

29 Julio

AED II

Tema: administración de memoria

→ El ciclo de vida de un programa:

1: Edición: construcción del código en un determinado lenguaje

2: Compilación: conversión del código a un lenguaje objetivo

(tradicionalmente a un lenguaje máquina).

Se chequen la consistencia del código, tipos, sintaxis...

3: Distribución: se empaqueta el programa en un ejecutable

4: Enlace: Se limpian dependencias y otras bibliotecas.

El ligado puede ser estático (al compilar) o dinámico.
(las bibliotecas son externas).

5: Carga: El programa se carga a la memoria principal.

Se le asignan recursos.

6 Ejecución:

→ A diferencia de Java, en C/C++ el programador participa activamente en el ciclo de vida del programa (1→1)

→ El sistema operativo

• Es software

• Colocado directamente sobre el hardware.

→ De cara al programador el S.O. provee un API que abstrae los recursos y que permite construir programas de aplicación para el usuario final.

→ De manera general las funciones del S.O. son:

- Administrar procesos (CPU)

- Administrar memoria (RAM)

- Administrar archivos y discos

- Administrar I/O

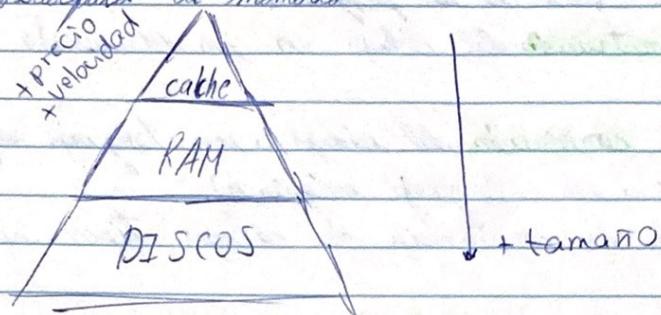
→ La memoria es un recurso valioso que debe ser administrado cuidadosamente

10.02

LOMA

Si no se administra bien, los programan "lucharán" por ocupar toda la memoria posible.

→ jerarquía de memoria



Libro: Modern Operating System - Andrew Tannenbaum - Minis.

Tema: Administración del memoria

⇒ Esquemas de administración de memoria:

- Esquema inexistente
- Espacios de direcciones
- Memoria virtual

⇒ Esquema inexistente

- Utilizado en los Mainframes de antaño (60's).
- No había ninguna abstracción de memoria.
- Los programar "tenían" la memoria física.

• Por ejemplo al ejecutar la instrucción:

MOV AX, 1000

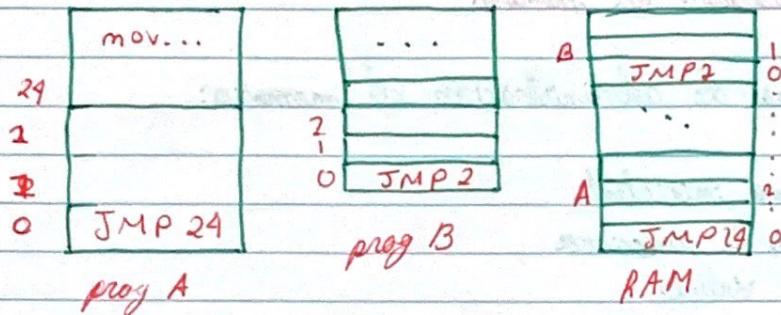
en realidad se direcciona la posición de memoria 1000.

- El "modelo" es un conjunto de direcciones de 0 al máximo.
- En un momento dado, solo estaba en memoria el S.O. y un solo programa de usuario.
- Algunas variaciones.



- Cuando es S.O. ejecuta un programa, lo copia por entero al RAM, cuando este termina, se ejecuta otro de la misma forma.
- Eventualmente soporta multiprogramación dividiendo la memoria en bloques. Cada bloque tiene una llave asignada por el S.O. Si un programa trata de leer memoria de un bloque que no le pertenece, se detiene sin ejecución.

- Sin embargo, existe un problema.



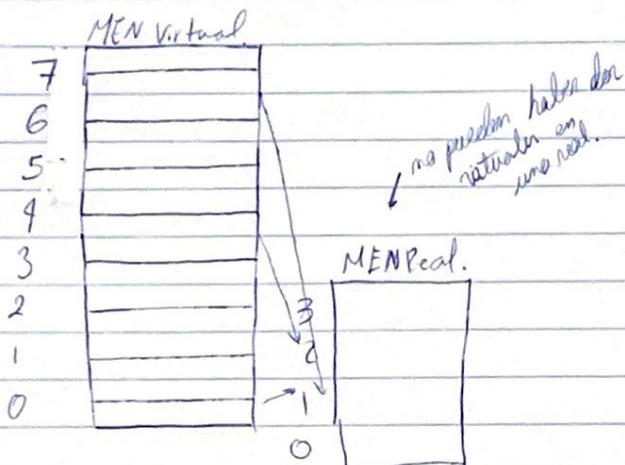
- Dado que los programadores manipulaban la memoria física, la instantaneidad 0 del programa B, causa una invocación al programa A.
- Para resolver esto, se usa **Static relocation**.
- Esto significa que en la etapa de **carga**, las direcciones a lo interno del código se cambian sumandole la dirección en la que se empezó a cargar.

→ Espacio de direcciones:

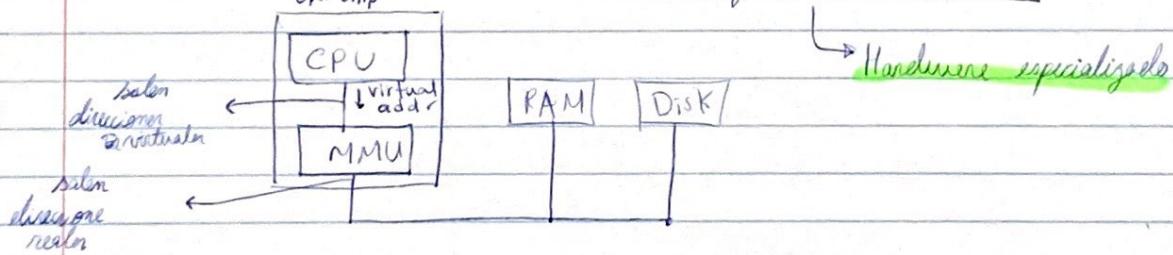
- Cada proceso tiene un conjunto fijo de direcciones de memoria que puede utilizar.
- Se implementa **dynamic relocation**
- Cada programa crece que empieza en la dirección 0
- El CPU tiene los registros especiales
 - **Base**
 - **Límite**
- Cuando el programa referencia memoria, se le suma la base y se verifica que no sea mayor que el límite.
- En esta generación de S.O. se introduce el concepto de **Swapping**
 - ~~Swapping~~ Swapping permitió tener un conjunto total de programas mayor que el total del RAM
 - Bajo uti organiza, un programa corre por cierto tiempo sin necesidad de bajar al disco y se reemplaza por otro programa en su lugar.

⇒ Memoria virtual

- "Virtual": "se ve pero no existe" vs "Transparente": "Existe pero no se ve".
- El tamaño de los programas crece más rápido que la cantidad de RAM disponible.
- Existe una memoria virtual de mayor tamaño que el RAM. El programa opera sobre dicha memoria virtual, creyendo que en su totalidad, todo en el RAM.



- El programa se divide en páginas. Cada página es un bloque de memoria contigua.
- La memoria real se divide en páginas. Las páginas reales y virtuales son del mismo tamaño.
- Cada página virtual se mapea a una a una página real. Este mapeo se hace mediante el MMU (Memory Management Unit).



- La MMU tiene la tabla de páginas. Dicha página tabla mantiene el mapeo entre páginas reales y virtuales.
- Cuando el programa referencia una página no cargada, se produce un page fault.

• En un pago suelt, se tira la página del disco y se acuña la tabla.

• Si la tabla de páginas está llena, se aplica un algoritmo de remplazo.

7 agosto.

AEDII

TEMA: Administración de memoria

- ⇒ A nivel de proceso (programa en ejecución)
- La memoria para un proceso tiene una estructura bien definida.
 - Esta estructura puede variar según el compilador.
 - Para el lenguaje C/C++, la estructura donde es:

Parámetros de línea de comandos y variables de ambiente	Código: código ejecutable
Pila	Datos inicializados: variables globales y estáticas. inicializadas por el programador.
Heap	Datos sin inicializar: variables globales no inicializadas.
Datos inicializados	HEAP y STACK: secciones específicas (más adelante las veremos).
Datos sin inicializar	
Código	

- Parámetros de línea de comando y variable de ambiente
- parámetros pasados al programa y variables definidas en el shell.

⇒ La pila

- Estructura del memory layout
 - Sigue comportamiento LIFO
 - Accesible para el programador
 - menor pulgar.
 - menor trabajo.
- Se compone de Stack Frames
- Un frame contiene:
- Storage para variables locales
 - Número de línea donde regresar.
 - Storage para parámetros.

- La pila hace transparente la memoria.
 - Cada llamada a un método crea un nuevo Stack frame.
 - Cuando el método termina, el frame se elimina por completo y la memoria se libera por completo.
- Variables locales se conocen como variables automáticas.
- Las variables automáticas son solo de tipo primitivo

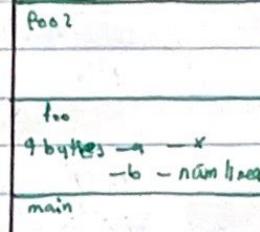
Pila.

<code>

```
int main() {
    foo(10, 20);
}

void foo(int a, int b) {
    int x = 0;
    foo2(b, 'a');
}

void foo2(int a, char c) {
    char z = 'z';
}
```



⇒ HEAP.

- Secuencia del memory layout
- No tiene estructura
- No es transparente
- Hay un API.

C { void * malloc (nBytes); : asigna n bytes en memoria y retorna un puntero
void free (void *): libera la memoria apuntada por el parametro.
realloc
calloc

C++ (delete)
(new)

• no impone límites a las variables. usadas.

• En java el Heap es manejado

→ transparente

→ Garbage Garbage collector.

(contador de referencias)

0x000 | 1 | } class a = new (clase)

{

2 ← class b = a ; → variable automatica

}

0 ←

⇒ Pointer / puntero

- Tipos de dato

int * → int * aPtr; = (int *) malloc (sizeof(int)); bytes

char *

double *

long *

float *

void *

Struct A {

 char a;

 int b;

}

Struct A *

class Persona

↳ Persona *P = new Persona();

- Todos los pointers son del mismo tamaño

→ 32 bits → 4 bytes

→ 64 bits → 8 bytes.

→ Se les pone tipo para detectar errores en tiempo de compilación

Operaciones

referenciador & → se le aplica una variable y retorna la dirección
desreferenciado * → se le aplica a un pointer y devuelve el valor apuntado.

- Un pointer inicializado tiene un valor conocido como
BAD VALUE

- Siempre inicializar puntero en null
char * = null

9 de agosto
AE DI

Resumen

Heap:

- Si hago un malloc y no hay nada en el programa retorna un -1.
- sizeof (—).

(C++)
new / delete.

Stack	Heap
<ul style="list-style-type: none">• Temporal+ Copias locales	<ul style="list-style-type: none">• lifetime: control total• Control sobre el tamaño de las variables.
<ul style="list-style-type: none">- Tiempo de vida corto- comunicación restringida.	<ul style="list-style-type: none">• más trabajo• más fallas

Tener dos pointers apuntando a la misma dirección de memoria

Sharing

Pointer

- Tipo de variable dato
 - Variable que apunta a una posición de memoria
- int a = 5
- int *aptr = & a;
- Cout << a Ptr → 0x0A
- Cout << & aptr → 0xFF
- int **aptr Ptr = & aptr;
- int *acopyPtr = aptr;
- copia la referencia. char *a; → BAD VALUE. (#define Null or preprocesador.)
- shallow copy → copia superficial.
- | | |
|------|------|
| 0x0A | s |
| 0xFF | 0x0A |
| 0xFF | 0xFF |
| 0xAA | 0x0A |
- aptr 1 → 0x0A → s
aptr 2 → 0x0A → s

Deep Copy

strcpy

int *a = (int*) malloc (sizeof (int));
*a = 666;
int *b = (int*) malloc (sizeof (int));
*b = *a;

int *z = (int*) malloc ~

≡

free (?) → ~~dangling~~
cout << *z dangling pointer.

char *c = (char*) z;

cout << c. <?>

memoria contigua.

Arreglos.

int vec[5] = {1,2,3,4,5}

funciona como
un pointer

0x01	1
0x05	2
0x09	3
0x13	4
0x19	5

int* veci = vec;

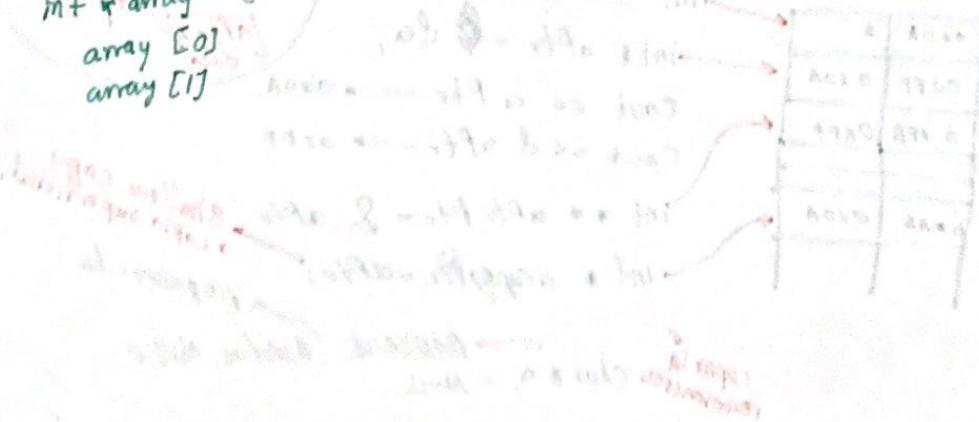
so to
MISMO. (vec[1] ← vec[1] ← &(vec + 1),
&(vec + 1) ← vec[4] ← &(vec + 4)).

on al Heap

int *array = (int*)malloc (sizeof(int) * 10)

array [0]

array [1]



int b = 500
int & a = b; tipo de dato referencia
ac 600.

- Paso parámetros por valor.

- El caller no ve los cambios hechos por el llamado a los parámetros pasados.
- El llamado recibe nuevas copias en su stack frame

```
void setvar (int x){  
    x = 8;
```

```
}
```

3

```
void test () {  
    int x = 10;  
    setvar (x);
```

```
} cout << x <> 10.
```

Paso de parámetros por referencia.

- El ~~caller~~ llamado recibe una referencia o un pointer como parámetro

- El caller verá cambios hechos por el llamado a los parámetros.

```
void setvar (int& p) { → void setvar (int& p) {  
    p = 8; } }
```

```
void test () {  
    int x = 10;  
    setvar (&x);  
    cout << x; } → 8;
```

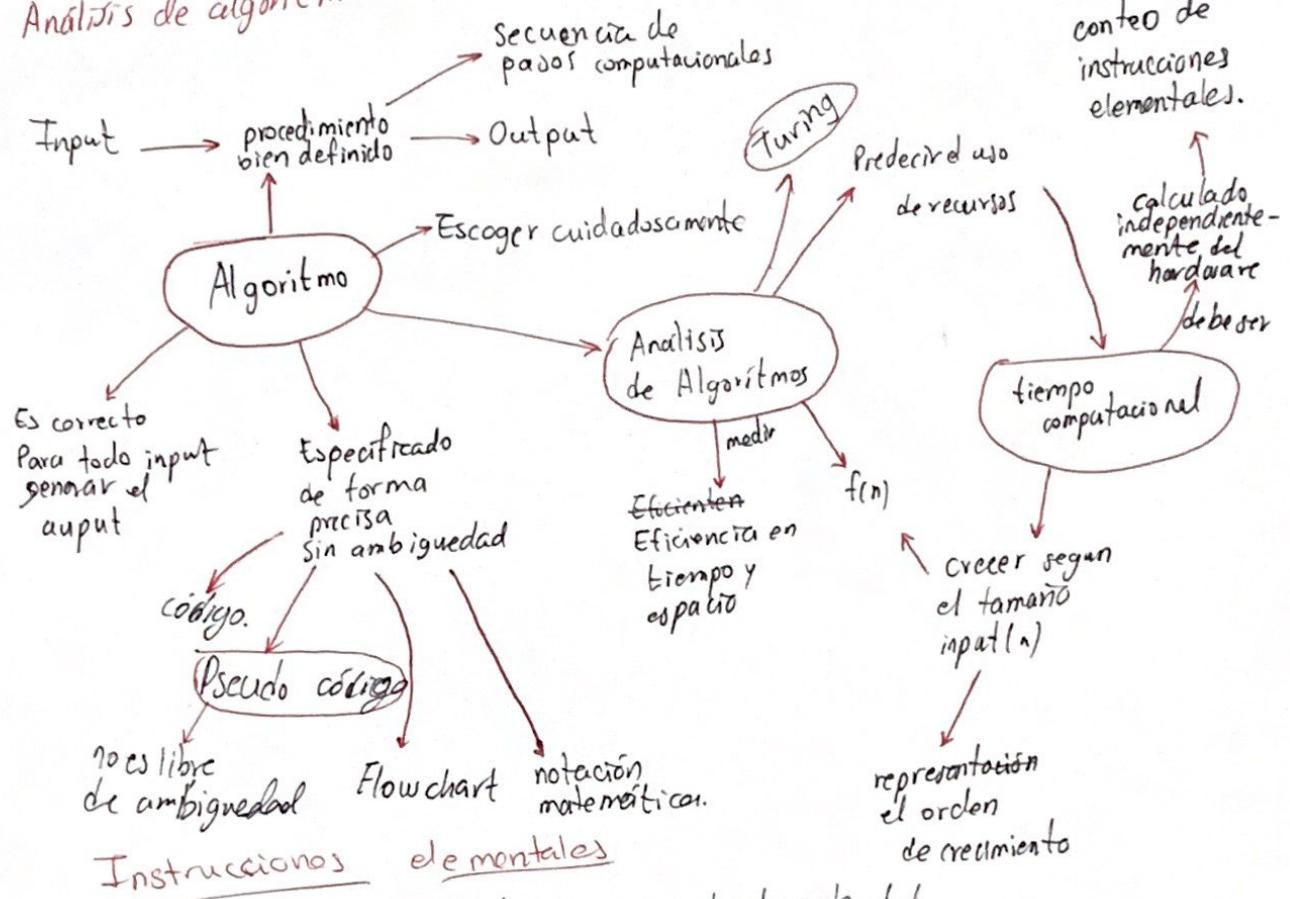
```
int test () {  
    setvar (x); } →
```

```
3  
int test () {  
    setvar (x); } →
```

Punteros a procedimientos y funciones.

```
void foo (int a, int b) {  
    =  
}  
main () {  
    foo (2,3);  
    * (void) (f (int x, int y) = & foo;  
    * f (2,3); }
```

TEMA:
Análisis de algoritmos.



- Toman el mismo tiempo independientemente del tamaño del output input.
- El tiempo de ejecución de una instrucción elemental se denota mediante T .

- Aritmética: $+, -, /, \%, \wedge$
- Bit. operaciones: $\ll, \gg, \wedge, \vee, \neg$

- Lógicos: $=, !=, ', \wedge, \vee$

- Jumps: (return, llamadas llamadas a métodos)

- Asignaciones, acceso a array

conteo de instrucciones

var $M = A[0]$ → ~~3T~~ $2T$

for (var $i = 0$; $i < A.size(); i++$) { → $IT + 2NT$

 if ($A[i] \geq M$) { → $3TN + 2TN$

$M = A[i]$, → }

$$\begin{aligned}
 & 2T \\
 & IT + 2NT \\
 & 3TN + 2TN \\
 & SNT \\
 \hline
 f(n) = & 7NT + 3T
 \end{aligned}$$

Análisis de algoritmos → continuación.

→ Análisis asintótico.

⇒ Dada una función obtenida mediante conteo de instrucciones, el análisis asintótico se enfoca en encontrar el término que determina el crecimiento de la función.

$$f(n) = 2n^3 + 3n^2 + \log_{10}(n) + 333$$

Término más importante

Despreciable

⇒ Segun el análisis asintótico, $f(n) = n^3$

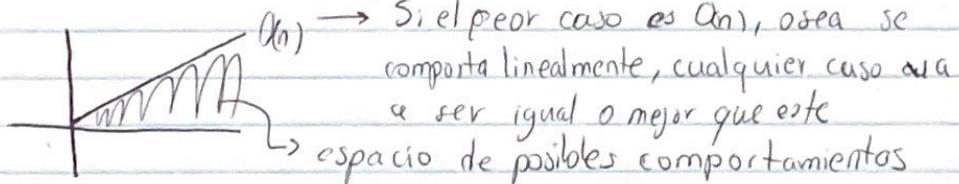
⇒ El análisis asintótico se puede hacer enfocándose en

- peor caso → Suelo ser suficiente
- Caso
- Mejor caso

→ Big O

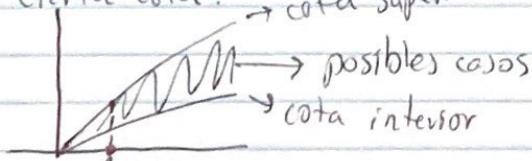
⇒ Encontrar el peor caso.

⇒ Encontrar la cota superior



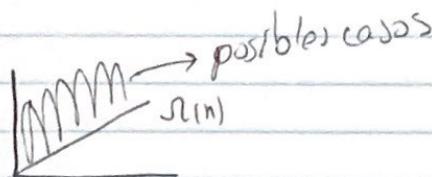
→ Big - Θ (+theta)

⇒ Encontrar la cota superior o inferior a partir de cierta cota.



→ Big (Ω) (Omega)

⇒ Es encontrar el límite inferior



→ Órdenes de crecimiento comunes.

⇒ Constante

⇒ Logarítmico $\log(n) \rightarrow \text{base } 2$

⇒ Lineal N

⇒ Lineal Logarítmico $N \log(n)$

⇒ Cuadrático N^2

⇒ Cúbico N^3

⇒ Exponencial 2^N

⇒ Factorial $N!$

Algoritmos de Búsqueda

⇒ Búsqueda secuencial

→ Es el algoritmo de búsqueda más sencillo

→ Busca un elemento en una lista o vector

→ Complejidad $O(n)$

↳ En el peor de los casos.



⇒ Binary Search

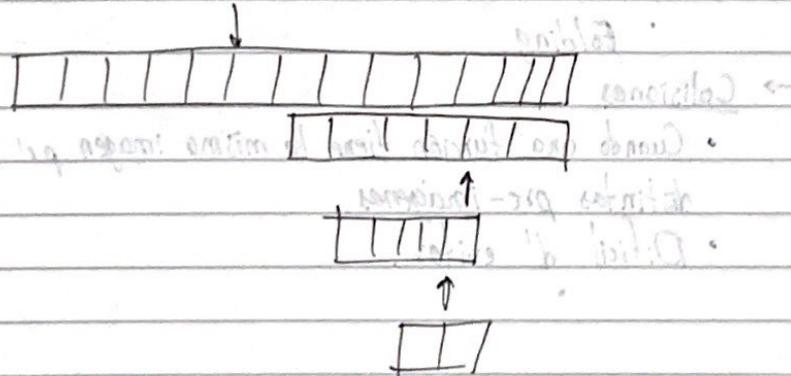
→ Busca un número en un arreglo ordenado

→ Compara el elemento buscado con el central.

→ Si el central es el elemento buscado termina

→ Si el central es menor que el buscado, se realiza la búsqueda en la mitad superior.

→ Si el central es mayor, se realiza la búsqueda en la mitad inferior



⇒ Interpolación Search

→ Modificación del binary Search

→ El código es prácticamente el mismo, a excepción del cálculo del elemento central.

→ En cada etapa, tratará de calcular donde está el elemento central.

$$\text{middle} = \text{low} + \frac{(\text{buscado} - a[\text{low}]) * (\text{high} - \text{low})}{a[\text{high}] - a[\text{low}]}$$

Algoritmos de Búsqueda

⇒ Búsqueda secuencial.

→ Es el algoritmo de búsqueda más sencillo

→ Busca un elemento en una lista o vector

→ Complejidad $O(n)$

↳ en el peor de los casos.



⇒ Binary Search

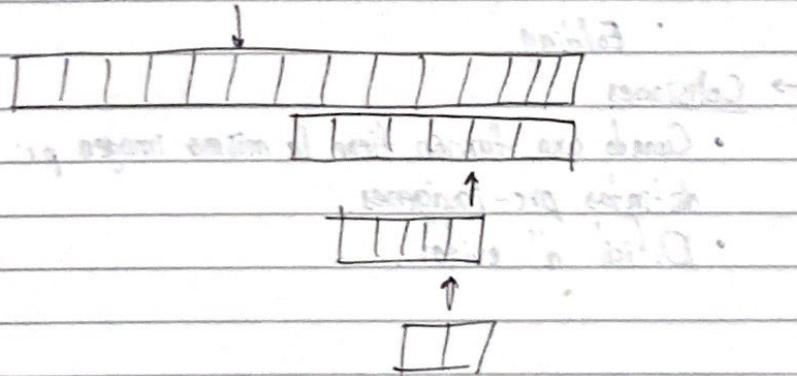
→ Busca un número en un arreglo ordenado

→ Compara el elemento buscado con el central.

→ Si el central es el elemento buscado termina

→ Si el central es menor q' el buscado, se realiza la búsqueda en la mitad superior.

→ Si el central es mayor, se realiza la búsqueda en la mitad inferior



⇒ Interpolación Search

→ Modificación d' binary Search

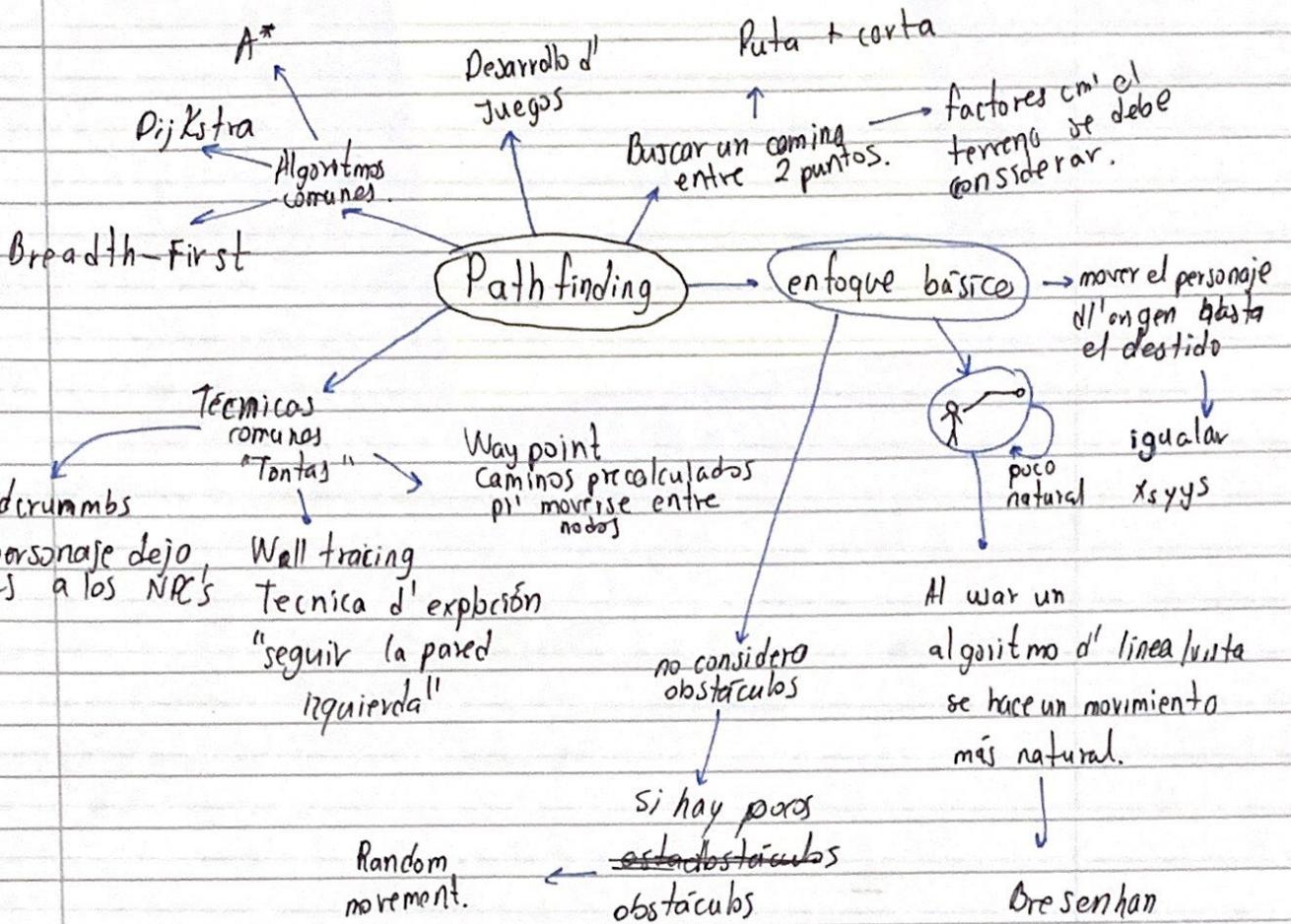
→ El código es prácticamente el mismo, a excepción d' el cálculo d' elemento central.

→ En cada etapa, tratará d' calcular donde está el elemento central.

$$\text{middle} = \text{low} + \frac{(\text{buscado} - a[\text{low}]) * (\text{high} - \text{low})}{a[\text{high}] - a[\text{low}]}$$

⇒ Hashing

- Mapea grandes datasets a pequeños datasets.
- Provee una forma d' buscar rápidamente.
- Determina una función de hash q' permite buscar y encontrar un índice p' una llave $f(\text{llave}) \rightarrow \text{índice único}$
- Se puede aplicar en encriptación.
SHA-256 / MD5...
- La forma más básica de hash es la función identidad.
- La función ideal es inyectiva \rightarrow difícil encontrarlos.
- Técnicas de Hashing
 - Restas sucesivas
 - Aritmética modular
 - Cuadrado medio
 - Truncado
 - Folding.
- Colisiones
 - Cuando una función tiene la misma imagen p' distintas pre-imágenes.
 - Difícil d' evitar.



Algoritmos genéticos.

- Durante los años 60, se estudiaron los sistemas evolucionarios de la naturaleza como una herramienta para la optimización de problemas de ingeniería.

- Formalmente desarrollados en 1970 por John Henry Holland.

- Basados en estos conceptos como:

- Selección natural
- Herencia
- Mutaciones

- Un algoritmo genético es un algoritmo de búsqueda adaptativo que aplica conceptos de la teoría de la evolución.

- Es un enfoque "inteligente" para hacer búsqueda aleatoria.

- => Explora la información histórica almacenada en la población

Genéticos cans

- Elementos por definir:

⇒ Cómo representar los individuos

- Cuantos cromosomas?

⇒ Definir una función de fitness para evaluar las soluciones

- Cuantos genes por cromosoma?

- El proceso

- Mutaciones

- Cruce

- Estructura de datos

p^{er} genes/cromosomas

[Generar población] --> Poniendo aleatoria

[Selección] --> Aplicar fitness para determinar quiénes se reproducen

[Cruce] --> Cruzar los individuos seleccionados

[Mutación]

[Insertar individuos]

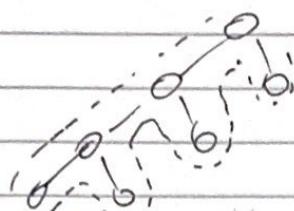
→ Formando → Si

Backtracking: "Vuelta atrás".

Es una técnica para probar secuencias de decisiones.

Hasta encontrar la profundidad o terminar sin ninguna opción.

- Las soluciones se pueden ver como si fueran un árbol de decisiones.



Arbol de decisiones

Son algoritmos naturalmente recursivos.

- Si un nodo conduce solo a errores, se retorna al nodo padre y se evalúan otros nodos.

- Uno de los problemas clásicos es el de n reinas.

→ Colocar n reinas en un tablero $n \times n$ sin que se ataquen.

→ Colocar 4 reinas.

Resolución del problema:

Se coloca una reina en la primera fila y se evalúan las demás.

Resolución:

Reina en la primera fila → Posible.

Reina en la segunda fila → Imposible.

Reina en la tercera fila → Imposible.

Resolución:

Resolución:

Diseño de Algoritmos

- Divide y conquista

- Técnica de diseño de algoritmos que divide el problema en sub problemas.
- Conquista los subproblemas resolviendo recursivamente
- Combina las soluciones d' los subproblemas p'nt obtener la solución del problema original.
- Es la técnica más básica (generalmente es veniente)

- Ejemplos de algoritmos divide y conquista:

- Merge Sort
- Binary Search

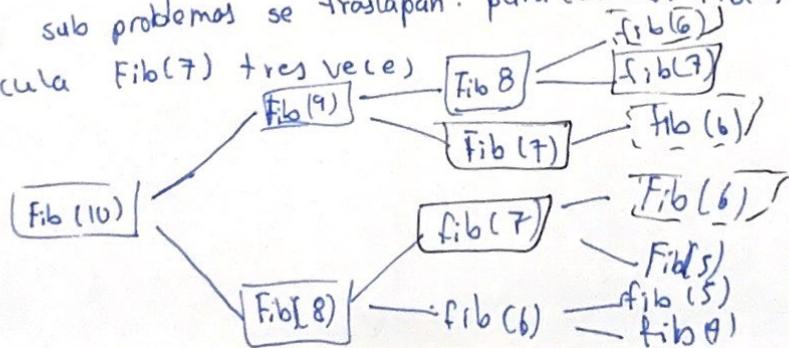
- Programación Dinámica.

- Similar a divide y conquista
- Siempre combina resultados parciales
- Los problemas que se resuelven con programación dinámica tienen las siguientes características
 - Subestructura óptima: la solución de un problema se puede obtener mediante la combinación de sus subproblemas.
 - Los subproblemas se traslapan, es decir los mismos problemas se resuelven n veces.

- Consideré fibonaci:

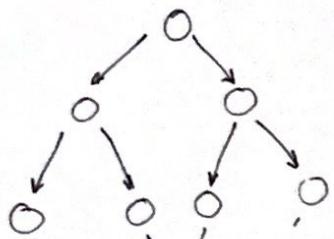
- Subestructura óptima: cada sub problema se soluciona con la combinación de otros. Hay dos casos base Fib(0) y Fib(1)

- Los sub problemas se traslapan: para calcular Fib(10) se calcula Fib(7) tres veces



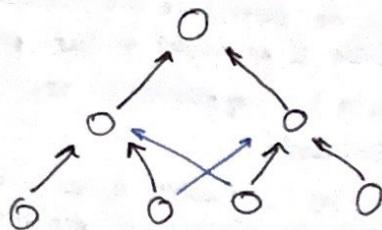
- programación dinámica calcula fib utilizando arreglos para mantener resultados y no recalcularlos

Divide y conquista.



cada problema
es diferente

Programación dinámica



2 oct.
AEAD II

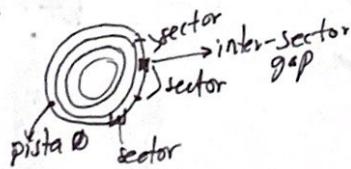
Estructuras de almacenamiento Externo.

- El almacenamiento en los computadoras se clasifica en memoria principal y memoria secundaria
- La memoria principal es volátil, "cara", "limitada" y rápida. La memoria secundaria es permanente, "barata", "ilimitada" y lenta.
- La memoria principal se llama: memoria de acceso aleatorio.
- La memoria secundaria actual se conoce como memoria de acceso directo.

↓
Toma el mismo tiempo leer cualquiera parte de la memoria.

HDD

- Hard-disk drives (discos duros)
- Son los dispositivos tradicionales para almacenamiento secundario
- Son permanentes
- Almacenan los datos en discos rígidos giratorios con superficies magnéticas
- Están selladas
- Contienen x cantidad de platos apilados.
- Tienen cabezas para leer ambas caras de cada plato
- Los platos se montan sobre un rotor.
- Los platos son de almacenamiento o fibros de vidrio.
Recubiertos con material magnético
- Cada plato se divide en pistas



- El sector es una unidad mínima de un HDD.