Soluciones OIFem 2020

Concurso de Práctica 3

Comparando número de galaxias

Teoría

- Entrada y salida de datos
- Condicionales

Solución

Usamos una serie de ifs para imprimir los resultados a las preguntas que se nos hacen.

Código

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

int main() {
  int a, b, c;
  cin >> a >> b >> c;
  // Primera pregunta: a < b?
  if (a < b)
    cout << "True ";
  else
    cout << "False ";
  // Segunda pregunta: c > a?
  if (c > a)
    cout << "True ";
  else
    cout << "False ";
  // Tercera pregunta: a = b?
  if (a == b)
    cout << "True ";
  else
    cout << "False ";
  // Cuarta pregunta: a != c?
  if (a != c)
    cout << "True ";
  else
    cout << "False ";
```

```
27    // Quinta pregunta: c <= b?
28    if (c <= b)
29      cout << "True";
30    else
31      cout << "False";
32    return 0;
33  }
```

Extra: este código puede hacerse más corto diseñando una función `imprimirRespuesta (bool condicion)` donde a la función le des la respuesta a la pregunta como entrada y esta imprima True/False al usuario como corresponda. ¿Sabrías implementar esta solución?

Dos gatos y un ratón

Teoría

- Entrada y salida de datos
- Condicionales
- Valor absoluto

Solución

El truco para resolver este problema es darse cuenta de que nos interesan las distancias absolutas, ya que no sabemos si el gato está a la izquierda o a la derecha del ratón. Una vez calculadas, imprimimos el mensaje que corresponda.

Código

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

int main() {
  int distA, distB, posA, posB, posC;
  cin >> posA >> posB >> posC;
  // calculamos las distancias absolutas
  distA = abs(posA - posC);
  distB = abs(posB - posC);
  // comparamos las distancias
  if (distA == distB)
    cout << "raton C";
  else if (distB < distA)
    cout << "gato B";
  else
    cout << "gato A";
  return 0;
}
```

Múltiplos dentro de un intervalo

Teoría

- Bucles

Solución

En este problema, la solución consiste en programar un bucle que empiece en a y suba en incrementos de a hasta toparse con un número mayor que b, donde frenará.

Código

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

int main() {
  int a, b;
  cin >> a >> b;
  for (int i = a; i <= b; i += a)
    cout << i << ' ';
  return 0;
}
```

Nombres y apellidos sanos

Teoría

- Strings
- Vectores

Solución

Leemos la entrada línea a línea. Para cada línea, primero procesamos las palabras para convertirlas en la salida que Susana espera y luego la imprimimos. Para procesarla, guardamos en todo momento las letras que llevamos de seguido sin ver un espacio y, cuando vemos uno, añadimos la palabra completa al vector con las anteriores y reiniciamos ese contador de letras.

Código

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

string procesar(string & S) {
  // Separamos la línea en sus palabras
  string ret = "";
  string palabraActual = "";
  vector<string> palabras;
  for (wchar_t i = 0; i < S.length(); i++) {
    if (S[i] != ' ')
      palabraActual += S[i]; // añadimos la letra a la palabra actual
    else {
      palabras.push_back(palabraActual); // añadimos la palabra a la lista de
      ↪  palabras
      palabraActual = ""; // hacemos borrón y cuenta nueva
    }
  }
  if (palabraActual != "")
    palabras.push_back(palabraActual); // importante acordarse de añadir la última
    ↪  palabra!

  // retornamos la construcción que busca Susana
  int n = (int) palabras.size();
  if (n < 2 || n > 4)
    return "*";
  else if (n == 2)
    return palabras[1] + " " + palabras[0];
  else if (n == 3)
    return palabras[2] + " " + palabras[0] + " " + palabras[1];
  else
    return palabras[2] + " " + palabras[3] + " " + palabras[0] + " " +
    ↪  palabras[1];
}

int main() {
```

```
34    string nombre;
35    vector<string> nombres;
36    while(getline(cin, nombre)) {
37      cout << procesar(nombre) << '\n';
38    }
39    return 0;
40  }
```

Mínimo común múltiplo de varios enteros

Teoría

- Funciones
- Método de Euclides para el MCD

Solución

Usamos el método de Euclides (explicado en las instrucciones), dándonos cuenta de que el MCM de una lista de enteros se puede encontrar de forma iterativa, es decir, que el MCM de a, b y c es el MCM de c y d, donde d es el MCM de a y b, y así sucesivamente.

Código

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

int mcd(int a, int b) {
  if (b == 0)
    return a;
  return mcd(b, a % b);
}

int mcm(int a, int b) {
  return a*b/mcd(a, b);
}

int main() {
  int n, mcmActual = 1, numeroActual;
  cin >> n;
  while(n--) {
    cin >> numeroActual;
    mcmActual = mcm(mcmActual, numeroActual);
  }
  cout << mcmActual;
  return 0;
}
```

Xornacci

Teoría

- Xor

Solución

La clave para resolver este problema es aprovechar varias propiedades de XOR: es asociativo, conmutativo y que b XOR b = 0 y b XOR 0 = b para todo b. Entonces vemos lo siguiente:

F(0) = a

F(1) = b

F(2) = a XOR b

F(3) = b XOR (a XOR b) = a XOR (b XOR b) = a XOR 0 = a

F(4) = (a XOR b) XOR a = (a XOR a) XOR b = 0 XOR b = b

F(5) = a XOR b

... y así sucesivamente.

Código

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

int main() {
  int a, b, n, T;
  cin >> T;
  while(T--) {
    cin >> a >> b >> n;
    int c = a^b;
    if (n % 3 == 0)
      cout << a << '\n';
    else if (n % 3 == 1)
      cout << b << '\n';
    else
      cout << c << '\n';
  }
  return 0;
}
```

Atsa y el torneo de programación

Teoría

- Álgebra

Solución

Para este problema, la clave era encontrar la siguiente fórmula iterativa:

`D[i] = P[i] * (P[0] + ... + P[i-1]) + D[i-1]`

con el caso base `D[0] = P[0]`, donde `D[i]` es la popularidad total de los encuentros entre los concursantes 0, 1, ..., i. Por lo tanto, la solución al problema será `D[N-1]`.

Así, se podía encontrar la solución de forma iterativa en $O(N)$, guardando siempre `D[i]` en `respuesta` y `P[0] + ... + P[i]` en `suma`.

Código

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

int main() {
  int N;
  cin >> N;
  long long int respuesta = 0, suma, P_i;
  cin >> suma;
  for (int i = 1; i < N; i++) {
    cin >> P_i;
    respuesta += suma * P_i;
    suma += P_i;
  }
  cout << respuesta;
  return 0;
}
```

Contando secuencias crecientes

Teoría

- Combinaciones
- Programación dinámica

Solución

La respuesta es el número de formas de escoger N números entre 1 y K, ya que para esa selección solo hay un orden posible (el ascendente). Por lo tanto, calculamos $\frac{K!}{N!(K-N)!}$ si $K >= N$ y 0 si no.

Como K puede ser 25, y 25! daría overflow en long long, calculamos el número de combinaciones con el método de programación dinámica visto en clase.

Código

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

vector<vector<int>> DP;

int combinaciones(int n, int k) {
  if (DP[n][k] != -1)
    return DP[n][k];
  if (n == k || k == 0)
    return DP[n][k] = 1;
  return DP[n][k] = combinaciones(n-1, k-1) + combinaciones(n-1, k);
}


int main() {
  int N, K;
  cin >> N >> K;
  DP.assign(K+1, vector<int>(N+1, -1));
  if (N > K)
    cout << '0';
  else
    cout << combinaciones(K, N);
  return 0;
}
```

Pies de hobbits

Teoría

- DFS
- Backtracking
- Máscaras de bits

Solución

Para ver el mínimo número de colores del camino, probaremos las 7 combinaciones posibles de 1, 2 y 3 (no son 8 ya que nos saltamos la que no contiene ninguno de los tres colores), probando en orden de cuántos colores tiene la combinación. Representaremos cada combinación con una máscara de bits, ya que así podremos comprobar rápidamente al llegar a una casilla nueva si esta está en la combinación.

Hacemos DFS con cada combinación hasta encontrar una que funcione. Solo iniciaremos la búsqueda con las máscaras que contengan el número de inicio y el de salida, recordando hacer backtracking con las casillas visitadas para evitar entrar en bucles.

Código

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

int N, M;
vector<vector<int>> mapa;
vector<vector<int>> visitado;

int ones(int mask) {
  return (mask & 1) + (mask & 2) / 2 + (mask & 4) / 4;
}

bool valido(int r, int c, int mask) {
  // esta función nos dice si en la casilla r, c del mapa (r = row, c = column),
  ↪  hay un número en la combinación representada por mask
  return (mask & 1<<(mapa[r][c]-1)) > 0;
}

bool dfs (int mask, int r, int c) {
  // la casilla se encuentra en la combinación? -> si no es el caso retornamos
  ↪  false
  if (!valido(r, c, mask))
    return false;
  // si ya hemos visitado la casilla, retornamos false, ya que el bucle no nos
  ↪  llevará a la salida si otro camino sin bucle no lleva a ella, ya que el
  ↪  bucle nos deja "donde empezamos"
  if (visitado[r][c])
    return false;
  // si llegamos a la casilla final, retornamos true :)
  if (r == N-1 && c == M-1)
    return true;
```

```
    // marcamos la casilla actual como visitada para evitar bucles
    visitado[r][c] = true;
    // visitamos las 4 direcciones, recordando borrar el "visitado" al retroceder
    if (r < N-1 && dfs(mask, r+1, c)) {
      visitado[r][c] = false;
      return true;
    }
    if (c < M-1 && dfs(mask, r, c+1)) {
      visitado[r][c] = false;
      return true;
    }
    if (r > 0 && dfs(mask, r-1, c)) {
      visitado[r][c] = false;
      return true;
    }
    if (c > 0 && dfs(mask, r, c-1)) {
      visitado[r][c] = false;
      return true;
    }
    // no encontramos camino :( -> retornamos false y borramos el "visitado"
    visitado[r][c] = false;
    return false;
  }

  int main() {
    cin >> N >> M;
    mapa = vector<vector<int>>(N, vector<int>(M));
    for (int i = 0; i < N; i++)
      for (int j = 0; j < M; j++)
        cin >> mapa[i][j];
    vector<int> masks = {1, 2, 4, 3, 5, 6, 7};
    for (int mask: masks) {
      if (!valido(0, 0, mask) || !valido(N-1, M-1, mask))
        continue;
      visitado.assign(N, vector<int>(M, false));
      if (dfs(mask, 0, 0)) {
        cout << ones(mask);
        break;
      }
    }
    return 0;
  }
```

Contando estudiantes

Teoría

- Balanced binary search trees

Solución

Construimos un BST balanceado (AVL en el caso de esta solución, pero otras alternativas valen igual), donde cada nodo del árbol es la edad de un estudiante. Cada vez que añadimos un estudiante (queries tipo 1), añadimos un nodo con su edad al árbol y, cada vez que recibimos una query de tipo 2, la resolvemos bajando por el árbol.

Código

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

struct Nodo {
        int edad, cantidad, cantidadIzq, cantidadDer, altura;
        // cantidadIzq = número de nodos en su subárbol izquierdo
        Nodo *izq, *der;
        Nodo(int e) {
                edad = e;
                izq = NULL;
                der = NULL;
                cantidad = altura = 1;
                cantidadIzq = cantidadDer = 0;
        }
};

int encontrarAltura(Nodo * nodoActual) {
        if (nodoActual == NULL)
                return 0;
        return nodoActual->altura;
}

int encontrarCantidad(Nodo * nodoActual) {
        if (nodoActual == NULL)
                return 0;
        return nodoActual->cantidadIzq + nodoActual->cantidad +
        ↪  nodoActual->cantidadDer;
}

void recalcularAltura(Nodo * nodoActual) {
        if (nodoActual == NULL)
                return;
        nodoActual->altura = 1 + max(encontrarAltura(nodoActual->izq),
        ↪  encontrarAltura(nodoActual->der));
}

void recalcularCantidad(Nodo * nodoActual) {
        if (nodoActual == NULL)
```

```
                return;
        nodoActual->cantidadIzq = encontrarCantidad(nodoActual->izq);
        nodoActual->cantidadDer = encontrarCantidad(nodoActual->der);
}

Nodo *rotarIzq(Nodo * nodoActual) {
    Nodo *hijoActual = nodoActual->der;
    Nodo *nietoActual = hijoActual->izq;

    hijoActual->izq = nodoActual;
    nodoActual->der = nietoActual;

    nodoActual->altura = max(encontrarAltura(nodoActual->izq),
                    encontrarAltura(nodoActual->der)) + 1;
    hijoActual->altura = max(encontrarAltura(hijoActual->izq),
                    encontrarAltura(hijoActual->der)) + 1;

    recalcularCantidad(nodoActual);
    recalcularCantidad(hijoActual);

    return hijoActual;
}

Nodo * derRotate(Nodo * nodoActual) {
    Nodo * hijoActual = nodoActual->izq;
    Nodo * nietoActual = hijoActual->der;

    hijoActual->der = nodoActual;
    nodoActual->izq = nietoActual;

    nodoActual->altura = max(encontrarAltura(nodoActual->izq),
                    encontrarAltura(nodoActual->der)) + 1;
    recalcularAltura(nodoActual);
    recalcularAltura(hijoActual);

    recalcularCantidad(nodoActual);
    recalcularCantidad(hijoActual);

    return hijoActual;
}

Nodo * insertar(int edad, Nodo * nodoActual) {
        if (nodoActual == NULL)
                return new Nodo(edad);
        if (edad < nodoActual->edad)
                nodoActual->izq = insertar(edad, nodoActual->izq);
        else if (edad == nodoActual->edad) {
                nodoActual->cantidad++;
                return nodoActual;
        } else
                nodoActual->der = insertar(edad, nodoActual->der);

        recalcularAltura(nodoActual);
        recalcularCantidad(nodoActual);

```

```
92          int equilibrio = encontrarAltura(nodoActual->izq) -
            ↪  encontrarAltura(nodoActual->der);
93
94          if (equilibrio > 1 && edad < nodoActual->izq->edad)
95                  return derRotate(nodoActual);
96
97          if (equilibrio < -1 && edad > nodoActual->der->edad)
98                  return rotarIzq(nodoActual);
99
100         if (equilibrio > 1 && edad > nodoActual->izq->edad) {
101                 nodoActual->izq = rotarIzq(nodoActual->izq);
102                 return derRotate(nodoActual);
103         }
104
105         if (equilibrio < -1 && edad < nodoActual->der->edad) {
106                 nodoActual->der = derRotate(nodoActual->der);
107                 return rotarIzq(nodoActual);
108         }
109
110         return nodoActual;
111 }
112
113 int buscarArbol(int edad, Nodo *nodoActual) {
114         if (nodoActual == NULL)
115                 return 0;
116         if (nodoActual->edad < edad)
117                 return nodoActual->cantidad + nodoActual->cantidadIzq
118                                 + buscarArbol(edad, nodoActual->der);
119         else if (nodoActual->edad == edad)
120                 return nodoActual->cantidad + nodoActual->cantidadIzq;
121         else
122                 return buscarArbol(edad, nodoActual->izq);
123 }
124
125 int buscar(int edad, Nodo *raiz) {
126         return buscarArbol(edad, raiz);
127 }
128
129 int main() {
130         ios::sync_with_stdio(false);
131         cin.tie(NULL);
132         int m, p, x;
133         Nodo * raiz = NULL;
134         cin >> m;
135         while (m--) {
136                 cin >> p >> x;
137                 if (p == 1)
138                         raiz = insertar(x, raiz);
139                 else
140                         cout << buscar(x, raiz) << '\n';
141         }
142         return 0;
143 }
```

Referencia para la implementación de los árboles AVL. Para esta solución, se ha usado la inserción algo modificada, ya que era lo único que nos hacía falta.