Depth First Search (DFS) and Breadth First Search (BFS)

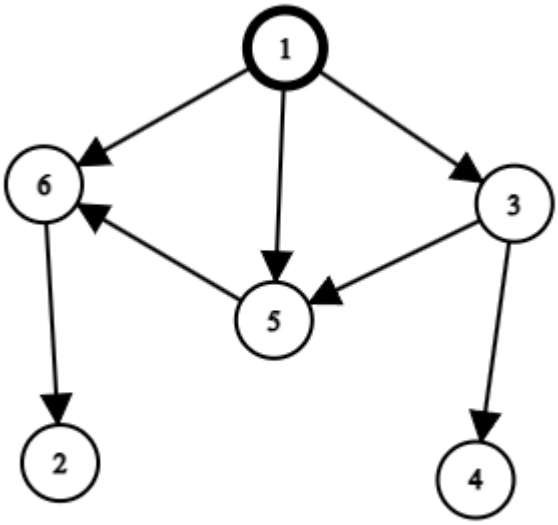
Un grafo es una estructura de datos que representa un conjunto de objetos (vértices o nodos) y las conexiones entre ellos (aristas o enlaces). Se puede utilizar para modelar varios escenarios del mundo real, como redes sociales, redes de carreteras y redes informáticas.

Dentro de lo que supone los diferentes algoritmos que se pueden aplicar sobre los grafos, tenemos dos algoritmos de búsqueda (y recorrido) populares: DFS y BFS.

Depth First Search (DFS)

La Búsqueda Primero en Profundidad (DFS en inglés) es un algoritmo que parte de un nodo inicial y luego va recorriendo recursivamente cada uno de los vecinos del nodo en profundidad. Es decir, visitando el primer vecino del nodo actual y continuando recursivamente hasta llegar al final de la rama. Luego se retrocede y se continúa el proceso con el siguiente vecino sin visitar del nodo actual (backtracking). Este proceso se repite hasta que se han visitado todos los nodos del grafo o hasta que se encuentra el nodo buscado en caso de estar realizando una búsqueda.

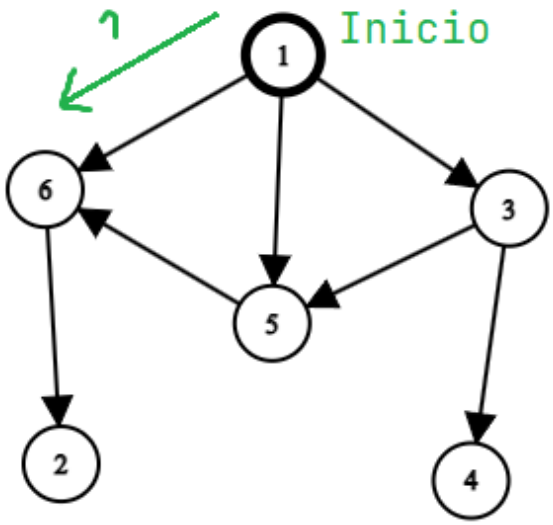
A continuación, se tiene el siguiente grafo dirigido de ejemplo:



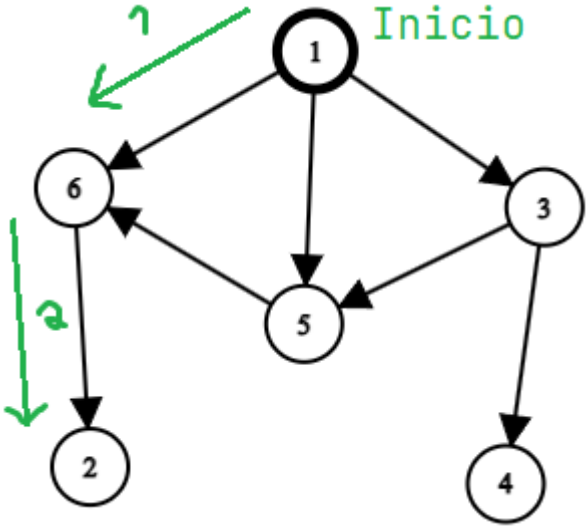
Para este caso, empezaremos del nodo 1 (aunque puede ser cualquiera) para mostrar cómo se lleva a cabo el recorrido DFS.

1. Al partir del nodo 1 notamos que tenemos 3 posibles caminos:
    a. Ir hacia el nodo 6.
    b. Ir hacia el nodo 5.
    c. Ir hacia el nodo 3.

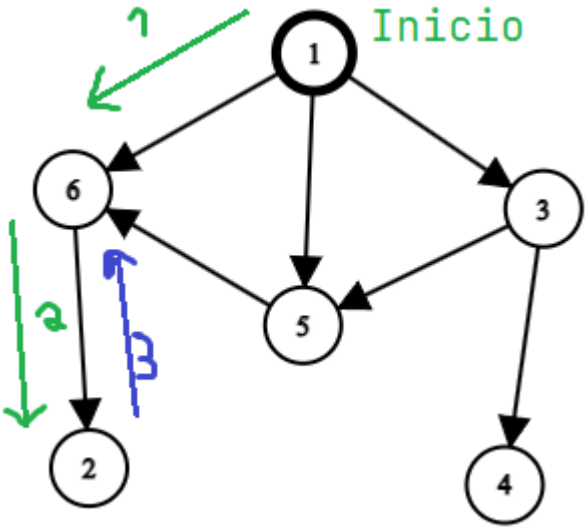
    Por lo tanto, aquí nosotros podemos elegir **arbitrariamente** cuál nodo visitar primero. Aunque se seleccione un nodo por visitar distinto, el algoritmo seguirá siendo DFS. Es decir, **puede existir más de un recorrido válido de DFS**. Para fines prácticos, iremos en orden de izquierda a derecha, así que iremos al nodo 6.



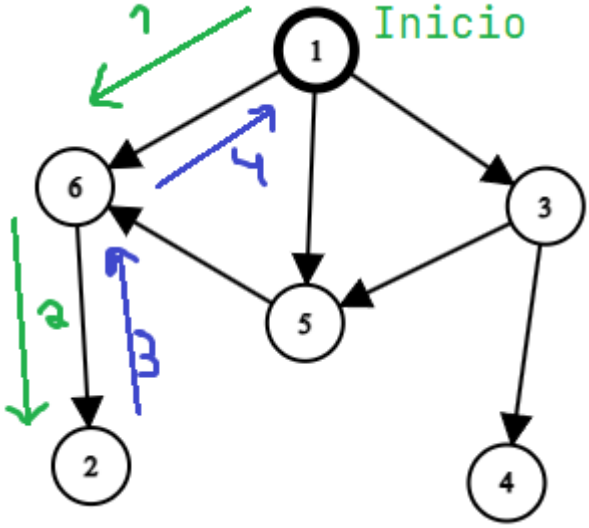
2. Una vez llegados al nodo 6, notamos que el único nodo que podemos visitar sería el nodo 2.



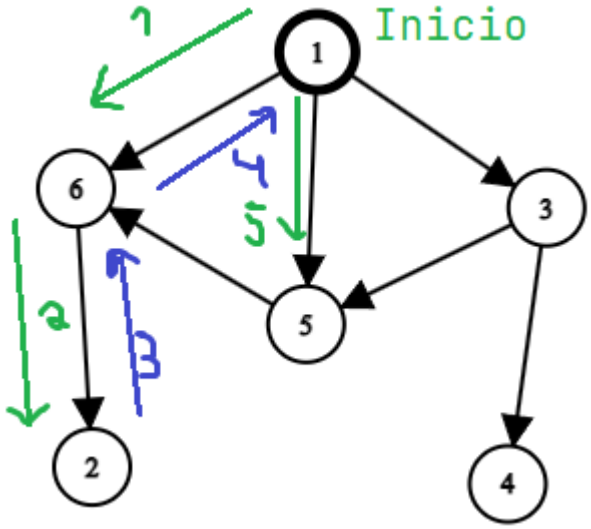
3. Cuando llegamos al nodo 2, notamos que ya no hay otro nodo posible por visitar. Así que volvemos al nodo del que vinimos. Es decir, volvemos al nodo 6.



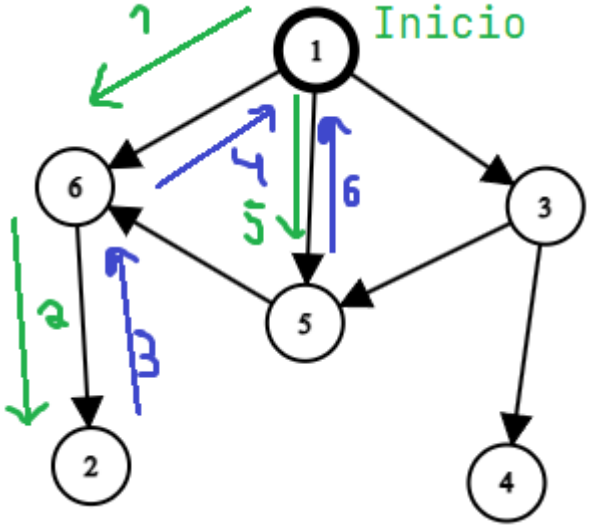
4. Al volver al nodo 6, nos damos cuenta que no hay otro nodos posible por visitar (ya visitamos el nodo 2). Así que volvemos al nodo inicial.



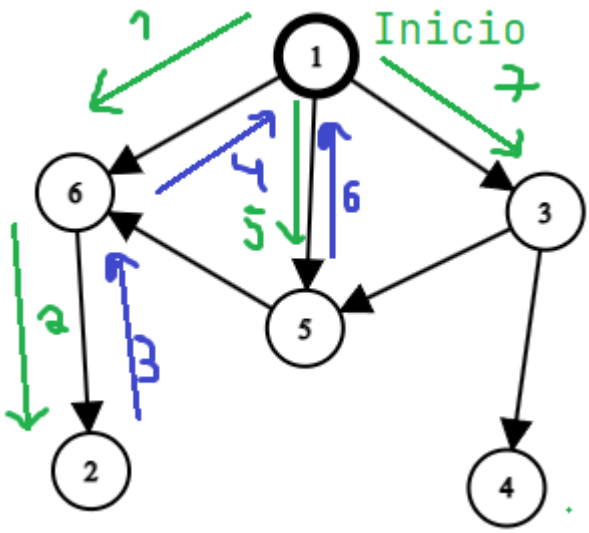
5. Una vez de vuelta al nodo inicial, vemos que nodos nos quedan por visitar. En este caso, como ya visitamos el nodo 6, podemos optar por el nodo 5 y 3. De este modo, como mencionamos anteriormente que íbamos a ir de izquierda a derecha, procederemos con visitar el nodo 5.



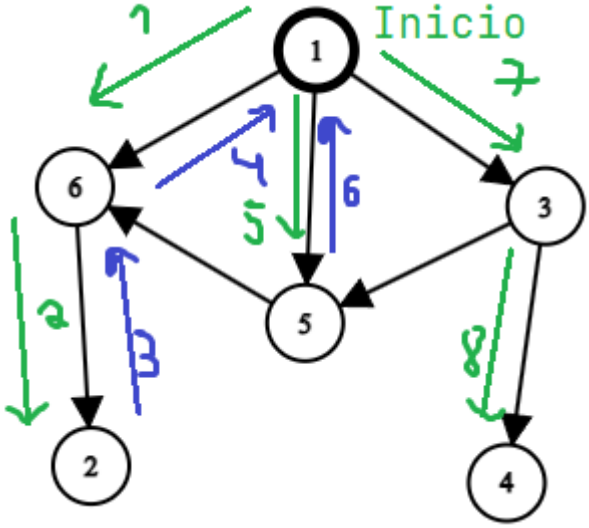
6. Ya en el nodo 5, vemos que nodos podemos visitar y notamos que la única opción sería el nodo 6. No obstante, este nodo ya fue visitado, así que (como ya no hay otra opción) volvemos al nodo del que vinimos (el inicial).



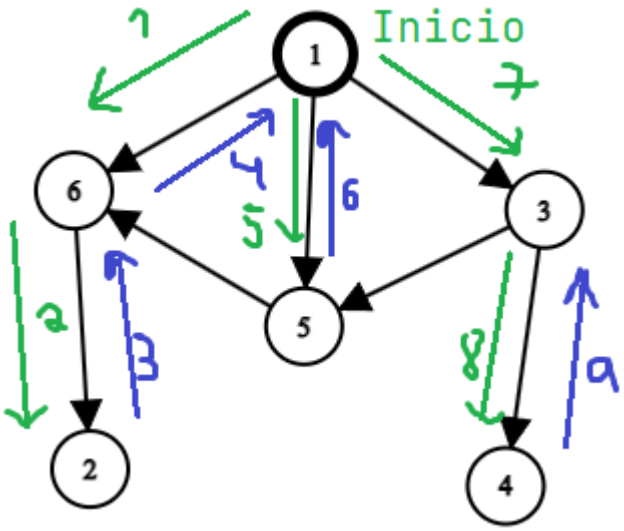
7. Ahora, el único nodo que nos queda por visitar desde el nodo inicial (1) sería nodo 3. Por lo tanto, nos dirigimos hacia allí.



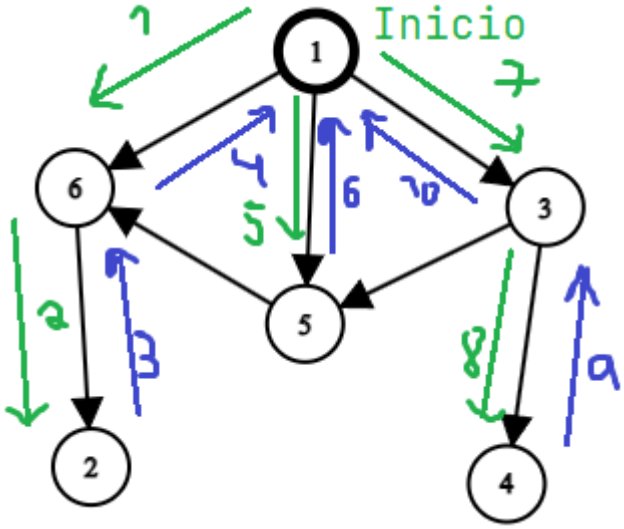
8. Desde el nodo 3, observamos que, siguiendo el orden de izquierda a derecha, deberíamos visitar el nodo 5. No obstante, este nodo ya ha sido visitado. Por lo tango continuaríamos con el siguiente: el nodo 4.



9. Ya llegados a este nodo, observamos que no podemos visitar a ningún otro nodo. Por lo tanto, volvemos al nodo del que vinimos.



10. De vuelta al nodo 3, como no tenemos nada más que visitar, volvemos al nodos del que vinimos. Es decir, al nodo inicial.



11. Finalmente, una vez llegamos al nodo inicial, notamos que ya no tenemos más nodos por visitar. De este modo, acabó nuestro algoritmos de DFS para recorrer el grafo.

Aquí es cuando notamos que el recorrido que sigue va desde el nodo inicial hasta el más profundo posible (un nodo que no tenga más nodos por visitar). Por eso se llama búsqueda en profundidad.

¿Y cómo hubiese sido para la búsqueda?

El algoritmo y recorrido hubiese sido el mismo. La única diferencia es que cada vez que visitamos un nodo revisamos si el valor del nodo es el que estamos buscando. Si es el caso, podemos parar el recorrido e indicar que sí se encontró. Caso contrario, se recorrerá todos los nodos e indicaremos que no fue encontrado.

Implementación Recursiva en Python

```
def dfs(graph, src):
  visited = [False] * len(graph)

  # START
  # Función recursiva embebida que ejecutará el algoritmo DFS

  def recursion(node):
    # El nodo en el que estamos actualmente está siendo
    # visitado justo ahora
    visited[node] = True

    # Aquí podemos procesar/operar con el nodo. En este
    # caso, sólo imprimiremos su valor.
    print(node, end = " -> ")

    # Vemos que nodos vecinos tiene el nodo actual
    for neighbour in graph[node]:
      # Si el nodo vecino no ha sido visitado, lanzamos
      # otro DFS desde ahí (en este caso, la función)
      # se llama recursion
      if not visited[neighbour]:
        recursion(neighbour)

  # END

  # Ejecutamos el algoritmos DFS desde el nodo inicial
  recursion(src)

if __name__ == "__main__":
  # Grafo del ejemplo
  # Como los nodos empiezan en 1, podemos
  # restarle 1 al valor de todos para que
  # encaje la lista desde el índice 0 o
  # colocamos que el nodo 0 no tiene conexión
  # con nadie. En este caso usaremos esta última opción
  graph = [
    [],
    [6, 5, 3],
    [],
    [5, 4],
    [],
    [6],
    [2]
  ]
```

```
# El nodo inicial desde el que partimos es el nodo 1
dfs(graph, 1)
```

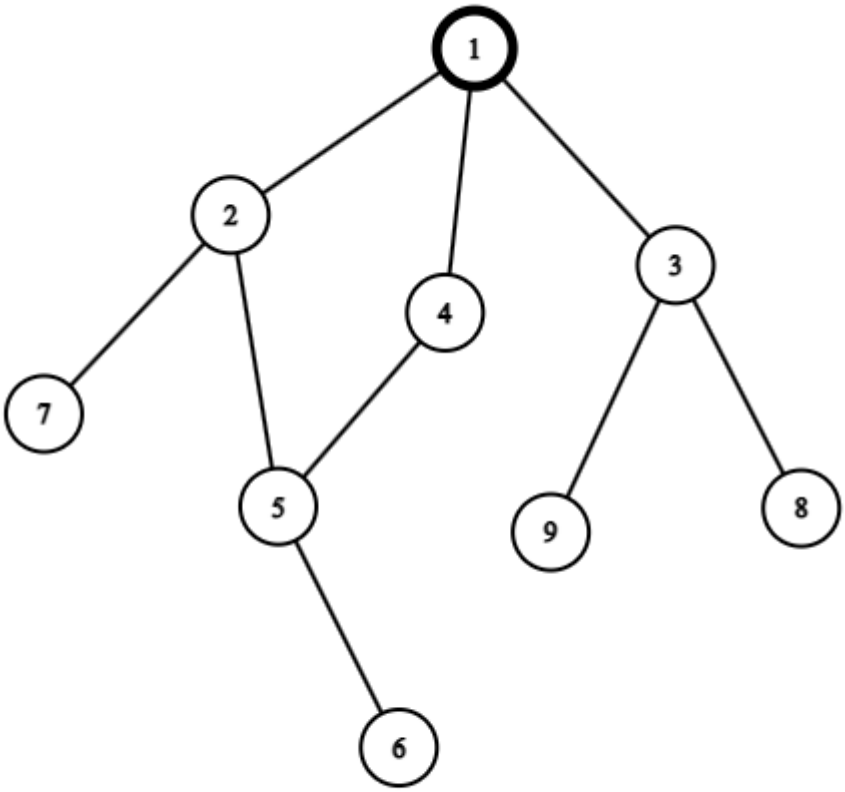
Breadth First Search (BFS)

El algoritmo de Búsqueda Primero en Anchura (Breadth-First Search, por sus siglas en inglés) empieza por un nodo inicial y se exploran todos los nodos vecinos a ese nodo en un nivel antes de pasar a explorar los vecinos del siguiente nivel.

En otras palabras, el algoritmo comienza por el nodo inicial, visita todos sus vecinos directos y, a continuación, visita todos los vecinos directos de esos vecinos y así sucesivamente, hasta que se hayan visitado todos los nodos accesibles desde el nodo inicial. Este algoritmo se basa en una estrategia de búsqueda primero en anchura, donde se prioriza la búsqueda en los nodos vecinos más cercanos antes de explorar los nodos más alejados.

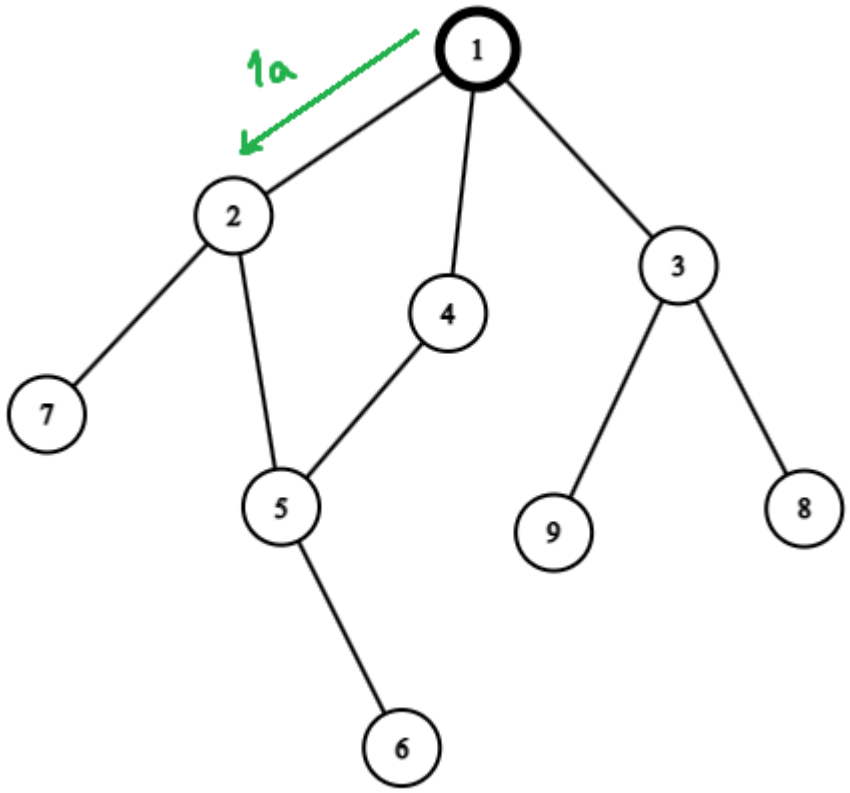
El algoritmo de BFS se utiliza comúnmente para encontrar la ruta más corta entre dos nodos en un grafo o para recorrer todos los nodos de un grafo. También se puede utilizar para encontrar el árbol de expansión mínimo en un grafo ponderado.

A continuación, se tiene el siguiente grafo no dirigido de ejemplo:

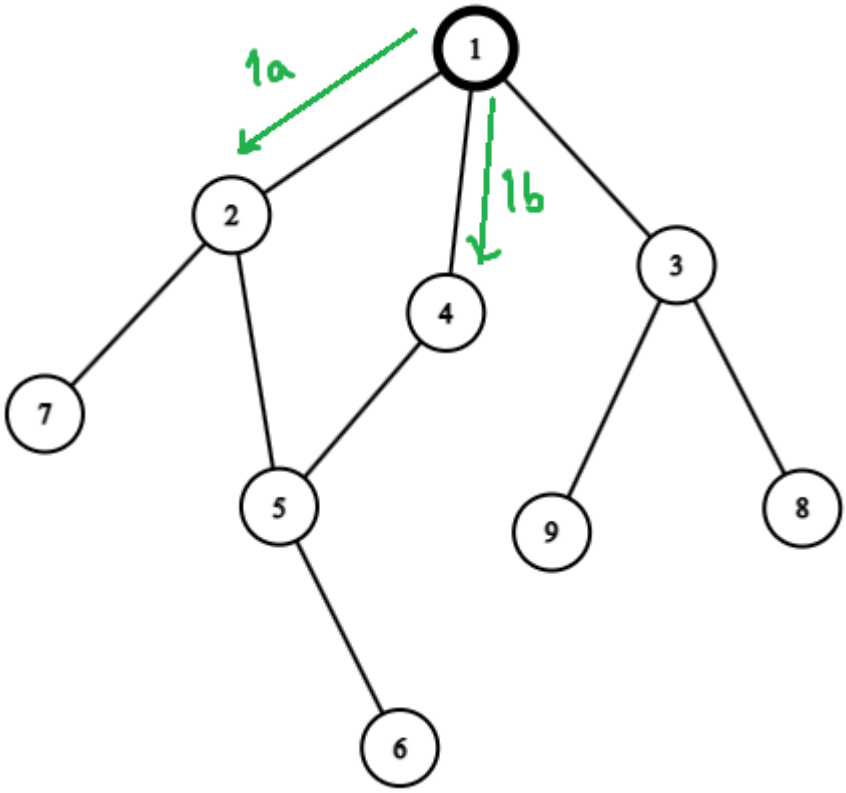


Asimismo, realizaremos el recorrido partiendo del nodo 1.

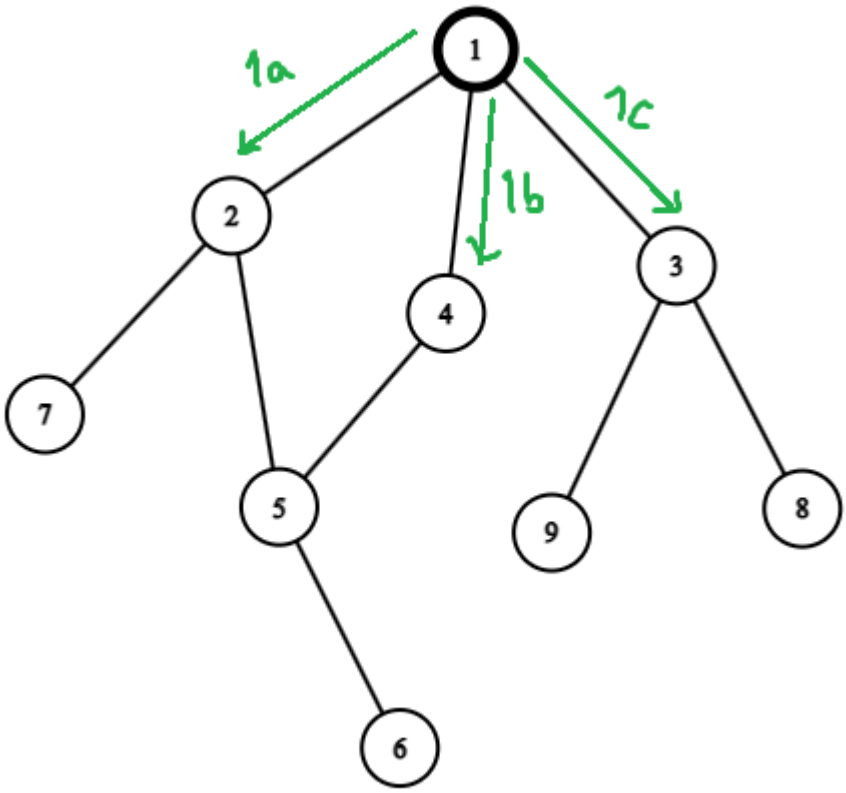
1. Como se mencionó anteriormente, partiremos el nodo inicial y visitaremos todos los vecinos de este. De este modo, si vamos de izquierda a derecha:
    a. Visitamos el nodo 2



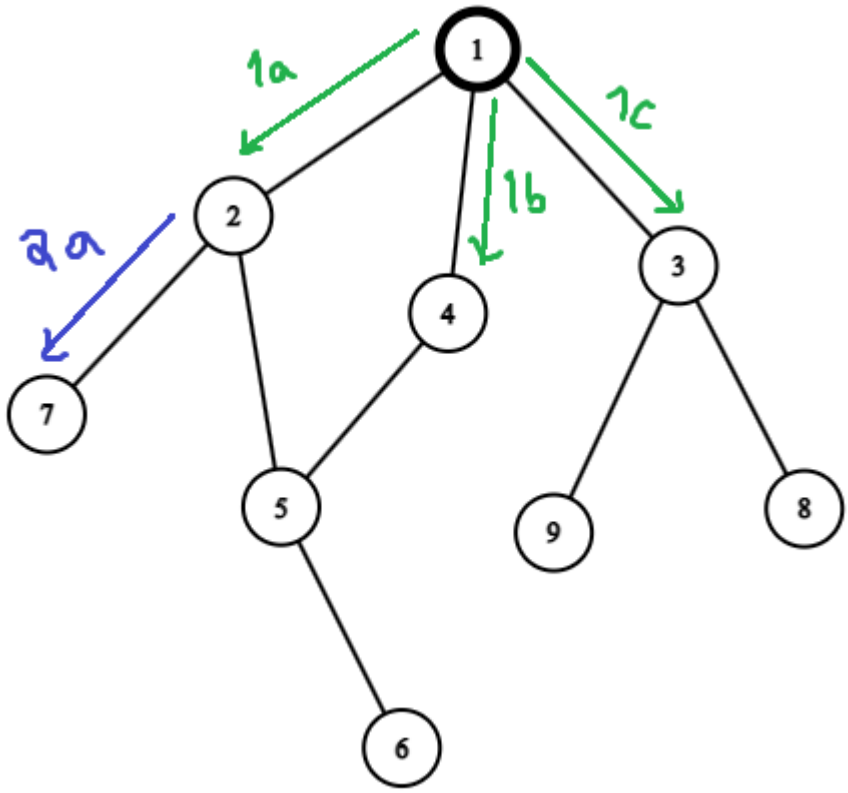
b. Visitamos el nodo 4



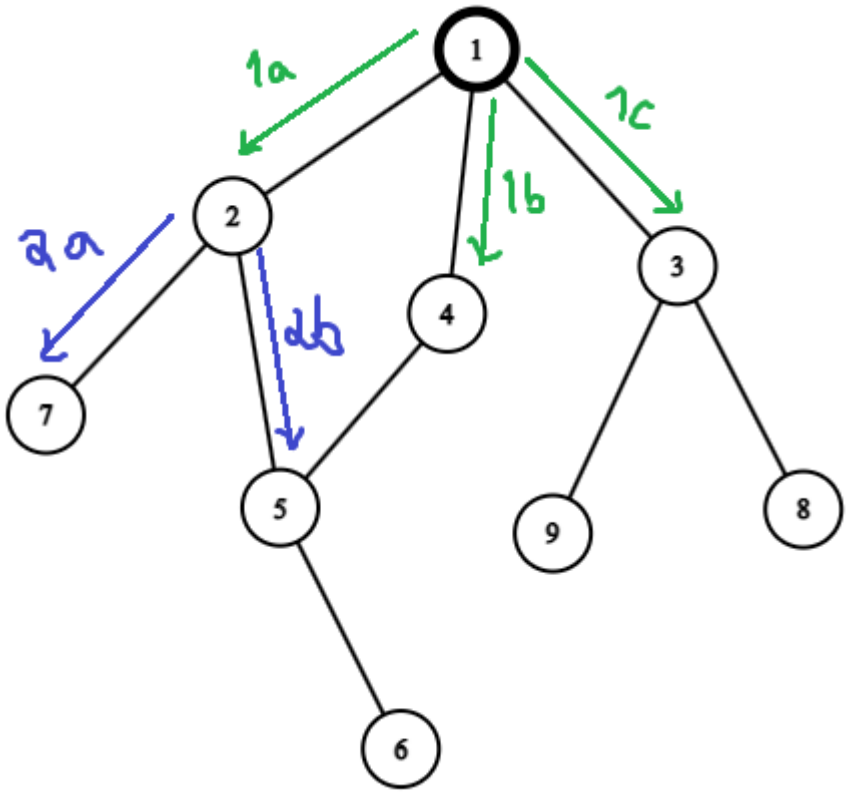
c. Visitamos el nodo 3



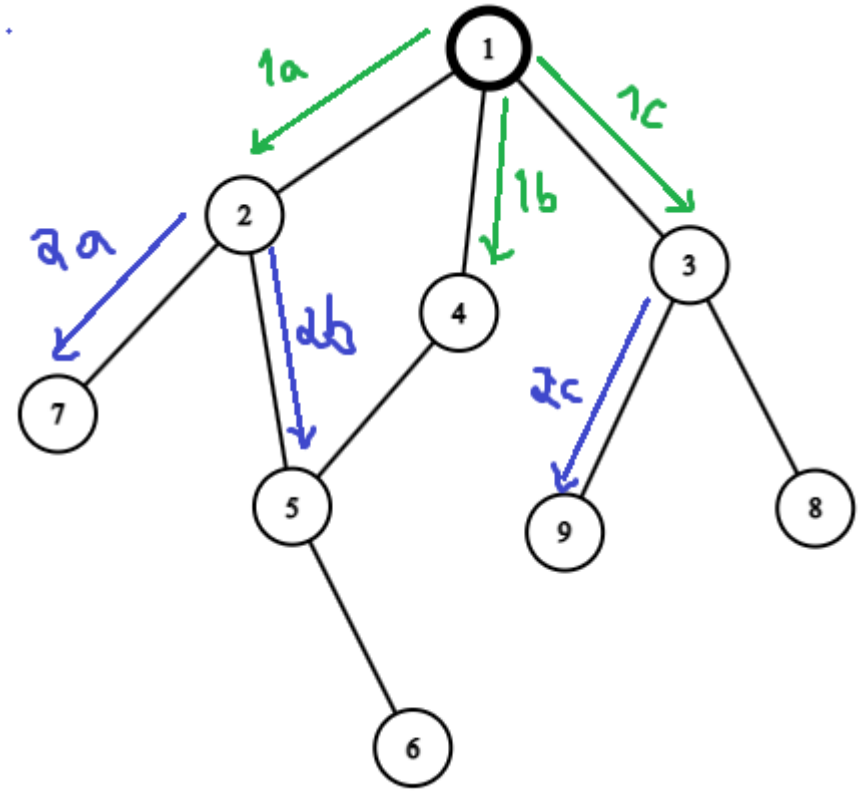
2. Cómo ya terminamos de visitar todos los nodos del nodo 1, procederemos a realizar lo mismo para cada uno de los nodos vecinos del nodo 1.
   a. Del nodo 2, visitamos al nodo 7



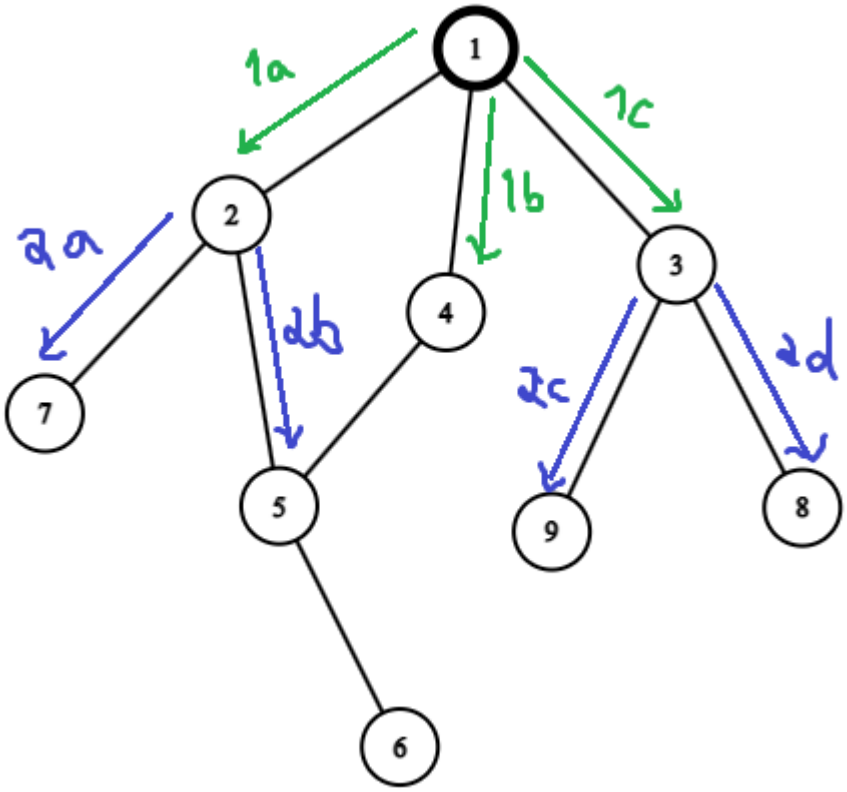
b. Del nodo 2, visitamos al nodo 5.



c. Del nodo 4, visitaríamos el nodo 5. No obstante, este nodo ya fue visitado. Por lo tanto, como no hay más nodos por visitar con 4, continuaremos con el nodo 3 que visitará al nodo 9.



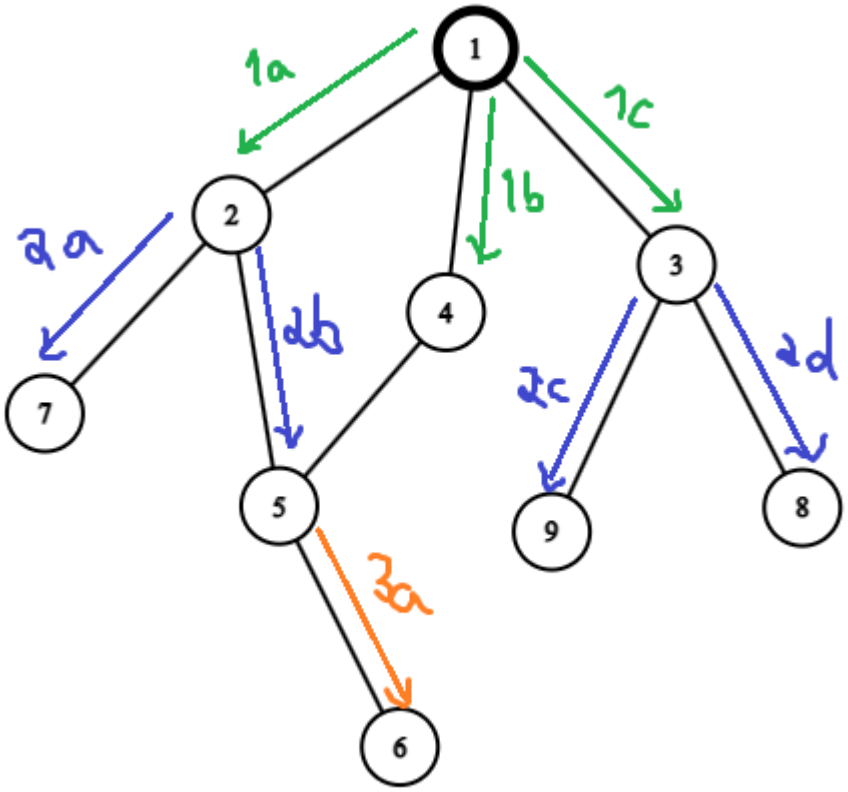
d. Del nodo 3, visitamos el nodo 8.



3. Al terminar de visitar todos los vecinos de la segunda capa/nivel. Ahora procederemos a realizar lo mismo para los vecinos del 3er nivel.

    a. Del nodo 7, notamos que no tenemos ningún vecino no visitado por visitar (dado que al ser un grafo no dirigido, tenemos como vecino al 2, pero este ya fue visitado anteriormente). Por lo tanto, procedemos a continuar con el siguiente: el nodo 5.

    Desde este nodo, notamos que tenemos por visitar al nodo 6.



4. Es así como continuaríamos con el nodo 9. Sin embargo, este nodo no tiene nodos vecinos por visitar. Por lo que, continuaríamos con el nodo 8, pero este nodo tampoco tiene más nodos por visitar. Así que, ya terminamos de revisar este nivel y continuamos con el siguiente.
5. Una vez llegados a este nivel, notamos que el nodo 6 (el único de ese nivel) no tiene más nodos vecinos por visitar. En consecuencia, el algoritmos de BFS terminaría aquí.

Una vez más, aquí es cuando notamos que el recorrido que sigue va desde el nodo inicial por todo el ancho que le rodea (los vecinos) y así consecutivamente (hasta que se llega a un nodo que no tiene más nodos por visitar). Por este motivo, se llama búsqueda en anchura.

Implementación en Python

```
def bfs(graph, src):
  visited = [False] * len(graph)
```

```
  # Iniciamos una cola para tener control del orden
  # secuencial en el que se deben visitar los nodos
  queue = []

  # El primer nodo a procesar es el inicial
  # y lo podemos marcar como visitado
  queue.append(src)
  visited[src] = True

  # El algoritmo se ejecuta mientra existan
  # elementos en la cola

  while len(queue):
    # Extraemos el primer elemento de la cola
    # ya que es el que tiene prioridad sobre
    # el resto
    node = queue.pop(0)

    # Aquí podemos procesar/operar con el nodo. En este
    # caso, sólo imprimiremos su valor.
    print(node, end = " -> ")

    # Recorremos los nodos vecinos al nodo actual:
    for neighbour in graph[node]:
      # Si el nodo no ha sido visitado, lo agregamos a la
      # cola, hasta que sea su turno. Dado que estamos
      # agregando los nodos de un nivel a la cola, estos
      # se procesarán primero antes que los de los demás
      # niveles.
      if not visited[neighbour]:
        # El nodo lo marcamos como visitado
        visited[neighbour] = True
        # Y lo agregamos a la cola
        queue.append(neighbour)


if __name__ == "__main__":
  # Grafo del ejemplo
  # Como los nodos empiezan en 1, podemos
  # restarle 1 al valor de todos para que
  # encaje la lista desde el índice 0 o
  # colocamos que el nodo 0 no tiene conexión
  # con nadie. En este caso usaremos esta última opción
  graph = [
    [],
    [2, 4, 3],
    [1, 7, 5],
    [1, 9, 8],
    [1, 5],
    [2, 4, 6],
    [5],
    [2],
    [3],
    [3]
  ]
```

```
  # El nodo inicial desde el que partimos es el nodo 1
  bfs(graph, 1)
```

Extra: Algoritmos de DFS Iterativo

Dado que podemos simular el stack de llamadas recursivas con la estructura de datos del mismo nombre, podemos implementar este algoritmo de manera iterativa. Para esto, haremos uso de la implementación del algoritmos de BFS y haremos un cambio ligero:

```
def dfs(graph, src):
  visited = [False] * len(graph)

  # Iniciamos un stack para tener la simulación
  # de las llamadas recursivas
  stack = []

  # El primer nodo a procesar es el inicial
  # y lo podemos marcar como visitado
  stack.append(src)
  visited[src] = True

  # El algoritmo se ejecuta mientra existan
  # elementos en el stack

  while len(stack):
    # Extraemos el último elemento del stack
    # ya que es el que tiene prioridad sobre
    # el resto
    node = stack.pop()

    # Aquí podemos procesar/operar con el nodo. En este
    # caso, sólo imprimiremos su valor.
    print(node, end = " -> ")

    # Recorremos los nodos vecinos al nodo actual:
    for neighbour in graph[node]:
      # Si el nodo no ha sido visitado, lo agregamos al
      # stack.
      if not visited[neighbour]:
        # El nodo lo marcamos como visitado
        visited[neighbour] = True
        # Y lo agregamos al stack
        stack.append(neighbour)


if __name__ == "__main__":
  # Grafo del ejemplo
  # Como los nodos empiezan en 1, podemos
  # restarle 1 al valor de todos para que
```

```
# encaje la lista desde el índice 0 o
# colocamos que el nodo 0 no tiene conexión
# con nadie. En este caso usaremos esta última opción
graph = [
  [],
  [2, 4, 3],
  [1, 7, 5],
  [1, 9, 8],
  [1, 5],
  [2, 4, 6],
  [5],
  [2],
  [3],
  [3]
]

# El nodo inicial desde el que partimos es el nodo 1
dfs(graph, 1)
```