Tu nombre
Algoritmos en gráficas 2021
Tarea 5
Fecha de entrega: 28 de febrero

El objetivo de esta tarea es que logres entender con mayor profundidad la estructura de los árboles generadores, además que, ejecutando un ejemplo, puedas mirar las diferencias principales entre los tres algoritmos clásicos para MSTs que estudiamos en clase.

1. Sea G una gráfica no dirigida con pesos en sus aristas, y sea T el árbol generador de peso mínimo de G (supongamos que es único).

a. Describe y analiza la complejidad de un algoritmo que actualice T cuando el peso de una arista $e \in E(T)$ se incremente.

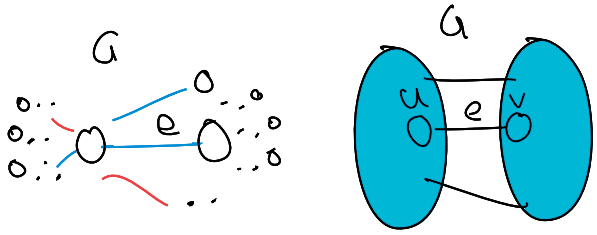
b. Describe y analiza la complejidad de un algoritmo que actualice T cuando el peso de una arista $e \in E(G)$ se decremente.

c. Describe y analiza la complejidad de un algoritmo que actualice T cuando se inserte una nueva arista e a la gráfica.

a. Para este algoritmo supondremos la gráfica G está representada por una lista de adyacencia, de igual manera para T, y los pesos están en una tabla hash que mapea a cada arista con su peso correspondiente.

Cuando una arista $e = (u, v)$ en T se incrementa, puede que siga perteneciendo al árbol de peso mínimo, o puede que tenga que ser reemplazada por otra arista.

Podemos primero remover la arista e, eso nos genera una partición del conjunto de aristas azules T en dos subárboles T_1 y T_2, esos subárboles pertenecen al MST T. Teniendo esto, para actualizar el árbol solo hay que aplicar la regla azul, seleccionando como corte para la regla cualquiera de los dos subárboles T, es decir, buscamos la arista de menor peso que una a los subárboles.



Para hacer eso primero removemos a $e = (u, v)$ de T y decimos arbitrariamente que u pertenece a T_1 y v a T_2, después hacemos un DFS sobre T tomando como vértice inicial a u, y en el recorrido vamos marcando a todos los vértices encontrados, eso lo hacemos para encontrar a todos los vértices que pertenecen a T_1, y por último hacemos un recorrido sobre todas las aristas de G y buscamos a las aristas que tengan en un extremo a una arista de T_1 y en el otro a un arista que no esté en T_1 (está en T_2).

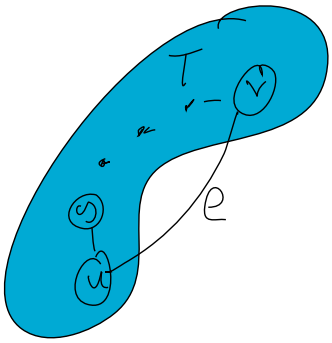
```
IncrementaAristaMST(e):
    remueve e de T    O(máx(deg(u), deg(v)))
    para todos vértice v en G:
      v.label ← 2
    (u, v) ← e
    LabelDFS(u)
    ActualizaArista(e)
```

```
LabelDFS(u):
    marca u
    / / este T ya no tiene a la arista e
    para toda arista u → v en T
      si v no está marcada
        v.label ← 1
        LabelDFS(v)
```

ActualizaArista(e):
  $e_{min} \leftarrow e$
  para toda arista $e' = (u, v) \in E(G)$
    si $u.label \neq v.label$ y $e' < w(e_{min})$
      $e_{min} \leftarrow e'$
  agrega e_{min} a T

La complejidad del algoritmo es la siguiente: para remover la arista e de T se necesita $O(\text{máx}(\deg(u), \deg(v)))$ tiempo, para asignarle la etiqueta a todos los vértices el valor 2 se necesita tiempo $O(V)$, la complejidad de la búsqueda y etiquetado del subárbol T_1 cuesta tiempo $O(V+E)$, y ActualizaArista tiene como complejidad $O(E)$, así que el costo total del algoritmo es $O(V + E)$

b Para modificar T al decrementar el peso de una arista $e = (u, v)$ de la gráfica G, si e también está en T entonces no hacemos nada, sigue perteneciendo. En el caso de que $e \notin T$, entonces podemos agregar ese vértice temporalmente a T, eso lo hacemos con el propósito de generar un ciclo en T para ver si esa arista e ahora debe de incluirse en el MST o no, en ese ciclo buscamos al vértice de mayor peso y lo removemos de T.



DecrementaAristaGráfica(e):
  agregar arista e a T
  $e' \leftarrow$ AristaMaxima(v, e_{max})
  remover e' de T

ComparaArista (e_1, e_2)
  si $w(e_1) < w(e_2)$
    return e_2
  de otra manera
    return e_1

AristaMaxima(x, e_{max}):
// Nos fuimos por el camino equivocado
// Ese $\deg(x)$ es el grado de x en T, no en G
  si $\deg(x) = 1$
    return -1
  para toda arista (x, y) en T
    si $x = u$
      return ComparaArista(e_{max}, (x, y)
    de otra manera
      $e_{max} \leftarrow$ ComparaArista(e_{max}, (x, y))
  return e_{max}

Como en el inciso anterior, la complejidad de remover una arista es $O(\text{máx}(\deg(u), \deg(v)))$ y para agregar una arista $O(1)$. La ejecución de AristaMaxima toma $O(V+E)$ tiempo, así que la complejidad del algoritmo es $O(V + E)$

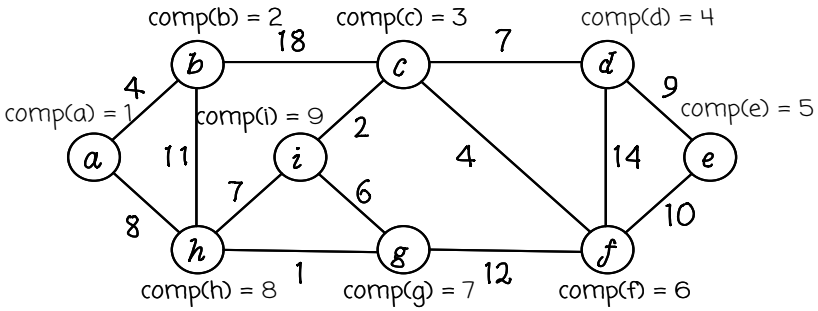
c De manera muy similar al inciso b), para determinar si la nueva arista $e = (u, v)$ agregada a G pertenece al MST T, la agregamos temporalmente a T y buscamos en el ciclo formado al vértice de mayor peso para removerlo de T. El análisis del algoritmo es idéntico al inciso b)

AgregarAristaGráfica(e):
    agrega arista e a G
    agregar arista e a T
    $e' \leftarrow$ AristaMaxima(v, e_{max})
    remover e' de T

■

2. Ejecuta el algoritmo de Boruvka, Jarnik y Kruskal en la gráfica que se muestra en la figura. Muestra tu trabajo paso a paso.

Algoritmo de Boruvka

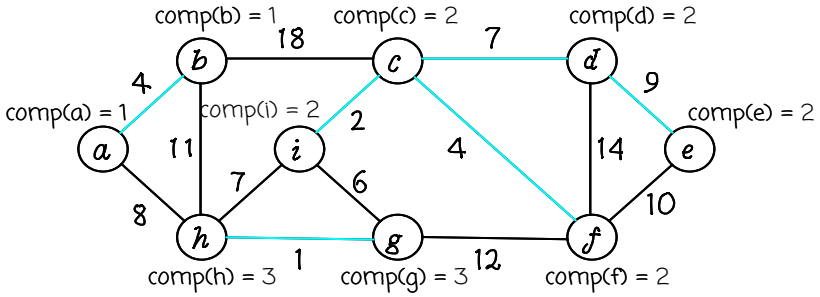


count = 9

edge uv = ab
safe[1] = Null
  safe[1] <- ab
safe[2] = Null
  safe[2] <- ab
edge uv = ah
safe[1] != Null and w(ah) = 8 > 4 = w(safe[1] = ab)
safe[8] = Null
  safe[8] <- ah
edge uv = bh
safe[2] != Null and w(bh) = 11 > 4 = w(safe[2] = ab)
safe[8] != Null and w(bh) = 11 > 8 = w(safe[8] = ah)
edge uv = bc
safe[2] != Null and w(bc) = 18 > 4 = w(safe[2])
safe[3] = Null
  safe[3] <- bc
edge uv = ci
safe[3] != Null and w(ci) = 2 < 18 = w(safe[3])
  safe[3] <- ci
safe[9] = Null
  safe[9] <- ci
edge uv = cd
safe[3] != Null and w(cd) = 7 > w(safe[3]) = 2
safe[4] = Null
  safe[4] <- cd
edge uv = hi
safe[8] != Null and w(hi) = 7 < w(safe[8]) = 8
  safe[8] <- hi
safe[9] != Null and w(hi) = 7 < w(safe[9]) = 2
edge uv = ig
safe[9] != Null and w(ig) = 6 > w(safe[9]) = 2
safe[7] = Null
  safe[7] <- ig
edge uv = hg
safe[8] != Null and w(hg) = 1 < w(safe[8]) = 7
  safe[8] <- hg
safe[7] != Null and w(hg) = 1 < w(safe[8) = 6
  safe[7] <- hg
edge gf
safe[7] != Null and w(gf) = 12 > w(safe[7]) = 1
safe[6] = Null
  safe[6] <- gf
edge fe
safe[6] != Null and w(fe) = 10 < w(safe[6]) = 12
  safe[6] <- fe
safe[5] = Null
  safe[5] <- fe
edge de
safe[4] != Null and w(de) = 9 > w(safe[4]) = 7
safe[5] != Null and w(de) = 9 < w(safe[5]) = 10
  safe[5] <- de
edge df
safe[4] != Null and w(df) = 14 > w(safe[4]) = 7
safe[6] != Null and w(df) = 14 > w(safe[6]) = 10
edge cf
safe[3] != Null and w(cf) = 4 > w(safe[3]) = 2
safe[6] != null and w(cf) = 4 < w(safe[6]) = 10
  safe[6] <- cf

safe

1	2	3	4	5	6	7	8	9
/	/	/	/	/	/	/	/	/
ab	ab	/	/	/	/	/	/	/
ab	ab	/	/	/	/	/	ah	/
ab	ab	bc	/	/	/	/	ah	/
ab	ab	ci	/	/	/	/	ah	ci
ab	ab	ci	cd	/	/	/	ah	ci
ab	ab	ci	cd	/	/	/	hi	ci
ab	ab	ci	cd	/	/	ig	hi	ci
ab	ab	ci	cd	/	/	hg	hg	ci
ab	ab	ci	cd	/	gf	hg	hg	ci
ab	ab	ci	cd	fe	fe	hg	hg	ci
ab	ab	ci	cd	de	fe	hg	hg	ci
ab	ab	ci	cd	de	cf	hg	hg	ci



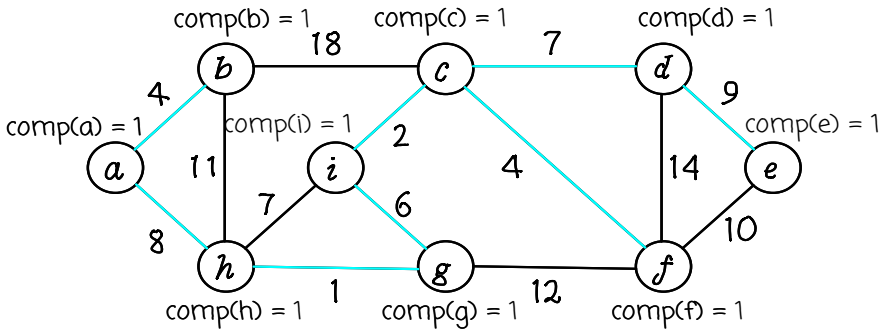
count = 3

edge uv = ab ; comp(a) = 1 = comp(b)
.....
<todas las aristas uv que tienen
comp(u) = comp(v) >
....
edge uv = ah
safe[1] = Null
  safe[1] <- ah
safe[3] = Null
  safe[3] <- ah
edge uv = bh
w(bh) = 11 > 8 = w(ah)
w(bh) = 11 > 8 = w(ah)
edge uv = hi
w(hi) = 7 < 8 = w(safe[3] = ah)
  safe[3] <- hi
safe[2] = Null
  safe[2] <- hi

edge uv = bc
w(bc) = 18 > 8 = w(safe[1] = ah)
w(bc) = 18 > 7 = w(safe[2] = hi)
edge uv = ig
w(ig) = 6 < 7 = w(safe[2] = hi)
  safe[2] <- ig
w(ig) = 6 < 7 = w(safe[3] = hi)
  safe[3] <- ig
edge uv = gf
w(gf) = 12 > 7 = w(safe[3])
w(gf) = 12 > 7 = w(safe[2] = ig)

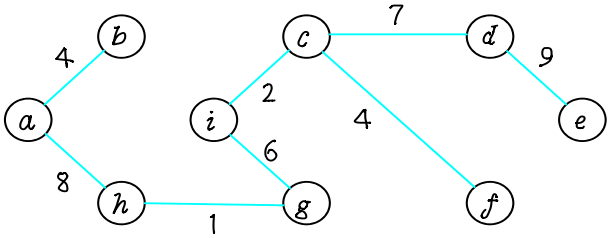
safe

1	2	3
/	/	/
ah	/	ah
ah	hi	hi
ah	ig	ig

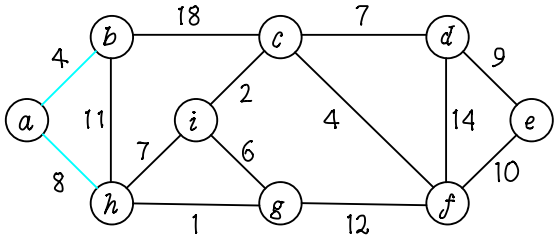


count = 1

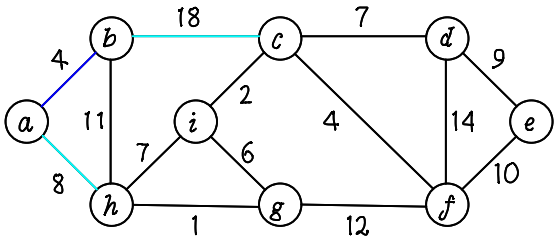
MST



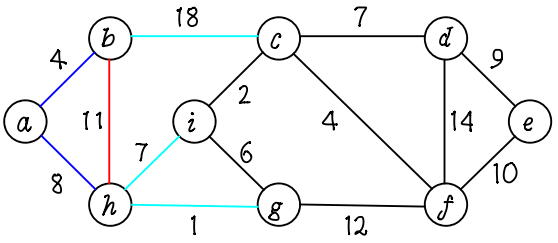
Algoritmo de Jarnik



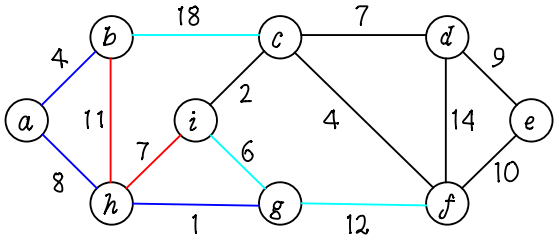
	Q	key	π
u →	r	0	NIL
	b	4	a
	c	∞	NIL
	d	∞	NIL
	e	∞	NIL
	f	∞	NIL
	g	∞	NIL
	h	8	a
	i	∞	NIL



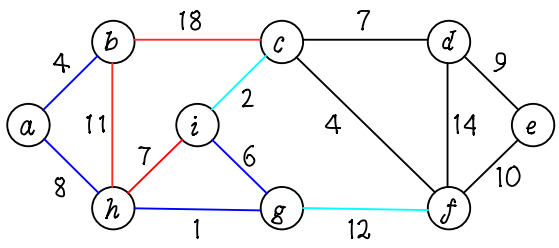
	Q	key	π
	r	0	NIL
u →	b	4	a
	c	18	b
	d	∞	NIL
	e	∞	NIL
	f	∞	NIL
	g	∞	NIL
	h	8	a
	i	∞	NIL



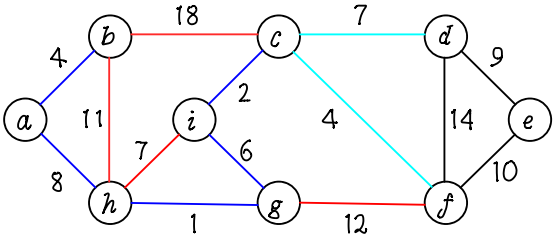
	Q	key	π
	r	0	NIL
	b	4	a
	c	18	b
	d	∞	NIL
	e	∞	NIL
	f	∞	NIL
	g	1	h
u →	h	8	a
	i	7	h



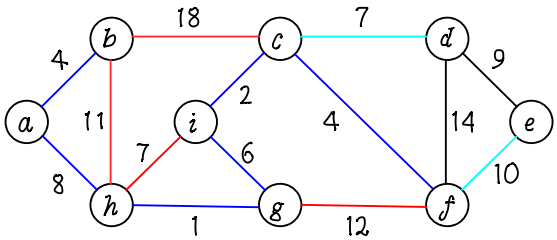
	Q	key	π
	r	0	NIL
	b	4	a
	c	18	b
	d	∞	NIL
	e	∞	NIL
	f	12	g
u →	g	1	h
	h	8	a
	i	6	g



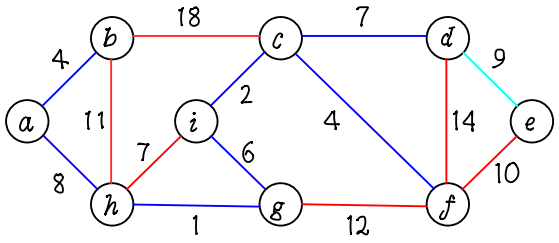
	Q	key	π
	r	0	NIL
	b	4	a
	c	2	c
	d	∞	NIL
	e	∞	NIL
	f	12	g
	g	1	h
	h	8	a
u ->	l	6	g



	Q	key	π
	r	0	NIL
	b	4	a
u ->	c	2	l
	d	7	c
	e	∞	NIL
	f	4	c
	g	1	h
	h	8	a
	l	6	g



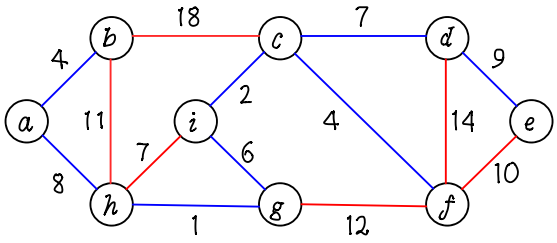
	Q	key	π
	r	0	NIL
	b	4	a
	c	2	l
	d	7	c
	e	10	f
u ->	f	4	c
	g	1	h
	h	8	a
	l	6	g



u ->

Q key π

r	0	NIL
b	4	a
c	2	i
d	7	c
e	9	d
f	4	c
g	1	h
h	8	a
i	6	g



u ->

Q key π

r	0	NIL
b	4	a
c	2	i
d	7	c
e	9	d
f	4	c
g	1	h
h	8	a
i	6	g

■

3. Sea $G = (V, E)$ una gráfica no dirigida. Una *partición de tamaño* r de los vértices de G es una colección $V_1, V_2, ..., V_r$ de subconjuntos de V tales que:

- $V_i \neq \emptyset, 1 \leq i \leq r$
- $V_i \cap V_j = \emptyset, i \neq j, 1 \leq i, j \leq r$
- $\bigcup_{i=1}^{r} V_i = V$.

Por otra parte, decimos que dos árboles generadores de G son *disjuntos (en aristas)* si sus conjuntos de aristas son disjuntos, es decir, si no existe ninguna arista de G que esté en ambos árboles.

En este problema vamos a investigar, usando Sage, cuántos árboles generadores disjuntos tiene una gráfica.

Considera cualquier partición de tamaño r de V, y considera cualquier árbol generador T de G, nota que hay al menos $r-1$ aristas que cruzan la partición (recuerda que una arista cruza la partición si sus extremos están en partes distintas). Hint: Intenta demostrar computacionalmente esto.

Usando Sage, reúne evidencia que sustente el siguiente Teorema.

Theorem 1. *Una gráfica contiene k árboles generadores disjuntos si y sólo si toda partición P de sus vértices tiene al menos $k(r-1)$ aristas que cruzan la partición.*

Haz un reporte discutiendo tus resultados.

Tu reporte debe contener: tu algoritmo, análisis de la complejidad de tu algoritmo como función de la complejidad de las funciones que uses de Sage, la lista de gráficas que utilizaste para tus experimentos, una discusión de los resultados de tus experimentos. En particular deberás argumentar si tu algoritmo comprueba computacionalmente el teorema o lo refuta.

Este ejercicio no lo pude terminar, tuve problemas en una parte de mi algoritmo

Para comprobar que el teorema es cierto necesitamos comprobar los dos sentidos de la equivalencia.

Para comprobar el primer sentido (Si una gráfica contiene k árboles generadores disjuntos entonces toda partición P de sus vértices tiene al menos $k(r-1)$ aristas que cruzan la partición, primero generaremos una gráfica aleatoria G, luego contaremos su número de arboles generadores disjuntos y después generaremos un número aleatorio de particiones con las que verificaremos el teorema, solo lo comprobamos un número aleatorio de veces para cada gráfica aleatoria G (en lugar de verificar el teorema para todas las particiones posibles para G). La razón por la que hacemos esto es porque el número de particiones del conjunto de vértices V es muy grande. Todo este proceso de generar una gráfica aleatoria G (conexa) lo repetimos un número suficientemente grande de veces.

Para contar el número de árboles disjuntos de una gráfica G hacemos una secuencia de recorridos DFS, en cada iteración removemos las aristas del árbol generador generado por DFS, eso hasta que la gráfica G se desconecte.

```
#retorna verdadero si fue posible remover un arbol
#generador, en caso contrario retorna falso
def remueve_arbol_generador(G, v):
    marcado = [False for i in G.vertices()]
    num_marcados = 0

    def remueve_arbol_gen(v):
        nonlocal marcado
        nonlocal num_marcados
        marcado[v] = True
        num_marcados += 1
```

```
        for w in G.neighbor_iterator(v):
            if (not marcado[w]):
                G.delete_edge(v,w)
                remueve_arbol_gen(w)

    remueve_arbol_gen(v)
    return True if (num_marcados == len(G.vertices())) else False
```

```
def cuenta_arboles_gen_disjuntos(G):
    arboles = 0
    #tomamos un vertice arbitrario como raiz del arbol
    #generador
    G_cpy = G.copy()
    while(remueve_arbol_generador(G_cpy,0)):
        arboles += 1
    return arboles
```

Esta es la parte que no me funciona de mi algoritmo, el conteo de los árboles generadores es erróneo, no se si sea un error del algoritmo para contar los árboles o del código

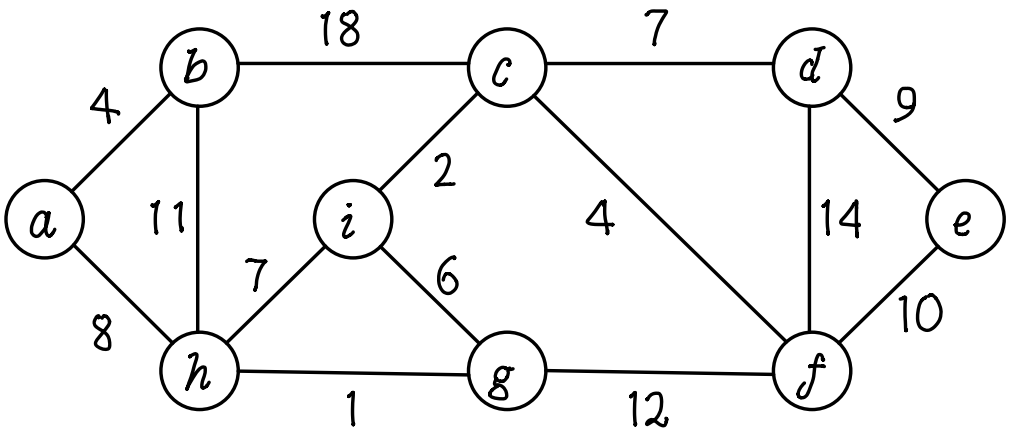
Para generar una partición aleatoria de tamaño r simplemente generamos un arreglo de tamaño N, donde N es el número de vértices de la gráfica, en donde cada índice del arreglo representa un vértice y cada valor del arreglo la partición a la que el vértice pertenece, como solo hay r conjuntos posibles a los que puede pertenecer un vértice v, simplemente le asignamos al arreglo un número aleatorio en el rango $[0, N-1]$

```
#regresa un arreglo tal que a[i] = k si
#y solo si el v rtice i pertenece a la particion indexada en k
def particion_aleatoria(N):
    particion = [0 for i in range(0,N)]
    for i in range(0,N):
        #valor aleatorio en el rango [0, N)
        particion[i] = randrange(0,N)
    return particion
```

Y para contar el número de aristas que cruzan una partición, iteramos sobre todos las aristas de G y contamos las aristas que tienen sus extremos en diferentes particiones

```
def cuenta_cruzan_particion(G, particion):
    cruzan = 0
    # aristas u -> v
    for (u, v, _) in G.edge_iterator():
        #sus extremos estan en particiones distintas
        if (particion[u] != particion[v]):
            cruzan += 1
    return cruzan
```

■


a
b
c
d
e
f
g
h
i
4
18
7
9
11
2
4
14
7
6
8
10
1
12