

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

***ENTRENAMIENTO
VERACRUZ 2019***

PARTE 2

TEMAS INCLUIDOS:

➤ BFS

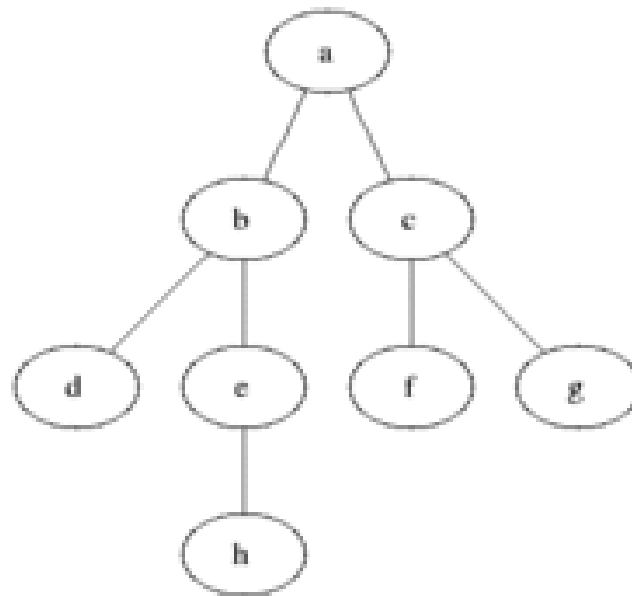
➤ DFS

➤ BÚSQUEDA BINARIA

➤ CORRIMIENTO DE BITS

BFS

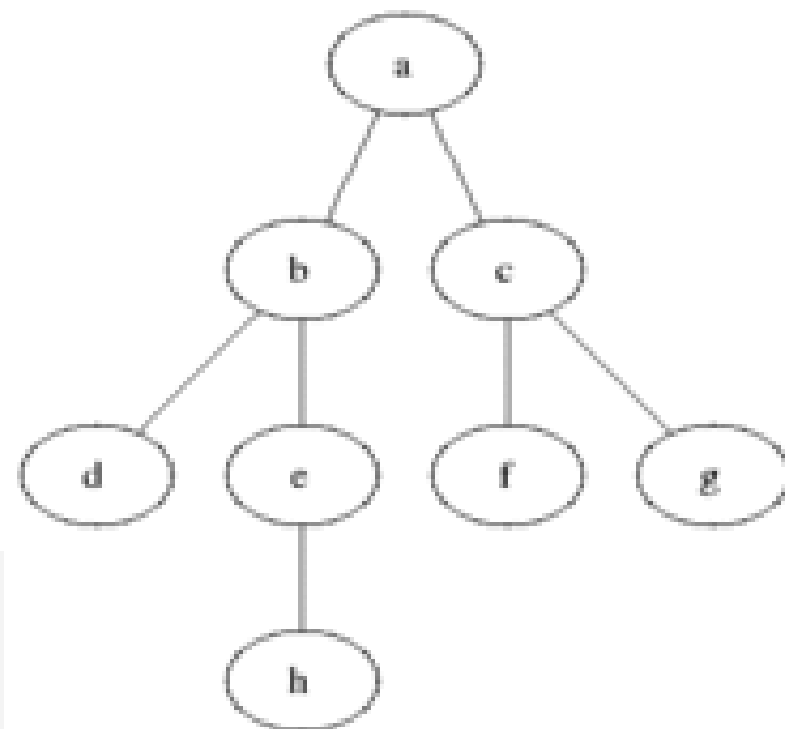
“Como llegar en el menor número de pasos de ‘A’ a ‘B’”



- **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**

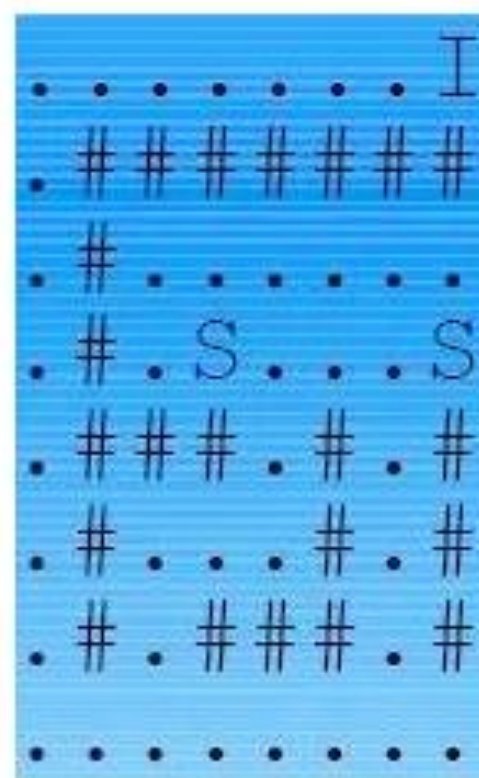
Espacio por el cual no puedo navegar

✓ **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.

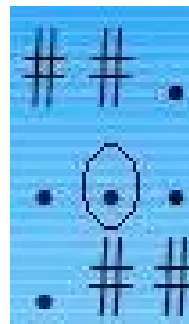
✓ ESPACIO INVALIDO

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.

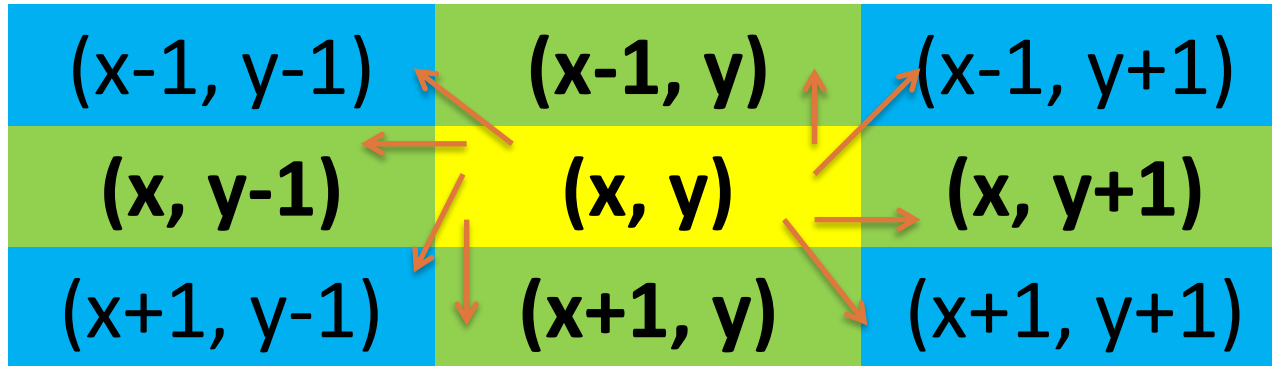
Así por ejemplo si estoy en la posición siguiente (círculo) y suponiendo que sea la coordenada 3,4 :



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.

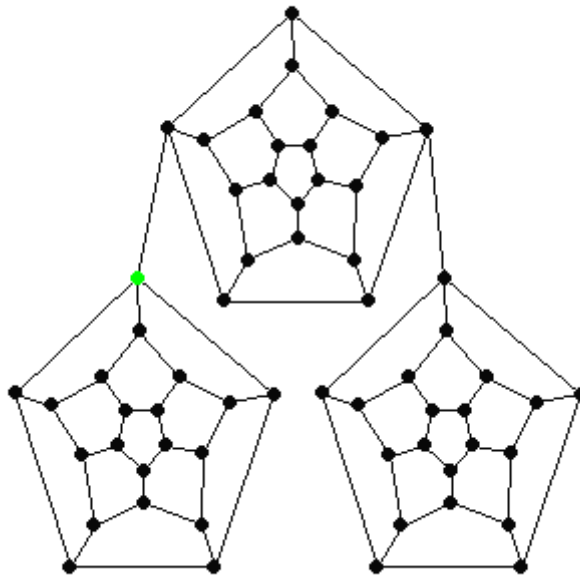
```
55 for(int i=0; i<n; i++)
56 {
57     for(int j=0; j<m; j++)
58     {
59         if((mat[i][j]=='.' || mat[i][j]=='S' || mat[i][j]=='#' || mat[i][j]=='I')
60             && (camino[i][j]==0 || camino[i][j]==-1) )
61             cout<<mat[i][j]<<" ";
62         else
63             cout<<camino[i][j]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


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 pseudocodigo:

```
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**

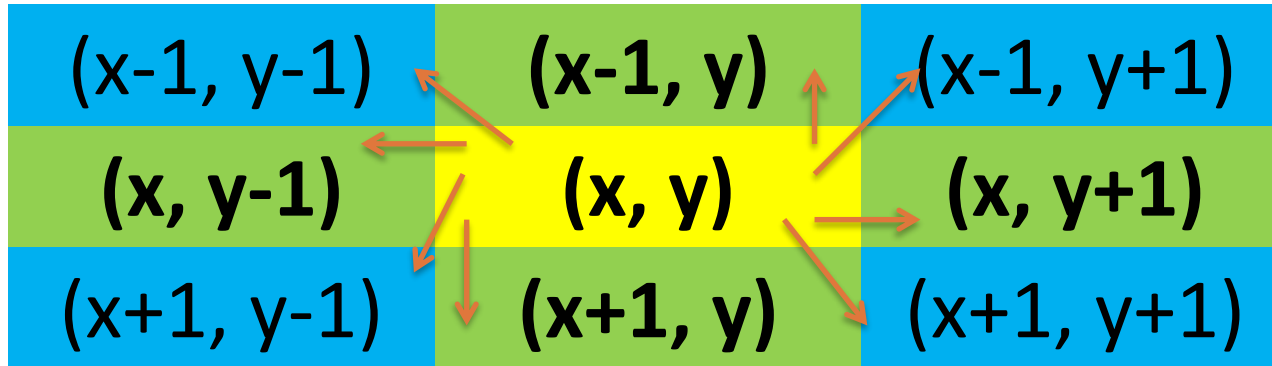
Espacio por el cual no puedo navegar

✓ **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

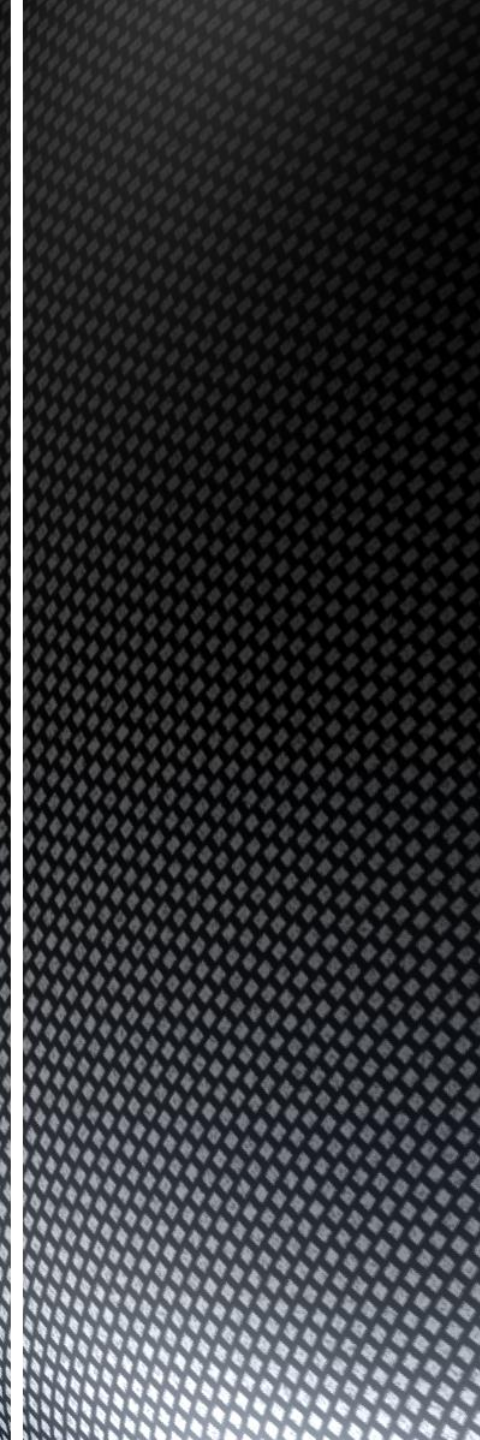
TOME COMO TRANSICIONES LAS COORDENADAS DE LAS **CASILLAS VERDES**

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

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



BÚSQUEDA BINARIA



- Es un algoritmo de búsqueda que encuentra la posición de un valor en un array ordenado. Compara el valor con el elemento en el medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.
- La búsqueda binaria es computada en el peor de los casos en un tiempo logarítmico, realizando $O(\log n)$ comparaciones, donde n es el número de elementos del arreglo y \log es el logaritmo.
- La búsqueda binaria requiere solamente $O(1)$ en espacio, es decir, que el espacio requerido por el algoritmo es el mismo para cualquier cantidad de elementos en el array. Aunque estructuras de datos especializadas en la búsqueda rápidas como las tablas hash pueden ser más eficientes, la búsqueda binaria se aplica a un amplio rango de problemas de búsqueda.

El algoritmo que rige la búsqueda binaria es sencillo:

Dado un arreglo de n elementos, comenzamos con un valor x (elemento a buscar), $\text{bool } \text{ban}=0$, $\text{mini}=1$, $\text{maxi}=n$ y $\text{med}=(1+n)/2$:

mientras $\text{mini} \neq \text{maxi}$

$\text{med} = (\text{mini} + \text{maxi}) / 2$

si $\text{vec}[\text{med}] == x$

$\text{ban} = 1$, break;

else if $(x > \text{vec}[\text{med}])$

$\text{mini} = \text{med} + 1$;

else

$\text{max} = \text{med} - 1$;

$\text{cout} << (\text{ban} ? \text{"SI ENCONTRADO"} : \text{"NO ENCONTRADO"}) ;$

Suena fácil, sin embargo, en nivel de competencia, usarás búsqueda binaria para resolver diversos problemas de una forma no implícita, ¿Cómo? Pues aprovechando una de las propiedades de la búsqueda binaria, segmentando el espacio de búsqueda.

Por ejemplo, si tras un análisis de un problema, nos percatamos que de alguna forma, nos conviene obtener el valor mínimo más cercano a k , para que a partir de ahí, se pueda interactuar con los valores, la búsqueda binaria nos asegura un panorama así:

1	k						n		
K-3	K-2	K-1	K+4	K+5	K+6	K+7	K+8	K+9	K+10

Gracias a esto, aseguramos que desde la posición donde nuestro medio estuvo por última vez, hasta el final, son valores aceptables.

Un problema de ejemplo de cómo se usa la técnica de búsqueda binaria, pero con un punto de vista distinto, es FULANITO:

Descripción

En una gasolinera hay una fila de N autos esperando ser atendidos. Cada persona i desea llenar su auto con exactamente x_i litros de gasolina.

Fulanito, que trabaja en la gasolinera, sabe que se ha terminado la gasolina, pero está en un momento de divagación y no quiere avisarle a las personas de esto hasta que sea respondida la siguiente pregunta:

Si hubiera G litros de gasolina para vender, ¿cuántos autos podrían ser llenados con la cantidad de gasolina deseada?, ¿cuánta gasolina sobraría después de atender a dichos autos?

Fulanito se hace Q veces la misma pregunta para distintos valores de G , ayúdalo a obtener las respuestas antes de que la gente descubra su engaño y lo ataquen.

Entrada


La primer línea de entrada contiene los números Q y N , indicando el número de preguntas y el número de autos en la fila, respectivamente.

La siguiente línea contiene N números enteros x_i , que representan la cantidad en litros de gasolina que desea obtener la persona i . Las siguientes Q líneas contienen un número entero G , indicando los litros de gasolina con que debes responder cada pregunta.

Salida

Dos números enteros separados por un espacio, indicando la cantidad de autos que pueden ser llenados con la cantidad deseada y los litros sobrantes de gasolina después de haber atendido a dichos autos.

Ejemplo

Entrada	Salida	Descripción
2 5 1 200 30 345 12 250 576 	3 19 4 0	Cuando $G = 250$ pueden llenarse por completo los primeros 3 autos, quedando $250-231=19$ litros sin vender.

Límites

- $1 \leq N, Q \leq 10^5$
 - $1 \leq x_i \leq 10^6$
 - $0 \leq G \leq 10^9$
 - Para el 50% de los casos, $Q \leq 1000$
-

Básicamente, el problema nos pide el número máximo de carros a los cuales le podremos colocar combustible a tanque lleno, una respuesta sencilla es navegar en tiempo $O(n)$, sin embargo, como nos dan queries, este método será arcaico, y por ende, tardío.

Es por ello que, procedemos con la búsqueda binaria, con esta, haremos a los más 17 movimientos, puesto que es $\log n$ base 2.

Hay que resaltar que no se buscará un elemento x , lo que se buscará es el elemento más alto que es aceptable ante un dato x , esto quiere decir, el elemento del arreglo más alto que sea menor o igual a x .

La idea central es hacer un arreglo de sumatorias y navegar de forma binaria a través de este, a continuación se muestra el código:


```

1 #include <bits/stdc++.h>
2 #define maximo 100002
3 #define optimiza ios_base::sync_with_stdio(0); cin.tie(0)
4 using namespace std;
5 long long int n, q, autos, x, sumatoria[maximo], l;
6 int ff()
7 {
8     int bajo=1, alto=n, centro;
9     centro=(bajo+alto+1)/2;
10    while(bajo<=alto)
11    {
12        if(sumatoria[centro]<=x)
13            bajo=centro+1;
14        else
15            alto=centro-1;
16        centro=(bajo+alto+1)/2;
17    }
18    return alto;
19 }
20 int main()
21 {
22     optimiza;
23     cin>>q>>n;
24     for(int i=1; i<=n; i++)
25     {
26         cin>>autos;
27         sumatoria[i]=(sumatoria[i-1]+autos);
28     }
29     for(int i=0; i<q; i++)
30     {
31         cin>>x;
32         l=ff();
33         cout<<l<<" "<<x-sumatoria[l]<<"\n";
34     }
35 }

```



CORRIMIENTO DE BITS

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

Vamos a manejar 2 operadores de bits, '<<' y '>>', los cuales desplazan ya sea a la derecha o a la izquierda (respectivamente) sus bits un número determinado de posiciones.

Por ejemplo:

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int x=10;
6      /// x = 00001010 = 10
7      cout<<x<<"\n";
8
9      x=x<<2;
10     /// x = 00101000 = 40
11     cout<<x<<"\n";
12
13     x=x>>2;
14     /// x = 00001010 = 10
15     cout<<x;
16 }
```

```
10
40
10
Proce
Press
```

NOTA: Al recorrer bits hacia la izquierda, multiplicas por 2, al recorrer bits hacia la derecha, divides entre 2.

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

18/MAR – 23/MAR

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!!!