiamos a, b si están mal ordenadas. Hacemos lo mismo con b, c. Entonces, faltaría mirar otra vez cómo están a, b porque hemos cambiado el valor de b.

```
#include <iostream>
using namespace std;

void swap2(int& a, int& b)
{
    int aux = a;
    a = b;
    b = aux;
}

void sort3(int& a, int& b, int& c)
{
    // Devuelve los valores de a,b,c en orden creciente
    if (a > b)
```

```
15          swap2(a, b);
16      // a <= b
17      if (b > c)
18          swap2(b, c);
19      // b <= c y c es el máximo
20      if (a > b)
21          swap2(a, b);
22      // a <= b
23  }
24
25  int main()
26  {
27      int a, b, c;
28      cin >> a >> b >> c;
29      sort3(a, b, c);
30      cout << a << ' ' << b << ' ' << c << endl;
31  }
```

Ejemplo práctico: Reinas atacadas

En el ajedrez, la reina es la pieza más poderosa, al poderse mover cualquier número de escaques en vertical, horizontal, o diagonal.

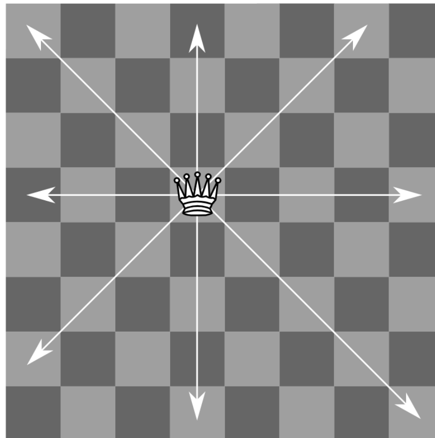

Figure 1: Movimientos de la reina.

En 1848, el alemán Max Bezzel planteó el puzzle de las 8 reinas, en el que retó a colocar 8 reinas sobre un tablero sin que se atacaran entre sí. Dos años después, se dieron algunas de las 92 soluciones.

Desde entonces, matemáticos y aficionados de todo el mundo han estudiado el problema, generalizándolo a tamaños de tableros de ajedrez de N×N. En 1972, Dijkstra, en plena crisis del software, usó el problema para demostrar el poder de la programación estructurada, y desde entonces es un ejemplo clásico de algoritmo de vuelta atrás.

Para poder colocar las reinas, el primer paso es saber cuándo un grupo de reinas sobre un tablero de ajedrez se atacan entre sí, es decir cuándo hay al menos una reina que podría comer a otra siguiendo las reglas del movimiento del juego.

ENTRADA: La entrada consta de un conjunto de casos de prueba. Cada uno comienza con una línea con dos números. El primero indica el ancho y alto del tablero de ajedrez (siempre será cuadrado de

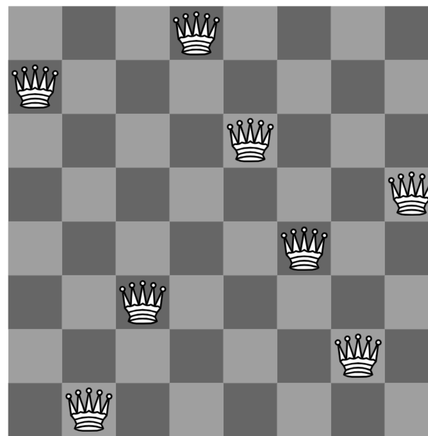

Figure 2: Una de las soluciones posibles.

como mucho 2000 x 2000). El segundo indica el número de reinas colocadas sobre él (entre 1 y 100).

A continuación vendrá una línea con la posición de todas las reinas. Para cada una, se indicará primero la coordenada X y luego la Y, separadas por espacio. Las posiciones de cada reina también se separarán por un único espacio. Todas las posiciones serán válidas (cada coordenada estará entre 1 y el tamaño del tablero) y se garantiza que no habrá dos reinas en la misma posición.

La entrada termina con un caso de prueba con un tablero de tamaño 0 x 0 y sin reinas que no debe procesarse.

SALIDA: Para cada caso de prueba, el programa escribirá, en la salida estándar, una línea con el texto "SI" si hay reinas atacadas en la configuración dada, y "NO" en otro caso (sin las comillas).

Link al problema: Reinas atacadas

Solución

Una reina está atacada cuando en las rectas diagonal, horizontal o vertical hay otra reina. Por lo tanto, tenemos que comprobar lo siguiente para cada par de reinas del tablero

- Si dos reinas están en la misma recta vertical, entonces tienen la misma coordenada X.
- Si están en a misma recta horizontal, tienen la misma coordenada Y.
- Si están en la misma diagonal, entonces el valor absoluto de la diferencia de las coordenadas X e Y, respectivamente, será el mismo. Este caso se puede separar en dos casos: cuando el valor es positivo, y cuando es negativo.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int atacada (vector< vector<int> > &posicion) {
6    for (int i = 0; i < (int) posicion.size(); ++i) {
7      for (int j = i+1; j < (int) posicion.size(); ++j) {
8        if (posicion[i][0] - posicion[j][0] == posicion[i][1] - posicion[j][1])
          ↪  return 1;
```

```
        if (posicion[i][0] - posicion[j][0] == posicion[j][1] - posicion[i][1])
         ↪  return 1;
        if (posicion[i][0] == posicion[j][0]) return 1;
        if (posicion[i][1] == posicion[j][1]) return 1;
      }
    }
    return 0;
  }

  int main() {
    int N, r;
    while (cin >> N >> r) {
      if (N == 0 and r == 0)
        break;
      vector< vector<int> > posicion(r, vector<int> (2));
      for (int i = 0; i < r; ++i) {
        cin >> posicion[i][0] >> posicion[i][1];
      }
      if (atacada(posicion) == 0)
        cout << "NO\n";
      else
        cout << "SI\n";
    }
  }
```

Ejemplo práctico: Juego 23

Policarpo juega al "Juego 23". Inicialmente tiene un número n y su objetivo es transformarlo en m. En un movimiento, puede multiplicar n por 2 o multiplicar n por 3. Puede realizar cualquier cantidad de movimientos.

Imprima el número de movimientos necesarios para transformar n en m. Imprime -1 si es imposible hacerlo.

Es fácil demostrar que cualquier forma de transformar n en m contiene el mismo número de movimientos (es decir, el número de movimientos no depende de la forma de transformación).

ENTRADA: La única línea de entrada contiene dos números enteros n y m $(1 \leq n \leq m \leq 5 \cdot 10^8)$.

SALIDA: Imprime el número de movimientos para transformar n a m, o -1 si no hay solución.

Link al problema: Game 23

Solución

Como solamente podemos multiplicar por 2 o por 3, observamos que podremos pasar de n a m solo cuando m sea n multiplicado por 2 y 3, es decir, $m = 2^i \cdot 3^j \cdot n$. Por lo tanto, debemos imprimir el número de veces que 2 y 3 dividen a $\frac{m}{n}$. Además, también debemos asegurarnos que $\frac{m}{n}$ no tenga ningún otro factor primo aparte de 2 y 3.

```
#include <iostream>
using namespace std;

int contar(int n, int k) {
```

```
  int contador = 0;
  while (n > 1) {
    if (n%k == 0) {
      n = n/k;
      contador++;
    }
    else n = 1;
  }
  return contador;
}

int main() {
  int n, m, a, b, num = 1;
  cin >> n >> m;
  if (m%n != 0) {
    cout << -1 << '\n';
  } else {
    a = contar(m/n, 2);
    b = contar(m/n, 3);
    for (int i = 1; i <= a; ++i)
      num = num*2;
    for (int j = 1; j <= b; ++j)
      num = num*3;
    if (num == m/n)
      cout << a + b << '\n';
    else
      cout << -1 << '\n';
  }
}
```