Ad hoc y Búsqueda Completa

MSc. (c) Jhosimar George Arias Figueroa

jariasf03@gmail.com

State University of Campinas
Institute of Computing

Contenido

Contenido

Problemas Ad hoc

- Problemas 'Ad hoc' son aquellos cuyos algoritmos no entran en las categorias standard.
- No existe una técnica general o especifica para este tipo de problema.
- Las soluciones podrian requerir una estructura de datos nueva o un inusual conjunto de bucles o condicionales.
- Normalmente solo es hacer lo que la descripción del problema nos indica.

Problemas Ad hoc

- El tiempo límite no es un problema.
- Algunas veces se tienen casos de prueba 'tricky'.
- Problemas complejos pueden ser dificiles de implementar.
- Por lo general requieren de una lectura cuidadosa a la descripcion del problema y usualmente gira en torno a una secuencia de instrucciones dadas en el problema **(Simulación)**.

Problemas Ad hoc

Problemas Ad hoc usualmente aparecen en el conjunto de problemas del ACM ICPC (1 o 2).

- Pueden ser de "Tipo General", usando simples estructuras de datos, matematica, strings, o geometria básica.

Lamentablemente, resolver solo problemas Ad hoc no dará un buen resultado en los concursos de programación.

- Aprenderemos más tipos de problemas en las próximas sesiones.

Contenido

4810 - Flowers Flourish from France

Latin America - South America - 2010/2011

Fiona has always loved poetry, and recently she discovered a fascinating poetical form. Tautograms are a special case of alliteration, which is the occurrence of the same letter at the beginning of adjacent words. In particular, a sentence is a tautogram if all of its words start with the same letter.

For instance, the following sentences are tautograms:

- Flowers Flourish from France
- Sam Simmonds speaks softly
- Peter pIckEd pePPers
- truly tautograms triumph

Fiona wants to dazzle her boyfriend with a romantic letter full of this kind of sentences. Please help Fiona to check if each sentence she wrote down is a tautogram or not.

Input

Each test case is given in a single line that contains a sentence. A sentence consists of a sequence of at most 50 words separated by single spaces. A word is a sequence of at most 20 contiguous uppercase and lowercase letters from the English alphabet. A word contains at least one letter and a sentence contains at least one word.

The last test case is followed by a line containing only a single character `*' (asterisk).

Output

For each test case output a single line containing an uppercase `Y' if the sentence is a tautogram, or an uppercase `N' otherwise.

Sample Input

```
Flowers Flourish from France
Sam Simmonds speaks softly
Peter pIckEd pePPers
truly tautograms triumph
this is NOT a tautogram
*
```

Sample Output

```
Y
Y
Y
Y
N
```

C++ Hints

- Posible uso de **tolower** – **libreria** <**ctype**> para volver un caractér a minuscula.
- Uso de **strtok** – **libreria** <**cstring**> para obtener tokens eliminando los espacios en blanco.
- Uso de **stringstream** – **libreria** <**sstream**> para obtener tokens.

JAVA Hints

- Posible uso de la clase **Character** para volver un caractér a minúscula.
- Uso de **StringTokenizer** para obtener tokens eliminando espacios en blanco.
- Posible uso del método **split**.

4473 - Brothers

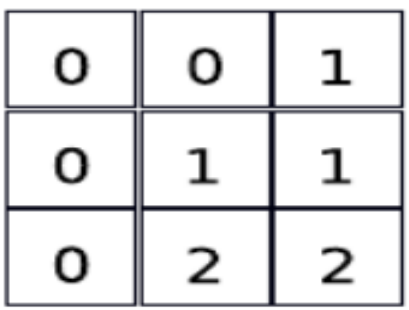
Latin America - South America - 2009/2010

In the land of ACM ruled a great King who became obsessed with order. The kingdom had a rectangular form, and the King divided the territory into a grid of small rectangular counties. Before dying, the King distributed the counties among his sons.

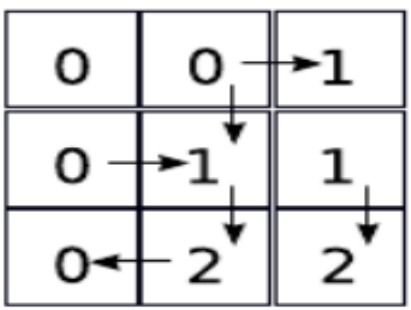
However, he was unaware that his children had developed a strange rivalry: the first heir hated the second heir, but not the rest; the second heir hated the third heir, but not the rest, and so on ...Finally, the last heir hated the first heir, but not the other heirs.

As soon as the King died, the strange rivalry among the King's sons sparked off a generalized war in the kingdom. Attacks only took place between pairs of adjacent counties (adjacent counties are those that share one vertical or horizontal border). A county *X* attacked an adjacent county *Y* whenever the owner of *X* hated the owner of *Y*. The attacked county was always conquered by the attacking brother. By a rule of honor all the attacks were carried out simultaneously, and a set of simultaneous attacks was called a battle. After a certain number of battles, the surviving sons made a truce and never battled again.

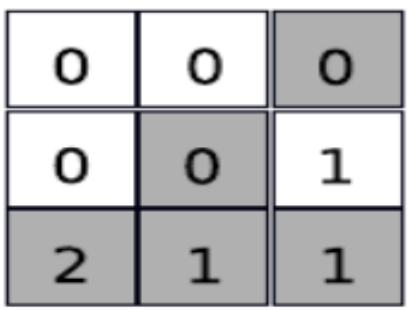
For example, if the King had three sons, named 0, 1 and 2, the figure below shows what happens in the first battle for a given initial land distribution:



Land before the first battle.



Attacks during the first battle.



Land after the first battle. Earned territory in gray.

You were hired to help an ACM historian determining, given the number of heirs, the initial land distribution and the number of battles, what was the land distribution after all battles.

Input

The input contains several test cases. The first line of a test case contains four integers N, R, C and K, separated by single spaces. N is the number of heirs ($2 \leq N \leq 100$), R and C are the dimensions of the kingdom ($2 \leq R, C \leq 100$), and K is the number of battles ($1 \leq K \leq 100$). Heirs are identified by sequential integers starting from zero (0 is the first heir, 1 is the second heir, ..., N - 1 is the last heir). Each of the next R lines contains C integers $H_{r,c}$ separated by single spaces, representing initial land distribution: $H_{r,c}$ is the initial owner of the county in row r and column c ($0 \leq H_{r,c} \leq N$ - 1).

The last test case is followed by a line containing four zeroes separated by single spaces.

Output

For each test case, your program must print R lines with C integers each, separated by single spaces in the same format as the input, representing the land distribution after all battles.

Sample Input

```
3 4 4 3
0 1 2 0
1 0 2 0
0 1 2 0
0 1 2 2
4 2 3 4
1 0 3
2 1 2
8 4 2 1
0 7
1 6
2 5
3 4
0 0 0 0
```

Sample Output

```
2 2 2 0
2 1 0 1
2 2 2 0
0 2 0 0
1 0 3
2 1 2
7 6
0 5
1 4
2 3
```

Contenido

Revisión de Adyacentes

Supongamos que estamos en la posición (1,1) en un tablero:

0,0	0,1	0,2
1,0	**1,1**	1,2
2,0	2,1	2,2

Deseamos saber los adyacentes horizontales y verticales.

Revisión de Adyacentes

Tendríamos que evaluar las siguientes posiciones:

0,0	0,1	0,2
1,0	**1,1**	1,2
2,0	2,1	2,2

Revisión de Adyacentes

Considerando que estamos en la posición (x, y), el tablero puede ser visto como lo siguiente:

(x - 1 , y - 1)	(x – 1 , y)	(x - 1 , y + 1)
(x , y - 1)	**(x , y)**	(x , y + 1)
(x + 1 , y - 1)	(x + 1 , y)	(x + 1 , y + 1)

Revisión de Adyacentes

Usaremos un arreglo de incrementos en ambas coordenadas:

```
int dx[ 4 ] = { 0 ,  0 , 1 , -1 };  //incremento en coordenada x
int dy[ 4 ] = { 1 , -1 , 0 ,  0 };  //incremento en coordenada y
```

Este arreglo de incrementos puede variar acorde a lo que me indique el problema por ejemplo podemos tener los siguientes arreglos.

```
//Movimientos cardinales
int dx[ 4 ] = { 0 ,  0 , 1 , -1 };
int dy[ 4 ] = { 1 , -1 , 0 ,  0 };

//Movimientos cardinales + diagonales
int dx[ 8 ] = { 1 , -1 , 0 ,  0 , 1 , 1 , -1 , -1 };
int dy[ 8 ] = { 0 ,  0 , 1 , -1 , 1 , -1 , 1 , -1 };

//Movimientos de caballo en un tablero de ajedrez
int dx[ 8 ]={ -2 , -2 , -1 , -1 ,  1 , 1 ,  2 , 2 };
int dy[ 8 ]={ -1 ,  1 , -2 ,  2 , -2 , 2 , -1 , 1 };
```

Revisión de Adyacentes

Usaremos dicho arreglo en cada coordenada iterando sobre el tamaño del arreglo.

```
//Revisar adyacentes para cada posicion de un tablero
void check(){
    int nx , ny;    //incrementos en x e y
    for( int x = 0 ; x < filas ; ++x ){
        for( int y = 0 ; y < columnas ; ++y ){
            //En cada posicion del tablero evaluo
            //los adyacentes de dicha posicion
            for( int i = 0 ; i < 4 ; ++i ){
                nx = dx[ i ] + x;   //Incremento en x
                ny = dy[ i ] + y;   //Incremento en y
                ...
            }
            ...
        }
    }
}
```

Revisión de Adyacentes

Que ocurre si estoy en un extremo del tablero?

(-1,-1)	(-1 , 0)	(-1,1)
(0,-1)	**(0,0)**	(0,1)
(1,-1)	(1,0)	(1,1)

Revisión de Adyacentes

Es necesario verificar que los adyacentes de cada coordenada no excedan los limites del tablero.

```
//Verificamos si estamos en una posicion valida o no
bool valid( int nx , int ny ){
    //Posiblemente algun adyacente se sale del tablero
    //por ello comprobamos que este dentro del tablero
    return ( nx >= 0 && ny >= 0 && nx < filas && ny < columnas );
}
```

Revisión de Adyacentes

Finalmente el código de revisión de adyacentes sería:

```
//Revisar adyacentes para cada posicion de un tablero
void check(){
    int nx , ny;    //incrementos en x e y
    for( int x = 0 ; x < filas ; ++x ){
        for( int y = 0 ; y < columnas ; ++y ){
            printf("\nCoordenada (%d,%d) tiene las siguientes coordenadas adyacentes:\n" , x , y );
            //En cada posicion del tablero evaluo
            //los adyacentes de dicha posicion
            for( int i = 0 ; i < 4 ; ++i ){
                nx = dx[ i ] + x;   //Incremento en x
                ny = dy[ i ] + y;   //Incremento en y

                //Si estamos en una posicion valida
                if( valid( nx , ny ) ){
                    printf("(%d,%d) " , nx , ny );
                }
            }
            printf("\n");
        }
    }
}
```

Volviendo al problema

El problema me indica que debo evaluar adyacentes horizontales y verticales mas no diagonales. Si tengo por ejemplo como parte del tablero lo siguiente:

(0,0) = 1	(0,1) = 1	(0,2) = 2
(1,0) = 1	**(1,1) = 0**	(1,2) = 1
(2,0) = 0	(2,1) = 1	(2,2) = 2

Tengo que evaluar los 4 adyacentes, pero si tengo:

(0,0) = 1	(0,1) = 2	(0,2) = 2
(1,0) = 1	**(1,1) = 0**	(1,2) = 1
(2,0) = 0	(2,1) = 2	(2,2) = 2

Tengo que evaluar solo el lado izquierdo y derecho

Hints

Usaremos el arreglo de movimientos cardinales.

```
int dx[ 4 ] = { 0 ,  0 , 1 , -1 };  //incremento en coordenada x
int dy[ 4 ] = { 1 , -1 , 0 ,  0 };  //incremento en coordenada y
```

Verificar que se cumpla lo siguiente en cada posición:

```
( tabla[ x ][ y ] + 1 ) % n == tabla[ nx ][ ny ] )
```

Posible uso de **memcpy(destino, fuente , número de bytes)** – libreria <**cstring**> para copiar los datos de un arreglo fuente a un arreglo destino.

Contenido

Problemas Búsqueda Completa

- Búsqueda Completa, también conocido como la fuerza bruta o backtracking recursivo, es un método para resolver un problema que atraviesa la totalidad o parte del espacio de búsqueda para obtener la solución requerida.
- Búsqueda completa debe ser la primera solución considerada, ya que suele ser fácil para llegar a una solución de este tipo y para codificar/depurarlo.
- Una solución de búsqueda completa, sin bugs, es libre de errores nunca deberia recibir Wrong Answer (WA), ya que explora todo el espacio de búsqueda

Problemas Búsqueda Completa

- Una solución de búsqueda completa puede recibir Time Limit Exceeded (TLE) como veredicto.
- Búsqueda completa se debe utilizar cuando no se puede llegar a una mejor solución - por lo menos nos permitirá recibir algunos puntos. Ejemplo en IOI, Codejam, Hackerrank, etc.
- Búsqueda completa puede actuar como un verificador para **pequeños casos**, proporcionando una prueba adicional para el algoritmo más rápido, pero no trivial que se desarrolle.

Contenido

Backtracking

- Definimos estados:
  - Tenemos un estado inicial "vacio".
  - Algunos estados son parciales.
  - Algunos estados son completos.
- Definimos transiciones desde un estado a otros posibles estados.
- Idea básica:
  1. Comenzar con un estado vacio.
  2. Usar recursion para recorrer todo el espacio de búsqueda por medio de los diferentes estados y sus transiciones.
  3. Si el estado actual es invalido, entonces paramos la exploración de esta rama.
  4. Procesamos todos los estados completados (estados que estabamos buscando).

Backtracking

Forma general de solución:

```
state S;

void generate() {
    if (!is_valid(S))
        return;

    if (is_complete(S))
        print(S);

    foreach (possible next move P) {
        apply move P;
        generate();
        undo move P;
    }
}

S = empty state;
generate();
```

Contenido

Búsqueda Completa Iterativa

Dado $6 < k < 13$ enteros, enumerar todos los posibles subconjuntos de tamaño 6 de estos enteros en orden. Por ejemplo:
k = 7, S = {1,2,3,4,5,6,7}
Los posibles subconjuntos en orden son:

{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 7}
{1, 2, 3, 4, 6, 7}
{1, 2, 3, 5, 6, 7}
{1, 2, 4, 5, 6, 7}
{1, 3, 4, 5, 6, 7}
{2, 3, 4, 5, 6, 7}

Búsqueda Completa Iterativa

Una posible solución es usar 6 bucles anidados que representen los subconjuntos de tamaño 6. Algo importante es que para el peor caso, es decir, k = 12, estos 6 bucles producirán $C_6^{12} = 924$ lineas de salida.

```
//O(n^6)
for( int a = 0 ; a < k - 5 ; ++a )
    for( int b = a + 1 ; b < k - 4 ; ++b )
        for( int c = b + 1 ; c < k - 3 ; ++c )
            for( int d = c + 1 ; d < k - 2 ; ++d )
                for( int e = d + 1 ; e < k - 1 ; ++e )
                    for( int f = e + 1 ; f < k ; ++f )
                        printf("%d %d %d %d %d %d\n",
                               S[a],S[b],S[c],S[d],S[e],S[f]);
```

Espacio de Búsqueda Complejo

Que ocurre cuando el espacio de búsqueda es más complejo?

- Todas las permutaciones de n items.
- Todos los subconjuntos de n items.

Como podriamos iterar sobre estos espacios de búsqueda?

Iterar sobre permutaciones

Dado un arreglo, mostrar todas las posibles permutaciones de dicho arreglo.

Podemos usar un método ya implementado **next_permutation** de la libreria **algorithm**.

```
//Hallar permutaciones - O(n!)
int n = 5;
vector<int> perm(n);
for (int i = 0; i < n; i++) perm[i] = i + 1;

do {
    for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]);
    }
    printf("\n");

} while (next_permutation(perm.begin(), perm.end()));
```

Búsqueda Completa Recursiva

Deseamos hallar todos los posibles subconjuntos de un tamaño determinado.

Una forma de resolver el problema es usando Backtracking.

```
const int n = 5;
bool pick[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            if (pick[i]) {
                printf("%d ", i+1);
            }
        }
        printf("\n");
    } else {
        // Considerar el elemento
        pick[at] = true;
        generate(at + 1);

        // No considerar el elemento
        pick[at] = false;
        generate(at + 1);
    }
}

generate(0); //Forma de llamar al m\'etodo.
```

Contenido

Conjuntos

- Útil y efectiva optimización de bajo nivel.
- Usando los bits de un entero podemos representar un conjunto.
- Todas las operaciones de conjuntos implican solamente la manipulación bits del entero correspondiente, lo que hace que sea una opción mucho más eficiente en comparación con **C++ STL vector<bool>**, **bitset**, o **set<int>**. Dicha velocidad es importante en la programación competitiva.

Bases

Las bases de la manipulación de bits son las operaciones sobre bits & (and), | (or), $\sim$ (not) and ^ (xor). Las primeras 3 operaciones deberian ya ser familiares con su forma booleana (&&, ||, ^)

A	B	!A	A && B	A || B	A^B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Las versiones de las operaciones booleanas sobre bits son las mismas, excepto que en lugar de interpretar sus argumentos como verdadero o falso, son operados sobre cada bit de los argumentos.

Operadores

De este modo, si A es 1010 y B es 1100, entonces:

- A & B = 1000
- A | B = 1110
- A ^ B = 0110
- ∼A = 11110101 (el número de 1's depende del tipo de dato de A).

Los otros dos operadores que necesitaremos son los operadores de desplazamiento a << b y a >> b.

Podemos pensar en el desplazamiento a la izquierda por b(a << b) como la multiplicación por 2^b y el desplazamiento a la derecha como la división por 2^b.

Otras operaciones

En general, usaremos un entero para representar un conjunto en el dominio de hasta 32 valores (o 64, usando un entero de 64 bits), con el bit 1 representando un miembro que esta presente y bit 0 uno que esta ausente. Teniendo en cuenta lo anteriores podemos realizar las siguientes operaciones:

- Set union: A | B
- Set intersection: A & B
- Set bit: A |= 1 << bit
- Toggle bit: A ^= (1 << bit)
- Clear bit: A &= ~(1 << bit)
- Test bit: (A & (1 << bit)) != 0

Contenido

Conteo de 1's

Deseamos hallar la cantidad de 1's que posee un número en su representación binaria.

Podemos hacer el conteo siguiendo una conversion a binario del número:

```
//O(log(n))
int countOneslogN(int x){
    int ones = 0;
    while(x){
        ones += (x & 1);
        x >>= 1;
    }
    return ones;
}
```

Conteo de 1's

Deseamos hallar la cantidad de 1's que posee un número en su representación binaria.

Podemos hacer el conteo tomando en cuenta el menor bit 1 de la representación binaria. Por ejemplo el número 10 = 1010 el menor bit 1 es 1010. Si restamos 1 al número, es decir, 9 = 1001. La posicion del menor bit 1 de 10 fue seteado a 0. Si realizamos la intersección entre ambos: x & (x-1) obtenemos el valor binario 1000.

```
//O(numero de 1's)
int countOnes( int x ){
    int ones = 0;
    while(x){
        x &= (x - 1);
        ones++;
    }
    return ones;
}
```

Todos los subconjuntos

Deseamos hallar todos los posibles subconjuntos de un tamaño determinado.

Podemos iterar sobre todos los posibles subconjuntos acorde al tamaño y verificar los bits activados y desactivados de cada subconjunto.

```
//O(2^n)
void allSubsets(int n){
    //Iterar sobre todos los posibles subsets entre 0 y (2^n - 1)
    for( int i = 0 ; i < 1<<n ; ++i){
        //Iterar sobre las posiciones de los bits
        for( int j = 0 ; j < n ; ++j ){
            //Verificar si el bit en la posicion j esta activado
            if( (x & (1 << j)) ){
                printf("1");
            }else
                printf("0");
        }
        printf("\n");
    }
}
```