Teoría de Grafos

Algoritmos y Estructuras de Datos

UNLA

Lic. Alejandro Sasin

Diego Cañete

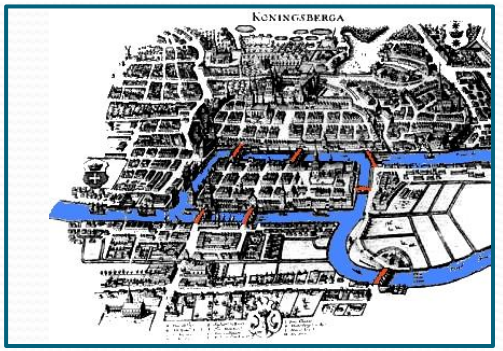
Contenido

- Teoría de grafos, definiciones, conceptos y ejemplos generales
- Aclaraciones puntuales sobre grafos
- Tipos de grafos, de representaciones y sus definiciones
- Recorridos básicos de grafos (BFS y DFS)
- Análisis de la complejidad algorítmica
- BigO notation & ejercicios explicativos

Grafos

- Un **grafo** (*graph*) es un T.D.A. que consta de un conjunto finito de nodos *V (vertex)* y unas relaciones *E* (*edge*) entre los nodos que lo componen.

  - Sirven para estudiar tanto la estructura de Internet, como una red de autopistas, la red de amigos en Facebook o hasta como interactúan las partículas elementales.
  - Creado por Euler, para solucionar un problema de interconexión de Puentes en Köenigsberg.



Definiciones

A nivel matemático se puede definir a un ***Grafo*** *(G) como un conjunto finito y no vacío de elementos llamados* ***Vértices*** *(V) y pares ordenados o desordenados de elementos llamados* ***Aristas*** *(E).*

Es decir:

- Donde e $\in$ E tiene forma {u, v}, donde u, v $\in$ V y u $\neq$ v $\neq \varnothing$

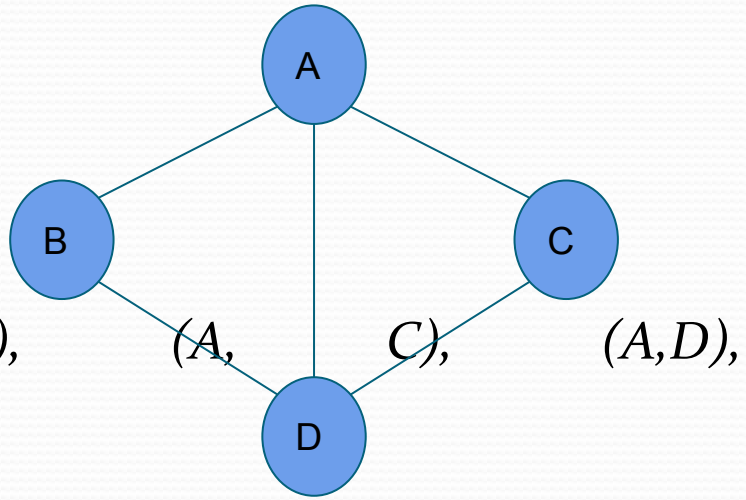
Entonces nuestro grafo representa la forma en que se relacionan binariamente entre sí los elementos dentro de él:

- G = (V, E)

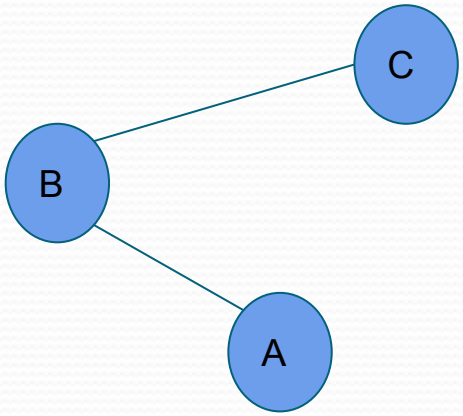
Ejemplos

Entonces si G = (V, E):

- V = 4
- E = 5
  - V = { *A, B, C, D* }
  - E = { *(A, B), (A, C), (A,D), (B, D), (C, D)* }



- V = 3
- E = 2
  - V = { *A, B, C* }
  - E = { *(A, B), (B, C)* }



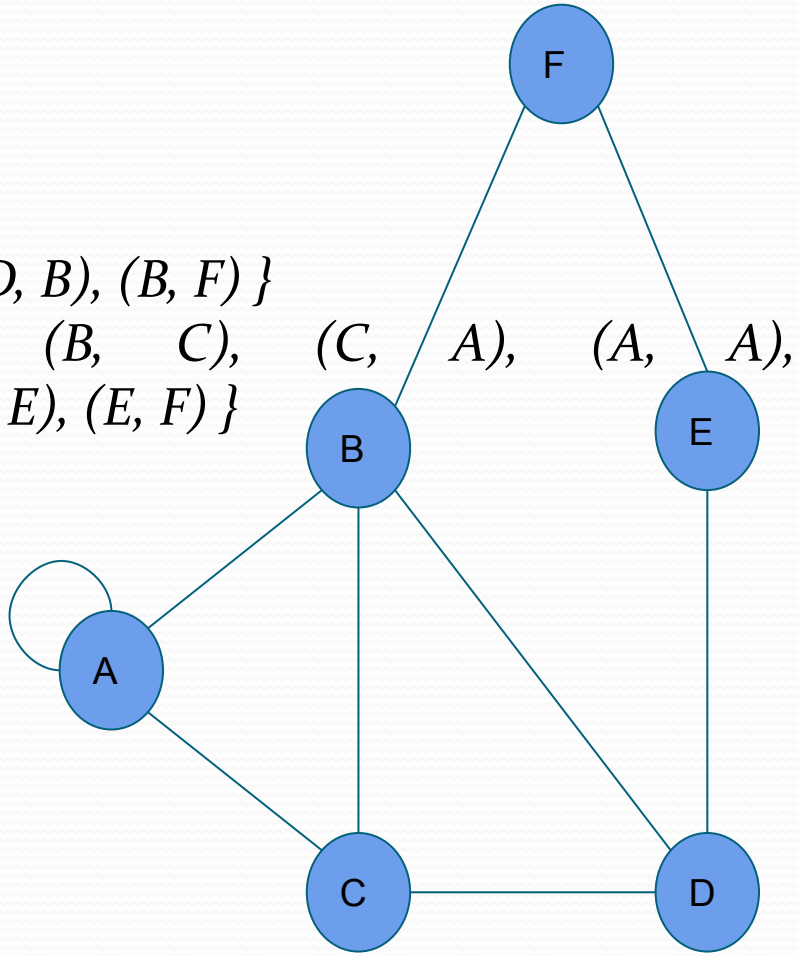
Conceptos Generales

Sea un Grafo (G), se llama:

- ***Grado de un Nodo:*** *Número de conexiones/relaciones que posee un nodo.*
- ***Cadena:*** *a toda sucesión finita alterna de Vértices (V) y Aristas (E).*
- ***Cadena Cerrada:*** *cadena en la que (V) inicial y final coinciden.*
- ***Camino:*** *cadena en la que no se repiten ni sus (V) ni sus (E).*
- ***Ciclo****: cadena en la que no se repiten ni sus (V) ni sus (E), a excepción del (V) inicial y final.*
- **Longitud de la Cadena:** Número de Aristas (E) que forman una cadena.

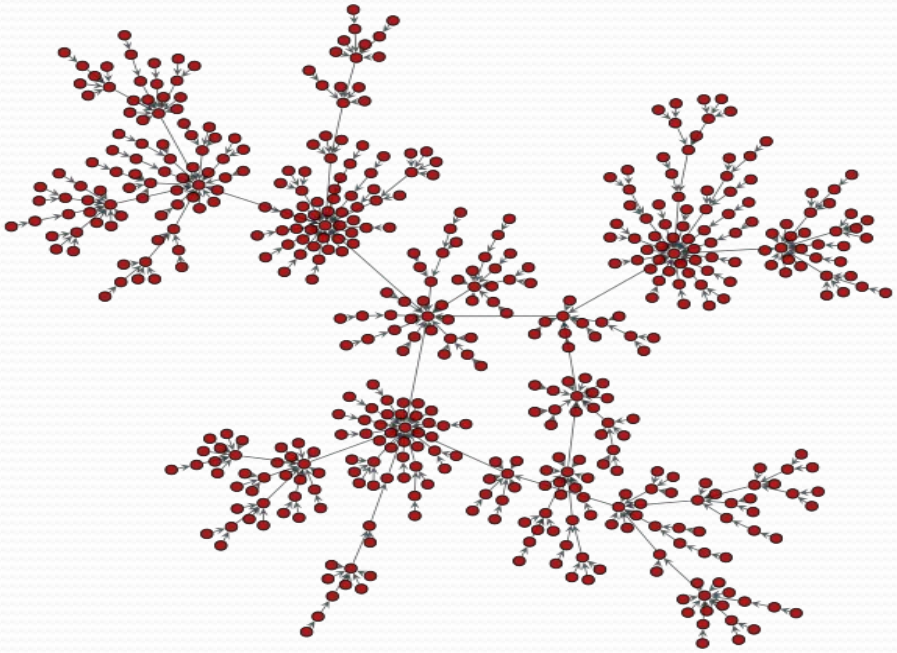
Ejemplos

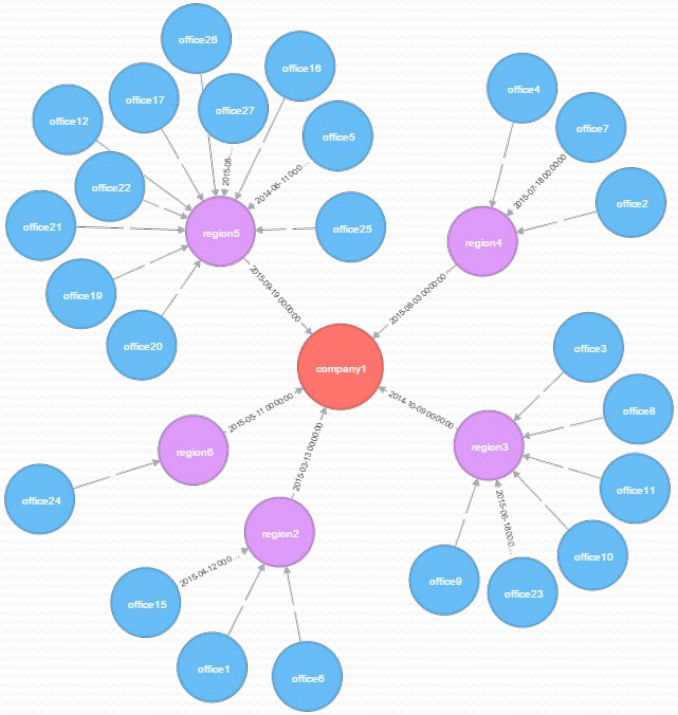
- ***Grado de un Nodo:***
  - *A: 3*
  - *B: 4*
  - *D:3*
- ***Cadena "C":*** *{ (A, B), (B, C), (C, D), (D, B), (B, F) }*
- ***Cadena Cerrada:*** *{ (F, B), (B, C), (C, A), (A, A), (A, B), (B, D), (D, E), (E, F) }*
- ***Camino:*** *{ (A, C), (C, D), (D, E) }*
- ***Ciclo****: { (A, B), (B, D), (D, C), (C, A) }*
- **Longitud de la Cadena: "C" = 5**



Aclaraciones

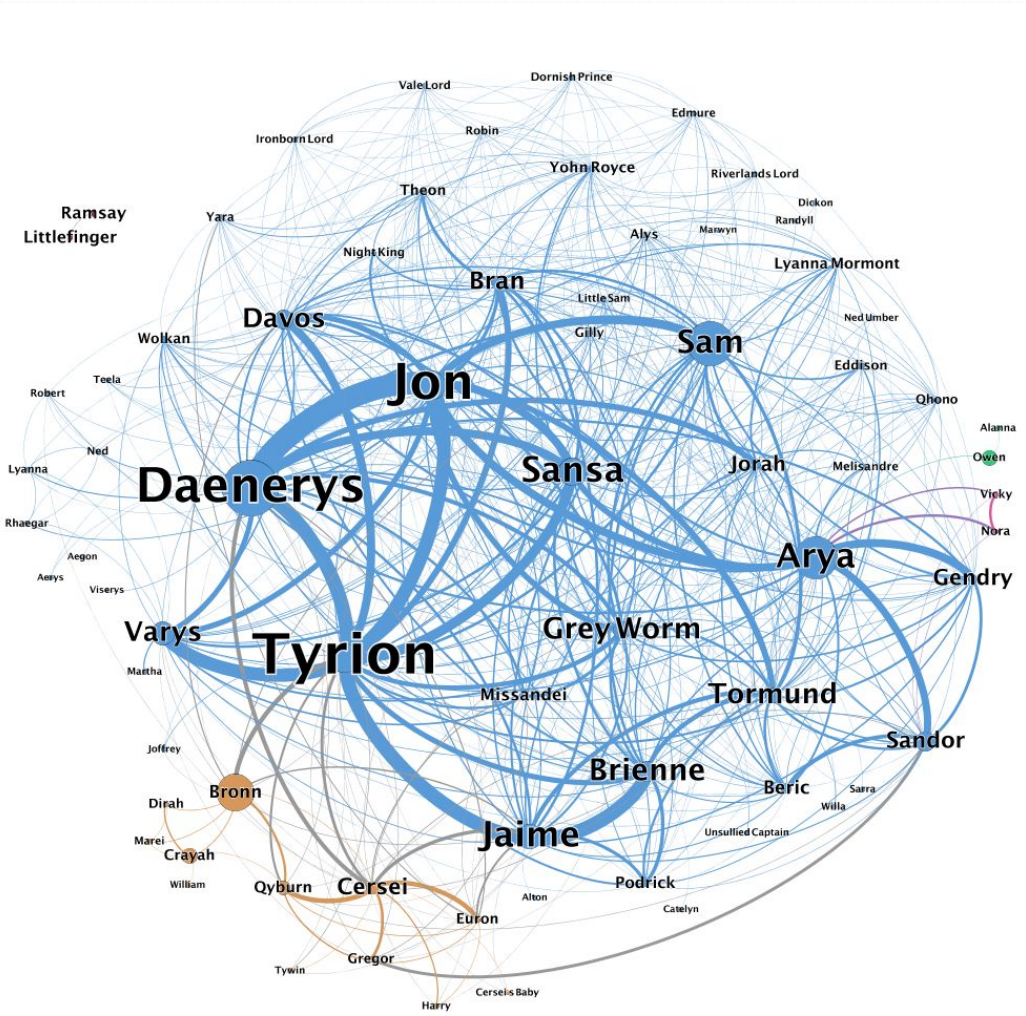
Con un Grafo se pueden estudiar miles de cosas, tienen millones de aplicaciones. Ejemplo, podemos saber cual es el nodo más importante de una red, si la red se puede recorrer visitando todos sus nodos, si se puede recorrer sin repetir ninguno o si hay pequeños grupos conectados entre sí.





Aclaraciones

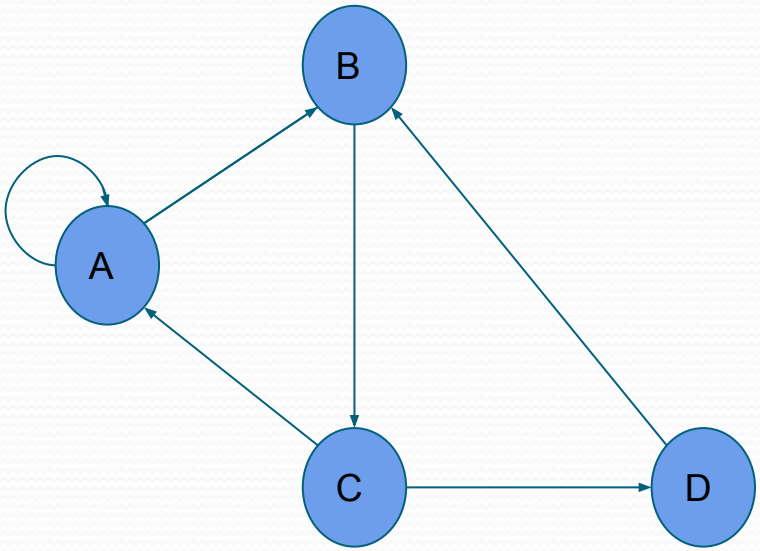
Andrew J. Beveridge y Jis Shan crearon una red social para determinar, con ciencia exacta, quién es el líder real en la historia.
El tamaño de cada punto corresponde al nivel de conexiones que el personaje tiene con otros individuos relevantes. El grosor de la línea muestra qué tan frecuente es la interacción entre ellos.

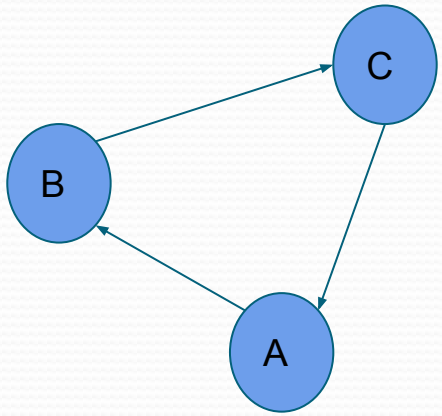


https://networkofthrones.wordpress.com/

Grafos Dirigidos

- *Un* ***Grafo Dirigido*** *G* es un par (V, E), donde V es un conjunto finito, no vacío y E es una relación binaria en V, es decir, un conjunto de pares **ordenados** de elementos de V.
  - Nos interesa la dirección (el sentido) de sus Aristas.





Representaciones

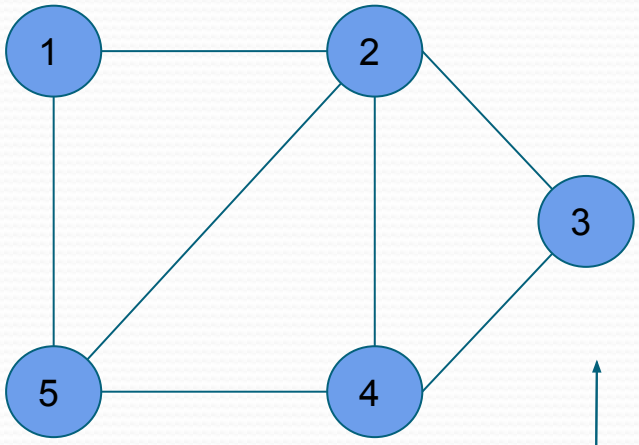
Para representar a esta estructura de datos de manera computacional y que sea sencillo de recorrerlas. Existen dos formas *standard* de representación, ambas se pueden utilizar de manera independiente para los dos tipos de grafos:

- ***Listas de Adyacencia:*** son útiles cuando el número de (V) son muy superior al número de (E).

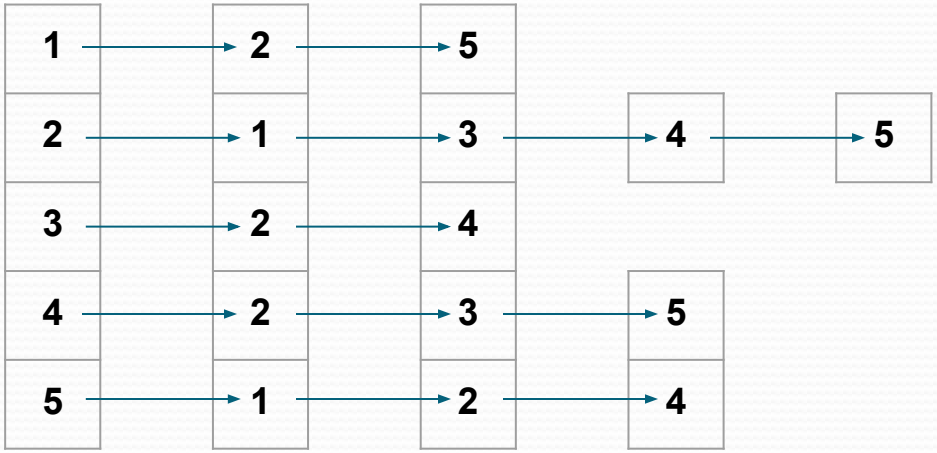
- ***Matriz de Adyacencia:*** van bien cuando buscamos conectividad rápida entre dos nodos.

Definiciones

*Las **Listas de Adyacencia:***

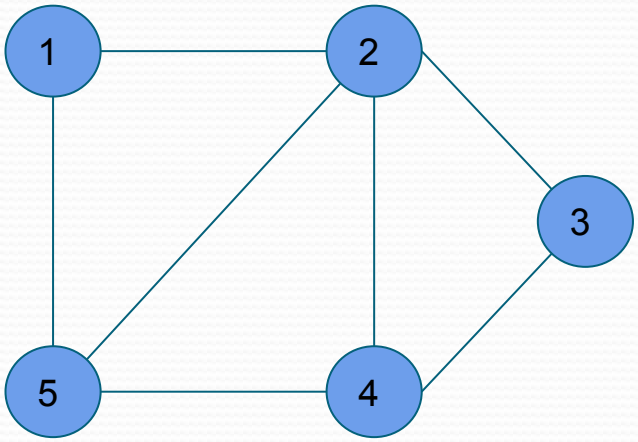


- ***Para GND nuestra L.A. => 2.E = 14***
- ***Para GD nuestra L.A => E = 7***



Definiciones

*Una **Matriz de Adyacencia:***



- → 0 si $\nexists$ V(i,j)
- → 1 si $\exists$ V(i, j)

- **N[i][j] == 1**
- **Coste = O(1)**

\	1	2	3	4	5
1	*0*	*1*	*0*	*0*	*1*
2	*1*	*0*	*1*	*1*	*1*
3	*0*	*1*	*0*	*1*	*0*
4	*0*	*1*	*1*	*0*	*1*
5	*1*	*1*	*0*	*1*	*0*

Recorridos básicos de Grafos

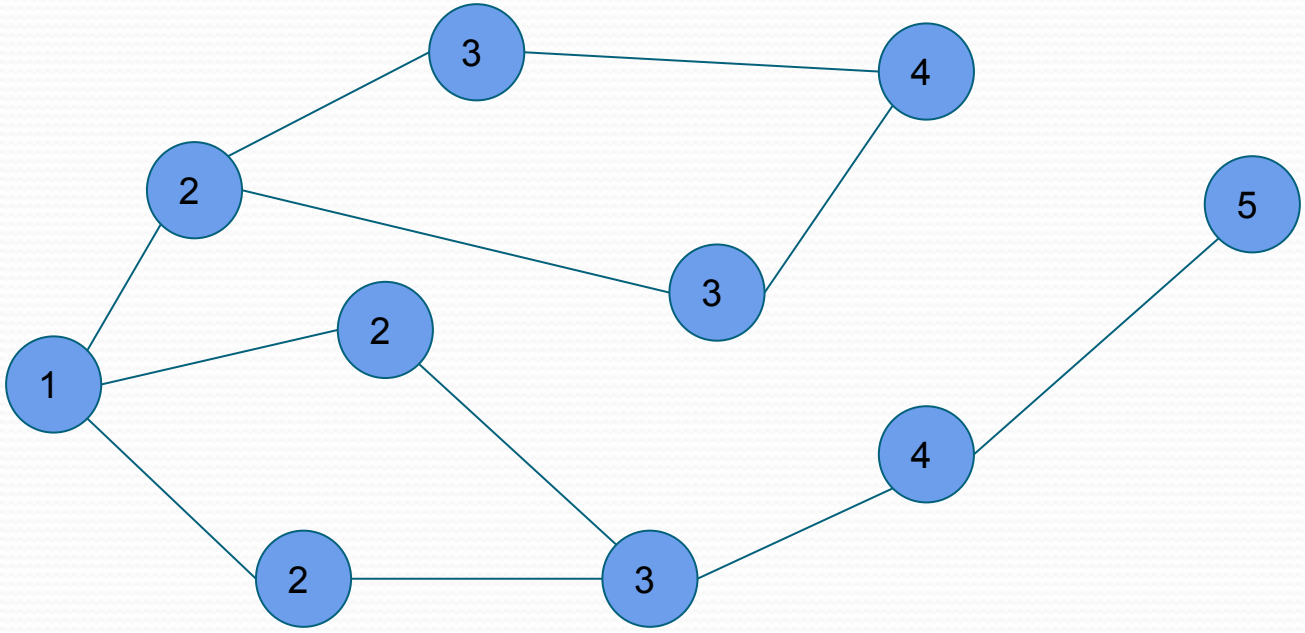
La operación de recorrer una estructura de datos consiste en visitar (procesar) cada uno de los nodos a partir de uno dado. Así, para recorrer un árbol se parte del nodo raíz y según el orden se visitan todos los nodos. De igual forma, recorrer un grafo consiste en visitar todos los vértices alcanzables a partir de uno dado.

Los recorridos de grafos se pueden realizar por:

- ***Recorrido en Anchura (BFS - Breadth First Search)***
- ***Recorrido en Profundidad (DFS - Depth First Search)***

Recorrido en Anchura(BFS)

- Comienza la búsqueda con los nodos adyacentes o los vecinos directos, adhiriéndolos en una cola de Vértices pendientes de visitar.
- Es simple y es una de las bases en las que se basa el algoritmo de Prim (para encontrar el árbol mínimo) y Dijkstra (para encontrar los caminos mínimos en un grafo dirigido ponderado).



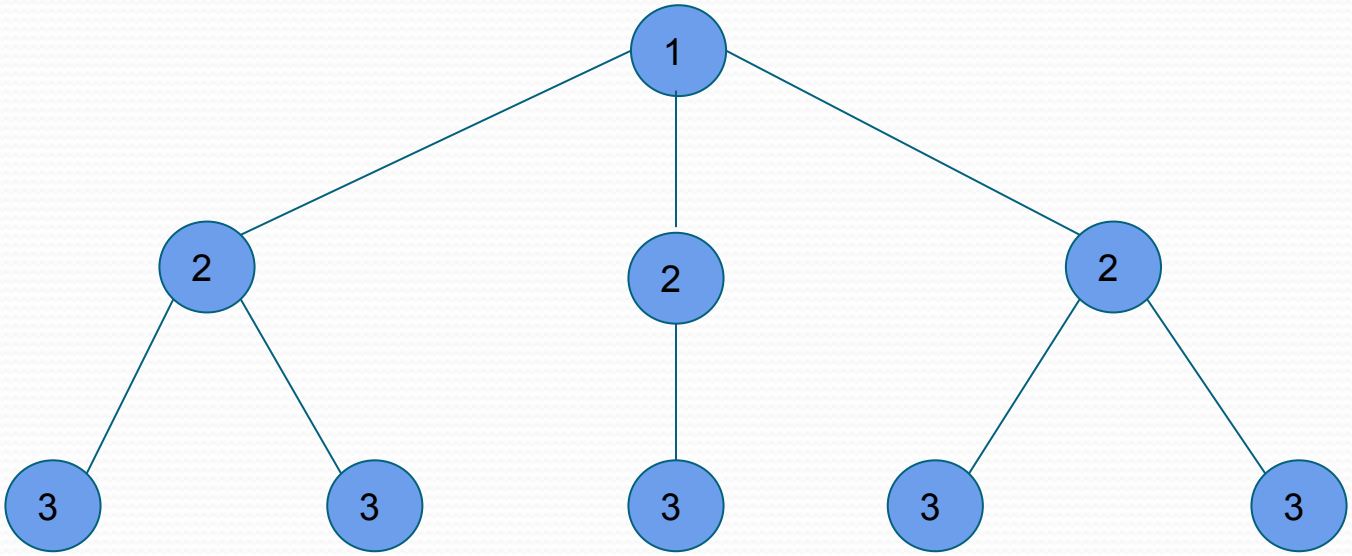
Pseudocódigo

Breadth-First Search Algorithm:

```
BFS (G, s) //Where G is the graph and s is the source node
      let Q be queue
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked
      mark s as visited.
      while ( Q is not empty)
            //Removing that vertex from queue,whose neighbour will be visited now
            v = Q.dequeue( )
            //processing all the neighbours of v
            for all neighbours w of v in Graph G
                  if w is not visited
                        Q.enqueue( w ) //Stores w in Q to further visit its neighbour
                        mark w as visited.
```

Recorrido en Profundidad (DFS)

- Admite formularse recursivamente, visitando todos los nodos hasta llegar a un callejón sin salida y, reiniciar el proceso de búsqueda.
- Es importante marcar como los nodos como visitados en el orden que se visitan, y luego continuar con la recursividad de los nodos adyacentes.



Pseudocódigo

Depth-First Search Algorithm:

```
DFS-iterative(G, s): //Where G is graph and s is source vertex
      let S be stack
      S.push(s)  //Inserting s in stack
      mark s as visited
      while(S is not empty):
            //Pop a vertex from stack to visit next
            v  =  S.top( )
            S.pop( )
            //Push all the neighbours of v in stack that are not visited
            for all neighbours w of v in Graph G:
                  if w is not visited :
                        S.push(w)
                        mark w as visited

DFS-recursive(G, s):
      mark s as visited
      for all neighbours w of s in Graph G:
            if w is not visited:
                  DFS-recursive(G, w)
```

¿Qué es un algoritmo?

Un algoritmo es un conjunto ordenado de instrucciones bien definidas, no-ambiguas y finitas que permite resolver un determinado problema computacional.
Un corolario de esta definición es que un determinado problema computacional puede ser resuelto por diversos (infinitos) algoritmos.

A partir de esto existen varios principios “importantes” para el buen diseño de algoritmos, como pueden ser:

- Un buen diseñador de algoritmos debe negarse a estar satisfecho con su primera solución.
- El buen diseñador de algoritmos debería pensar en forma de mantra, se puede hacer mejor?
- La primer solución que se nos ocurre o la solución más obvia generalmente no es la mejor.

¿Mejor algoritmo o peor algoritmo?

¿Cómo podemos poner juicio de valor a un determinado algoritmo?

- Necesitamos una escala de medición común.
- Mejor no puede ser que es más lindo, o que tiene más líneas de código o que está escrito en un lenguaje X o lenguaje Y.
- Se necesita definir con plena objetividad que significa mejor o peor, en el mundo de los algoritmos.

Una forma lógica de hacer esto podría ser quizás es medirlo por medio de las operaciones básicas que debe realizar ese algoritmo dada una entrada de tamaño ***n***, digamos, no es lo mismo 1 registro que 10, 100, 1000 o millones. Podríamos decir que este parámetro ***n*** me podría decir que tan grande es mi problema, verdad?.

Ejemplo de Ordenamiento

¿Es lo mismo ordenar un arreglo de 100 elementos que uno de 1 millón?

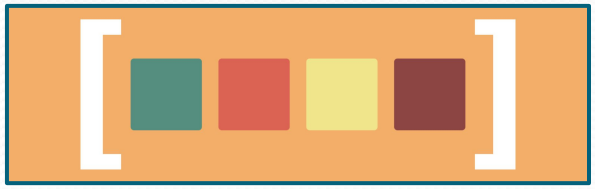
Entonces, ¿Cómo podríamos decir que un algoritmo es mejor o peor que otro?

- *"Podríamos decir que mi algoritmo es mejor si realiza menos operaciones básicas para ese mismo tamaño del problema."*

Un ejemplo de todo esto es pensar en los algoritmos de ordenamiento, ¿cuántas comparaciones tiene que hacer el algoritmo "Select Sort" dado un arreglo ***n*** de números enteros diferentes?.

★ **Select Sort:**

n-1	1
n-2	2
n-3	3
…	…
1	**n-1**



Orden de Complejidad del Select Sort

- ¿Cuánto equivale la sumatoria de todos los números naturales de 1 hasta n?
  - $\sum_{i=n}^{1} i \quad == \quad n.(n+1)/2$
  - Ejemplo de dónde es que sale esto?, sumemos los números del 1 al 100.
    - La sumatoria de 1 hasta 100 es igual a (n/2).(n+1) => (50).(101)

- Tomando esto: ¿Qué diferencias existen con el Select Sort? ¿y Cuántas comparaciones hace?
  - ¿No era que el Select Sort iba hasta n-1?
  - $\sum_{i=(n-1)}^{1} i \quad == \quad ((n-1)/2).((n-1)+1) \quad == \quad \mathbf{n.(n-1)/2}$ }

+operaciones == ***+malo***

-operaciones == ***+bueno***

¿Pero el select sort no debería hacer otras operaciones además de comparar?

Orden de Complejidad del Select Sort

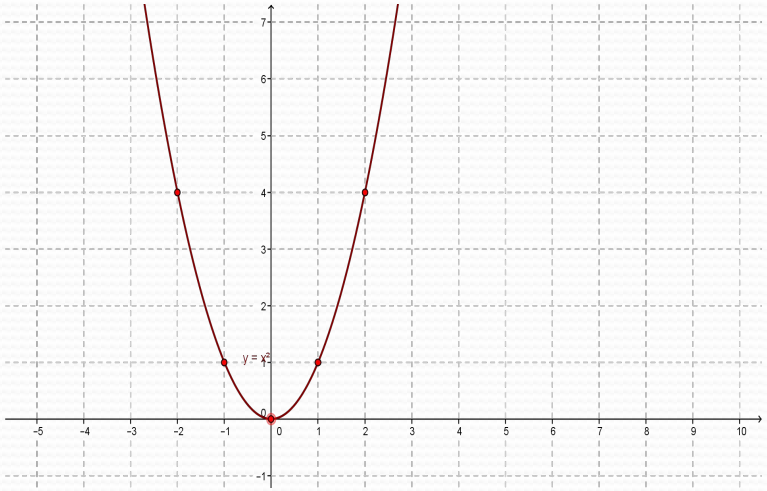
- Entonces tenemos claro que para considerar que un algoritmo es bueno, es a partir de la cantidad de operaciones básicas pero, ¿Qué considero que son las operaciones? asignaciones? comparaciones? todo esto?

- En términos prácticos para estandarizar y dejar fuera subjetividades:
  - Considerar el ***peor escenario*** *(array completamente desordenado)*
  - Enfocarse solamente en ***valores grandes de n***
  - Olvidarse de los ***valores constantes*** y de ***orden menor,***
    - ¿Qué significa esto?:

      n.(n-1)/2 =

      ((n^2)/2) - (n/2) =

      ((½)(n^2)) - (n/2) ==> **O(n^2)**



Analizando el “Grado de Complejidad”

- Los distintos tipos de Algoritmos dependen de las ***Cantidades de datos*** que van a manejar. Por ende, esta anotación no es un predictor de tiempo de ejecución, la notación nos dice que si aumentamos el tamaño del problema en una cantidad ***n,*** el tiempo que se va a demorar va a ser ***n cuadrado*** *(para el caso puntual del Select Sort)*

- El ***hardware no define al algoritmo***, no nos interesa el tiempo “real” de cómputo, sino el crecimiento asintótico que va a tomar el problema dependiendo de la cantidad de datos

- Para esto fue sumamente necesario ***definir una nomenclatura***, una simbología a través de la cual podamos hablar de la complejidad de un algoritmo (**BigO Notation**)

- Decimos entonces que un alg
otro (***más eficiente***) si su “***O***” es menor. Ya que el tiempo de ejecución, considerando el peor escenario, crece más lentamente a medida que aumenta el tamaño de la entrada.

Orden de Complejidad

La mejor técnica para diferenciar la eficiencia de los algoritmos es el estudio de los "**Órdenes de Complejidad**". El orden de complejidad se expresa, generalmente, en términos de la cantidad de datos procesados por el programa, denominado ***n***, que puede ser el tamaño dado o estimado.

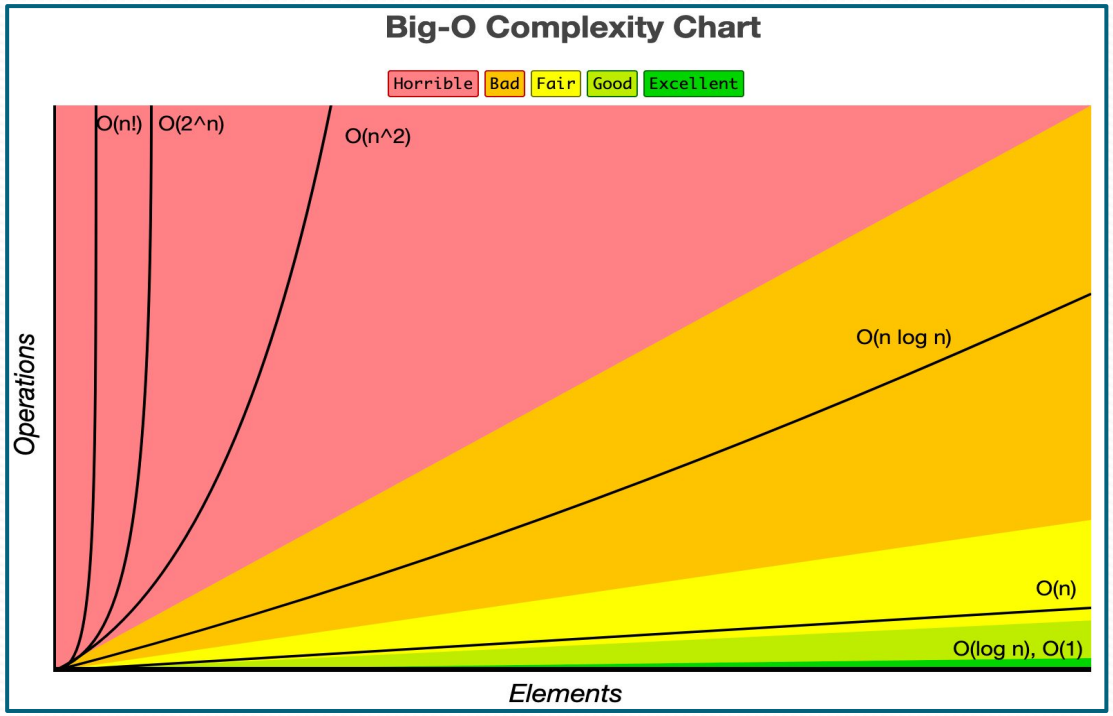
Notación asintótica: analiza el comportamiento de las funciones en el ***límite***, es decir, en su tasa de crecimiento e indica como se comporta el algoritmo para datos muy grande, captura el comportamiento de la función para valores grandes de ***n***.

- Agrupa las funciones que definen su coste en diferentes conjuntos:
  - O-Grande o Cotas Superiores
  - Ω-Grande o Cotas Inferiores
  - Θ-Grande o Cotas Entre Superior e Inferior
  - o-pequeña o Cotas estrictamente Superiores
  - ω-pequeña o Cotas estrictamente Inferiores

BigO Notation

O(1)	*-> Orden Constante*
O(log n)	*-> Orden Logarítmico*
O(n)	*-> Orden Lineal*
O(n log n)	*-> Orden casi Lineal*
O(n^2)	*-> Orden Cuadrático*
O(2^n)	*-> Orden Polinomial*
O(a^n)	*-> Orden Exponencial*
O(n!)	*-> Orden Factorial*

**(para todo a > 2)*



https://www.bigocheatsheet.com/

Ejercicios

¿Cuál es la eficiencia del Bubble Sort en términos de la Notación Big-O?

```java
public class Burbuja
{
    public static void MostrarLista(int [] lista)
    {
        for(int i = 0; i<lista.length;i++)
        {
            System.out.print(lista[i]);
            if(i<lista.length-1)
                System.out.print(",");
            else
                System.out.println("");
        }
    }

    public static void main(String[] args)
    {
        int[] numeros={6,8,2,4,1,5,3,7,9,0};
        System.out.println("ORDENACIÓN BURBUJA");

        System.out.print("Lista original:");
        MostrarLista(numeros);

        for(int i = 0; i<numeros.length;i++)
        {
            for(int j = 0; j<numeros.length-1;j++)
            {
                if(numeros[j]>numeros[j+1])
                {
                    int temp = numeros[j+1];
                    numeros[j+1] = numeros[j];
                    numeros[j] = temp;
                }
            }

            System.out.print("Iteración " + i + ":");
            MostrarLista(numeros);
        }

        System.out.print("Lista final:");
        MostrarLista(numeros);
    }
}
```

Ejercicios

```
a = b;
```

:: >> La asignación toma tiempo constante **O(1)**.

```
sum = 0;
for(i=1; i<=j; i++)
      sum += n;
```

:: >> El ciclo "***for***" es repetido ***n*** veces y, la primera y tercer línea son de coste constante. Entonces, el costo por el entero fragmento de código total es **O(n).**

```
sum = 0;
for(j=1; j<=n; j++)
      for(i=1; i<=j; i++)
            sum++;
for(k=1; k<=n; k++)
      A[k] = k-1;
```

:: >> Primero lo dividimos en 3 secciones:

- La asignación va a tomar tiempo constante, llamemoslo **C1**.
- El segundo ciclo es similar al anterior y tomaría **C2n = O(n)**.
- El ciclo del medio es un doble ciclo anidado y lo vamos a analizar de adentro hacia fuera. La expresión *sum++* es de tiempo constante, **C3**. El ciclo interno es ejecutado ***j*** veces, tiene un coste de ***C3j***. El ciclo exterior es ejecutado ***n*** veces, pero cada vez el costo del ciclo interior es diferente.

El costo total del ciclo es ***C3*** veces la suma de los números ***1*** a ***n***, es decir: ***n(n+1)/2 = O(n^2)***.

Por lo tanto ***O(C1+C2n+C3n^2) es simplemente O(n^2)***.

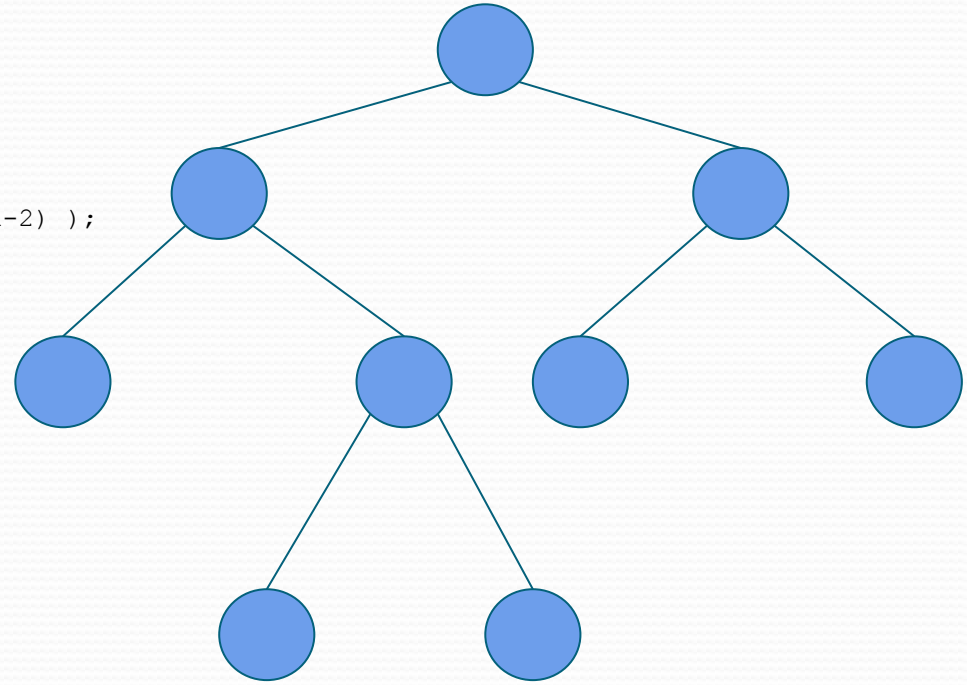
Ejercicios

¿Cuál de los dos algoritmos es mejor y porque?

- **Dado un valor n, calcular el valor enésimo término de la serie de Fibonacci:**

```
int fibonacciA(int n) {
      if (n <= 1) {
            return n;
      }
      return ( fibonacci(n-1) + fibonacci(n-2) );
}
```

```
int fibonacciB(int n) {
      f[0] = 0;
      f[1] = 1;
      for (i = 2; i <= n; ++i) {
            f[i] = f[i-1] + f[i-2];
      }
      return f[n];
}
```



- **fibonacciA :: >> O(2^n)**
- **fibonacciB :: >> O(n)**

Fin

Dudas, consultas, sugerencias?