Contenido

Introducción

Principio DRY (Don't repeat yourself): Si hay alguien que ya lo ha hecho, no repetir. Algoritmo: Conjunto finito y ordenado de instrucciones para resolver un problema dado. Debe ser:

- Correcto
- Eficaz
- Fácil de entender
- Haga uso eficiente de los

recursos Método de trabajo:

- Se diseña un algoritmo sencillo
- Se hacen simulaciones y mediciones
- Se plantea un algoritmo más eficiente
- Análisis de

mejora Análisis de

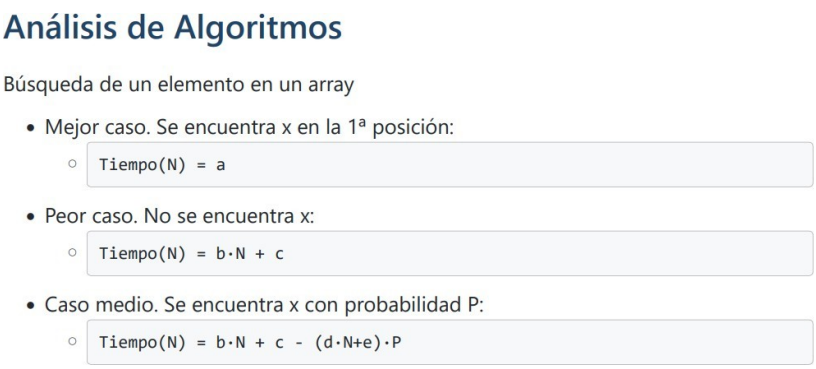
un programa:

- Memoria
- Tiempo de ejecución
- Otros recursos

Rendimiento de un algoritmo = Complejidad

Tipos de algoritmos:

Algoritmos deterministas: para los mismos datos de entrada se producen los mismos datos de salida. Algoritmos no deterministas: para los mismos datos de entrada pueden producirse diferentes de salida. Recursivos: de forma directa o indirecta se llama a sí mismo.
Iterativos: utiliza bucles.



Notaciones:

- f (n) ∈ O(g(n))

  si existen números positivos c y N tales que f(n) ≤ c g(n) ∀ n ≥ N.

- f (n) ∈ Ω(g(n))

  si existen números positivos c y N tales que f(n) ≥ c g(n) ∀ n ≥ N.

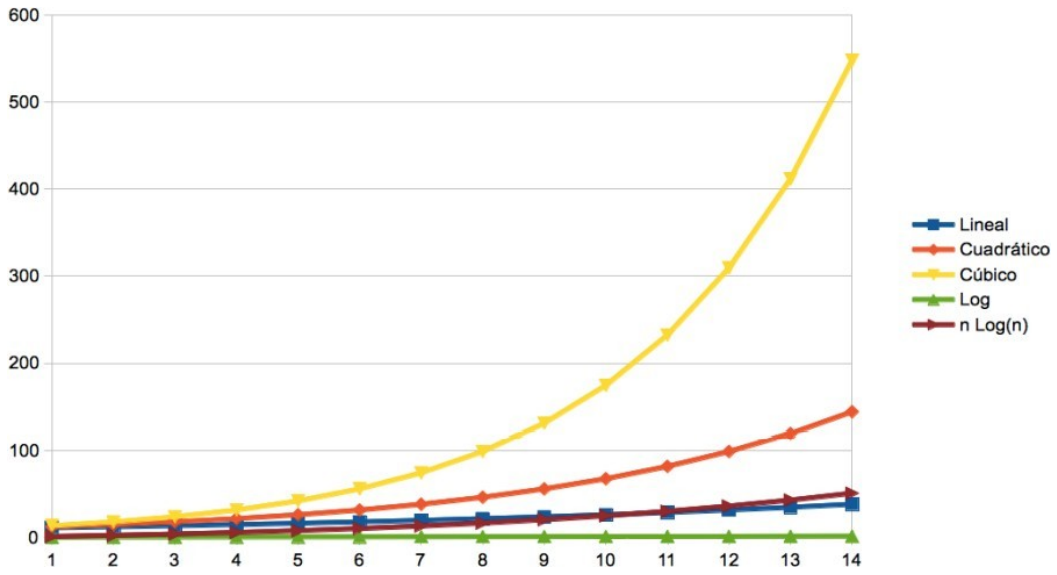
- f (n) ∈ Θ(g(n))

  si existen números positivos c1 ,c2 , y N tales que c1 g(n) ≤ f(n) ≤ c2 g(n) ∀ n ≥ N.

- f(n) ∈θ(g(n))

  Sí y sólo si: f(n) ∈O(g(n)) && f(n)∈Ω(g(n))

Rendimiento:



Cuanto más complejo es el algoritmo, más tiempo requerirá su ejecución.

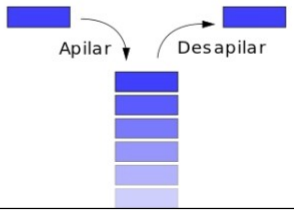
- Órdenes de complejidad habituales:

O(1);	Orden constante
O(log(n));	Orden logarítmico
O(n);	Orden lineal
O(n*log(n));	Orden cuasilineal
$O(n^2)$;	Orden cuadrático
$O(n^a)$;(a>2)	Orden polinomial
$O(a^n)$;	Orden exponencial
O(n!);	Orden factorial

TAD

Pila:

- Pila (stack):
  - LIFO (Last Input First Output)
  - Ejemplos: una pila de platos, etc,... la pila de cada proceso en un sistema operativo.
  - Operaciones básicas:
    - Apilar (push)
    - Desapilar (pop)

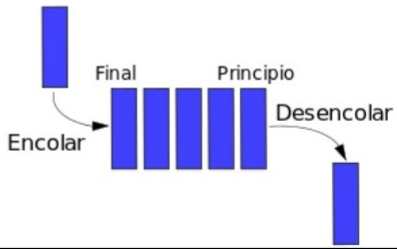


- Sólo tenemos acceso a la "parte alta" (TOS: Top Of Stack)
- Operaciones generales:
  - Crear
  - Tamaño (devuelve el tamaño de la pila)
  - Push / Pop (mete o saca dato del TOS)
  - Leer TOS (lee el valor accesible)
  - Vacío (devuelve 1 si está vacía, 0 si tiene datos)

- Hemos dicho que era secuencia finita de datos:
  - ¿Si está llena? -> Stack overflow
  - ¿Si está vacía? -> Stack underflow
- ¿Cómo podemos implementar una pila?
  - Encapsulando estructuras de datos conocidas: código
  - Dinámicamente: ...código...

Cola:

- Cola (queue):
  - FIFO (First Input First Output)
  - Ejemplos: fila del mercado, etc..., cola de trabajos pendientes en una impresora
  - Operaciones básicas:
    - Encolar (push)
    - Desencolar (pop)



- Tipos: lineal, circular, con prioridad, etc.
- Operaciones generales:
  - Crear
  - Tamaño (devuelve el tamaño de la cola)
  - Push / Pop (mete o saca dato)
  - Leer (lee el valor frontal)
  - Vacío (devuelve 1 si está vacía, 0 si tiene datos)

- ¿Cómo podemos implementar una cola?
  - Encapsulando estructuras de datos conocidas: código
  - Dinámicamente: ...código…

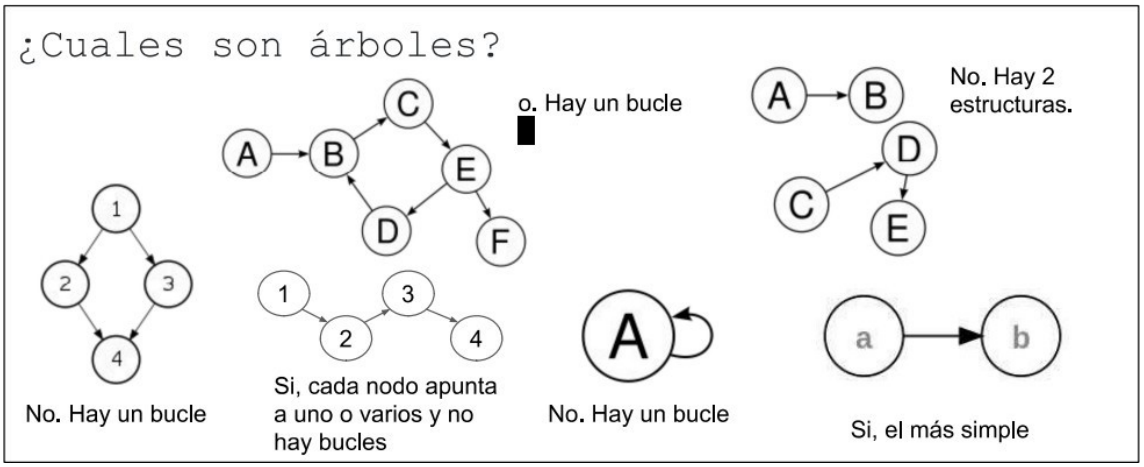
- El stack y el queue son casos particulares de las listas enlazadas (son comportamientos específicos de un TAD lineal general)

Hashtable:

https://www.youtube.com/watch?v=LluB6jU-SwY

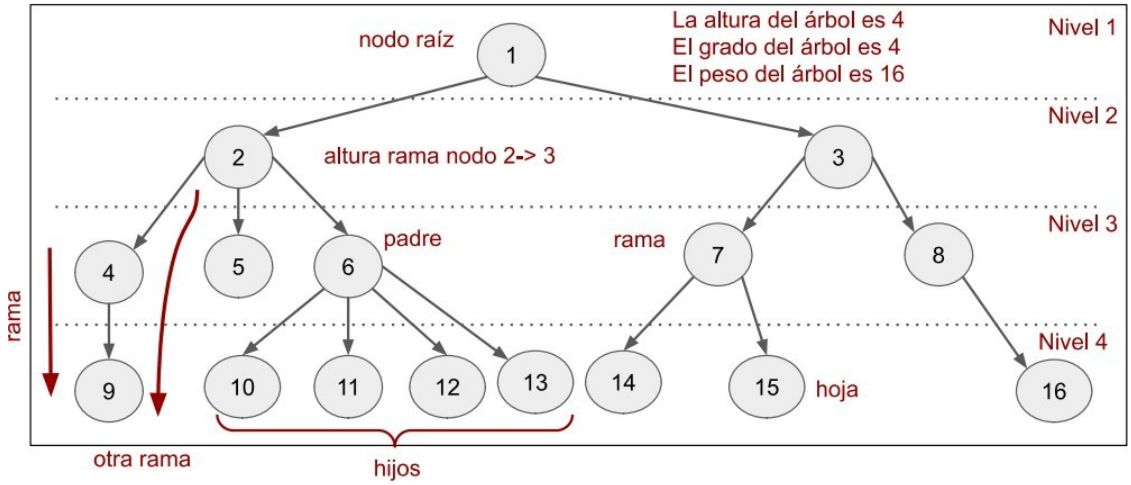
https://www.youtube.com/watch?v=9tZsDJ3JBUA&list=LL&index=44&t=419s

Árboles:



- Terminología:
  - Nodo: cada elemento que contiene un árbol
  - Padre: nodo del que parten otros nodos(más cerca de la raíz)
  - Hijo: nodos que parten de otro nodo (se alejan de la raíz)
  - Hermanos: nodos de un mismo padre
  - Raíz: El nodo superior (todos los nodos descienden de él y no tiene padre. Sólo puede haber un nodo Raíz)
  - Nodo Hoja: nodo sin hijos
  - Nodo Rama: nodo que no es raíz ni hoja y tienen al menos un hijo
  - Descendiente: Un nodo accesible por descenso repetido de padre a hijo
  - Ancestro: Un nodo accesible por ascenso repetido de hijo a padre.

- Terminología:
  - Bosque: conjunto de árboles disjuntos
  - Camino o rama: secuencia de nodos conectados con un nodo descendiente
  - Orden: nº máximo de hijos que puede tener un nodo
  - Grado: nº de hijos que tiene el elemento con más hijos dentro del árbol
  - Nivel: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos.
  - Altura: nivel del nodo con mayor nivel
  - Peso: número de nodos que tiene un árbol



- Tipos de árboles:
  - Árboles n-arios: árbol de orden n (cada nodo puede tener como máximo n hijos)
  - Árboles Binarios: árbol de orden 2 (cada nodo puede tener como máximo 2 hijos)
  - Árboles de Búsqueda: árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, y los de la otra rama serán menores.
  - Árboles Balanceados (AVL): estos árboles están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa.

- Árboles binarios:
  - Recorridos:
    - En profundidad, se diferencia según donde se procese el dato:
      - Pre order (antes de recorrer)
      - In order (entre medias)
      - Post order (después de recorrer)
    - En anchura (o amplitud)

(Saber recorrer un árbol para cada tipo de recorrido)

Montículos (heaps):

Un montículo es un árbol binario balanceado que cumple con la premisa de que: ningún padre tiene un hijo mayor (montículo de máximos) o menor (montículo de mínimos) a él.

Grafos:

Un grafo G es una tupla G=(V,A), donde V es un conjunto no vacío de vértices y A es un conjunto de aristas. Cada arista es un par (v,w), donde v,w pertenecen a V.

Grafo no dirigido: las aristas no están ordenadas.

Grafo dirigido: los pares sí están ordenados.

Longitud de un camino: número de aristas del camino = nº de nodos – 1

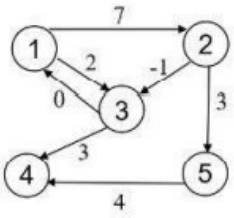
Camino simple: aquel en el que todos los vértices son distintos (excepto el primero y el último que pueden ser iguales).

Ciclo: es un camino en el cual el primer y el último vértice son iguales. Se llama ciclo simple si el camino es simple.

Grado de un vértice: número de aristas que inciden en él. Para grafos dirigidos, grado de entrada y grado de salida.

Representación mediante matrices de adyacencia.

- El conjunto de aristas es representado mediante una matriz M[nodo, nodo] de booleanos, donde M[v, w]= 1 sí y sólo si (v, w) ∈ A.
- Si el grafo está etiquetado, la matriz será de elementos de ese tipo, por ejemplo, caracteres o enteros. Tomará un valor nulo si no existe ese arco.



	1	2	3	4	5
1		7	2		
2			-1		3
3	0			3	
4					
5				4	

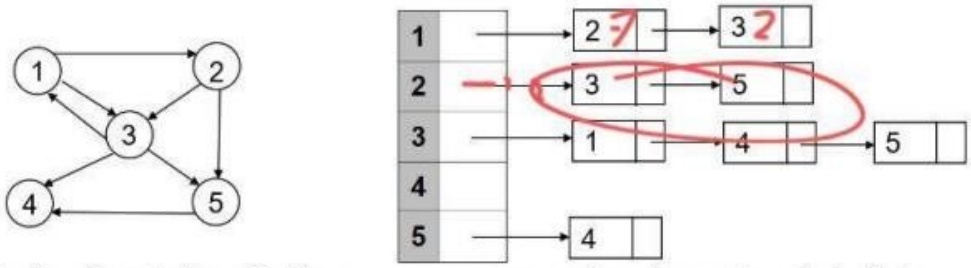
Si el grafo es no dirigido, M[v, w] = M[w, v]. La matriz es simétrica.

Problemas.

- Si el número de nodos es muy grande y hay poca conectividad (pocos arcos, en relación al máximo posible) se desperdicia memoria.
- Debemos conocer los tamaños aproximados que van a tener los grafos.

Representación mediante listas de adyacencia.

- Para cada nodo de V tendremos una lista de aristas que parten de ese nodo. Estas listas están guardadas en un array de nodos cabecera.



Grafo etiquetado: añadir un nuevo campo a los elementos de la lista.

Si el grafo es no dirigido entonces cada arista (v, w) será representada dos veces, en la lista de v y en la de w.

En grafos dirigidos.

- Calcular el grado de salida: recorrer la lista correspondiente. Grado de entrada: recorrer todas las listas.
- Otra posibilidad: tener otra lista con las aristas que llegan a un nodo dado.

La mejor representación dependerá de las características del problema.

Igual que los recorridos en árboles.

Existen dos tipos de recorridos:

- **Búsqueda primero en profundidad.** Es equivalente a un recorrido en preorden de un árbol. Se elige un nodo v de partida. Se marca como visitado y se recorren los nodos no visitados adyacentes a v, usando recursivamente la búsqueda primero en profundidad.
- **Búsqueda primero en amplitud o anchura.** Es equivalente a recorrer un árbol por niveles. Dado un nodo v, se visitan primero todos los nodos adyacentes a v, luego todos los que están a distancia 2 (y no visitados), a distancia 3, y así sucesivamente hasta recorrer todos los nodos.

El recorrido puede ser para grafos dirigidos o no dirigidos.

En los dos casos es necesario llevar una cuenta de los nodos visitados.

Algoritmo de Prim

Escoger un vértice cualquiera v. El árbol consta sólo del nodo v.

Del resto de vértices, buscar el que esté más próximo a v (con una arista (w, v) de mínimo costo). Añadir w y la arista (w, v) al árbol.

Buscar el vértice más próximo a cualquiera de estos dos, añadir ese vértice y la arista al árbol de expansión.

Así sucesivamente hasta haber añadido los n vértices.

Algoritmo de Kruskal

- Dado un grafo ponderado G=(V, A), el algoritmo parte de un grafo G'= (V, Ø). Cada nodo es una componente conexa en sí misma.
- En cada paso de ejecución se elige la arista de menor costo de A.
  - Si une dos nodos que pertenecen a distintas componentes conexas entonces se añade al árbol de expansión G'.
  - En otro caso no se coge, ya que formaría un ciclo en G'.
- Acabar cuando G' sea conexo: cuando tengamos n-1 aristas.

Definición: Dado un grafo ponderado G= (V, A) (dirigido o no) y un camino $w_1, w_2, ..., w_q$ en G, el **costo del camino** será la suma de los costos asociados a las aristas $(w_1, w_2), ..., (w_{q-1}, w_q)$.
Si el grafo es no ponderado, normalmente el costo se asocia con la longitud del camino.

Problema de los caminos más cortos por un origen:
Encontrar los caminos más cortos entre un nodo origen dado y todos los demás nodos.

Algoritmo de Dijkstra

Supongamos un grafo ponderado G (con pesos ≥ 0) y un nodo origen v.
El algoritmo trabaja con dos conjuntos:

- **S: conjunto de nodos escogidos**, para los cuales se conoce el camino de distancia mínima al origen.
- **C: conjunto de nodos candidatos**, pendientes de calcular el camino mínimo. Conocemos los caminos mínimos al origen pasando por nodos de S.

En cada paso coger del conjunto de candidatos el nodo con distancia mínima al origen. Recalcular los caminos de los demás candidatos pasando por el nodo cogido.

```
DijkstraAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    toBeChecked = all vertices;
    while toBeChecked is not empty
        v = a vertex in toBeChecked with minimal currDist(v);
        remove v from toBeChecked;
        for all vertices u adjacent to v and in toBeChecked
            if currDist(u) > currDist(v) + weight(edge(vu))
                currDist(u) = currDist(v) + weight(edge(vu));
                predecessor(u) = v;
```

Algoritmos

- Divide y vencerás

Dividir: Descomponer el problema en sub-problemas del mismo tipo. Este paso involucra descomponer el problema original en pequeños sub-problemas. Cada sub-problema debe representar una parte del problema original. Por lo general, este paso emplea un enfoque recursivo para dividir el problema hasta que no es posible crear un sub-problema más.

Vencer: Resolver los sub-problemas recursivamente. Este paso recibe un gran conjunto de sub- problemas a ser resueltos. Generalmente a este nivel, los problemas se resuelven por sí solos.

Combinar: Combinar las respuestas apropiadamente. Cuando los sub-problemas son resueltos, esta fase los combina recursivamente hasta que estos formulan la solución al problema original. Este enfoque algorítmico trabaja recursivamente y los pasos de conquista y fusión trabajan tan a la par que parece un sólo paso.

- Programación dinámica: es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.
    - Habitualmente optimización
    - Rellena resultados intermedios en tabla
    - Enfoque ascendente
    - Principio de optimalidad
    - Cómo trabajar
        - Ecuación recurrente
        - Definir tablas y cómo rellenarlas

- Algoritmos voraces: es un algoritmo que encuentra una solución globalmente óptima a un problema a base de hacer elecciones localmente óptimas. Es decir: el algoritmo siempre hace lo que “parece” mejor en cada momento, sin tener nunca que reconsiderar sus decisiones, y acaba llegando directamente a la mejor solución posible.
    - Habitualmente optimización
    - Cómo trabajar
        - Se parte de solución vacía
        - De entre los candidatos se añade uno
        - Continuar hasta solución

Backtracking: Se trata de una estrategia normalmente recursiva para resolver problemas como los de los laberintos, la colocación de piezas y similares, en los que mediante una búsqueda en profundidad se puede dar con la solución.

El nombre vuelta atrás (backtracking) viene del hecho de que en la búsqueda de la solución se va volviendo a un punto anterior para probar alternativas.