Contents

Indice

Estructuras de datos

C++

Vector <vector> `vector <data_type> vector_name`

Método	Complejidad	Funcionalidad
V.begin()	O(1)	Devuelve un iterador de la primera posición
V.end()	O(1)	Devuelve un iterador de la última posición
V.size()	O(1)	Devuelve el número de elementos
V.at()	O(1)	Devuelve la posición de un elemento
V.empty()	O(1)	Devuelve si está vacío
V.assign()	O(n)	Asigna un nuevo valor
V.fill()	O(1)	Rellena el vector con un valor
V.pop_back()	O(1)	Elimina el último valor
V.push_back()	O(1)	Añade un valor en la última posición
V.insert()	O(n)	Inserta un valor en la posición especificada
V.erase	O(n)	Borra un elemento en la posición especificada
V.clear()	O(n)	Elimina todos los elementos

Linked List <list> `list <data_type> list_name`

Método	Complejidad	Funcionalidad
L.begin()	O(1)	Devuelve un iterador de la primera posición
L.end()	O(1)	Devuelve un iterador de la última posición
L.size()	O(1)	Devuelve el número de elementos
L.push_back()	O(1)	Inserta un elemento al final de la lista
L.pop_back()	O(1)	Elimina el último elemento
L.push_front()	O(1)	Inserta un elemento al principio de la lista
L.pop_front()	O(1)	Elimina el primer elemento
L.insert()	O(n)	Inserta un elemento en la posición dada
L.erase()	O(n)	Elimina el elemento dado por la posición

Método	Complejidad	Funcionalidad
L.remove()	O(n)	Elimina todas las copias de un elemento dado

Deque <deque> `deque <data_type> deque_name`

Método	Complejidad	Funcionalidad
D.begin()	O(1)	Returns iterator to the first element.
D.end()	O(1)	Returns an iterator to the theoretical element after the last element.
D.at()	O(1)	Access specified element.
D.[ ]	O(1)	Access element at the given index.
D.front()	O(1)	Returns the first element.
D.back()	O(1)	Returns the last element.
D.size()	O(1)	Returns the number of elements.
D.push_back()	O(1)	Add the elements at the end.
D.pop_back()	O(1)	Removes the elements from the end.
D.push_front()	O(1)	Add the elements at the front.
D.pop_front()	O(1)	Removes the element from the front.

Stack <stack> `stack <data_type> stack_name`

Método	Complejidad	Funcionalidad
empty()	O(1)	Devuelve true si la pila está vacía, false en caso contrario.
size()	O(1)	Devuelve el número de elementos en la pila.
top()	O(1)	Devuelve el elemento superior.
push(g)	O(1)	Añade un elemento en la pila.
pop()	O(1)	Elimina un elemento de la pila.

Queue <queue> `queue <data_type> queue_name`

Método	Complejidad	Funcionalidad
empty()	O(1)	Devuelve true si la cola está vacía, false en caso contrario.
size()	O(1)	Devuelve el número de elementos en la cola.
front()	O(1)	Devuelve el elemento del frente de la cola.
back()	O(1)	Devuelve el elemento del final de la cola.
push()	O(1)	Añade un elemento a la cola.
pop()	O(1)	Elimina un elemento de la cola.

Priority Queue <queue> `priority_queue <data_type> priority_queue_name`

Método	Complejidad	Descripción
`push(elem)`	O(log n)	Inserta un elemento en la cola de prioridad.
`pop()`	O(log n)	Elimina el elemento de mayor prioridad.
`top()`	O(1)	Devuelve el elemento de mayor prioridad.
`empty()`	O(1)	Verifica si la cola de prioridad está vacía.
`size()`	O(1)	Devuelve el número de elementos en la cola.
`emplace(args)`	O(log n)	Construye e inserta un elemento en la cola.
`swap(other)`	O(1)	Intercambia el contenido con otra cola de prioridad.

Set <set> `set <data_type> set_name;`

Método	Complejidad	Funcionalidad
begin()	O(1)	Devuelve un iterador al primer elemento.
end()	O(1)	Devuelve un iterador al último elemento.
size()	O(1)	Devuelve el número de elementos.
empty()	O(1)	Verifica si el contenedor está vacío.
insert()	O(logn)	Inserta un solo elemento.
erase()	O(logn)	Elimina el elemento dado.
clear()	O(n)	Elimina todos los elementos.
find()	O(logn)	Devuelve un puntero al elemento dado si está presente, de lo contrario, un puntero al final.

Unordered_set `<unordered_set>` `unordered_set <data_type> set_name`

Función	Complejidad	Descripción
begin()	O(1)	Devuelve un iterador al primer elemento.
end()	O(1)	Devuelve un iterador al elemento teórico que sigue al último.
size()	O(1)	Devuelve el número de elementos.
empty()	O(1)	Devuelve true si el conjunto no ordenado está vacío, de lo contrario, false.
insert()	O(1)	Inserta un elemento en el contenedor.
erase()	O(1)	Elimina un elemento del contenedor.
find()	O(1)	Devuelve un puntero al elemento dado si está presente, de lo contrario, un puntero al final.

Map `<map>` `map <key, value_type> map_name`

Se implementa con un árbol de búsqueda balanceado, por lo que los elementos están ordenados

Función	Complejidad	Descripción
begin()	O(1)	Devuelve un iterador al primer elemento.
end()	O(1)	Devuelve un iterador al elemento teórico que sigue al último.
size()	O(1)	Devuelve el número de elementos en el mapa.
insert()	O(logn)	Agrega un nuevo elemento al mapa.
erase(iterator)	O(logn)	Elimina el elemento en la posición señalada por el iterador.
erase(key)	O(logn)	Elimina la clave y su valor del mapa.
clear()	O(n)	Elimina todos los elementos del mapa.

Unordered_map `<unordered_map>` `unordered_map <key_type, value_type> map_name`

Se implementa con una tabla hash, por lo que los elementos no tienen porqué estar ordenados

Función	Complejidad	Descripción
begin()	O(1)	Devuelve un iterador al primer elemento.
end()	O(1)	Devuelve un iterador al elemento teórico que sigue al último.
size()	O(1)	Devuelve el número de elementos.
empty()	O(1)	Devuelve true si el conjunto no ordenado está vacío, de lo contrario, false.
find()	O(1)	Devuelve un puntero al elemento dado si está presente, de lo contrario, un puntero al final.
bucket()	O(1)	Devuelve el número de cubo donde se almacena el dato.
insert()	O(1)	Inserta un elemento en el contenedor.
erase()	O(1)	Elimina un elemento del contenedor.

Python

List `list = []`

Método	Complejidad	Descripción
`list.append()`	O(1)	Añade un elemento al final de la lista.

Método	Complejidad	Descripción
`list.extend()`	O(k)	Añade los elementos de un iterable al final de la lista.
`list.insert()`	O(n)	Inserta un elemento en una posición específica de la lista.
`list.remove()`	O(n)	Elimina la primera aparición del elemento especificado.
`list.pop()`	O(1) o O(n)	Elimina y devuelve el último elemento o un elemento específico.
`list.clear()`	O(1)	Elimina todos los elementos de la lista.
`list.index()`	O(n)	Devuelve el índice de la primera aparición del elemento especificado.
`list.count()`	O(n)	Cuenta el número de apariciones de un elemento.
`list.sort()`	O(n log n)	Ordena los elementos de la lista.
`list.reverse()`	O(n)	Invierte el orden de los elementos en la lista.

Método	Complejidad	Descripción
`list.index()`	O(n)	Devuelve el índice de la primera aparición del elemento especificado.
`list.count()`	O(n)	Cuenta el número de apariciones de un elemento.

Método	Complejidad	Descripción
`list.remove()`	O(n)	Elimina la primera aparición del elemento especificado.
`list.pop()`	O(1) o O(n)	Elimina y devuelve el último elemento o un elemento específico.

Método	Complejidad	Descripción
`list.sort()`	O(n log n)	Ordena los elementos de la lista.

Dictionary `dict = {}`

`dict = {key:value}`

Método	Complejidad	Descripción
`dict[key] = value`	O(1)	Escribe en una clave o la crea sino existe
`dict.copy()`	O(n)	Devuelve una copia superficial del diccionario.
`dict.clear()`	O(1)	Elimina todos los elementos del diccionario.
`dict.fromkeys(seq)`	O(n)	Crea un nuevo diccionario con claves de `seq` y valores `None`.
`dict.get(key)`	O(1)	Devuelve el valor asociado a la clave, o `None` si no existe.
`dict.items()`	O(n)	Devuelve una vista de pares clave-valor en el diccionario.
`dict.keys()`	O(n)	Devuelve una vista de las claves en el diccionario.
`dict.values()`	O(n)	Devuelve una vista de los valores en el diccionario.
`dict.pop(key)`	O(1)	Elimina la clave y devuelve el valor asociado.
`dict.popitem()`	O(1)	Elimina y devuelve un par clave-valor arbitrario.
`dict.setdefault()`	O(1)	Devuelve el valor asociado a la clave, o la establece si no existe.
`dict.update(iterable)`	O(len(iterable))	Actualiza el diccionario con pares clave-valor de `iterable`.
`dict.fromkeys(keys, value)`	O(n)	Crea un nuevo diccionario con claves de `keys` y valores `value`.
`dict.__contains__()`	O(1)	Verifica si la clave está presente en el diccionario.

Set `set = set()`

`set = {value}`

Método	Complejidad	Descripción
`set.add(elem)`	O(1)	Añade un elemento al conjunto.
`set.clear()`	O(1)	Elimina todos los elementos del conjunto.
`set.copy()`	O(n)	Devuelve una copia superficial del conjunto.

Método	Complejidad	Descripción
`set.discard(elem)`	O(1)	Elimina un elemento del conjunto si está presente.
`set.pop()`	O(1)	Elimina y devuelve un elemento arbitrario del conjunto.
`set.remove(elem)`	O(1)	Elimina un elemento del conjunto; genera un error si no existe.
`set.update(iterable)`	O(len(iterable))	Agrega elementos del iterable al conjunto.
`set.__contains__()`	O(1)	Verifica si el elemento está presente en el conjunto.
`set.__len__()`	O(1)	Devuelve el número de elementos en el conjunto.

Método	Complejidad	Descripción
`set.union()`	O(len(set))	Devuelve la unión de dos conjuntos.
`set.intersection()`	O(min(len(set), len(other_set)))	Devuelve la intersección de dos conjuntos.
`set.difference()`	O(len(set))	Devuelve la diferencia entre dos conjuntos.
`set.symmetric_difference()`	O(len(set))	Devuelve la diferencia simétrica entre dos conjuntos.
`set.issubset()`	O(len(set))	Verifica si el conjunto es un subconjunto de otro.
`set.issuperset()`	O(len(set))	Verifica si el conjunto es un superconjunto de otro.

Strings

C++ `<string>`

```
string string_name
```

Método	Complejidad	Descripción
`length()`	O(1)	Devuelve la longitud de la cadena.
`size()`	O(1)	Devuelve la longitud de la cadena.
`empty()`	O(1)	Verifica si la cadena está vacía.
`clear()`	O(1)	Elimina todos los caracteres de la cadena.
`at(index)`	O(1)	Accede al carácter en la posición especificada.
`operator[]`	O(1)	Accede al carácter en la posición especificada.
`front()`	O(1)	Devuelve una referencia al primer carácter.
`back()`	O(1)	Devuelve una referencia al último carácter.
`c_str()`	O(1)	Devuelve un puntero a una cadena de estilo C.
`substr(pos, len)`	O(len)	Devuelve una subcadena desde la posición especificada con la longitud dada.
`find(str, pos)`	O(n)	Busca la primera aparición de la cadena especificada a partir de la posición dada.
`find_first_of(chars, pos)`	O(n)	Busca la primera aparición de cualquiera de los caracteres dados a partir de la posición dada.
`find_last_of(chars, pos)`	O(n)	Busca la última aparición de cualquiera de los caracteres dados hasta la posición dada.
`replace(pos, len, str)`	O(n)	Reemplaza una parte de la cadena con otra cadena.

Python

```
string_name = ""
```

Método	Descripción
`len(str)`	Devuelve la longitud de la cadena.
`str.lower()`	Devuelve una copia de la cadena en minúsculas.
`str.upper()`	Devuelve una copia de la cadena en mayúsculas.
`str.capitalize()`	Devuelve una copia de la cadena con la primera letra en mayúscula.
`str.title()`	Devuelve una copia de la cadena con cada palabra en mayúscula.
`str.strip()`	Elimina los espacios en blanco al principio y al final de la cadena.
`str.lstrip()`	Elimina los espacios en blanco al principio de la cadena.
`str.rstrip()`	Elimina los espacios en blanco al final de la cadena.
`str.startswith(prefix)`	Verifica si la cadena comienza con el prefijo dado (p es la longitud del prefijo).

Método	Descripción
`str.endswith(suffix)`	Verifica si la cadena termina con el sufijo dado (s es la longitud del sufijo).
`str.find(substring)`	Busca la primera aparición de la subcadena y devuelve la posición, o -1 si no se encuentra.
`str.replace(old, new)`	Reemplaza todas las ocurrencias de la subcadena antigua con la nueva.
`str.split(sep)`	Divide la cadena en una lista de subcadenas usando el separador dado.
`str.isalpha()`	Devuelve `True` si todos los caracteres de la cadena son letras.
`str.isdigit()`	Devuelve `True` si todos los caracteres de la cadena son dígitos.
`str.isalnum()`	Devuelve `True` si todos los caracteres de la cadena son alfanuméricos.
`str.islower()`	Devuelve `True` si todos los caracteres de la cadena están en minúsculas.
`str.isupper()`	Devuelve `True` si todos los caracteres de la cadena están en mayúsculas.
`str.capitalize()`	Devuelve una copia de la cadena con la primera letra en mayúscula.
`str.count(substring)`	Cuenta el número de ocurrencias de la subcadena en la cadena .

Tabla ASCII

Decimal	Carácter	Decimal	Carácter	Decimal	Carácter	Decimal	Carácter
0	NUL	32	espacio	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

KMP

El algoritmo de Knuth-Morris-Pratt (KMP) es un algoritmo de búsqueda de patrones que se utiliza para encontrar todas las ocurrencias de un patrón en un texto. Se aplica cuando deseas encontrar todas las ocurrencias de un patrón, no solo la primera.

Complejidad

- Tiempo:
  - El tiempo de ejecución del algoritmo es lineal en el tamaño del texto y el patrón O(n+m)
- Espacio:
  - La complejiadd espacial del algoritmo KMP es O(m)

Implementación

```
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    # create lps[] that will hold the longest prefix suffix values for pattern
    lps = [0]*M
    j = 0 # index for pat[]

    # Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps)

    i = 0 # index for txt[]
    while (N - i) >= (M - j):
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print("Found pattern at index " + str(i-j))
            j = lps[j-1]

        # mismatch after j matches
        elif i < N and pat[j] != txt[i]:
            # Do not match lps[0..lps[j-1]] characters, they will match anyway
            if j != 0:
                j = lps[j-1]
            else:
                i += 1

# Function to compute LPS array
def computeLPSArray(pat, M, lps):
    len = 0 # length of the previous longest prefix suffix
    lps[0] = 0 # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i] == pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            # This is tricky. Consider the example. AACAAAA and i = 7. The idea is similar to search step.
            if len != 0:
                len = lps[len-1]

                # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1

# Driver code
if __name__ == '__main__':
```

```
txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)
```

Sorting

C++

`#include <algorithm>` En C++ hay dos métodos de ordenamiento:

- `sort()`: El cual ordena los elementos de un rango, dado dos iteradores para indicar el rango de efecto. Se puede indicar el predicado booleano de ordenamiento
- `stable_sort()`: El cual ordena los elementos en un rango de manera estable, preservando el orden relativo de elementos con claves iguales

Python

- `.sorted()`: Ordena los elementos y devuelve una nueva lista ordenada, se le puede especificar la función de ordenamiento
- `.sort()`: Ordena la lista in-place

Teoría de Grafos

- $\sum_{i=0}^{|V|} g(V_i) = 2E$
- **Colorario**: Un grafo contiene un número par de vertices con grado impar

Conexo

Un grafo es conexo si hay un camino entre todos los pares de vértices. La existencia de un *spanning tree* es suficiente para verificar que es conexo.

- Si un grafo conexo tiene un vértice que si es eliminado se convierte en un grafo inconexo se llama *vertice articulado*. Si este vértice no existe es un grafo *grafo biconectado*.
- Si un grafo conexo tiene una arista que si es eliminado se convierte en un grafo inconexo se llama *puente*.

Encontrar vertices articulados o puentes es fácil por **fuerza bruta**. Por cada vértice/arista, se borra y se comprueba si sigue siendo conexo.

En grafos dirigidos son conocidos como *componentes fuertemente conexos* si para cada par de vértices u y v existe un camino de u hacia v y un camino de v hacia u. Se puede encontrar un ciclo en un grafo conexo de manera sencilla a través de una búsqueda en profundidad.

Ciclos en Grafos

Son particularmente interesantes los ciclos que visitan todos las aristas o vertices del grafo.

- **Ciclo Euleriano**: Es aquel que visita cada arista del grafo exactamente una vez. Los ciclos Eulerianos son *circuitos* ya que pueden visitar varias veces el mismo vértice
  - Un grafo no dirigido contiene un ciclo Euleriano si es conexo y cada vertice tiene grado par.
    - Si existe el ciclo Euleriano es fácil encontrarlo, tan solo hay que aplicar **profundidad**.
  - En un grafo dirigido la condición es que todos los vértices tengan el mismo grado de entrada que de salida
- **Camino Euleriano**: Es aquel que visita cada arista del grafo exactamente una vez, pero puede no terminar donde ha comenzado
  - Para que exista un camino Eurleriano es necesario que tan solo dos vértices violen la restriccion de la paridad del grado, los cuales serán el vértice inicial y el vértice final
- **Ciclo Hamiltoniano**: Es aquel que vista cada vértice exactamente una vez.
  - El problema del ciclo Euleriano puede ser reducido al problema del ciclo Hamiltoniano, de manera que cada arista del problema original se convierta en un vértice del problema reducido.
  - No existen algoritmos eficientes para resolver el problema del ciclo Hamiltoniano.
    - Si el grafo es suficientemente pequeño puede resolverse por backtracking
      - Cada ciclo Hamiltoniano peude ser descrito como una permutación de vértices, por lo tanto realizamos backtraking cuando no exista una arista desde el último vértice visitado a un vértice no visitado

Grafos Planares

Los grafos planares son aquellos que pueden ser dibujados en un plano tal que dos aristas no se crucen entre ellas.

> Todos los árboles son planares

Los grafos planares tienen propiedades muy importantes:

- Existe relación entre el número de vértices n, de arístas m, y de caras f de cualquier grafo planar, la **Fórmula de Euler**.

$$n - m + f = 2$$

  - Existen algoritmos eficientes para probar la planidad de un grafo pero son complicados de implementar. La fórmula de Euler es una manera fácil de probar si un grafo es no planar. Todos los grafos planares continenen como mucho $3n$- 6 aristas para n> 2. Esto significa que todos los grafos deben tener un vértice con grado de al menos 5, y borrarlo significa cada grafo resultante será planar y por lo tanto tendrá esta misma propiedad

Spanning Trees

Un *spanning tree* de un grafo G(V,E) es un subconjunto de aristas E que forman un arbol conectando todos los vértices. Para arístas ponderadas, es interesante pararse en el *spanning tree mínimo*, el cual es el spanning tree cuya suma de pesos de las aristas es la mínima posible.

El algoritmo que aportamos para encontrar el spanning tree mínimo es el **Algoritmo de Prim** que se encuentra más adelante en la sección de **Algoritmos/Algoritmo de Grafos**

Algoritmos

Algoritmos de Grafos

Algoritmo de Prim Es un algoritmo de teoría de grafos utilizado para encontrar el spanning tree mínimo en un grafo no dirigido y conexo con pesos en sus aristas.

Complejidad

- Tiempo:
  - La complejidad temporal del algoritmo es O(V^2)
- Espacio:
  - La complejidad espacial es O(V+E) para almacenar el grafo

C+

```
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from the set of vertices not yet included in
int minKey(int key[], bool mstSet[]){
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V]){
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t"
```

```
            << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for a graph represented using adjacency matrix representation
void primMST(int graph[V][V]){
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;
        // Update key value and parent index of the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // Print the constructed MST
    printMST(parent, graph);
}

// Driver's code
int main(){
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

Python

```
# Library for INT_MAX
import sys

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

    # A utility function to find the vertex with minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph represented using adjacency matrix representation
    def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
        parent = [None] * self.V # Array to store constructed MST
        # Make key 0 so that this vertex is picked as first vertex
        key[0] = 0
        mstSet = [False] * self.V

        parent[0] = -1 # First node is always the root of

        for cout in range(self.V):

            # Pick the minimum distance vertex from the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minKey(key, mstSet)

            # Put the minimum distance vertex in the shortest path tree
            mstSet[u] = True

            # Update dist value of the adjacent vertices of the picked vertex only if the current
            # distance is greater than new distance and the vertex in not in the shortest path tree
            for v in range(self.V):

                # graph[u][v] is non zero only for adjacent vertices of m
                # mstSet[v] is false for vertices not yet included in MST
                # Update the key only if graph[u][v] is smaller than key[v]
                if self.graph[u][v] > 0 and mstSet[v] == False \
```

```
                and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)

# Driver's code
if __name__ == '__main__':
    g = Graph(5)
    g.graph = [[0, 2, 0, 6, 0],
            [2, 0, 3, 8, 5],
            [0, 3, 0, 0, 7],
            [6, 8, 0, 0, 9],
            [0, 5, 7, 9, 0]]

    g.primMST()
```

Búsqueda en Amplitud Algoritmo utilizado para recorrer o buscar en un grafo de manera nivelada. Es útil cuando se quiere recorrer un grafo nivel a nivel y para encontrar caminos más cortos en grafos no ponderados.

Complejidad

- Tiempo:
  - La complejidad temporal es $O(x^n)$
- Espacio:
  - La complejidad espacial es $O(x^n)$ para almecenar la información de visitados y la cola de nodos por visitar

C++

```
#include <bits/stdc++.h>
using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph {

    // No. of vertices
    int V;
    // Pointer to an array containing adjacency lists
    vector<list<int> > adj;

public:
    // Constructor
    Graph(int V);
    // Function to add an edge to graph
    void addEdge(int v, int w);
    // Prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj.resize(V);
}

void Graph::addEdge(int v, int w)
{
    // Add w to v's list.
```

```
    adj[v].push_back(w);
}

void Graph::BFS(int s){
    // Mark all the vertices as not visited
    vector<bool> visited;
    visited.resize(V, false);

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    while (!queue.empty()) {

        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s.
        // If an adjacent has not been visited, then mark it visited and enqueue it
        for (auto adjacent : adj[s]) {
            if (!visited[adjacent]) {
                visited[adjacent] = true;
                queue.push_back(adjacent);
            }
        }
    }
}

// Driver code
int main(){
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "<< "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Python

```
from collections import defaultdict
# This class represents a directed graph using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # Default dictionary to store graph
```

```
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print(s, end=" ")

            # Get all adjacent vertices of the dequeued vertex s.
            # If an adjacent has not been visited, then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


# Driver code
if __name__ == '__main__':

    # Create a graph given in the above diagram
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is Breadth First Traversal"
        " (starting from vertex 2)")
    g.BFS(2)
```

Búsqueda en Profundidad Es un algoritmo utilizado para recorrer o buscar en un grafo. Es útil para encontrar ciclos

Complejidad

- Temporal
  - La complejidad temporal del algoritmo es O(x^N)
- Espacial
  - La complejidad espacial es O(N)

C++

```cpp
#include <bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph using adjacency list representation
class Graph {
public:
    map<int, bool> visited;
    map<int, list<int> > adj;

    // Function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices reachable from v
    void DFS(int v);
};

void Graph::addEdge(int v, int w){
    // Add w to v's list.
    adj[v].push_back(w);
}

void Graph::DFS(int v){
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}

// Driver code
int main(){
    // Create a graph given in the above diagram
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";

    // Function call
    g.DFS(2);

    return 0;
}
```

Python

```python
def iterative_dfs(graph, start):
    stack = [start]
    visited = set()

    while stack:
```

```
        current = stack.pop()
        if current not in visited:
            visited.add(current)
            for neighbor in graph.get(current, []):
                stack.append(neighbor)

    return visited

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

visited = iterative_dfs(graph, 'A')
print(visited)  # Output: {'A', 'C', 'F', 'E', 'B', 'D'}
```

Dijkstra Es un algoritmo de búsqueda de caminos mínimos utilizados para encontrar el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo dirigido.

Complejidad

- Temporal:
  - La complejidad temporal es de $O(V^2)$
- Espacial:
  - La complejidad espacial es de O(V+E)

C++

Python

```
# Library for INT_MAX
import sys
class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print("Vertex \tDistance from Source")
        for node in range(self.V):
            print(node, "\t", dist[node])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initialize minimum distance for next node
        min = sys.maxsize
        # Search not nearest vertex not in the shortest path tree
        for u in range(self.V):
            if dist[u] < min and sptSet[u] == False:
                min = dist[u]
                min_index = u
```

```
        return min_index

    # Function that implements Dijkstra's single source
    # shortest path algorithm for a graph represented using adjacency matrix representation
    def dijkstra(self, src):

        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
            # Pick the minimum distance vertex from
            # the set of vertices not yet processed. X is always equal to src in first iteration
            x = self.minDistance(dist, sptSet)
            # Put the minimum distance vertex in the shortest path tree
            sptSet[x] = True
            # Update dist value of the adjacent vertices of the picked vertex only if the current
            # distance is greater than new distance and the vertex in not in the shortest path tree
            for y in range(self.V):
                if self.graph[x][y] > 0 and sptSet[y] == False and \
                        dist[y] > dist[x] + self.graph[x][y]:
                    dist[y] = dist[x] + self.graph[x][y]

        self.printSolution(dist)


# Driver's code
if __name__ == "__main__":
    g = Graph(9)
    g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
            [4, 0, 8, 0, 0, 0, 0, 11, 0],
            [0, 8, 0, 7, 0, 4, 0, 0, 2],
            [0, 0, 7, 0, 9, 14, 0, 0, 0],
            [0, 0, 0, 9, 0, 10, 0, 0, 0],
            [0, 0, 4, 14, 10, 0, 2, 0, 0],
            [0, 0, 0, 0, 0, 2, 0, 1, 6],
            [8, 11, 0, 0, 0, 0, 1, 0, 7],
            [0, 0, 2, 0, 0, 0, 6, 7, 0]
            ]

    g.dijkstra(0)
```

```
import heapq

def dijkstra(graph, start):
    queue = [(0, start, [])]
    visited = set()
    while queue:
        (cost, current, path) = heapq.heappop(queue)
        if current not in visited:
            visited.add(current)
            path = path + [current]
            for neighbor, edge_cost in graph[current].items():
                heapq.heappush(queue, (cost + edge_cost, neighbor, path))
    return visited

# Example usage:
graph = {
    'A': {'B': 1, 'C': 3},
```

```
    'B': {'A': 1, 'D': 2, 'E': 3},
    'C': {'A': 3, 'F': 5},
    'D': {'B': 2},
    'E': {'B': 3, 'F': 4},
    'F': {'C': 5, 'E': 4},
}

visited = dijkstra(graph, 'A')
print(visited)  # Output: {'A', 'B', 'C', 'D', 'E', 'F'}
```

Floyd Warshall El algoritmo de Floyd-Warshall es un algoritmo de búsqueda de caminos más cortos entre todos los pares de nodos en un grafo, pero no puede manejar ciclos negativos

Complejidad

- Temporal:
  - La complejidad temporal es $O(V^3)$
- Espacio:
  - La complejidad espacial es $O(V^2)$ para almacenar la matriz de distancias

C++

```
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value.This value will be used for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{

    int i, j, k;

    /* Add all vertices one by one to the set of intermediate vertices.
    ---> Before start of an iteration, we have shortest distances between all pairs of vertices such that the
    shortest distances consider only the vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
    ----> After the end of an iteration, vertex no. k is added to the set of
    intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][j] > (dist[i][k] + dist[k][j])
                    && (dist[k][j] != INF
                        && dist[i][k] != INF))
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
```

```
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V]){
    cout << "The following matrix shows the shortest "
            "distances between every pair of vertices \n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                cout << "INF"
                    << " ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }
}

// Driver's code
int main(){
    /* Let us create the following weighted graph
            10
    (0)------->(3)
        |    /|\
    5 |  |
        |    | 1
    \|/  |
    (1)------->(2)
            3    */
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    // Function call
    floydWarshall(graph);
    return 0;
}
```

Python

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

```python
# Example usage:
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'D': 2, 'E': 3},
    'C': {'A': 3, 'F': 5},
    'D': {'B': 2},
    'E': {'B': 3, 'F': 4},
    'F': {'C': 5, 'E': 4},
}

dist = floyd_warshall(graph)
print(dist)
```

Algortimos Generales

Búsqueda Binaria Es un algoritmo de búsqueda eficiente que se aplica en conjunto de datos ordenados

Complejidad

- Temporal:
  - La complejidad temporal es O(log n)
- Espacial
  - La complejidad espacial es O(1)

C++

```cpp
// C++ program to implement iterative Binary Search
#include <bits/stdc++.h>
using namespace std;

// An iterative binary search function.
int binarySearch(int arr[], int l, int r, int x){
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was not present
    return -1;
}

// Driver code
int main(void){
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
```

```
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Python

```python
# It returns location of x in given array arr
def binarySearch(arr, l, r, x):

    while l <= r:
        mid = l + (r - l) // 2
        # Check if x is present at mid
        if arr[mid] == x:
            return mid
        # If x is greater, ignore left half
        elif arr[mid] < x:
            l = mid + 1
        # If x is smaller, ignore right half
        else:
            r = mid - 1
    # If we reach here, then the element was not present
    return -1
```

Eliminación Gaussiana Es un algoritmo utilizado para resolver sistemas de ecuaciones lineales y encontrar la forma escalonada de una matriz.

Complejidad

- Temporal: La complejidad temporal es $O(n^3)$

Python

```python
M = 10

# Function to print the matrix
def PrintMatrix(a, n):
    for i in range(n):
        print(*a[i])

# function to reduce matrix to reduced row echelon form.
def PerformOperation(a, n):
    i = 0
    j = 0
    k = 0
    c = 0
    flag = 0
    m = 0
    pro = 0

    # Performing elementary operations
    for i in range(n):
        if (a[i][i] == 0):
            c = 1
            while ((i + c) < n and a[i + c][i] == 0):
                c += 1
            if ((i + c) == n):
                flag = 1
                break
```

```
            j = i

            for k in range(1 + n):
                temp = a[j][k]
                a[j][k] = a[j+c][k]
                a[j+c][k] = temp

        for j in range(n):
            # Excluding all i == j
            if (i != j):
                # Converting Matrix to reduced row
                # echelon form(diagonal matrix)
                p = a[j][i] / a[i][i]
                k = 0
                for k in range(n + 1):
                    a[j][k] = a[j][k] - (a[i][k]) * p

    return flag

# Function to print the desired result if unique solutions exists, otherwise
# prints no solution or infinite solutions depending upon the input given.
def PrintResult(a, n, flag):

    print("Result is : ")

    if (flag == 2):
        print("Infinite Solutions Exists<br>")
    elif (flag == 3):
        print("No Solution Exists<br>")

    # Printing the solution by dividing constants by their respective diagonal elements
    else:
        for i in range(n):
            print(a[i][n] / a[i][i], end=" ")

# To check whether infinite solutions exists or no solution exists
def CheckConsistency(a, n, flag):

    # flag == 2 for infinite solution
    # flag == 3 for No solution
    flag = 3
    for i in range(n):
        sum = 0
        for j in range(n):
            sum = sum + a[i][j]
        if (sum == a[i][j]):
            flag = 2

    return flag

# Driver code
a = [[0, 2, 1, 4], [1, 1, 2, 6], [2, 1, 1, 7]]

# Order of Matrix(n)
n = 3
flag = 0

# Performing Matrix transformation
flag = PerformOperation(a, n)
```

```python
if (flag == 1):
    flag = CheckConsistency(a, n, flag)

# Printing Final Matrix
print("Final Augmented Matrix is : ")
PrintMatrix(a, n)
print()

# Printing Solutions(if exist)
PrintResult(a, n, flag)
```

Euclides Extendido Es un algoritmo que se utiliza para encontrar el máximo común divisor de dos números enteros, además de encontrar los coeficientes enteros x e y que satisfaces la ecuación de Bézout

C++

```cpp
// C++ program to demonstrate
// Basic Euclidean Algorithm

#include <bits/stdc++.h>
using namespace std;

// Function to return gcd of a and b
int gcd(int a, int b){
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

// Driver Code
int main(){
    int a = 10, b = 15;

    // Function call
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
        << endl;
    a = 35, b = 10;
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
        << endl;
    a = 31, b = 2;
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
        << endl;
    return 0;
}
```

Python

```python
# function for extended Euclidean Algorithm
def gcdExtended(a, b):
    # Base Case
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = gcdExtended(b % a, a)

    # Update x and y using results of recursive call
    x = y1 - (b//a) * x1
    y = x1
    return gcd, x, y
```

```
# Driver code
a, b = 35, 15
g, x, y = gcdExtended(a, b)
print("gcd(", a, ",", b, ") = ", g)
```

Fórmulas Matemáticas

Suma Gaussiana

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

Coeficientes Binomiales

$$\binom{n}{k} = \binom{n}{n-k} = \binom{n-1}{k} + \binom{n-1}{k-1} = \frac{n!}{k!(n-k)!}$$

Principio de Inclusión-Exclusión

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

MCD

$$\operatorname{mcm}(a, b) = \frac{|a \cdot b|}{\operatorname{mcd}(a, b)}$$