Explique como es que se pueden comparar dos algoritmos sin tener en cuenta el Hardware en el que estan siendo ejecutados. ¿Cómo es posible que se puedan comparar?. Ponga un ejemplo.

Remitirse al apunte de complejidad computacional. Lo que se pide es explicar de forma sintética como establacemos el modelo matemático que nos permite calcular complejidades de diferentes algorítmos como vimos en clase (notación Big O).

Ordene en orden creciente de complejidad los algoritmos con las siguientes ecuaciones de recurencia asociadas:

a. T(n)= 4T(n/4)+n^2
b. G(n)= 16T(n/4) + n
c. F(n)=3T(n/2)+ n^2

Basta con aplicar teorema maestro para luego ordenar los resultados. Primero calculamos el coeficiente **n^(log_b(a))** para los 3 casos

a -> n^log_4(4) = n
b -> n^log_4(16) = n^2
c -> n^log_2(3) = n^1.585

Luego comparamos los coeficientes con el término **f(n)** de cada ecuación y determinamos la solución en base a si dicho coeficiente es menor, mayor o igual al término.

a -> n **<** n^2 => **Ta(n) = Θ(n^2)**
b -> n^2 **>** n => **Tb(n) = Θ(n^2)**
c -> n^1.585 **<** n^2 => **Tc(n) = Θ(n^2)**

Por lo tanto, los 3 algoritmos poseen, según teorema maestro, la misma complejidad computacional.

Dada la siguiente definición de lista doblemente enlazada recursiva:

```c
typedef struct lista{
  struct lista* siguiente;
  struct lista* anterior;
  void* elemento;
}lista_t;
```

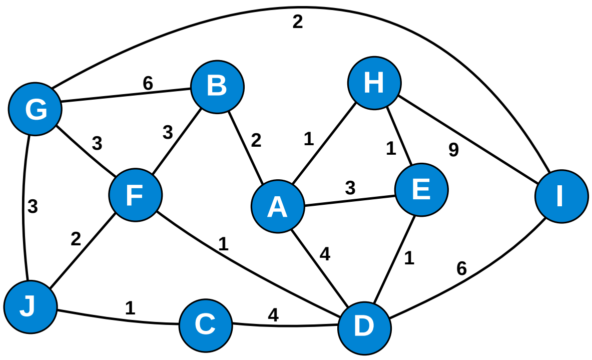
Implemente la siguiente función de forma completamente recursiva (no se admiten for/while/do, etc).

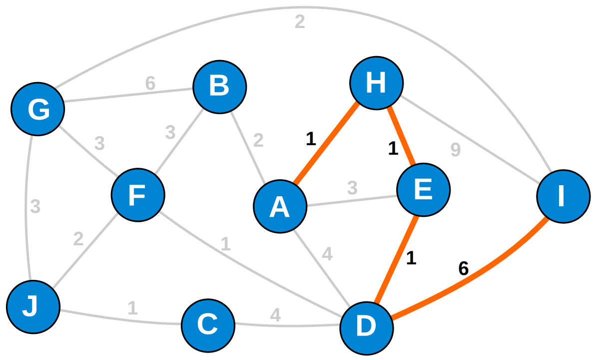
Nota: A los efectos de este ejercicio, puede asumir que malloc/calloc nunca fallan.

```c
lista_t* lista_insertar(lista_t* lista, void* elemento);

lista_t* lista_eliminar(lista_t* lista, void* elemento, int (*comparador)(void*,void*));
```

```c
typedef struct lista{
  struct lista* siguiente;
  struct lista* anterior;
  void* elemento;
}lista_t;

lista_t* crear_lista(void* elemento){
  lista_t* lista = calloc(1, sizeof(lista_t));
  lista->elemento = elemento;
  return lista;
}

lista_t* lista_insertar(lista_t* lista, void* elemento){
  if(!lista)
    return crear_lista(elemento);
  lista->siguiente = lista_insertar(lista->siguiente, elemento);
  lista->siguiente->anterior = lista;
  return lista;
}

lista_t* lista_eliminar(lista_t* lista, void* elemento, int (*comparador)(void*,void*)){
  if(!lista || !comparador)
    return lista;

  if(comparador(elemento, lista->elemento)==0){
    lista_t* siguiente = lista->siguiente;
    if(siguiente)
      siguiente->anterior = lista->anterior;
    free(lista);
    return siguiente;
  }

  lista->siguiente = lista_eliminar(lista->siguiente, elemento, comparador);
  return lista;
}
```

Dado el siguiente grafo:





Aplique **Dijkstra** desde **A** hasta **I** mostrando la tabla a cada paso.
Aplique **DFS,** empezando por el vértice **A,** explicando el procedimiento.

Armamos una tabla que contenga los vértices del grafo. Inicializamos la distancia al vértice inicial con 0 y el resto a infinito. En cada paso del algoritmo, tomamos el vértice con menor distancia y actualizamos las distancias (si son menores) desde ese vértice a sus vecinos próximos. Cuando nos toca visitar el vértice destino, ya finalizamos.

Vértice	A	B	C	D	E	F	G	H	I	J
Paso (1)	0+	2	-	4	3	-	-	1	-	-
Paso (2)	0*	2	-	4	2	-	-	1+	10	-
Paso (3)	0*	2+	-	4	2	5	8	1*	10	-
Paso (4)	0*	2*	-	3	2+	5	8	1*	10	-
Paso (5)	0*	2*	7	3+	2*	4	8	1*	9	-
Paso (6)	0*	2*	7	3*	2*	4+	7	1*	9	6
Paso (7)	0*	2*	7	3*	2*	4*	7	1*	9	6+
Paso (8)	0*	2*	7+	3*	2*	4*	7	1*	9	6*
Paso (9)	0*	2*	7*	3*	2*	4*	7+	1*	9	6*
Paso (10)	0*	2*	7*	3*	2*	4*	7*	1*	9+	6*

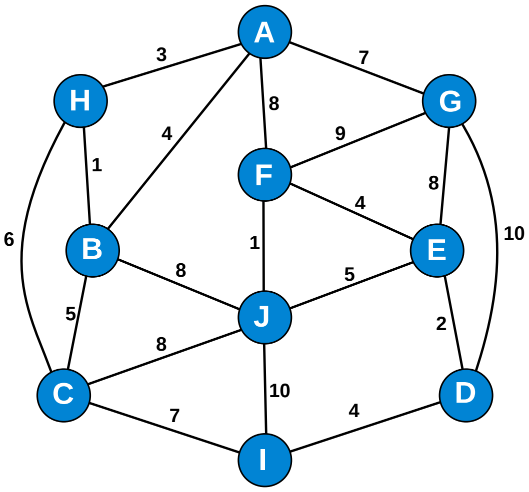
En base a la tabla podemos reconstruir el trayecto siguiendo el órden inverso, subiendo por la tabla y cambiando al vértice que estaba activo cada vez que el valor cambia. El trayecto en este caso fue **A-H-E-D-I**

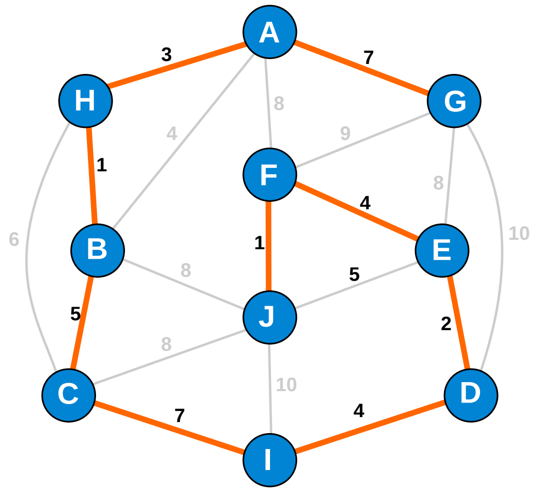
Para el recorrido **DFS**, podemos usar una impĺementación iterativa con pilas o una recursiva. Para la iterativa: metemos el vértice de inicio en una pila. Luego en cada paso, quitamos un vértice de la pila, si aun no fue visitado, lo marcamos como visitado y apilamos a los vecinos directos. Repetimos hasta quedarnos sin vértices.

Pila	Visitados
A	-
B,D,H,E	A
B,D,H,A,D,H	A,E
B,D,H,A,D,A,E,I	A,E,H
B,D,H,A,D,A,E,D,H	A,E,H,I
B,D,H,A,D,A,E,A,C,E,F,I	A,E,H,I,D
B,D,H,A,D,A,E,A,C,E,B,D,G,J	A,E,H,I,D,F
B,D,H,A,D,A,E,A,C,E,B,D,G,C,F,G	A,E,H,I,D,F,J
B,D,H,A,D,A,E,A,C,E,B,D,G,C,F,B,I,J	A,E,H,I,D,F,J,G
B,D,H,A,D,A,E,A,C,E,B,D,G,C,F,A,F,G	A,E,H,I,D,F,J,G,B
B,D,H,A,D,A,E,A,C,E,B,D,G,D,J	A,E,H,I,D,F,J,G,B,C
-	A,E,H,I,D,F,J,G,B,C

Los vértices coloreados son los ya visitados

Dado el siguiente grafo:





Aplique **Kruskal**, explicando el procedimiento.
Aplique **BFS,** empezando por el vértice **F**, explicando el procedimiento

Ordenamos las aristas de menor a mayor. Al empezar el algoritmo, cada uno de los vértices del grafo es un árbol de un solo vértice. Mientras quede mas de un árbol y tengamos aristas por recorrer, tomamos la menor arista y nos fijamos si une dos árboles diferentes. Si es asi, unimos los árboles y repetimos. Caso contrario se descarta la arista.
Las aristas ordenadas son: F-J(1), H-B(1), E-D(2), H-A(3), I-D(4), F-E(4), B-A(4), J-E(5), B-C(5), H-C(6), C-I(7), A-G(7), G-E(8), C-J(8), B-J(8), A-F(8), F-G(9), G-D(10), J-I(10). Luego:

```
[A],[B],[C],[D],[E],[F],[G],[H],[I],[J]
[A],[B],[C],[D],[E],[F-J],[G],[H],[I]    : Arista F-J(1)
[A],[C],[D],[E],[F-J],[G],[H-B],[I]      : Arista H-B(1)
[A],[C],[E-D],[F-J],[G],[H-B],[I]        : Arista E-D(2)
[C],[E-D],[F-J],[G],[A-H-B],[I]          : Arista H-A(3)
[C],[E-D-I],[F-J],[G],[A-H-B]            : Arista I-D(4)
[C],[J-F-E-D-I],[G],[A-H-B]              : Arista F-E(4)
[J-F-E-D-I],[G],[A-H-B-C]                : Arista B-C(5)
[G],[A-H-B-C-I-D-E-F-J]                  : Arista C-I(7)
[G-A-H-B-C-I-D-E-F-J]                    : Arista A-G(7)
```

Para el recorrido **BFS**, metemos el vértice de inicio en una cola. Luego en cada paso, quitamos un vértice de la cola, si el vértice no habia sido visitado, lo marcamos como visitado y encolamos a los vecinos directos. Repetimos hasta quedarnos sin vértices.

```
| Cola                                  | Visitados             |
|---------------------------------------+-----------------------|
| F                                     | -                     |
| A,E,G,J                               | F                     |
| E,G,J,B,F,G,H                         | F,A                   |
| G,J,B,F,G,H,D,F,G,J                   | F,A,E                 |
| J,B,F,G,H,D,F,G,J,A,D,E,F             | F,A,E,G               |
| B,F,G,H,D,F,G,J,A,D,E,F               | F,A,E,G,J             |
| F,G,H,D,F,G,J,A,D,E,F,A,C,H,J         | F,A,E,G,J,B           |
| D,F,G,J,A,D,E,F,A,C,H,J,A,B,C         | F,A,E,G,J,B,H         |
| F,G,J,A,D,E,F,A,C,H,J,A,B,C,E,G,I     | F,A,E,G,J,B,H,D       |
| H,J,A,B,C,E,G,I,B,H,I,J               | F,A,E,G,J,B,H,D,C     |
| B,H,I,J                               | F,A,E,G,J,B,H,D,C,I   |
| -                                     | F,A,E,G,J,B,H,D,C,I   |
```

Los vértices coloreados son los ya visitados

Justifique si la siguiente función de hashing es buena o no para ser utilizada en una tabla hash:

```
size_t funcion_hash(const char* string){
  if(!string || strlen(string)<3) return 0;

  size_t factor1 = string[0];
  size_t factor2 = string[1];
  size_t factor3 = string[2];

  return ((factor1*NUMERO_MAGICO)+(factor2*factor3))/FACTOR_NORMALIZADOR;
}

size_t funcion_hash(const char* string){
  if(!string || !*string) return 0;

  size_t acumulado = 0;

  for(size_t i=0;string[i];i++)
    acumulado = acumulado + string[i]*(i+1);

  return acumulado/(size_t)string;
}

size_t funcion_hash(const char* string){
  if(!string || !*string) return 0;

  size_t acumulado = 0;

  for(size_t i=0;string[i];i++){

    for(size_t j=0;string[j];j++)
      acumulado = acumulado + i + string[j]/(2*j+1);

    acumulado = acumulado / (i+1);
  }

  return acumulado;
}
```

En los 3 casos la respuesta podía ser sí o no. Lo importante, como siempre, es que pongan una justificación acorde (y que sea coherente con lo aprendido). Por ejemplo, podemos decir que la primer función de hashing no es apropiada porque dado cualquier string menor a 3 caracteres, el resultado siempre es 0, todos esos strings terminan colisionando. Adicionalmente, si tenemos strins de mas de 3 caracteres, sin importar que tan largos sean, si coinciden los primeros 3 caracteres, la dirección asignada también es la misma. Si miramos aún mas, se utilizan multiplicados directamente los caracteres 2 y 3 del string, por lo tanto cualquier string que tenga la forma 'ABCxxxx...' y 'ACBxxxx....' terminan también colisionando. En general las funciones de hashing para tablas de hash utilizan la totalidad de la clave.
Para las otras dos se puede hacer un análisis similar. Por ejemplo la segunda hace unos cálculos con la totalidad de la clave, pero antes de devolver el número, lo divide por la dirección de memoria del string. En este caso, pasarle la misma clave 2 veces a la función no garantiza que el resultado sea el mismo (ya que los strings pueden tener el mismo contenido pero estar ubicados en partes diferentes de la memoria).
El último caso podría ser o no. El cálculo que hace es **O(n^2)** para cada clave. Dependiendo del tipo de claves que se vayan a utilizar, puede que no sea lo mejor. Siempre buscamos que las funciones de hashing para tablas de hash sean (además de repetibles) rápidas de calcular.

Dado el siguiente Heap binario, insertar los valores **4** y **7** y luego eliminar 2 veces la raíz. Muestre el estado del heap luego de cada operación y justifique.

9	8	3	2	1			

1	2	8	3	9			

Primero hay que verificar que efectivamente sean heaps binarios los vectores (en este caso son). Para ello, comenzando desde el primer elemento (n=0), sabemos que el hijo izquierdo deberia estar almacenado en 2*n y el derecho en 2*n+1. Verificamos que se cumplam las reglas de heap (todo nodo es mayo/menor que sus nodos hijos). El primer heap es maximal, el segundo es minimal.

Para insertar un elemento en el heap, se agrega el elemento en la primer posición libre del vector y se hace **sift-up** (intercambiamos el elemento con su padre mientras que se violen las propiedades de heap). Para eliminar la raíz, se toma el último elemento del heap, se mueve a la posición de la raíz y se hace **sift-down** (se intercambie el nodo con el mayor/menor de sus hijos mientras se violen las propiedades de heap).

		PADRE					
9	8	3	2	1	4		

		PADRE					
1	2	8	3	9	4		

En ambos casos, luego de insertar el 4, se viola la propiedad de heap. En el primero, 4 (hijo) es mayor a 3 (padre) en un heap maximal, en el segundo, 4 (hijo) es menor a 8 (padre) en un heap minimal. Hacemos **sift-up** hasta que quede en su lugar. Insertamos 7 y repetimos lo mismo.

PADRE							
9	8	4	2	1	3		
		PADRE					
9	8	4	2	1	3	7	
PADRE							
9	8	7	2	1	3	4	

PADRE							
1	2	4	3	9	8		
		PADRE					
1	2	4	3	9	8	7	

Y ahora eliminamos 2 veces la raíz, moviendo el último elemento al inicio y acomodando

	HIJO_I	HIJO_D					
4	8	7	2	1	3		
			HIJO_I	HIJO_D			
8	4	7	2	1	3		

	HIJO_I	HIJO_D					
7	2	4	3	9	8		
			HIJO_I	HIJO_D			
2	7	4	3	9	8		
						HIJO_I	HIJO_D
2	3	4	7	9	8		

	HIJO_I	HIJO_D					
3	4	7	2	1			
					HIJO_I	HIJO_D	
7	4	3	2	1			

	HIJO_I	HIJO_D					
8	3	4	7	9			
			HIJO_I	HIJO_D			
3	8	4	7	9			
3	7	4	8	9			

Dados los siguientes recorridos de un **ABB** cuya forma es desconocida, reconstruya el **ABB** y explique el procedimiento.

Inorden: **!,%,=,#,@,&,$,+**
Preorden: **#,!,=,%,@,$,&,+**

Inorden: **!,%,=,#,@,&,$,+**
Preorden: **&,%,!,=,#,@,$,+**

Inorden: **!,%,=,#,@,&,$,+**
Preorden: **&,=,%,!,@,#,+,$**

El recorrido inorden nos da la relación de órden entre los diferentes símbolos. Los símbolos mas a la izquierda son menores que los símbolos a la derecha. Por ejemplo, **!** es menor que **%**, **=**, **#**, etc. Las tres variantes conservan la misma relación de órden entre los símbolos.
Luego, sabemos que en un recorrido preorden, primero se recorre la raíz, luego el hijo izquierdo y por último el derecho. Esto en principio nos indica que el primer elemento del recorrido es la raíz del árbol. El elemento que le sigue, puede ser hijo izquierdo o derecho (en ese órden de prioridad). Para saber si es hijo izquierdo o derecho, basta con fijarse en la relación de órden entre los símbolos. Como los símbolos menores siempre van a la izquierda y los mayores a la derecha, podemos determinar de que lado debemos dibujar el próximo elemento. De esta forma, recursivamente podemos ir recorriendo la lista preorden, y agregando nodos a árbol en el lugar correcto.

Tomando como ejemplo el primero, tenemos # como raíz, el siguiente elemento es **!**, que según el recorrido inorden es menor a # y por lo tanto va a la izquierda. El siguiente elemento (**=**) deberia ser el hijo izquierdo de **!**, derecho de **!** o derecho de #, con ese órden de prioridad. Como **=** es mayor a **!** y menor a #, va a la derecha de **!**. Y asi repetimos hasta al final.