

# EDDA

## Ordenes de complejidad

Temporal: cuanto se demora un algoritmo en ejecutar su cometido

Espacial: cuanto espacio ocupará el algoritmo

O: Cota superior (el algoritmo nunca superará ese tiempo)

$\Omega$ : Entrega el mejor caso posible

$\Theta$ : Impone una cota por arriba y abajo

$C^*f(n) > \text{algoritmo}$  (existe  $N_0$ , desde ahí no superará la cota); con  $n \geq n_0$

## Estructuras Fundamentales

### LISTA

Indexada

Ligada

Doblemente ligada

OP: prepend(x), append(x), remove(x), head(), tail(), get(i)

variations: Circular

### PILA -> LIFO

OP: push(x), pop(), peek()

### COLA -> FIFO

OP: enqueue(x), dequeue(x)

variations: priorizada, deque

### TABLAS DE HASH

Diccionarios asociativos

Open addressing (tantos registros como elementos existan)

Chaining (listas ligadas por entrada, para las colisiones)

### HEAP

Arbol binario "Aplanado"

los hijos de i son  $2i+1$  y  $2i+2$

Max-Heap (todo nodo padre es mayor q sus hijos)

Min-Heap (todo nodo padre es menor que sus hijos)

## GRAFOS\*

conexion de aristas entre vertices

**vertices:** pueden tener datos y son los nodos del grado

**aristas:** son las conexiones, pueden tener costos asociados y tmbn

direccionalidad

Representaciones:

Grafica, Matriz de adyacencia, Lista de adyacencia, lista de aristas



ALGORITMOS en Grafos:

Prim y Kruskal: buscan encontrar el arbol de cobertura

Floyd-Warshal y Dijkstra: buscan la rutas de menor costo

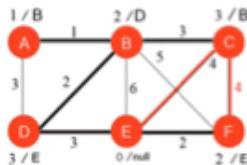
Arbol de cobertura: Conecta todos los nodos con un subconjunto de aristas,  
(grafo no dirigidos y las aristas tienen costos),

cuando es minimo quiere decir que la conexion  
entre todos los nodos debe ser minimo.

Greedy:

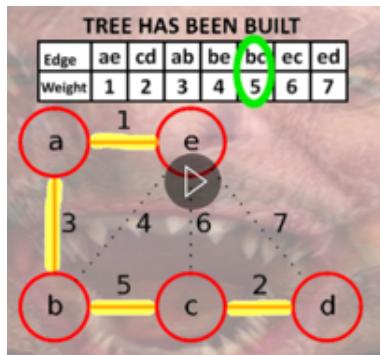
Prim: parte de un nodo aleatorio y busca la ruta de menor costo  
desde los nodos que va agregando al ST

SET: {EFDBAC}



Update all the adjacent nodes of the chosen node.

Kruskal: Toma las aristas y las ordena en costos y parte por el costo minimo, y va seleccionando las aristas de menor costo forme ciclos.  
mientras no



Rutas de costo minimo

Floyd-Warshall: Obtiene los costos de todas las mejores rutas entre cualquier par de vertices.

Trabaja sobre grafos dirigidos

Construye una matriz D<sub>k</sub> para cada vertice V<sub>k</sub>

$$D^k(i,j) = \min(D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j))$$

$$D_0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$
  

$$D_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix} \quad D_2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{bmatrix} \text{ etc}$$

Dijkstra: Calcula los costos de todas las rutas minimas de un vertice a a todos los otros vertices.

Utiliza D0

Crea un vector V del tamaño de la cantidad de vertices en el conjunto y se inicializa en infinito en todos los puntos

Se crea W que guarda los vertices visitados (initialmente vacio)

Seleccionar el vértice S que no esté en W de menor costo ( $C(S)$ ) a algún vértice en W.

Para cada vértice N vecino de S,  $C(N) = \min(C(N), C(S) + D^0(S,N))$ . Agregar S a W.

## ARBOLES

AVL: arbol binario de busqueda, que se autobalancea (no existe una diferencia mayor a 1 entre los subarboles del arbol), pueden tener complejidad  $O(\log(n))$  siempre y cuando esten balanceados.

para balancear un arbol se hacen rotaciones mediante un vertice

---

---

## Tipos de algoritmos

Divide and Conquer: divide el problema en subsproblemas mas pequeños y  
esas soluciones se organizan de manera que  
solucionan el problema grande

Greedy: Miran las alternativas inmediatas y de mejor progreso

Dynamic programming: Utilizan parte ua calculada de un subproblema para  
resolver un problema mas grande, utilizan estructuras de  
datos para llevar le recuento de las soluciones intermedias y  
no recalcularlas.

<i>Name of the algorithm</i>	<i>Average case time complexity</i>	<i>Worst case time complexity</i>	<i>Stable?</i>
Bubble sort	$\Theta(n^2)$	$O(n^2)$	Yes
Selection sort	$\Theta(n^2)$	$O(n^2)$	No
Insertion sort	$\Theta(n^2)$	$O(n^2)$	Yes
Merge sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	Yes
Quick sort	$\Theta(n \log_2 n)$	$O(n^2)$	No
Bucket sort	$\Theta(d(n+k))$	$O(n^2)$	Yes
Heap sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	No

---

---

## Normalización

### Primera Forma Normal (1NF)

Ninguna Relación (**Tabla**) tiene atributos multivariados

**Todos los atributos de la Tabla deben tener un valor único**

No hay listas

## Segunda Forma Normal (2NF)

Estar previamente en 1NF

Atributos no deben depender de nada que no sea la clave primaria de la relación

Ej:

1NF:

PRESTAMO( **num\_socio**, nombre\_socio, **cod\_libro**, **fec\_prest**, editorial, país)

2NF:

PRESTAMO1( **num\_socio**, **cod\_libro**, **fec\_prest**)

LIBRO(**cod\_libro**, editorial, país)

SOCIO(**num\_socio**, nombre\_socio)

## Tercera Forma Normal (3NF)

Estar previamente en 2NF

Atributos no primos únicamente entregan información de la clave completa y no  
de otros atributos

Ej:

2NF:

LIBRO(**cod\_libro**, editorial, país)

3NF:

LIBRO1(**cod\_libro**, editorial)

EDITORIAL(**editorial**, país)

# Modelamiento

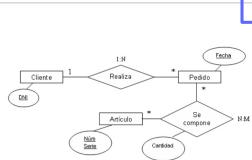
## Entidad Relación

**Entidad:** todo Objeto o concepto que comparte o interactua con el problema tienen Propiedades o atributos que almacenar  
Responde a ¿Dese saber mas (propiedades) de este Objeto?

No necesariamente material  
Animada

**Vinculo o Relación:** representan asociaciones entre los entidades que son explícitas en el contexto del Problema

Poscen:



Cardinalidad numero de entidades en la relación (1:1, 1:n, n:m)

Aridad Con cuantas entidades se conecta (numero)

Atributos propiedades

## Relacional

Clave

Super clave → Super clave Definen a la relación de manera única

Super clave candidata → Clave candidata son una superclave sin subconjuntos que sea superclave

• Clave Primaria → clave candidata seleccionada para identificar las tuplas

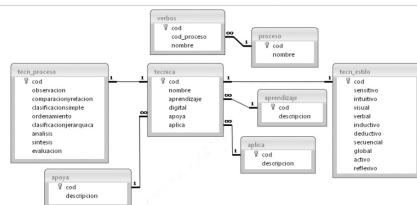
• Clave Externa → clave que posee valores de una clave primaria de otra tabla

Algoritmo de transformación



la participación completa es cuando no hay relaciones nulas

Si un vinculo posee atributos asociados al vinculo en si, crear tabla de relación con estos atributos



## Sentencias SQL

create db CREATE DATABASE dbname;

→ REMEMBER: ; ←

drop db DROP DATABASE dbname;



create table CREATE TABLE table\_name (column1 datatype, ...);

drop table DROP TABLE table\_name;

alter table ALTER TABLE table\_name ADD column\_name datatype; Variations: ADD; DROP COLUMN; ALTER COLUMN

Primary key (ID SERIAL PRIMARY KEY, ...);

foreign key (... , PersonID int REFERENCES Persons (ID));

insert INSERT INTO table\_name (column1, ...) VALUES (value1, ...);

delete DELETE FROM table\_name WHERE condition;

update UPDATE table\_name SET column\_1 = value1, ... WHERE condition;

auto increment (PersonID SERIAL PRIMARY KEY, ... );

select SELECT column1 FROM table\_name

Select distinct SELECT DISTINCT column1, ... FROM table\_name;

where SELECT column1, ... FROM table\_name WHERE condition adds: AND, OR, NOT

count SELECT COUNT(column\_name) FROM table\_name WHERE condition; Variations: AVG, SUM

join SELECT Orders.id, Customers.name, ... FROM Orders JOIN Customers ON orders.customerID=customers.id; LEFT | FULL | RIGHT JOIN

min SELECT MIN(column\_name) FROM table\_name WHERE condition; MAX(...)

Groupby SELECT Column\_name FROM table WHERE condition GROUP BY column\_name; ORDER BY

first SELECT TOP number | Percent column\_name WHERE condition;

as SELECT column\_name FROM table AS alias select column AS alias\_name FROM table

like SELECT column1 FROM table WHERE columnN LIKE pattern; wildcards: %, \_ ; NOT LIKE

in SELECT column\_name FROM table WHERE column\_name IN (value1, ...); (SELECT ..)

exists SELECT column\_name FROM table WHERE EXISTS (SELECT .... WHERE)

having SELECT column\_name FROM table WHERE condition GROUP BY column HAVING Condition

Union SELECT ... UNION SELECT

COUNT  
ORDER BY  
GROUP BY

# Sistemas Operativos

## Semaphore

```
from threading import Condition, Thread
```

### Semáforo

```
Semaphore(value) #Crea un objeto de tipo semáforo.  
acquire() #Se adquiere un semáforo  
release() #Libera un semáforo e incrementa la variable semáforo.
```

### Mutex

```
mutex = threading.Lock()  
mutex.acquire()  
mutex.release()
```

crea mutex  
lock  
libera

## Variable Condicion

```
cond = Condition()  
cond.acquire()  
cond.notify()
```

— Crea Condicion

— Cerra

— notifica a los que esperan

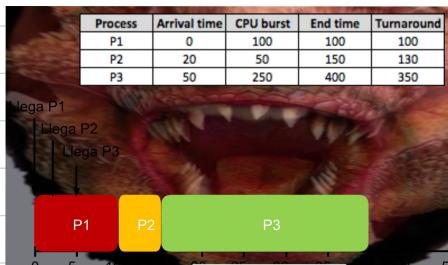
```
cond.wait()  
cond.release()
```

— espera la notificación

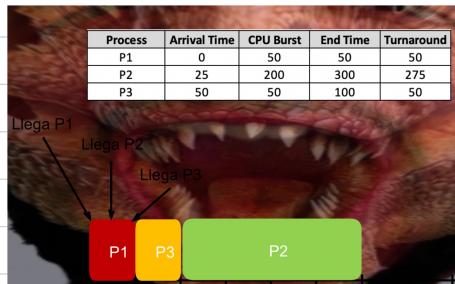
— suelte el lock

## Scheduling

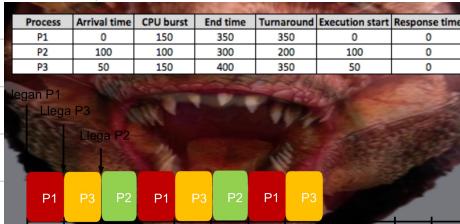
### First Come First Served



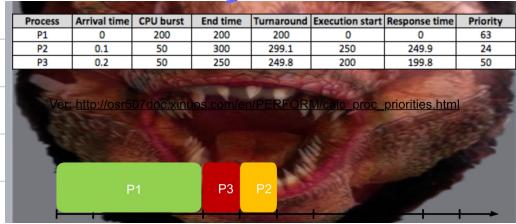
### Shortest Job first



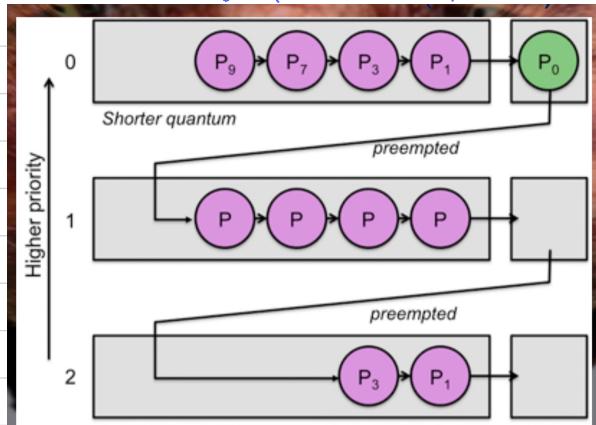
### Round-robin (q=50)



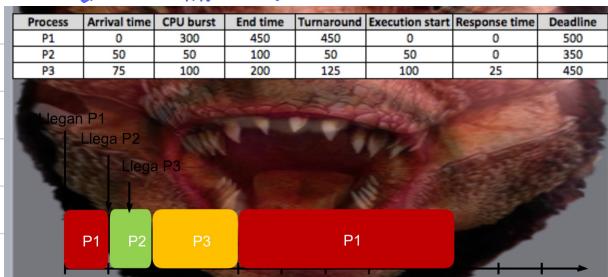
### Priority Scheduling



## Multilevel Feedback Queue (MLFQ)



## Earlier deadline first



# fabulas

## Lectores - Escritores

- dos procesos acceden al mismo bloque de memoria, con la restriccion de que solo se puede realizar una accion a la vez sobre el bloque de memoria (leer o escribir)

Por lo que si el bloque se esta leyendo o escribiendo por un proceso ningun otro proceso puede acceder a realizar alguna de estas acciones

R:

```

1 semaphore mutex = 1;           // controla el acceso al contador de procesos leidos
2 semaphore rc = 1;             // controla el acceso a la region critica
3 int reader_count;            // el numero de procesos lectores que accede la region critica
4
5 actor()
6 {
7     while (true) {
8         down(mutex);
9         reader_count++;
10        if (reader_count == 1)
11            down(rc);
12
13        up(mutex);
14        leer_rc();
15        down(mutex);
16        reader_count--;
17        if (reader_count == 0)
18            up(rc);
19        up(mutex);
20    }
21 }
22
23 Escritor()
24 {
25     while (true) {
26         down(mutex);
27         crear_datos();
28         down(rc);
29         write_rc();
30         up(mutex);
31     }
32 }
```

## Productores Consumidores

- dos procesos acceden a un buffer de tamaño fijo y lo utilizan como una cola.

Un proceso productor genera datos y los almacena en la cola, por su parte el consumidor consume los datos producidos, siempre que estos existan

R:

```

1 semaphore lleno = 0; // items produced
2 semaphore vacio = BUFFER_SIZE; // remaining space
3
4 proceso productor() {
5     while (true) {
6         item = producir();
7         down(lleno);
8         publicar();
9         up(vacio);
10        up(lleno);
11    }
12 }
13
14 proceso consumidor() {
15     while (true) {
16         down(vacio);
17         item = pop();
18         up(lleno);
19         consumir(item);
20     }
21 }
```

## Filosofos Comensales

5 filosofos se encuentran en una mesa redonda con un plato de spaghetti en medio. en medio de 2 filosofos hay tenedores. Cada filosofo se pasa la vida pensando o comiendo pero precisan de 2 tenedores para comer. Los tenedores solo se pueden utilizar por un filosofo a la vez y cuando terminan son devueltos a su lugar.

R:

Pensar (p)  
Comer (c)  
Hambriento (h)  
Dormir (d)

```

1 Semaphore Em;
2 Semaphore[] senns = new Semaphore[5]; // arreglo de semafros
3 Status = {'p', 'p', 'p', 'p', 'p'}; //que este haciendo cada filosofo en determinado momento
4 Sime(); // inicializar exclusion mutua
5
6 //Inicializar el arreglo de semafors (uno por filosofo)
7 for (int i = 0; i < 5; i++) {
8     int iems[i];
9 }
10
11 //proceso del i-ximo filosofo
12 filosofo(i) {
13     pensar();
14     tomar_tenedores(i);
15     comer();
16     dejar_tenedores(i);
17     dormir(i);
18 }
19
20 //Tomar tenedor del i-ximo filosofo
21 tomar_tenedores(i) {
22     down(1); //bloquear los recursos de rc
23     status[i] = 'h'; //indicar que este hambriento
24     test(i); //intentar tomar los recursos de rc
25     up(1); //si no los recursos de rc
26     down(senns[i]); //bajar el semaforo de este filosofo
27 }
28
29 //Intentar comer para el i-ximo filosofo
30 test(i) {
31     if (
32         status[i] == 'c' || 'c' <= i <= 4 // el filosofo de la derecha no esta comiendo
33         status[i-1] == 'c' || 'c' <= i // el filosofo de la derecha no esta comiendo
34         status[i] == 'd' //este filosofo tiene hambre
35     ) {
36         status[i] = 'c'; // como hay tenedores podemos comer
37         up(senns[i]); // bloquemos la region critica
38     }
39
40     dejar_tenedores(i) {
41         down(1);
42         status[i] = 'd';
43     }
44     test (i-1) && i > 0; //dejar tenedor izquierdo
45     test (i+1) && i < 4; //dejar tenedor derecho
46 }
```

El problema se encuentra en que el spaghetti es infinito y los filosofos no saben cuando los otros quieren comer o pensar. Pero no deben morir de hambre

# Memoria

localización dinámica de variables los procesos se ubican en distintas ubicaciones en la memoria con sus distintas ejecuciones. estas ubicaciones están parametrizadas (direcciones virtuales)

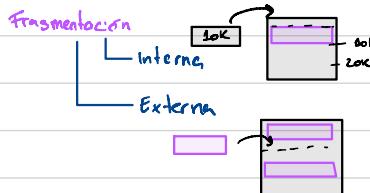
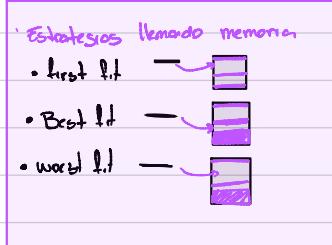
protección de memoria

Esto es realizado por MMU (Memory management unit)

Swap-in = RAM → HDD

Swap-out = HDD → RAM

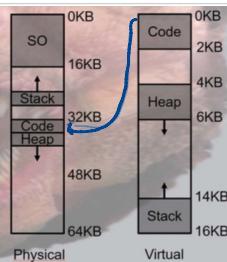
Swapping  
Cuando RAM se queda sin memoria



Segmentación: dividir la memoria de un proceso en segmentos más pequeños

Segment Table		
Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)

Si buscamos la dirección 100 (virtual):  
• Sabemos que está en el segmento Code  
• Se calcula como base + dirección:  
32768 + 100 = 32868  
• Dirección física: 32868



Segment Table		
Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)

Si buscamos la dirección 4200 (virtual):  
• Sabemos que está en el segmento Heap  
• Offset dentro de Heap: 4200 - 4096 = 104  
• Se calcula como base + dirección:  
34816 + 104 = 34920  
• Dirección física: 34920

