

**MATERIAL VISUAL CON
RESPECTO A LENGUAJE
**SE INCLUYEN
ALGORITMOS Y CÓDIGO
FUENTE
*ENTRENAMIENTO
VERACRUZ 2019***

TEMAS INCLUIDOS:

➤ **QUEUE**

➤ **BFS**

➤ **STACK**

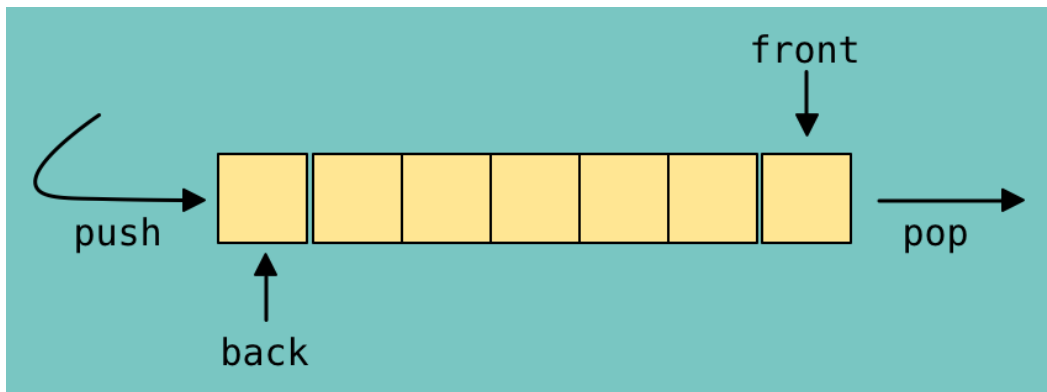
➤ **DFS**

➤ **BACKTRACKING**

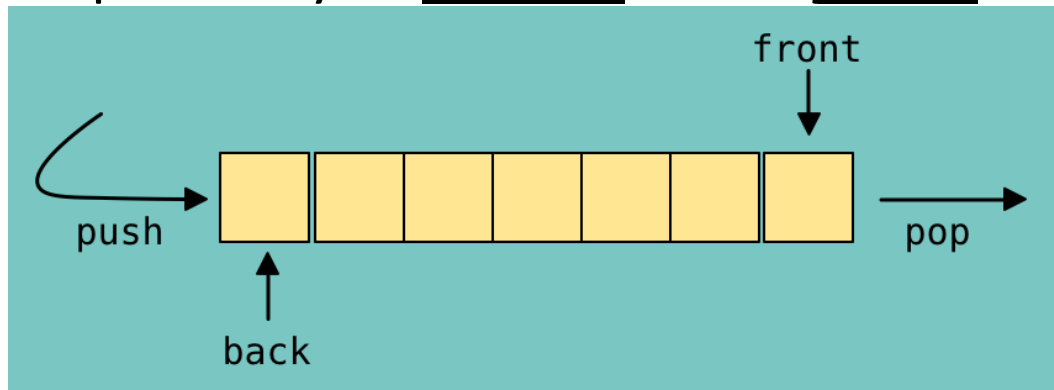
➤ **FLOODFILL**

QUEUE

*Estructura de Datos tipo FIFO (First- In
/ First- Out)*



- Una **queue** es un tipo de adaptador de contenedor, específicamente diseñado para operar en un contexto **FIFO** (primero en entrar, primero en salir [“first-in first-out”]), donde los elementos se insertan en un extremo del contenedor y se extraen del otro.
- Son clases que utilizan un objeto encapsulado de una clase de contenedor específica como su *contenedor subyacente*, proporcionando un conjunto específico de **funciones miembro** para acceder a sus elementos.
- Los elementos se **insertan** en la **"parte posterior"** del contenedor específico y se **extraen** de su **"frente"** .



■ Entre las funciones miembro de una queue resaltan:

- **empty()** *verifica si la queue está vacía, retorna true si es cierto, false de lo contrario.
- **size()** *retorna el tamaño de una queue, sólo números enteros positivos.
- **front()** *accede al elemento del frente de la queue.
- **back()** *accede al elemento último de la queue.
- **push()** *inserta un elemento (en la parte posterior)
- **pop()** *borra (retira) un elemento (de la parte frontal)

****NOTA:** Las funciones push y el pop se realizan según el principio del FIFO,

ESTRUCTURA DE QUEUE CON EL USO DE STRUCT

- Entre las ventajas de usar un struct, se encuentran, declarar métodos para usar con dicha estructura, y con nuestra queue, los usaremos:


```
4  struct dato
5  {
6      int vec[10000], ini=0, fin=0, cont=0, x;
7
8      void push(int x)
9      {
15
16      void pop()
17      {
22
23      int size()
24      {
27
28      bool empty()
29      {
32  };
33  dato cola;
```

- Vamos a utilizar 3 variables básicas:
 - **ini** *Representa la posición donde nuestra queue tiene su frente
 - **fin** *Representa la posición donde nuestra queue tiene su final
 - **cont** *Representa el tamaño de nuestra queue (cuantas casillas tiene).
 - **NOTA:** Si quieres omitir la creación del **contador**, puedes utilizar la siguiente operación para obtener el tamaño de tu queue:
 - ***fin-ini***

■ PUSH(VARIABLE)

- La función *push()* es declarada de tipo **void**, pues no devolverá ningún tipo de dato, simplemente la necesitamos para modificar nuestro arreglo (*vec[]*).
- En la declaración *void push(int x)*, subrayamos int x pues, esto declara que en este caso, utilizamos una queue de *int* e insertamos un valor del mismo tipo, si desea realizar su queue de otro tipo de dato, puede hacerlo, ya sea char's, float's, double's, etc.

8
9
10
11
12
13
14



```
void push(int x)
{
    vec[fin]=x;
    fin++;
    cont++;
    return;
}
```

1. Asigna x a la posición final
2. Incrementa la posición final una posición
3. Aumenta el tamaño de la queue en un valor

■ POP()

- La función *pop()* es declarada de tipo **void**, pues no devolverá ningún tipo de dato, simplemente la necesitamos para modificar nuestro arreglo (*vec[]*).
- En la declaración *void pop()*, vamos a “**eliminar**” un elemento de la queue, sin embargo, basta con “cancelar” esa posición de nuestro arreglo, para que no la usemos a futuro.

16
17
18
19
20
21



```
void pop()  
{  
    ini++;  
    cont--;  
    return;  
}
```

`ini++;`

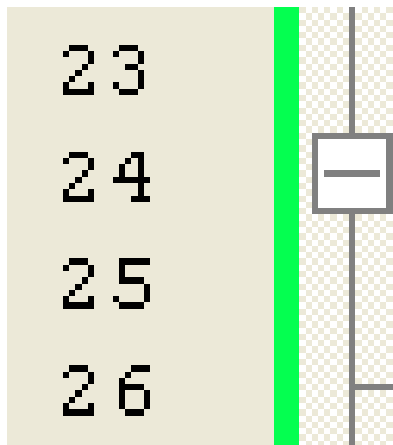
`cont--;`

`return;`

1. Incrementa el inicio de nuestra queue en una posición
2. Decrementa el tamaño de la queue en un valor

■ SIZE()

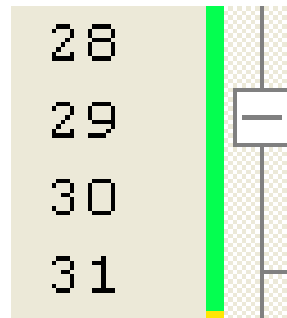
- La función *size()* es declarada de tipo **int**, pues devolverá el tamaño de nuestro arreglo (*vec[]*) en forma de int.
- En la declaración *int size()*, retornaremos como tal la variable **cont**, pues esta nos devuelve el tamaño de nuestra queue.



```
int size()  
{  
    return cont;  
}
```

■ EMPTY()

- La función *empty()* es declarada de tipo **bool**, pues devolverá si nuestro arreglo (*vec[]*) está o no vacío, en forma de **true** o **false**.
- En la declaración *bool empty()*, retornaremos **1** si nuestra queue esta **vacía**, y **0** si **tiene al menos un elemento**.



```
bool empty()  
{  
    return !cont;  
}
```

!		cont	Resultado
0	*	1, 2, ..., fin (con datos)	0 [FALSE]
0	*	0 (vacío)	1 [TRUE]

- Ejemplo de ejecución de programa, usando push(dato), pop(), size(), empty().

```
33 dato cola;  
34 int main()  
35 {  
36     cout<<cola.empty()<<" " <<cola.size()<<"\n";  
37     cola.push(1);  
38     cout<<cola.empty()<<" " <<cola.size()<<"\n";  
39     cola.push(2);  
40     cout<<cola.empty()<<" " <<cola.size()<<"\n";  
41     cola.push(3);  
42     cout<<cola.empty()<<" " <<cola.size()<<"\n";  
43     cola.pop();  
44     cout<<cola.empty()<<" " <<cola.size()<<"\n";  
45     cola.pop(), cola.pop();  
46     cout<<cola.empty()<<" " <<cola.size()<<"\n";  
47     return 0;  
48 }
```

```
1 0  
0 1  
0 2  
0 3  
0 2  
1 0
```

```
Process returned 0 (0x0)   execution time : 0.063 s  
Press any key to continue.
```

■ BACK() & FRONT()

- Ambas funciones serán del tipo de dato que contiene nuestra queue, pues harán referencia al valor que posee nuestra queue en esos momentos.

```
33     int front_()
34     {
35         return vec[ini];
36     }
37
38     int back_()
39     {
40         return vec[fin-1];
41     }
42 };
43 dato cola;
44 int main()
45 {
46     cola.push(1);
47     cola.push(2);
48     cola.push(3);
49     cout<<cola.front_()<<" "<<cola.back_();
50     return 0;
51 }
```

```
1 3
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
-
```

QUEUE DE LA STL

- En caso de carecer de tiempo para construir nuestra queue, podemos utilizar la de la STL, se llama igualmente queue, pero requiere algunas especificaciones para usarse:
 - 1. Incluir la librería “*queue*” en nuestro apartado de librerías
 - 2. Declarar nuestra queue de la siguiente manera:
 - `queue < tipo_de_dato > nombre;`

EJEMPLO

```
1  #include <iostream>
2  #include <queue>
3
4  using namespace std;
5
6  queue <int> cola;
7
8  int main()
9  {
10     cout<<cola.empty()<<" " <<cola.size()<<"\n";
11     cola.push(1), cola.push(2), cola.push(3);
12     cout<<cola.front()<<" " <<cola.back()<<"\n";
13     cout<<cola.empty()<<" " <<cola.size()<<"\n";
14     cola.pop(), cola.pop(), cola.pop();
15     cout<<cola.empty()<<" " <<cola.size()<<"\n";
16     return 0;
17 }
```

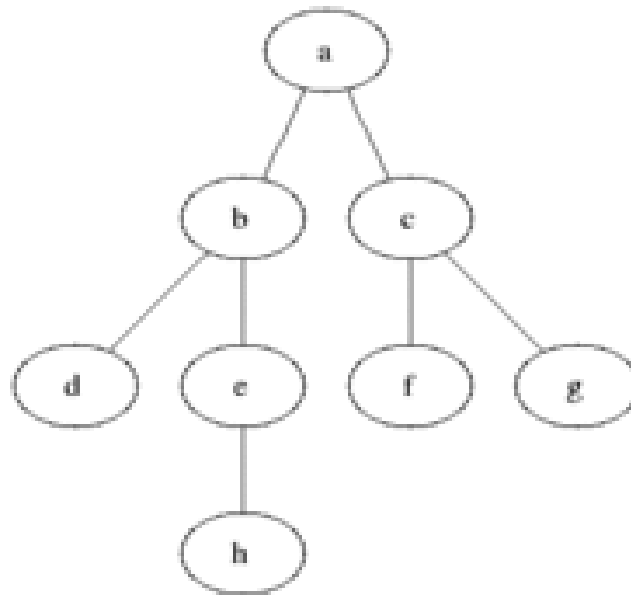
```
1 0
1 3
0 3
1 0
```

```
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```



BFS

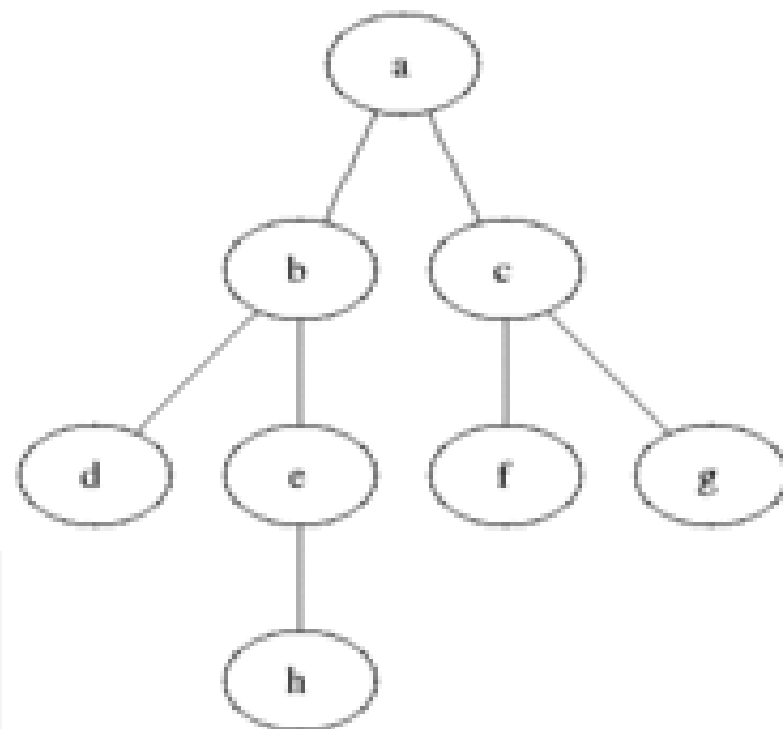
“Como llegar en el menor número de pasos de ‘A’ a ‘B’”



- **La Búsqueda en anchura** (en inglés *BFS -Breadth First Search*) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles).
- Intuitivamente, **se comienza en la raíz** (eligiendo algún nodo como elemento raíz en el caso de un grafo) **y se exploran todos los vecinos de este nodo**. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.
- Formalmente, *BFS* es un algoritmo de *búsqueda sin información*, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Algoritmo en pseudocódigo

```
1  método BFS(Grafo, origen):  
2      creamos una cola Q  
3      agregamos origen a la cola Q  
4      marcamos origen como visitado  
5      mientras Q no este vacío:  
6          sacamos un elemento de la cola Q llamado v  
7          para cada vertice w adyacente a v en el Grafo:  
8              si w no ah sido visitado:  
9                  marcamos como visitado w  
10                 insertamos w dentro de la cola Q
```



Entre los principales puntos que debemos ubicar antes de hacer una BFS, están:

✓ **ESTADO INICIAL**

Origen de la búsqueda

✓ **ESTADO FINAL**

Final de la búsqueda

✓ **ESPACIO VALIDO**

Espacio por el cual puedo navegar

✓ **ESPACIO INVALIDO**

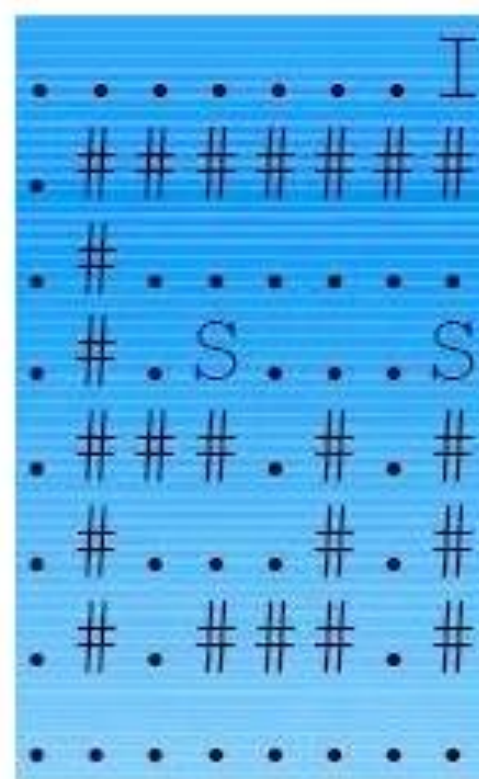
Espacio por el cual no puedo navegar

✓ **TRANSICIONES**

Movimientos

Ejemplo Aplicativo:

Tenemos una matriz de caracteres que representa un laberinto 2D un '#' implica un muro, un '.' implica un espacio libre, un 'I' indica la entrada del laberinto y una 'S' indica una salida.



¿Cuánto mide la ruta más corta para escapar?

SOLUCIÓN:

El problema nos da un Estado inicial y final, en este caso “I” y “S”, nos pide el **menor numero de pasos** para ir de “I” a “S” tomando en consideración que solo podremos avanzar por los “.”

Dicha la descripción anterior, el problema puede ser resuelto por BFS al tener un estado inicial y final e ir avanzando por algun adyacente que no sea “#”.

SALIDA

ACEPTABLE:

8 8

n, m (límites)

```

. . . . . I
. #####
. #. . . . .
. #. 5. . . . S
. ###. #. #
. #. . . #. #
. #. ###. #
. . . . .
    
```

Plano original

Pasos mínimos para
llegar de 'I' a la 'S'
más cercana

Camino a seguir

23

```

  7  6  5  4  3  2  1  I
  8  #  #  #  #  #  #  #
  9  #  .  .  .  .  .  .
 10  #  . 23 22  .  .  S
 11  #  #  # 21 #  .  #
 12  # 18 19 20 #  .  #
 13  # 17  #  #  #  .  #
 14 15 16  .  .  .  .  |
    
```

PUNTOS BÁSICOS DE ANÁLISIS:

✓ ESTADO INICIAL

Casilla con la letra 'I'.

✓ ESTADO FINAL

Casilla con la letra 'S'.

✓ ESPACIO VALIDO

Casillas dentro de los límites de la matriz que tengan la letra 'I', 'S' o '.' y que no hayan sido visitadas con anterioridad.

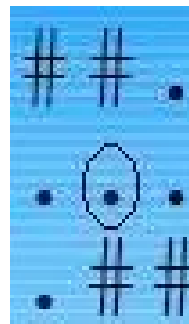
✓ ESPACIO INVALIDO

Casillas fuera de los límites de la matriz, casillas con '#' o casillas ya visitadas.

✓ TRANSICIONES

Dado que nos podemos mover en las casillas adyacentes de los cuatro puntos cardinales, podemos movernos hacia arriba, abajo, izquierda y derecha.

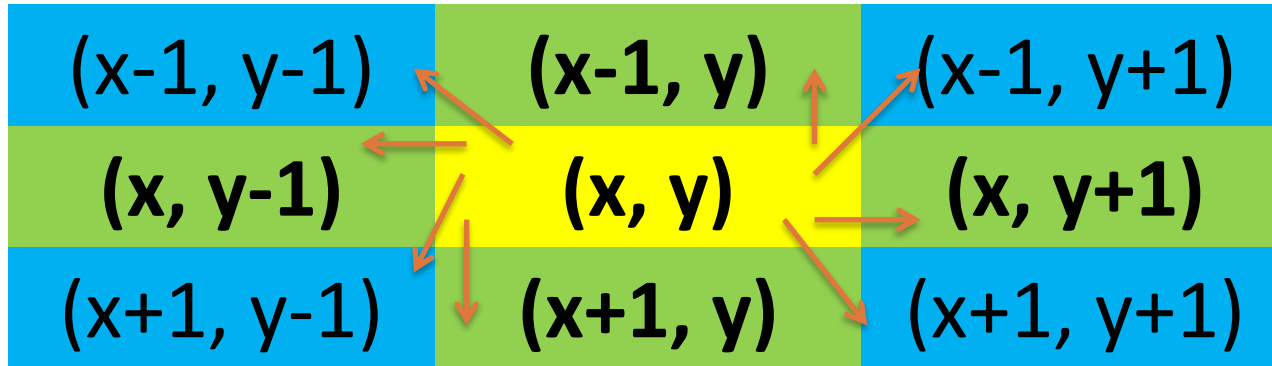
Así por ejemplo si estoy en la posición siguiente (círculo) y suponiendo que sea la coordenada 3,4 :



Los posibles Estados adyacentes serían: izquierda (3,2 -> "."), derecha (3,5 -> "."), arriba (2,4 -> "#") y abajo (4,4 -> "#") por lo tanto para ese caso solo podríamos avanzar por la derecha o izquierda.

NOTA:

TRANSICIONES POSIBLES DESDE UN PUNTO (x, y) SI DESARROLLAMOS UNA BFS EN UNA MATRIZ



TOMANDO EN CUENTA LA CASILLA AMARILLA COMO EL ORIGEN:

Si le piden avanzar a las **4 casillas adyacentes**: NORTE, SUR, ESTE, OESTE

TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES**

Si le piden avanzar a las **8 casillas adyacentes**

TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES Y AZULES**

A CONTINUACIÓN SE PRESENTARÁ UN CÓDIGO CON LA SOLUCIÓN DEL PROBLEMA, ES RECOMENDABLE LEERLO Y ADAPTARLO A SU ESTILO DE PROGRAMACIÓN

Este código contiene:

- Obtener los pasos mínimos para llegar de la coordenada 'A' hacia 'B'
- Obtener el camino a seguir, según los pasos mínimos

Primero, vamos a definir las variables que usaremos:

```
6 struct dato
7 {
8     int x, y, pasos, x_ant, y_ant;
9 };
10
11 struct dato1
12 {
13     int x_pas, y_pas;
14 };
15
16 int n, m, camino[101][101]=(-1); ///limites y camino es usado para guardar la ruta más corta
17 bool marcas[101][101]={0}; ///una matriz de bool's para guardar las casillas ya visitadas
18 char mat[101][101]; /// una matriz de char's para guardar el plano original
19 queue <dato> cola; /// queue para encolar mis estados y proceder con la BFS
20 dato1 anterior[101][101], actual; ///matriz y variable para guardar la posición de donde provengo
21 dato padre, hijo, fun_especial; /// auxiliares los cuales guardan mi posición actual,
22                                     /// la posición de donde provengo y la distancia de mi origen
23                                     /// los cuales, usaré para encolar y proceder con la BFS
24
```

Nuestro programa principal contiene:

- **Línea 62:** Lectura de la dimensión de la matriz
- **Líneas 63-69:** Lectura del mapa
- **Líneas 67-68:** Identificamos la casilla donde está la 'I' y la encolamos en la **línea 71**, con ello nos ahorramos otra búsqueda de $O(n*m)$ para ubicarla.

```
60  int main()  
61  {  
62      cin>>n>>m;  
63      for(int i=0; i<n; i++)  
64          for(int j=0; j<m; j++)  
65          (  
66              cin>>mat[i][j];  
67              if(mat[i][j]=='I')  
68                  padre={i, j, 0, -1, -1};  
69          )  
70  
71      cola.push(padre);  
72      ///padre.x=0  
73      ///padre.y=7  
74      ///padre.pasos=0  
75      ///padre.x_ant=-1  
76      ///padre.y_ant=-1
```

Para **encolar un dato**,
ingresaremos nuestras
coordenadas **(i, j)**, las casillas
recorridas **“pasos”** y la casilla de la
cual provenimos **“x_ant”** y **“y_ant”**

A partir de la línea 78, comenzamos con nuestra queue, con ello, procede la BFS.

Importante: Gracias a las propiedades de la queue, siempre tomaremos el frente de la queue para trabajar, con ello, nuestra búsqueda se expandirá a los 4 lados simultáneamente siempre que sea posible.

```
78 while(!cola.empty())
79 {
80     padre=cola.front();    ///la posicion que tomo para proceder
81                           ///será el frente de la queue (.front())
82     cola.pop(); /// elimino ese nodo, pues lo procesaré a futuro
83
84     if(mat[padre.x][padre.y]=='S') /// pregunto si he llegado a 'S'
85     {
86         final_encontrado(padre);
87         return 0;
88     }
89     ///si el pasado if devuelve un false, quiere decir que estoy en un '.'
90
91     marcas[padre.x][padre.y]=true; /// entonces, marco esta posición en mi
92     /// matriz de bool's, para no volver a visitar y así, evitar un bucle
93
94     ///marco en mi matriz de "pasados", la casilla de la cual provengo
95     anterior[padre.x][padre.y].x_pas=padre.x_ant;
96     anterior[padre.x][padre.y].y_pas=padre.y_ant;
```

El último paso es definir las transiciones, en este caso, en los 4 puntos adyacentes, para armar los if's, debemos tener presente algunos puntos al momento de proceder con el movimiento:

- ✓ Permanecer dentro de la matriz (o mapa) --PRIMER CONDICIÓN--
- ✓ No moverse hacia una casilla inválida --SEGUNDA CONDICIÓN--
- ✓ No moverse hacia una casilla ya visitada --TERCERA CONDICIÓN--

```
98      ///encolar transiciones
99
100     ///arriba
101     if (padre.x-1>=0 && mat[padre.x-1][padre.y]!='#' && !marcas[padre.x-1][padre.y])
102         cola.push(hijo={padre.x-1, padre.y, padre.pasos+1, padre.x, padre.y});
103
104     ///abajo
105     if (padre.x+1<n && mat[padre.x+1][padre.y]!='#' && !marcas[padre.x+1][padre.y])
106         cola.push(hijo={padre.x+1, padre.y, padre.pasos+1, padre.x, padre.y});
107
108     ///izquierda
109     if (padre.y-1>=0 && mat[padre.x][padre.y-1]!='#' && !marcas[padre.x][padre.y-1])
110         cola.push(hijo={padre.x, padre.y-1, padre.pasos+1, padre.x, padre.y});
111
112     ///derecha
113     if (padre.y+1<m && mat[padre.x][padre.y+1]!='#' && !marcas[padre.x][padre.y+1])
114         cola.push(hijo={padre.x, padre.y+1, padre.pasos+1, padre.x, padre.y});
115     }
116 }
```

Vamos a regresar al if de la línea 84, gracias a las propiedades de una BFS, se nos asegura que si estamos en una casilla que no hemos visitado antes, hemos llegado a dicha casilla de la forma más corta.

Por lo tanto, si llegamos a una 'S', según el problema, **imprimiremos los pasos mínimos que hemos recorrido, y el camino que seguimos**, por lo tanto, como proceden varias actividades, declaramos una función para realizarlas, para mantener el programa ordenado.

```
78 while(!cola.empty())
79 {
80     padre=cola.front();    ///la posicion que tomo para proceder
81                           ///será el frente de la queue (.front())
82     cola.pop();    /// elimino ese nodo, pues lo procesaré a futuro
83
84     if(mat[padre.x][padre.y]=='S')    /// pregunto si he llegado a 'S'
85     {
86         final_encontrado(padre);
87         return 0;
88     }
89     ///si el pasado if devuelve un false, quiere decir que estoy en un '.'
90
91     marcas[padre.x][padre.y]=true;    /// entonces, marco esta posición en mi
92     /// matriz de bool's, para no volver a visitar y así, evitar un bucle
93
```

Función utilizada para imprimir las premisas

La función va a tomar como parámetro “fun_especial”, una variable de tipo “dato”, la cual en realidad es nuestra variable “padre” que antes había estado encolada, justamente en la diapositiva anterior.

Después de ello, vamos a trabajar con nuestra matriz donde guardamos nuestro camino recorrido, ¿Cómo? Sencillo: procederemos desde la casilla con ‘S’ hasta la casilla ‘I’, marcándola con nuestros pasos, por lo cual, procederemos de manera descendente, regresando a la casilla de donde alguna vez me moví, dichas coordenadas están guardadas en “x_ant” y “y_ant”.

```
25 void final_encontrado(dato fun_especial)
26 {
27     cout<<fun_especial.pasos<<"\n";    ///imprime los pasos que he realizado
28
29     ///uso mi matriz de camino usado, para declarar que mi a mi 'S'
30     ///he llegado con los pasos que mi función me retorna
31     camino[fun_especial.x][fun_especial.y]=fun_especial.pasos;
32     fun_especial.pasos--; ///disminuyo mis pasos
33
34     ///uso una variable "actual" para regresar a través de mis pasos
35     actual.x_pas=fun_especial.x_ant;    ///actualizo mi coordenada x
36     actual.y_pas=fun_especial.y_ant;    ///actualizo mi coordenada y
37
```

El procedimiento que sigo es sencillo, no es el único método, pero es útil y sencillo de programar:

En todo momento, vas a usar la matriz del camino recorrido, debes tomar tu posición actual, y en tu matriz de caminos en esa misma posición colocarás los pasos que llevas, una vez hecho eso, disminuyes en un valor los pasos, para dejar preparados los pasos para la anterior posición.

Luego de ello, como tu variable “actual” tiene las coordenadas de donde vienes, simplemente tienes que actualizar tu “actual” (x, y), a las coordenadas de (x_ant, y_ant), de la siguiente manera:

```
38 ///uso un ciclo para regresar a mis pasos mientras no esté en 'I'
39 while (mat[actual.x_pas][actual.y_pas] != 'I')
40 {
41     ///declaro que en esa casilla he llegado con una cantidad k de pasos:
42     camino[actual.x_pas][actual.y_pas]=fun_especial.pasos;
43     ///disminuyo los pasos en una unidad, para que la siguiente casilla a
44     ///visitar, tenga los pasos que yo, menos uno
45     fun_especial.pasos--;
46
47 ///uso dos variables auxiliares para guardar temporalmente mis pasadas coordenadas
48     int aux=anterior[actual.x_pas][actual.y_pas].x_pas;
49     int aux2=anterior[actual.x_pas][actual.y_pas].y_pas;
50     ///colocamos nuestra casilla pasada a la variable "actual", para volver al ciclo
51     actual.x_pas=aux;
52     actual.y_pas=aux2;
53 }
```

Al final, imprimiremos la matriz con la ayuda de dos matrices que declaramos, de la siguiente manera:

- ✓ Si en el mapa hay un '.', una 'S', un '#' o una 'I' y en nuestra matriz de caminos tenemos un número '0' o '-1', debemos imprimir lo que tenemos en el mapa, la letra, en resumen; pues, el '-1' quiere decir que no es parte de un camino válido.
- ✓ En caso de que, en sencillas palabras, nuestra matriz de caminos tenga un número que no sea '-1', significa que es parte del camino válido, por lo tanto, debemos imprimir el número.

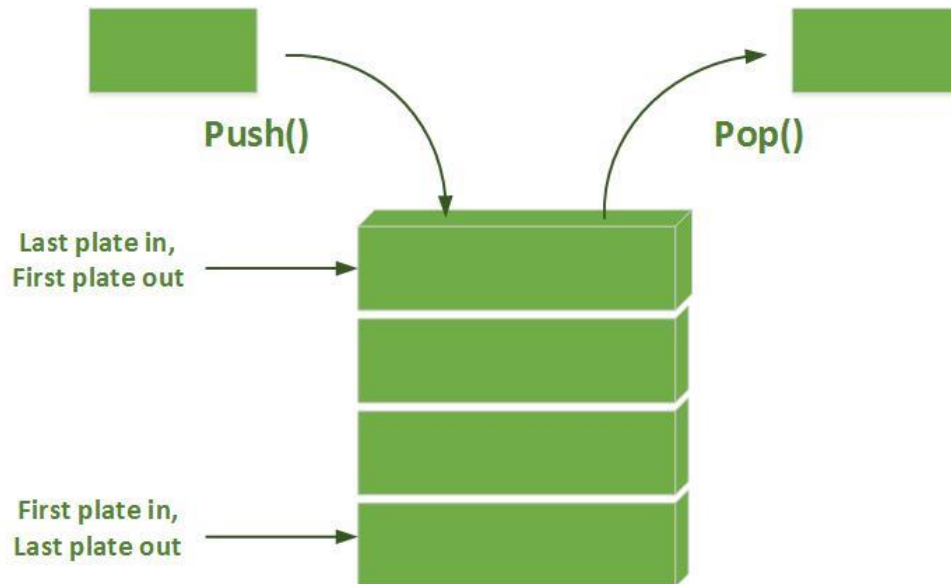
Por ello, debemos usar ambas matrices y alternar las impresiones.

```
55 for(int i=0; i<n; i++)
56 {
57     for(int j=0; j<m; j++)
58     {
59         if((mat[i][j]=='.' || mat[i][j]=='S' || mat[i][j]=='#' || mat[i][j]=='I')
60             && (camino[i][j]==0 || camino[i][j]==-1) )
61             cout<<mat[i][j]<<" ";
62         else
63             cout<<camino[i][j]<<" ";
64     }
65     cout<<"\n";
66 }
```

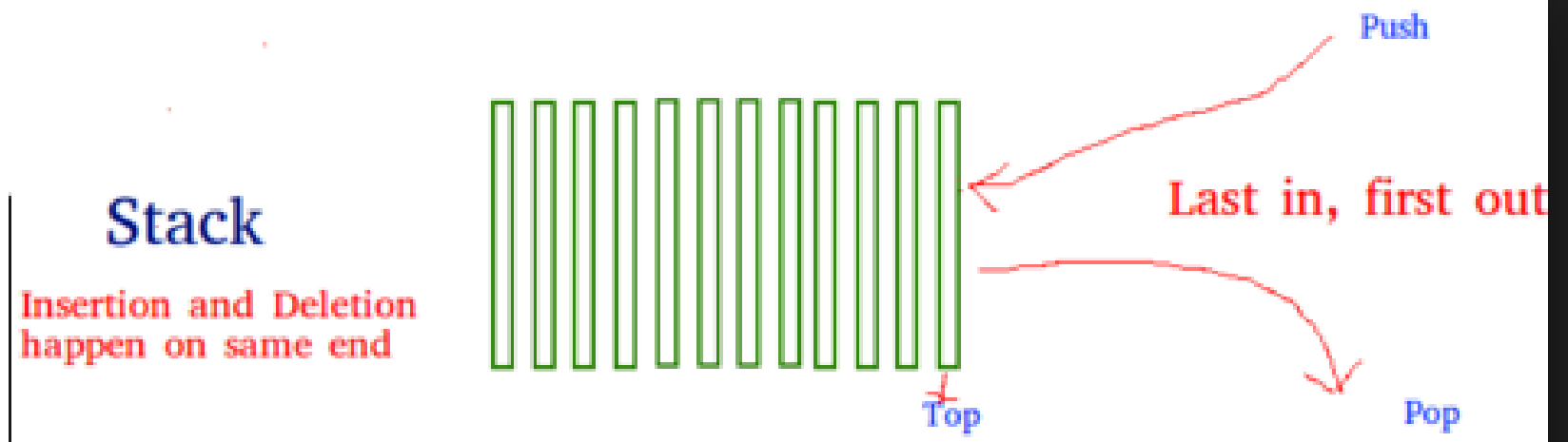


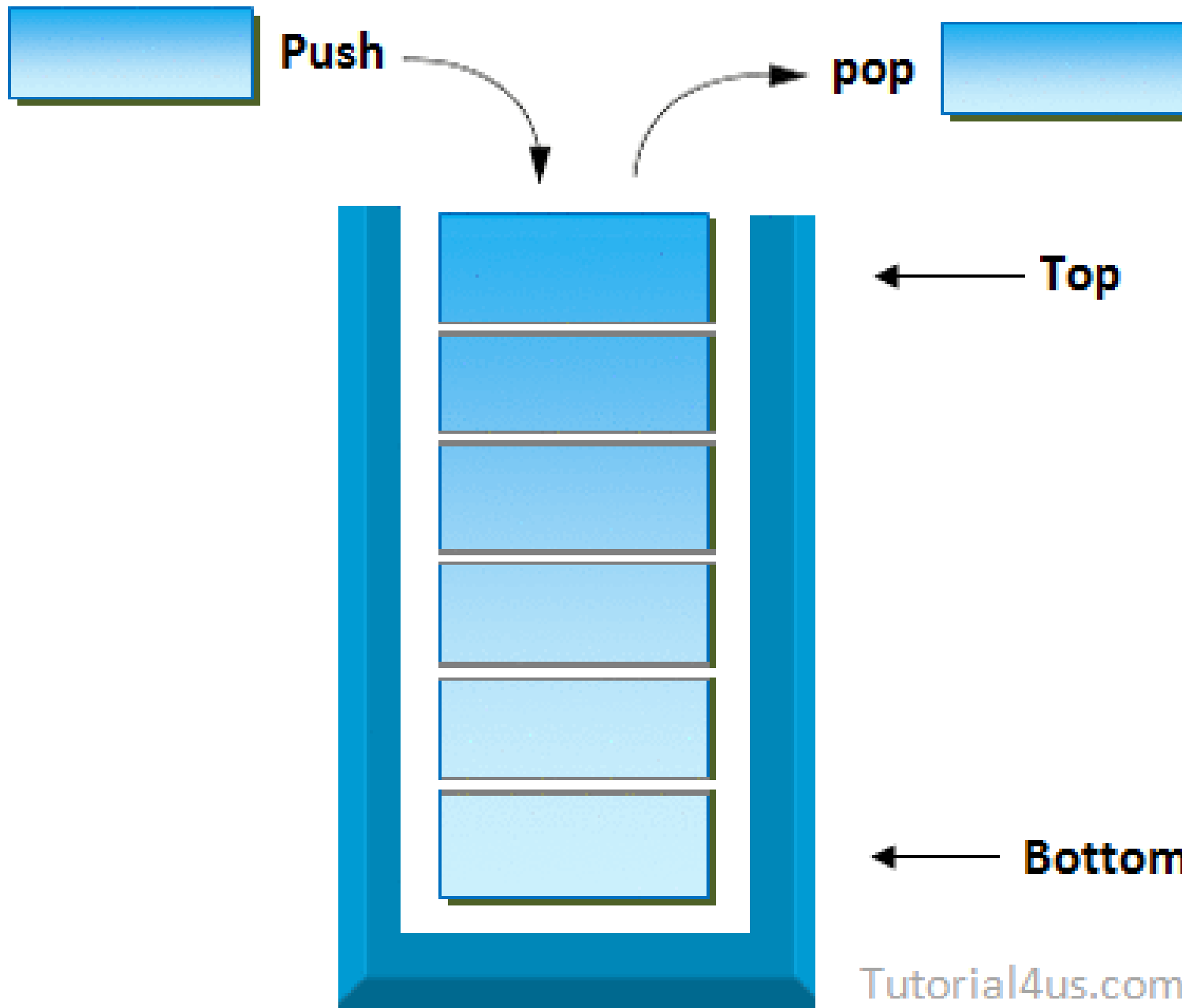
STACK

*Estructura de Datos tipo LIFO (Last- In
/ First- Out)*



- Las **pilas** son un tipo de adaptador de contenedor, específicamente diseñado para operar en un contexto **LIFO (último en entrar, primero en salir)**, donde los elementos se insertan y se extraen solo de un extremo del contenedor.
- Las **pilas** se implementan como *adaptadores de contenedores*, que son clases que usan un objeto encapsulado de una clase de contenedor específica como su *contenedor subyacente*.
- Proporcionando un conjunto específico de funciones miembro para acceder a sus elementos. Los elementos **se empujan / saltan desde la "parte posterior"** del contenedor específico, que se conoce como la *parte superior* de la pila.





■ Entre las funciones miembro de una stack resaltan:

- **empty()** *verifica si la queue está vacía, retorna true si es cierto, false de lo contrario.
- **size()** *retorna el tamaño de una queue, sólo números enteros positivos.
- **top()** *accede al “tope” de la pila, básicamente al último elemento insertado.
- **push()** *inserta un elemento (en la parte posterior)
- **pop()** *borra (retira) un elemento (de la parte frontal)

****NOTA: Las funciones push y el pop se realizan según el principio del LIFO**

ESTRUCTURA DE STACK CON EL USO DE STRUCT

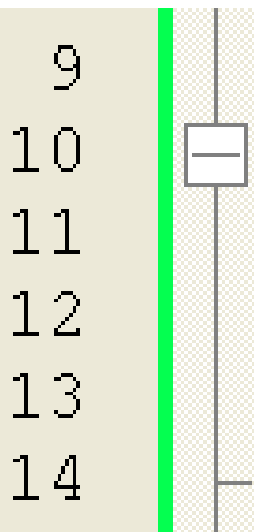
- Entre las ventajas de usar un struct, se encuentran, declarar métodos para usar con dicha estructura, y con nuestra stack, los usaremos:

```
5 struct dato
6 {
7     int vec[10000], tope=0, x;
8
9     void push(int x)
10    {
11
12
13
14
15
16    void pop()
17    {
18
19
20
21
22    int size()
23    {
24
25
26
27    bool empty()
28    {
29
30
31    };
32    dato pila;
```

- Vamos a utilizar 1 variable básica:
 - **tope** *La cual nos va a servir para determinar:
 - Tamaño de la pila
 - Final de la pila, tanto para hacer push, como para hacer pop
 - Si está o no vacía la pila

■ PUSH(VARIABLE)

- La función *push()* es declarada de tipo **void**, pues no devolverá ningún tipo de dato, simplemente la necesitamos para modificar nuestro arreglo (*vec[]*).
- En la declaración *void push(int x)*, subrayamos int x pues, esto declara que en este caso, utilizamos un stack de *int* e insertamos un valor del mismo tipo, si desea realizar su queue de otro tipo de dato, puede hacerlo, ya sea char's, float's, double's, etc.

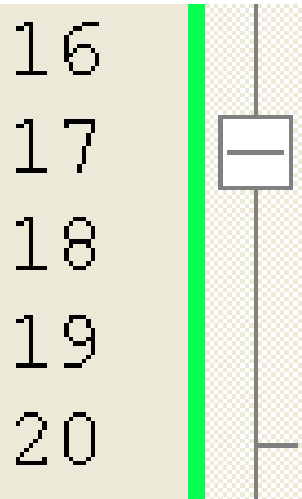


```
void push(int x)
{
    vec[tope]=x;
    tope++;
    return;
}
```

1. Asigna x al tope de la pila
2. Incrementa el tope una posición

■ POP()

- La función *pop()* es declarada de tipo **void**, pues no devolverá ningún tipo de dato, simplemente la necesitamos para modificar nuestro arreglo (*vec[]*).
- En la declaración *void pop()*, vamos a “**eliminar**” un elemento de la stack, sin embargo, basta con retroceder nuestro tope, y en caso de un futuro *push*, sobrescribir el valor



```
void pop()  
{  
    tope--;  
    return;  
}
```

1. Decrementa el tope
en una posición

■ SIZE()

- La función *size()* es declarada de tipo **int**, pues devolverá el tamaño de nuestro arreglo (*vec[]*) en forma de int.
- En la declaración *int size()*, retornaremos como tal la variable **tope**, pues el tope nos devuelve la cantidad de elementos.

22

23

24

25



```
int size()
```

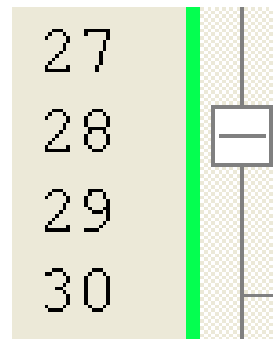
```
{
```

```
    return tope;
```

```
}
```

■ EMPTY()

- La función *empty()* es declarada de tipo **bool**, pues devolverá si nuestro arreglo (*vec[]*) está o no vacío, en forma de **true** o **false**.
- En la declaración *bool empty()*, retornaremos **1** si nuestra stack esta **vacía**, y **0** si **tiene al menos un elemento**.



```
bool empty()  
{  
    return !tope;  
}
```

!		tope	Resultado
0	*	1, 2, ..., fin (con datos)	0 [FALSE]
0	*	0 (vacío)	1 [TRUE]

- Ejemplo de ejecución de programa, usando push(dato), pop(), size(), empty().

```
32 dato pila;  
33 int main()  
34 {  
35     cout<<pila.empty() <<" " <<pila.size() <<"\n";  
36     pila.push(1);  
37     cout<<pila.empty() <<" " <<pila.size() <<"\n";  
38     pila.push(2);  
39     cout<<pila.empty() <<" " <<pila.size() <<"\n";  
40     pila.push(3);  
41     cout<<pila.empty() <<" " <<pila.size() <<"\n";  
42     pila.pop();  
43     cout<<pila.empty() <<" " <<pila.size() <<"\n";  
44     pila.pop(), pila.pop();  
45     cout<<pila.empty() <<" " <<pila.size() <<"\n";  
46     return 0;
```

```
1 0  
0 1  
0 2  
0 3  
0 2  
1 0
```

Process returned 0 (0x0) execution time : 0.031 s
Press any key to continue.

STACK DE LA STL

- En caso de carecer de tiempo para construir nuestra stack, podemos utilizar la de la STL, se llama igualmente stack, pero requiere algunas especificaciones para usarse:
 - 1. Incluir la librería “*stack*” en nuestro apartado de librerías
 - 2. Declarar nuestra stack de la siguiente manera:
 - `stack < tipo_de_dato > nombre;`

EJEMPLO

```
1  #include <iostream>
2  #include <stack>
3  using namespace std;
4  stack <int> pila;
5  int main()
6  {
7      cout<<pila.empty()<<" " <<pila.size()<<"\n";
8      pila.push(1), pila.push(2), pila.push(3);
9      cout<<pila.top()<<"\n";
10     pila.pop();
11     cout<<pila.empty()<<" " <<pila.size()<<"\n";
12     cout<<pila.top()<<"\n";
13     pila.pop();
14     cout<<pila.empty()<<" " <<pila.size()<<"\n";
15     cout<<pila.top()<<"\n";
16     pila.pop();
17     cout<<pila.empty()<<" " <<pila.size()<<"\n";
18     return 0;
```

```
1 0
3 3
0 2
2 2
0 1
1 1
1 0
```

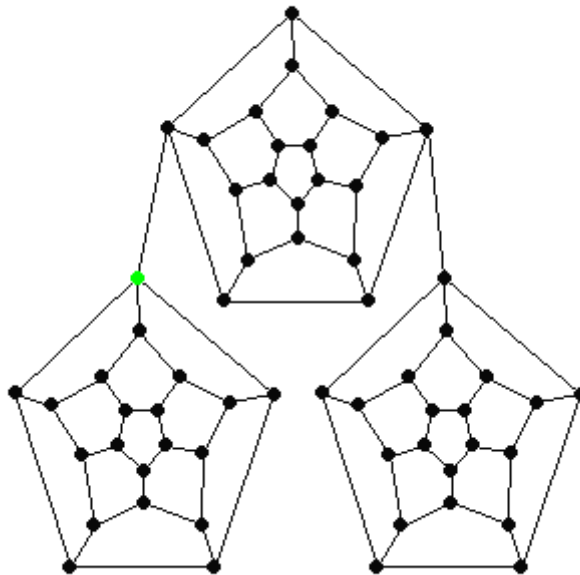
Process returned 0 (0x0) execution time : 0.031 s
Press any key to continue.

DFS

“¿Qué tan profundo es este camino?”

“¿De cuántas formas puedo llegar de ‘A’ a ‘B’?”

Depth-First Search



- El **algoritmo DFS** posee varias aplicaciones la más importante es para **problemas de conectividad**, si un **grafo es conexo**, **detectar ciclos en un grafo**, **numero de componentes conexas**, etc y es bastante útil en otros algoritmos como para hallar las componentes fuertemente conexas en un grafo, para hallar puntos de articulación o **componentes biconexas**, para recorrido en un circuito o **camino euleriano**, **topological sort**, **flood fill** y otras aplicaciones.
- DFS va formando un árbol al igual que BFS pero lo hace a profundidad. Existen dos formas de hacer el recorrido una es usando una Pila y otra de manera recursiva:

USANDO STACK

El concepto es el mismo que BFS solo que se cambia la Cola por una Pila, el proceso es como sigue: **visitar el nodo inicial y ponerlo en la pila**, ahora para ver los siguientes nodos a visitar **sacamos el nodo tope de la pila y vemos sus adyacentes**, los que **no han sido visitados** los **insertamos en la pila**. El proceso **se repite hasta que la pila se encuentre vacía** (se han visitado todos los nodos).

Algoritmo en pseudocódigo:

```
1  método DFS( origen):
2      creamos una pila S
3      agregamos origen a la pila S
4      marcamos origen como visitado
5      mientras S no este vacío:
6          sacamos un elemento de la pila S llamado v
7          para cada vertice w adyacente a v en el Grafo:
8              si w no ah sido visitado:
9                  marcamos como visitado w
10                 insertamos w dentro de la pila S
```

NOTA: PSEUDOCÓDIGO RECOMENDADO PARA TEORÍA DE GRAFOS

Usando Recursión

Usar la recursión es mucho mas fácil y ademas muy útil, es la forma mas usada en la solución de problemas con este algoritmo.

Algoritmo en pseudocódigo:

```
1  método DFS( origen ) :  
2      marcamos origen como visitado  
3      para cada vertice v adyacente a origen en el Grafo:  
4          si v no ah sido visitado:  
5              marcamos como visitado v  
6              llamamos recursivamente DFS( v )
```

Entre los principales puntos que debemos ubicar antes de hacer una DFS, están:

✓ **ESTADO INICIAL**

Origen de la búsqueda

✓ **ESTADO FINAL**

Final de la búsqueda

✓ **ESPACIO VALIDO**

Espacio por el cual puedo navegar

✓ **ESPACIO INVALIDO**

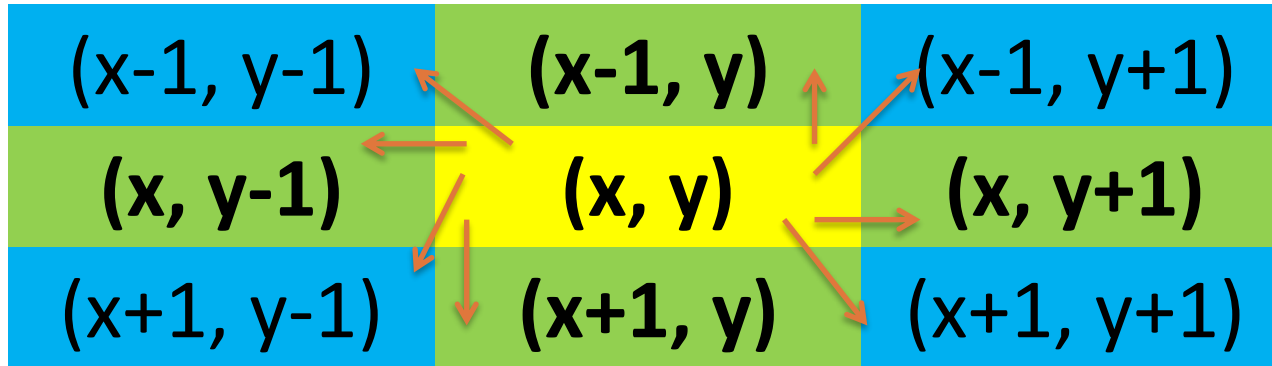
Espacio por el cual no puedo navegar

✓ **TRANSICIONES**

Movimientos

NOTA:

TRANSICIONES POSIBLES DESDE UN PUNTO (x, y) PARA DESARROLLAR UNA DFS EN LA MATRIZ



TOMANDO EN CUENTA LA CASILLA AMARILLA COMO EL ORIGEN:

Si le piden avanzar a las **4 casillas adyacentes**: NORTE, SUR, ESTE, OESTE

TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES**

Si le piden avanzar a las **8 casillas adyacentes**

TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES Y AZULES**



BACKTRACKING

“Usar la recursión, pero con una pequeña ventaja, tener una respuesta de por medio”

Los problemas que deben satisfacer un determinado tipo de restricciones son problemas completos, donde el orden de los elementos de la solución no importa. **Estos problemas consisten en un conjunto (o lista) de variables a la que a cada una se le debe asignar un valor sujeto a las restricciones del problema.**

La técnica va creando todas las posibles combinaciones de elementos para obtener una solución. **Su principal virtud es que en la mayoría de las implementaciones se puede evitar combinaciones, estableciendo funciones de acotación (o poda) reduciendo el tiempo de ejecución.**

Esencialmente, la idea es encontrar la mejor combinación posible en un momento determinado, por eso, se dice que este tipo de algoritmo es una búsqueda en profundidad.

Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa.

Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción (hijo [si nos referimos a un árbol]).

Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja).

Normalmente, se suele implementar este tipo de algoritmos como un procedimiento recursivo.

Así, en cada llamada al procedimiento se toma una **variable** y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados.

La diferencia con la búsqueda en profundidad es que **se suelen diseñar funciones de cota**, de forma que no se generen algunos estados si no van a conducir a ninguna solución, o a una solución peor de la que ya se tiene. De esta forma **se ahorra espacio en memoria y tiempo de ejecución**.

Algunas implementaciones muy sofisticadas usan una función de cotas, que examina si es posible encontrar una solución a partir de una solución parcial.

Además, **se comprueba si la solución parcial que falla puede incrementar significativamente la eficiencia del algoritmo.** Por el uso de estas funciones de cota, se debe ser muy minucioso en su implementación de forma que sean poco costosas computacionalmente hablando, ya que lo más normal es que se ejecuten en para cada nodo o paso del algoritmo.

Cabe destacar, que **las cotas eficaces se crean de forma parecida a las funciones heurísticas**, esto es, relajando las restricciones para conseguir mayor eficiencia.

PROBLEMA DE EJEMPLO:

Dado un entero n y una frase de n letras (sin espacios), genere todas las combinaciones posibles de dicho conjunto de letras.

SALIDA

ACEPTABLE:

3

ABC

Output:

ABC

ACB

BAC

BCA

CBA

CAB

La solución es relativamente sencilla de entender, sólo **imprimir las $n!$ diferentes cadenas**, que representan las permutaciones de nuestra frase, para ello, entre una de las soluciones encontramos el uso del **backtracking**, que a continuación explicaremos, iniciando por las variables:

```
5  int n;          ///TAMAÑO DE LA FRASE
6  char frase[101]; ///FRASE
```

Nuestro programa principal consiste en la lectura normal de las variables, luego de esto, debemos llamar a nuestra función “permuta” con tres parámetros, nuestro char, el inicio de la cadena (0) y el fin de la cadena (n-1, pues un char se indexa en 0).

```
31  int main()  
32  {  
33      cin>>n;  
34      for(int i=0; i<n; i++)  
35          cin>>frase[i];  
36  
37      permuta(frase, 0, n-1);  
38  
39      return 0;  
40  }
```

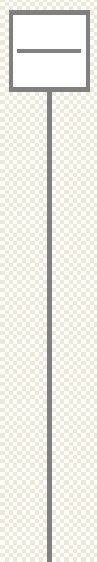
En la función, manejaremos el char con ayuda de un puntero, seguido de las dos variables de límite. El procedimiento implica un método que se describe como *swapy*, básicamente es un swap, pero con una modificación que adelante se describirá.

Intercambiamos siempre los inicios con los fines de nuestro arreglo, sólo **debemos intermediar con una llamada a función** donde el ini que mandaremos, incrementará un valor.

```
16 void permuta(char *a, int ini, int fin)
17 {
18     if (ini==fin)
19         cout<<a<<"\n";
20     else
21     {
22         for(int i=ini; i<=fin; i++)
23         {
24             swapy( (a+ini), (a+i) );
25             permuta( a, ini+1, fin );
26             swapy( (a+ini), (a+i) ); /// backtrack
27         }
28     }
29 }
```

Swapy trabaja como un swap normal, a diferencia de que usaremos punteros para intercambiar las letras de nuestra frase, es sólo jugar un poco con la asignación, puede eliminar esta función e implementarla directamente a la función *permuta*.

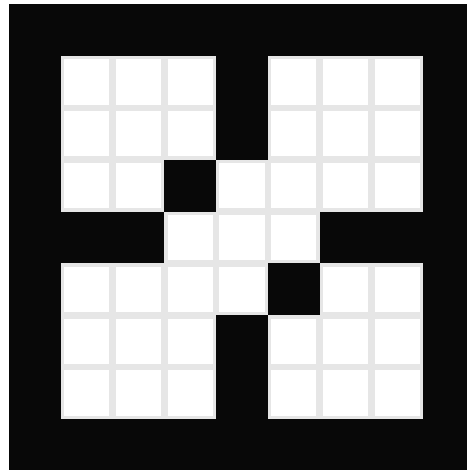
```
8 void swapy(char *x, char *y)
9 {
10     char aux;
11     aux=*x;
12     *x=*y;
13     *y=aux;
14 }
```





FLOODFILL

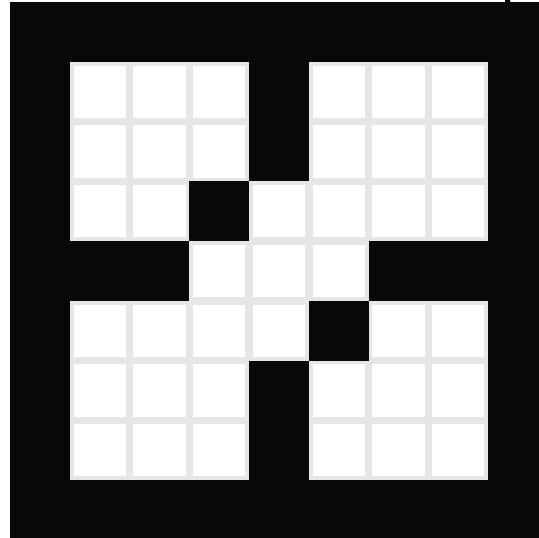
“¿Qué tan grande es una mancha?”



FLOODFILL

Requiere tres parámetros: un nodo inicial, un color para sustituir, y otro de relleno. El algoritmo rastrea todos los nodos que sean del color seleccionado, y a la vez contiguos entre sí y con el inicial, y los sustituye por el color de relleno.

Hay muchas maneras en las que el algoritmo de relleno por difusión puede ser estructurado, pero todas ellas hacen uso de tipos de datos tales como la cola o la pila, explícita o implícitamente.



Una implementación del algoritmo de relleno por difusión basada en pilas se define de la siguiente manera (para un arreglo bidimensional):

- 1. Si el color de un *node* es distinto del que se pretende sustituir, se termina el algoritmo.**
- 2. Asigna el color de *node* a *replacement-color*.**
- 3. Se ejecuta de nuevo el algoritmo, usando el nodo situado a la izquierda del presente, y los mismos parámetros de color.**
 1. Se ejecuta de nuevo el algoritmo, usando el nodo situado a la derecha del presente, y los mismos parámetros de color.
 2. Se ejecuta de nuevo el algoritmo, usando el nodo situado inmediatamente superior al presente, y los mismos parámetros de color.
 3. Se ejecuta de nuevo el algoritmo, usando el nodo situado inmediatamente inferior al presente, y los mismos parámetros de color.
- 4. Fin del Algoritmo**

Entre los principales puntos que debemos ubicar antes de hacer un FLOODFILL, están:

✓ **ESTADO INICIAL**

Origen de la búsqueda

✓ **ESTADO FINAL**

Final de la búsqueda

✓ **ESPACIO VALIDO**

Espacio por el cual puedo navegar

✓ **ESPACIO INVALIDO**

Espacio por el cual no puedo navegar

✓ **TRANSICIONES**

Movimientos

PROBLEMA DE EJEMPLO:

Tenemos una matriz de $n*m$ dígitos, dichos dígitos son positivos, representan la calificación de un grupo de personas en una prueba. Entonces, por órdenes superiores, **debes cambiar la calificación de una persona ubicada en la posición (x, y) , en vez de su calificación original, debe tener una calificación k** , sin embargo, para poder tapar dicho cambio y que no sea notable, **cambiarás todas las calificaciones de los chicos en los 4 lados adyacentes (N,S, E, O) del chico, que hayan tenido la misma calificación que el chico (x, y)** . Si los chicos ***adyacentes tienen más chicos adyacentes*** con la misma nota, también deberás cambiarla. ¿Por qué? Sencillo: MATEMÁTICAS.

SOLUCIÓN:

El problema nos brinda un estado inicial para lanzar el floodfill, una coordenada, llamémosla $A(x, y)$, también nos enmarca que debemos cambiar de calificación a todos los vecinos de A , por lo tanto, llamemos A' al vecino del norte de A ; sin embargo, si A' tiene más vecinos que tengan la calificación igual a la que A alguna vez tuvo, deben ser marcados.

Por lo tanto, la solución nos enmarca realizar un floodfill desde (x, y) e ir cambiando la calificación siempre que sea posible. Imprime la matriz con el cambio hecho.

SALIDA

ACEPTABLE:

8 8 ← n, m (límites)
4 4 3 ← x, y, k

1	1	1	2	2	2	2	2
1	1	1	1	1	1	0	2
1	0	0	1	1	0	1	2
1	2	2	2	2	0	1	0
1	1	1	2	2	0	1	0
1	1	1	2	2	2	2	0
1	1	1	1	1	2	1	1
1	1	1	1	1	2	2	1

Plano original

Matriz con los
cambios ya
aplicados.

1	1	1	2	2	2	2	2
1	1	1	1	1	1	0	2
1	0	0	1	1	0	1	2
1	3	3	3	3	0	1	0
1	1	1	3	3	0	1	0
1	1	1	3	3	3	3	0
1	1	1	1	1	3	1	1
1	1	1	1	1	3	3	1

NOTE QUE LOS 2'S
DE LA PRIMER FILA
NO FUERON
CAMBIADOS, PUES
NO SON
ADYACENTES AL
PUNTO DE CAMBIO
ORIGINAL

PUNTOS BÁSICOS DE ANÁLISIS:

✓ ESTADO INICIAL

Coordenada (x, y)

✓ ESTADO FINAL

Hasta que ya no pueda hacer cambios.

✓ ESPACIO VALIDO

Casillas que tengan una calificación igual a la que tuvo (x, y) .

✓ ESPACIO INVALIDO

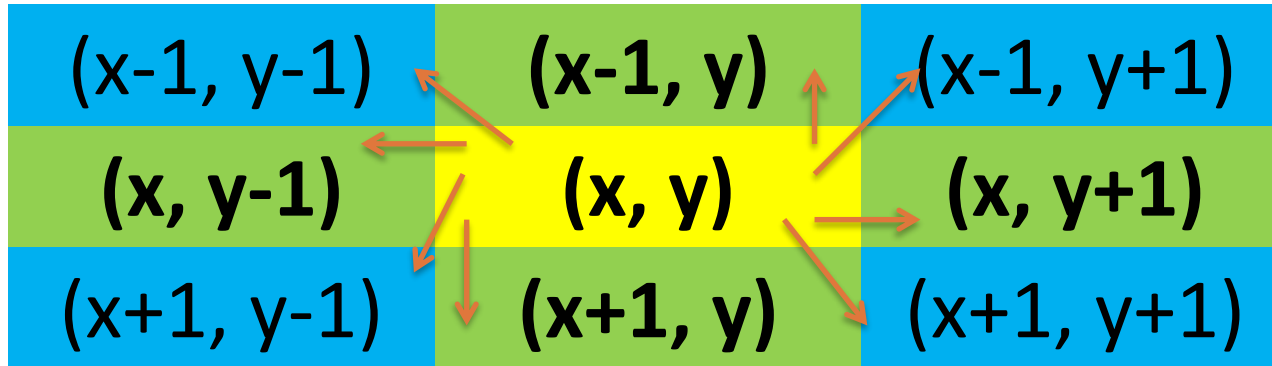
Casillas fuera de los límites de la matriz, casillas con calificación diferente a lo que tuvo (x, y) y casillas ya visitadas.

✓ TRANSICIONES

Dado que nos podemos mover en las casillas adyacentes de los cuatro puntos cardinales, podemos movernos hacia arriba, abajo, izquierda y derecha.

NOTA:

TRANSICIONES POSIBLES DESDE UN PUNTO (x, y) PARA DESARROLLAR EL FLOODFILL EN LA MATRIZ



TOMANDO EN CUENTA LA CASILLA AMARILLA COMO EL ORIGEN:

Si le piden avanzar a las **4 casillas adyacentes**: NORTE, SUR, ESTE, OESTE

TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES**

Si le piden avanzar a las **8 casillas adyacentes**

TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES Y AZULES**

**A CONTINUACIÓN SE PRESENTARÁ UN
CÓDIGO CON LA SOLUCIÓN DEL PROBLEMA,
ES RECOMENDABLE LEERLO Y ADAPTARLO A
SU ESTILO DE PROGRAMACIÓN**

Este código contiene:

- La matriz ya transformada

Definimos las variables a utilizar, n y m para los límites, x y y para la coordenada elegida y k para el número que usaremos para cambiar.

Añadimos una matriz de booleanos para marcar los lugares ya visitados, siguiendo un poco los recursos usados en la BFS y DFS.

```
5  int n, m, x, y, k, mat[101][101];  
6  bool marcas[101][101]={false}; ///matriz de bool's para marcados
```


Nuestro programa principal contiene la **lectura de las variables y de la matriz**, luego “tiramos” el **floodfill** desde la **coordenada (x, y)** y a lo último, **imprimimos la matriz**.

A continuación se explicará la estructura del floodfill.

```
22  int main()
23  {
24      cin>>n>>m>>x>>y>>k;
25      for(int i=1; i<=n; i++)
26          for(int j=1; j<=m; j++)
27              cin>>mat[i][j];
28
29      floodfill(x, y, k, mat[x][y]);
30
31      cout<<"\n";
32      for(int i=1; i<=n; i++)
33      {
34          for(int j=1; j<=m; j++)
35              cout<<mat[i][j]<<" ";
36          cout<<"\n";
37      }
38  }
```

El Floodfill lo vamos a llamar como una **FUNCIÓN CON PARÁMETROS**, una de las soluciones es tomar en cuenta los siguientes parámetros:

```
floodfill(x, y, k, mat[x][y]);
```

1. Coordenada x

2. Coordenada y

**3. Número cual usaremos para cambiar la
nota**

4. Número que existe en la coordenada x, y
(para que sepamos cual era nuestro número original, y si los vecinos coinciden con este, los cambiamos)

El floodfill es un algoritmo muy sencillo de ejecutar, y aunque gasta mucha memoria y es algo tonto con respecto al tiempo, es eficaz en conjuntos de mediano tamaño. Se divide en:

RESTRICCIONES

MODIFICACIONES

SIGUIENTES LLAMADAS

(relacione los puntos con los colores de la imagen)

```
8 void floodfill(int x, int y, int color_deseado, int color_base)
9 {
10     if(x<1 || y<1 || x>n || y>m || marcas[x][y] || mat[x][y]!=color_base)
11         return; ///en caso de estar en una posición invalida, retornamos
12
13     mat[x][y]=color_deseado; ///sobreescribimos el nuevo color
14     marcas[x][y]=true; ///marcamos como visitado
15
16     floodfill(x-1, y, color_deseado, color_base); ///arriba
17     floodfill(x, y+1, color_deseado, color_base); ///derecha
18     floodfill(x+1, y, color_deseado, color_base); ///abajo
19     floodfill(x, y-1, color_deseado, color_base); ///izquierda
20 }
```

RESTRICCIONES

En esta parte, debe anotar todos los **estados inválidos posibles**, pues el **if** será **determinante para no procesar dichos estados**, en caso de que la prueba del **if** regrese un **true**, **debemos cancelar ese proceso**. ¿Cómo se cancela? Fácil: Como nuestra **función floodfill** es una **variable de tipo void**, pues **sólo modificará nuestra matriz**, debemos apegarnos a la regla para **retornar funciones void**, con el uso del **“return;”**

```
10     if(x<1 || y<1 || x>n || y>m || marcas[x][y] || mat[x][y]!=color_base)
11         return; ///en caso de estar en una posición invalida, retornamos
```

MODIFICACIONES

En esta parte, debemos realizar todos los cambios que necesitamos, pues nuestro estado ha sido comprobado como válido, entonces podemos editar variables en dicha posición, por ejemplo.

Por lo regular, siempre se usa una matriz de marcas (bool's), entonces, esta es la parte donde debemos marcar como true la posición, para asumir que ya ha sido visitada.

```
13 mat[x][y]=color_deseado; ///sobreescribimos el nuevo color  
14 marcas[x][y]=true; ///marcamos como visitado
```

SIGUIENTES LLAMADAS

En esta parte, debemos realizar las siguientes llamadas recursivas, básicamente llamamos nuestra función, pero con parámetros modificados, casi siempre en las coordenadas, según tengamos que movernos.

```
16 floodfill(x-1, y, color_deseado, color_base); ///arriba  
17 floodfill(x, y+1, color_deseado, color_base); ///derecha  
18 floodfill(x+1, y, color_deseado, color_base); ///abajo  
19 floodfill(x, y-1, color_deseado, color_base); ///izquierda
```

MODIFICACIÓN DE LAS COORDENADAS



Material creado para el ENTRENAMIENTO PRE- SELECTIVO VERACRUZ DE COBAEV 2019

28/ENE – 01/FEB

Este material puede usarse para explicar los temas
ya establecidos.

El material fue creado para distribución a partir del
COMITÉ DE PROGRAMACIÓN DE COBAEV

*Fabricio Cruz López
@MrKristarlx07 – GitHub
@marbasz - Twitter

BUEN VIAJE!!!