

Curadilla

Algoritmos y Estructuras de Datos

Parámetros	Procedimientos
TP 1	
TP 2	
<u>TP R</u>	
L_1	3/5
L_2	
L_R	

(A) Ordenar por Selección

agarra el menor elemento y lo pone en la primera posición

Proc

```
proc Selection_Sort (in/out a: array [1..n] of T)
```

```
    var i, minp : nat
```

```
    for i := 1 to n do
```

```
        minp = min_pos_from(a, i)
```

```
        Swap(a, i, minp)
```

```
    od
```

```
end proc
```

```
fun min_pos_from(a: array [1..n] of T, i: nat) ret minp: nat
```

```
    minp := i
```

```
    for j := i + 1 to n do
```

```
        if a[j] < a[minp] then
```

```
            minp := j
```

```
    fi
```

```
od
```

```
end fun
```

```
proc Swap (in/out a: array [1..n] of T, i, j: int)
```

```
    var tmp: T
```

```
    tmp := a[i]
```

```
    a[i] := a[j]
```

```
    a[j] := tmp
```

```
end proc
```

se busca el mínimo a partir de una función, retorna el índice

Intercambian 2 variables

operación repetitiva

la op principal en los algoritmos de ordenamiento va a ser la comparación (el if)

en este caso

cuando $i=1$ hago $n-1$ comparaciones (contra todos lo que estén por delante)
 $i=2$ $n-2$ comparaciones
 \vdots
 $i=n$ $n-k$ comparaciones
 \vdots
 $i=n-1$ $n-(n-1) = 1$ (solo contra el último)

sumando los operadores $1+2+\dots+n-2+n-1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

(Esto está mal, solo para ver de forma rápida que es $\Theta(n^2)$ pero
 Ver que tiene n por unidad)

orden cuadrático

ejemplos

3 1 8 2 5 // lo ya ordenado
 1 1 3 8 2 5 // intercalados
 2 1 2 3 3 5
 3 1 2 3 8 5
 4 1 2 3 5 8 *

noticia: como en cada paso tenemos una parte del arreglo ya ordenada y otra que no lo está necesariamente

Otra cosa a considerar es que puedo contar de maneras exactas la cantidad de operaciones, lo que no siempre es posible

B) Inversos

va tomando un elemento nuevo, y lo va moviendo hasta que pida este en su lugar lo hace veces solo una vez del vector

Ejemplos

9 3 1 3 5 2 7
 9 3 1 3 5 2 7
 9 9 1 3 5 2 7
 3 1 9 3 5 2 7

// ordenado

o elementos a mover
 "lo que veo"

1 3 9 3 5 2 7

proc insertionSort (in/out a: array [1..n] of T)

1 3 3 9 1 5 2 7

for i := 2 to n do

1 3 3 5 9 2 7

inInsert(a, i)

1 3 3 5 2 9 7

od

1 3 3 2 5 9 7

end proc

1 3 2 3 5 9 7

proc insert (in/out a: array [1..n] of T, in i: Nat)

1 2 3 3 5 9 7

var j: Nat

1 2 3 3 5 7 9 *

J := i

do j > 1

^ a[j] < a[j-1] \rightarrow swap(a, j, j-1)

j := j - 1

od

end proc

(*) lo voy a "invertir" siempre que el valor del lado derecho sea menor al del lado izquierdo (ver ejemplos)

en el final tiene mucho el
mismo valor es porque es largo.
Lo mismo depende del largo.

	A	B	C	D
	Comparaciones	Intercambios		
1	mn	mn	mn	mn
2	1	1	0	1
3	1	2	0	2
4	1	3	0	3
⋮				
n	1	n-1	0	n-1
Suma	n	$\frac{n(n-1)}{2}$	0	$\frac{n(n-1)}{2}$

- (A) siempre teje que hacer una permutación como misma (un elemento y su anterior)
- (B) suponiendo que el elemento que este intercambiado es el mismo haga i-1 comparaciones (el de la celda en la primera fila)

- (C) si el elemento que queremos meter ya está en su lugar no hagas sumas
- (D) analogo a b

también se dice que el algoritmo es de $O(n^2)$ porque se va a permutar en la práctica si el arreglo este bastante ordenado o mejor que es de selección

Ordenación avanzada

C) MergeSort

~~la idea principal va a ser separar el arreglo en 2 de forma sucesiva y ordenar cada parte de manera recursiva para juntar cada parte~~

la idea principal consiste en dividir el arreglo en 2 mitades ordenar cada mitad y unirlo o mezclar ambas partes, todo esto de forma recurrente a través de la Recursión

1 Proc merge-sort(a:left; a:right; T)

2 merge-sort-rec(a, 1, n)

3 End Proc

4 Pde merge-sort-rec(1:left; a:right; T, lft, rgt; mid)

5 Var mid : nat

6 If rgt > lft then

7 mid := (lft + rgt) div 2

8 merge-sort-rec(a, lft, mid)
9 merge-sort-rec(a, mid+1, rgt)

10 merge(a, lft, mid, rgt)

11 fi

12 End Proc

ahora vamos como esta implementada merge

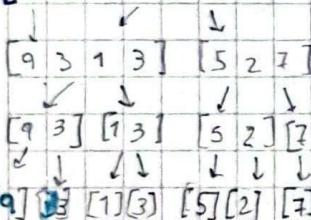
```

13 7+6 merge(in/out a:array [1..n] of T, in lft, mid, rgt :Nat)
14  Var tmp: array [1..n] of T
15  Var J,k :Nat
16  for i := lft to mid do
17      tmp[i] := a[i]
18  od
19  J := lft
20  k := mid + 1
21  for i := lft to rgt do
22      if J < mid ^ (k > rgt v tmp[J] <= a[k]) then
23          a[i] := tmp[J]
24          J := J + 1
25      else
26          a[i] := a[k]
27          k := k + 1
28      fi
29  od
30 end proc

```

Ejemplo de merge

[9 3 1 3 5 2 7]



[9 9 1 3 5 2 7]



[3 3 3 9 5 2 7]

[1 3 3 9 5 2 7]

Otra declaración lanza (22) que nos dice cada condición del for

- $J \leq mid \rightarrow$ todos los elementos en tmp (\leq no quedaron, entonces, todos lo que quedó debió estar en la parte derecha de a)
- $(k > rgt \vee tmp[j] \leq a[k])$
 - La tmp[j] $\leq a[k]$ está en la posición entre el inicio de la divisible de tmp y a, es decir, en la parte derecha de a.
 - $\rightarrow k > rgt$, esto significa cuando todos me quedan (que es la parte derecha de a) (hasta a $J \leq mid$)

* lo que mergea

algoritmo: omite los pasos de merge 1 solo elemento

Ejemplo del último merge

		[1 3 3 9 2 5 7]
①	[3 3 9]	[2 5 7]
②	[3 9]	[1 2 5 7]
③	[3 9]	[1 2 3 5 7]
④	[3 9]	[1 2 3 5 7]
⑤	[1 2 3 5 7]	[1 2 3 5 7]
⑥	[1 2 3 5 7]	[1 2 3 5 7]
⑦	[1 2 3 5 7]	[1 2 3 5 7]
⑧	[1 2 3 5 7]	[1 2 3 5 7]

num: elementos a 129 (ordenado)

num: elementos a orden (ordenado)

num: elementos ordenados

○ elementos que se comparan

Caso generalizado $\mathcal{O}(n \log(n))$

Si algoritmo merge-sort(a) llama a merge-sort-rec(a, 1, n) para todos los casos de este

merge-sort-rec(a, 1, n) llama a merge-sort-rec(a, 1, $\lfloor \frac{n+1}{2} \rfloor$) y a merge-sort-rec(a, $\lfloor \frac{n+1}{2} \rfloor + 1, n$)
Por lo que contiene las operaciones de este tipo

Sea $t(m)$ el número de comparaciones que realizan merge-sort-rec(a, lft, rgt)
(donde desde lft hasta rgt hay m celdas), o sea $m = rgt - lft + 1$

• Si $m=0$

$lft = rgt + 1$ la condición del if es falsa $t(m)=0$

• Si $m=1$

$lft = rgt$ la condición del if también es falsa $t(m)=0$

• Si $m > 1$

$lft > rgt$ la condición del if es verdadera

$t(m)$ es el número de comparaciones de las 2 llamadas recursivas
más el número de comparaciones que hace la intercalación

$$t(m) \leq t\left(\lceil \frac{m}{2} \rceil\right) + t\left(\lfloor \frac{m}{2} \rfloor\right) + m$$

Ahora sea $m = 2^k$ con $k > 1$

$$t(m) = t(2^k) \leq t\left(\lceil 2^{\frac{k}{2}} \rceil\right) + t\left(\lfloor 2^{\frac{k}{2}} \rfloor\right) + 2^k$$

$$= t(2^{k-1}) + 3t(2^{\frac{k-1}{2}}) + 2^k = 2t(2^{k-1}) + 2^k$$

$$\frac{t(2^k)}{2^k} \leq \frac{2t(2^{k-1}) + 2^k}{2^k} = \frac{t(2^{k-1})}{2^{k-1}} + 1$$

$$\boxed{\frac{t(2^k)}{2^k} \leq \frac{t(2^{k-1})}{2^{k-1}} + 1}$$

$$\frac{t(2^k)}{2^k} \leq \frac{t(2^{k-1})}{2^{k-1}} + 1 \leq \frac{t(2^{k-2})}{2^{k-2}} + 1 + 1 \leq \dots \leq \frac{t(2^0)}{2^0} + k$$

$$= t(1) + k = k$$

entonces $t(2^k) \leq 2^k k$

Luego $t(m) \leq m \cdot \log_2(m)$ para m potencia de 2

Como inferior y superior
partiendo de que $t(m) \leq t\left(\lceil \frac{m}{2} \rceil\right) + t\left(\lfloor \frac{m}{2} \rfloor\right) + m$ y llegamos a que $t(m) \leq m \log_2(m)$

Porque m potencia de 2

también vale que: $t(m) \geq t\left(\lceil \frac{m}{2} \rceil\right) + t\left(\lfloor \frac{m}{2} \rfloor\right) + \frac{m}{2}$

lo que nos permite ver $t(m) \geq \frac{m \log_2(m)}{2}$ para m potencia de 2

lo que visto que $m = 2^k \rightarrow O(n \log(n))$

Si $n \geq 0$ es potencia de 2, sea k tq $2^k \leq n < 2^{k+1}$ y por lo tanto $k \leq \log_2(n) \leq k+1$

$$t(n) \leq t(2^{k+1})$$

$$\leq 2^{k+1} + \frac{1}{2} \cdot (k+1) \cdot 2^{k+1} \quad \text{Pasar } f \text{ a creciente}$$

$$\begin{aligned} t(n) &\leq t(2^{k+1}) && \text{así q. } f \text{ es creciente} \\ &\leq 2^{k+1} (k+1) && \text{dado q. } 2^{k+1} \text{ es potencia de 2} \\ &= 2^{k+1} k + 2^{k+1} \\ &\leq 2^{k+1} k + 2^{k+1} \cdot k \\ &= 2^{k+1} \cdot 2k \\ &= 4 \cdot 2^k k \leq 4n \leq \log_2(n) \quad \text{por } 2^k \leq n \leq 2^{k+1} \end{aligned}$$

Resumen Vimos que $t(n) \leq 4 \cdot n \cdot \log_2(n)$ y podemos demostrar que $t(n) \geq \frac{1}{3} n \log_2(n)$
Por lo tanto el algoritmo de ordenación por intercambio tiene un orden $n \log_2(n)$ incluso cuando n no es potencia de 2.

① Quicksort

Idea: separar en 2 mitades el arreglo (igual que merge), una los mas que otro y principio y otra al final respecto al pivote, luego ordenar las mitades y separarlos y juntar los 2 mitades, todo esto de forma recursiva

b) dif c) el merge sort o que se usa en función de un pivote

algoritmo

```

1 Proc Quick_Sort (In/Out a:array[1..n] of T)
2   Quick_Sort_rec(a, 1, n)
3 End Proc
```

```

4 Proc_Quick_Sort_rec (In/Out a:array[1..n] of T, In lft, rgt: Nat)
5   Var pvt: Nat
6   If rgt > lft Then
7     Partition (a, lft, rgt, pvt)
8     Quick_Sort_rec (a, lft, pvt-1)
9     Quick_Sort_rec (a, pvt+1, rgt)
10  End If
11 End Proc
```

```

12 Proc_Partition (In/Out a:array[1..n] of T, In lft, rgt: Nat, Out pvt: Nat)
13   Var i, j: Nat
14   pvt := lft
15   i := lft + 1
16   j := rgt
17   Do : < j -> i / a[i] <= a[pvt] -> i := i + 1
18   Do : >= a[pvt] -> j := j - 1
19   Do : > a[pvt] ^ a[j] < a[pvt] -> Swap (a, i, j)
20   End
21   Swap (a, pvt, j)
22   pvt := j
23 End Proc
```

1) ultima 2 veces lo que hace es dejar al pivot en la posición intermedia del arreglo (ya que todos los elementos serán menores y todos los a derecha serán mayores)

Ejemplo

$\begin{bmatrix} 3 & 9 & 2 & 1 & 7 & 3 & 5 \end{bmatrix}$	$\begin{bmatrix} 3 & 9 & 2 & 1 & 7 & 3 & 5 \\ 3 & 1 & 2 & 9 & 7 & 3 & 5 \\ 2 & 1 & 3 & 9 & 7 & 3 & 5 \\ 2 & 1 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \end{bmatrix}$
$\begin{bmatrix} 3 & 9 & 2 & 1 & 7 & 3 & 5 \\ 3 & 1 & 2 & 9 & 7 & 3 & 5 \\ 2 & 1 & 3 & 9 & 7 & 3 & 5 \\ 2 & 1 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 5 & 3 & 7 & 9 \\ 1 & 2 & 3 & 9 & 3 & 7 & 9 \\ 2 & 1 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 9 & 7 & 3 & 5 \\ 1 & 2 & 3 & 3 & 7 & 9 & 5 \\ 1 & 2 & 3 & 3 & 5 & 7 & 9 \\ 1 & 2 & 3 & 3 & 5 & 7 & 9 \end{bmatrix}$

→ números intercambiados

O pivot

elementos a la izq

elementos a la der

Ejemplo partición

$\begin{bmatrix} 3 & 9 & 2 & 1 & 7 & 3 & 5 \end{bmatrix}$

lo que viene es de ordenar, uno recorre la primera mitad en orden ascendente y el otro la segunda (izquierda mayor) en el orden reverso (siguiendo hacia atrás)

$\begin{bmatrix} 3 & 9 & 2 & 1 & 7 & 3 & 5 \end{bmatrix}$

(vendo encuentra un elemento mal en alguna mitad lo guarda y borra el siguiente elemento mal en la otra mitad para intervallitos)

$\begin{bmatrix} 3 & 1 & 2 & 9 & 7 & 3 & 5 \end{bmatrix}$

el elem. mal digo un cte para mover mayor al pivot o uno q la segunda mitad menor al pivot

O pivot

análisis

blanca en la 2da mitad

blanca en la 4ta mitad

orden

E) Juntar o merge sort en ambos casos hay un desdoblamiento que le llama recursivamente con los mismos parámetros, con el mismo if y then.

Pero la diferencia radica en que en merge se hace los llamados recursivos y luego intercalar (que es de orden n), mientras que en quicksort se llaman a partición, y después se hacen los llamados recursivos lo que en el primer algoritmo particiones son q los mitades otras, mientras que en el segundo depende del pivot por esto ~~esta vez~~ las particiones quedan más equilibradas.

Sin embargo es importante mencionar que tanto merge como partición son ambas $\Theta(n \log n)$ con $O(n)$

Vemos que partición lo es.

Sea n el tamaño del arreglo en la llamada a partición ($n = \text{len}(A) - 1$). El ciclo do se ejecuta a lo sumo $n-1$ veces ya que si cada vez la brecha entre i, j se agonta en 1 o 2.

En cada iteración del ciclo se realiza un número cte de comparaciones.

$\Rightarrow O(n)$

Caso el nodo que se va a dividir al medio, se puede realizar el "mejor caso" y llegar a que siempre que el arreglo particione sobre al medio en partition entonces $O(n \log n)$

Un embargo esto no ocurre de la, por lo que podemos estar en casos

Caso medio $O(n \log n)$

Pero (caso) si el arreglo ya esta ordenado o invertido $\Rightarrow O(n^2)$
mejor caso. cuando se parte el punto al medio (el pivot es el elem del medio) $\Rightarrow O(n \log n)$

Recurrimos, divide y vencerás, jerarquía de funciones

Surgan cuando tenemos una solución para casos sencillos de un problema, y para los casos mas complejos se divide en subproblemas, donde

- cada subproblem es de la misma naturaleza que el original
- el tamaño del subproblem es una fracción del problema original
- se resuelven los subproblems aplicando el mismo algoritmo

finalmente se sobrivan esas soluciones para obtener una solución del original

El caso general es el siguiente

fun $D_rV(x)$ ret y

if x suficientemente pequeño/simple then $y := \text{ad_hoc}(x)$

else descomponer x en x_1, x_2, \dots, x_n ad

for $i := 1$ to n do $y_i := D_rV(x_i)$ ad

Combinar y_1, y_2, \dots, y_n para obtener solución y de x

fi

end fun

normalmente los x_i son una fracción de x , es decir $|x_i| = \frac{|x|}{b}$
para algún b fijo con $b > 1$

los algoritmos mergesort y quicksort son ambos casos de divide y vencerás y para ambos

- "x simple": fragmentos de longitud 0 ó 1

- "descomponer": partir al medio ($b=2$) para merge y dejar mayores de menor ($b=2$) para quick

- $a=2$

- "combinar": intercalar para merge y juxtaponer para quick

○ a : número de llamadas recursivas a D_rV

○ b : relación entre el tamaño de x y el de $x_i \rightarrow |x_i| = \frac{|x|}{b}$

○ k : orden de un operador (descomponer y combinar) $\geq k$

ahora, si queremos ^{contar} el costo computacional (nro de operaciones) $t(n)$ de la función D_2V , se obtiene

$$\rightarrow t(n) = \begin{cases} c & \text{si la entrada } n \text{ es pequeña o simple} \\ a \cdot t\left(\frac{n}{b}\right) + g(n) & \text{caso contrario} \end{cases}$$

dpt recursivo.
divide y
vencera!

Si suponemos que el costo de resolver el caso simple ad-hoc es el cte c y $g(n)$ es el costo computacional de las operaciones de descomposición y combinación.

Esta definición de $t(n)$ se conoce como relación de recurrencia, existen múltiples tipos de relaciones de recurrencia, pero ésta se conoce como recursiva.

luego si $t(n) = \begin{cases} c & \text{n entrada simple} \\ a t\left(\frac{n}{b}\right) + g(n) & \text{caso contrario} \end{cases}$ entonces por teoría, simple que

$t(n)$ es no decreciente y $g(n)$ es de orden n^k (g tiene un cte)

(confundir caso de polinomio)

luego $\mathcal{O}(t(n)) = \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$

demonstración:

resulta más sencillo calcular $t(n)$ cuando n es potencia de b , por lo que lo separaremos en casos

$$n \text{ potencia de } b \Rightarrow n = b^m$$

$$\text{si } g(n) \text{ es de } O(n^k) \Rightarrow g(n) \leq d n^k \text{ cuando } n \rightarrow \infty$$

$$t(n) = t(b^m) = a t\left(\frac{b^m}{b}\right) + g(b^m)$$

$$\leq a t(b^{m-1}) + d(b^m)^k = a t(b^{m-1}) + d(b^k)^m$$

$$\leq a [a t(b^{m-2}) + d(b^k)^{m-1}] + d(b^k)^m$$

$$\leq a^2 t(b^{m-2}) + a d(b^k)^{m-1} + d(b^k)^m$$

$$\leq a^3 t(b^{m-3}) + a^2 d(b^k)^{m-2} + a d(b^k)^{m-1} + d(b^k)^m$$

$$\leq \dots \leq a^m t(1) + a^{m-1} d(b^k)^{m-1} + \dots + a d(b^k)^{m-1} + d(b^k)^m$$

$$= a^n c + d(b^k)^n \left[\left(\frac{a}{b^k}\right)^{n-1} + \dots + \frac{a}{b^k} + 1 \right]$$

$$= a^n c + d(b^m)^n \left[r^{m-1} + \dots + r + 1 \right] \quad \text{con } r = \frac{a}{b^k}$$

$$t(n) \leq a^{\log_b(n)} c + d n^k \left[r^{m-1} + \dots + r + 1 \right]$$

$$= n^{\log_b(a)} c + d n^k \left[r^{m-1} + \dots + r + 1 \right]$$

y recordar $n = b^m \Rightarrow m = \log_b(n)$

• Si $r=1 \Rightarrow a=b^k \Leftrightarrow \log_b(a)=k$

$$t(n) \leq n^k c + d n^k \log_b n \quad \cancel{\text{por orden } n^k \log n}$$

$$\Rightarrow \mathcal{O}(n^k \log n) \quad \text{para potencia de } b$$

• $r \neq 1$

$$t(n) \leq n^{\log_b a} c + d n^k \left[\frac{r^m - 1}{r - 1} \right]$$

despejando a como de nuevo

$$r > 1 \Rightarrow a > b^k \Rightarrow \log_b(a) > k \quad \text{y ordenes}$$

$$\begin{aligned} t(n) &\leq n^{\log_b a} c + d n^k \left(\frac{r^m - 1}{r - 1} \right) \\ &\leq n^{\log_b a} c + \frac{d}{r-1} n^k r^m = n^{\log_b a} c + \frac{d}{r-1} n^k \frac{a^m}{(b^k)^m} \\ &= n^{\log_b a} c + \frac{d}{r-1} n^k \frac{a^m}{b^{km}} \approx n^{\log_b a} c + \frac{d}{r-1} n^k a^m \\ &= n^{\log_b a} c + \frac{d}{r-1} a^{\log_b n} = n^{\log_b a} c + \frac{d}{r-1} n^{\log_b a} \end{aligned}$$

$$\Rightarrow \mathcal{O}(n^{\log_b a}) \quad \text{para } n \text{ potencia de } b$$

$$r < 1 \Rightarrow a < b^k \Rightarrow \log_b(a) < k \quad \text{ademas } r-1, r^m-1 < 0$$

$$t(n) \leq n^{\log_b a} c + d n^k \left[\frac{1 - r^m}{1 - r} \right] = n^{\log_b a} c + \frac{d}{1-r} n^k (1 - r^m)$$

$$\leq n^{\log_b a} c + \frac{d}{1-r} n^k \quad \log_b(a) < k$$

$$\Rightarrow \mathcal{O}(n^k) \quad \text{para } n \text{ potencia de } b$$

Entonces para n potencia de b

$$\text{si } t(n) = \begin{cases} a + t\left(\frac{n}{b}\right) + g(n) & \text{extrae simple caso contrario} \\ & \text{con OTR} \end{cases} \quad \mathcal{O}(g(n)) = n^k$$

$$t(n) \text{ es de orden } \begin{cases} \log_b a & a > b^k \\ n^k \log n & a = b^k \\ n^k & a < b^k \end{cases}$$

Ahora, se puede comprobar que si

* $t(n)$ es no decreciente

* $t(n)$ es del orden de $h(n)$ para n potencias de b para cualquier de las 3 funciones $h(n)$ que vienen ~~señaladas~~, pues

entonces $t(n)$ es del orden $h(n)$ para n , arbitrario, no necesariamente n potencia de b por lo que el resultado anterior vale para n , arbitrario.

Ejemplo: Busqueda binaria

fun binary-search-rec($a: \text{array}[1, \dots, n] \text{ of } T, x: T, \text{left}, \text{right}$) ret $i: \text{nat}$

Var mid: nat

if left > right $\rightarrow i := 0$

left < right $\rightarrow mid := (\text{left} + \text{right}) / 2$

if $x < a[mid]$ $\rightarrow i := \text{binary-search-rec}(a, x, \text{left}, \text{mid}-1)$

$x = a[mid]$ $\rightarrow i := mid$

$x > a[mid]$ $\rightarrow i := \text{binary-search-rec}(a, x, \text{mid}+1, \text{right})$

fi

fi

end fun

Para este caso

$a: \text{conjunto de elementos finitos}$

2

$b: \text{dimensión}$

2

$h: \text{migración}$ fija

0

$$t(n) = \begin{cases} 0 & n=0 \\ t(n/2) + 1 & n > 0 \end{cases} \rightarrow a = 1, b = 2, h = 0$$

$$a = b^h$$

cond. 2

$$\mathcal{O}(n^h \log_2 n) = \mathcal{O}(\log_2 n)$$

Comparación de funciones

algunas func. crecen más rápidas cuando $n \rightarrow \infty$, por estos motivos escribiremos $f(n) \asymp g(n)$ para decir que g crece más que f o se dice escribiremos $f(n) \approx g(n)$ para decir que f y g tienen el mismo orden.

No solo interesa $\begin{cases} \cdot \text{ctas multiplicativas} \\ \cdot \text{terminos menos significativos} \end{cases}$

$$\text{Sea } f, g \in \mathbb{R}^+ \quad \lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty, \exists c_1, c_2 \text{ s.t. } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c_1$$

$$\Rightarrow \cdot \text{Si } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \asymp g(n)$$

$$\cdot \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \asymp g(n)$$

$$\cdot \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = h \text{ con } h > 0 \Rightarrow f(n) \asymp g(n)$$

Este es particularmente cierto para $\log n$

$$\text{cte } E \in \log n \approx \log n \in n^{0.01} \in n^2 \in n^{100}$$

$$\text{E.g. } 1.01^n \in 2^n \in 100^n \in n! \in n!$$

Parte 2 Tipos y estructuras de datos

conceptualmente existen 2 tipos de datos

- **Concretos** son aquellos provista por el lenguaje de programación, por los que los lenguajes nativos, o en concepto dentro del lenguaje de programación.
- **Abstractos** surgen de analizar un problema en particular, siendo los independientes del lenguaje de programación.

Tipos Concretos más complejos (tipos numerados, tuplas, arreglos y enteros)

A Tipos Numerados: serie finita de ctes pertenecientes a un mismo tipo

```
type E = enumrate  
    elem1  
    elem2  
    :  
    elemK  
end enumrate
```

El tipo E así definido tiene una constante entre los elementos.
En un programa se puede declarar una variable del tipo enumrado E y asignarle a una de las ctes.
Por e:E
e := elem5

y se permite utilizar el operador for donde el índice toma los elementos de este tipo numerado
Por ejemplo for i = elem1 to elemK do

Al definir un tipo enumerado, estos deben cumplir una relación de orden.

B Tuplas: también llamadas registros o estructuras, representan el producto cartesiano de diferentes tipos, por ejemplo podemos definir

```
type Person = tuple  
    name: String  
    age: int  
    weight: real  
end tuple
```

en este caso se trata del producto string x int x real
name, age y weight son dominios compuestos y
esta accede a un campo se puede escribir un ".": ej
Ver Anti:Person
Anti.age = 23

C Arreglos: Colección ordenada de tamaño fijo de elementos del mismo tipo, para ellos un arreglo de α debe cumplir el primer y último índice N, M respectivamente
que son el tipo de datos de los elementos. Ver $\alpha: array[N..M] of T$ si
el arreglo tiene N-M elementos.

también podemos utlizar otras rango a los elementos de un tipo enumrado
ver $\alpha: array[elem1..elemK] of T$

tablas podemos tener arreglos multidimensionales

Ver `b: array[1..N, 1..M] of String` } y para acceder a un elemento `b[2,3] = "holo"`
 Ver `c: array[1..N, 'a'..'z', 1..M] of nat`

$$c['b', 'f', 6] = 25$$

D) Punteros dando un tipo T un puntero a T, es un tipo de datos que representa el lugar en la memoria donde estan alojados algunos elementos de tipo T, por ejemplo podemos declarar un puntero a nat como

Ver `p: Puntoe to nat`

dentro punteros podemos manejar la memoria:

- Para **reservar** un bloque de memoria, donde puedo almacenar un elemento de tipo al que apunta el puntero, se utiliza la operacion `alloc(p)`
- Puedo a su vez, acceder al valor en el bloque de memoria apuntado por p mediante la operacion * ejemplo `*p := 10`
- Puedo liberar un bloque de memoria que hasta ahora reservado anteriormente con la operacion `free(p)`, luego de liberar memoria p ya no apunta al bloque de memoria ya que no se sabe que valor tiene la expresion `*p`, a su vez existe una cte para representar puntero que no apunta a nada a la que llamamos `null`, ejemplo `p := null`

casos de un puntero

① $p \rightarrow$ el valor de p es null, p no apunta a ninguna direccion de memoria

② $p \rightarrow$ la posicion de memoria ligada por p no esta reservada, por ejemplo free hace `free(p)`

③ $p \rightarrow \boxed{\quad \quad \quad}$ el valor de p sea la direccion de memoria donde se aloja la memoria que weight tuple persona

en la situacion ③ podemos acceder a los elementos de la tupla de la siguiente forma `*p.name`, `*p.age`, `*p.weight` o de manera anloga `p → name`

Por otro lado en la situacion ②, el valor de p es manejable, no debe utilizarlo, ya que a priori no se sabe, los punteros que estan en esta situacion se llaman como **punteros colgantes (dangling pointers)**

Por ultimo en la situacion ① el valor de p dejara a null, para que no tiene derechos acceso a la posicion de memoria

al manejar punteros evitaremos cosas como

$q \rightarrow p \rightarrow \boxed{\quad \quad \quad}$ y tras realizar $q := p$ $q \rightarrow \boxed{\quad \quad \quad}$

en este caso q, p señalan a la misma dir de memoria lo que significa contra a q q tambien modifica *p y viceversa, esto es lo que se conoce como aliasing

Hasta ahora siempre se almacena que no es necesario ocupar de reservar y liberar espacios de memoria, los punteros p, q son variables. Por lo que también es necesario reservar y liberar espacios. Para ello, para las operaciones, malloc y free son responsables de reservar y devolver liberas explícitamente espacio para los objetos que p y q señalan.

Ya vimos los tipos constructores, ahora vamos a algunas características de los tipos abstractos (TADs).

- Se definen específicamente constructores y operaciones
- Podemos tener múltiples implementaciones para un mismo TAD
- Jerga de un problema particular
- Este problema evaluará que operaciones serán necesarias

Para especificar un TAD, debemos:

- + Indicar su nombre
- + Especificar sus constructores o procedimientos o funciones mediante los cuales crea elementos del tipo que estás especificando
- + Especificar operaciones: todos los proc/fun que me permitan manipular los elem
- + Indicar los tipos de cada constructor y operación (el encabezado) y mediante lenguaje natural explicar qué hacen
- + Algunas operaciones pueden tener restricciones a los que llamo: precondiciones
- + Especificar la operación de destrucción que libera memoria utilizada por los elementos del tipo

A partir de esta especificación ^{para} implementarlo, sera necesario:

- Definir un nuevo tipo con el nombre del TAD
- Implementar cada constructor por separado
- Implementar cada op. (respetando los tipos tal como fueron especificados)
- Implementar la op de destrucción, liberando memoria

Además pueden surgir nuevas restricciones que dependerán de como implementamos el tipo, o si vez puede llegar necesitar op auxiliares que no están especificadas en el tipo.

Ejemplo: Lista Encabezada

Se trata de una colección de elementos del mismo tipo, de tamaño variable, todos los cuales apuntan a otra o la misma dirección.

Algunas operaciones:

- Devolver el valor de la raíz
- obtener la cantidad de elem
- insertar un elemento en la lista
- concatenar dos listas
- eliminar el primer elemento
- obtener un elem por su pos.
- agregar un elem al final
- borrar una cantidad arbitraria de elem

otra cosa es borrar un elemento arbitrario

de la lista

o crear

una lista

nueva

Spec List of T where

- Constructors

fun empty() ret l: list of T
 {
 }
 falso una lista vacia -?

proc addl(in e: T /out l:list of T)
 {agrega un elem al final de la lista-}

- destructor

proc destroy(/in/out l:list of T)
 {libera memoria de los numeros-}

- Operations

fun is_empty(l:list of T) ret b:boolean
 {-devuelve True si la lista es vacia-}

fun head(l:list of T) ret e:T
 {-devuelve el primer elem de la lista-}
 {Pre: not is_empty(l)-}

proc tail(/in/out l:list of T)
 {elimina el primer elem-}
 {Pre: not is_empty(l)-}

proc addl(in/eve L:list of T, in e:T)
 {agrega un elem al final de la lista-}

fun length(L:list of T) ret n:nat
 {-retorna la cantidad de elementos de la lista-}

proc concat(in/out L:list of T, in l0:list of T)
 {agrega al final de L, los elem de l0 en orden-}

fun index(l:list of T, n:nat) ret e:T
 {-devuelve el n-ultimo elemento de la lista-}
 {Pre: not is_empty(l)-}

proc take(in/out l:list of T, in n:nat)
 {elimina los primeros n elementos de l, solo los primeros n elementos, eliminando el resto-}

proc drop(in/out l:list of T, in n:nat)
 {-elimina los primeros n elementos de l-}

fun copy_list(l1:list of T), ret l2:list of T
 {copia todos los elementos de l1 en otra lista l2-}

Con todo esto basta para utilizar el TAD ~~list~~ (toda la especificacion), mediante los constructores empty y addl puedes crear listas, que son vacias o no, en decir que No es necesario saber como estan implementados estos tipos en el TAD

ejemplos de TAD lists

```

fun average(L:list of float) ret r:float
var largo : nat
var elem : float
var lAux : list of float
lAux := copy(L)
r := 0.0
largo := length(L)
do (not is_empty(lAux))
  elem := head(lAux)
  r := r + elem
  lAux := tail(lAux)
  od
destroy(lAux)
r := r / largo
end fun
  
```

Implementación de listas mediante Punteros

Elemento de la lista contendrá un nodo, el nodo contiene un elemento del tipo T (tipos a elegir) y en Punteros al siguiente elemento de la lista.

La lista en si será un puntero al primer nodo y el último elemento de la lista tendrá un puntero a null, así es la lista vacía.

Implement List of T where

type Node of T = tuple

elem : T
next : pointer to (Node of T)

end tuple

type List of T pointer to (Node of T)

fun Empty() ret L:int or T

L := null

end fun

① Proc addl (in e:T, inout L:int of T)

var p:=pointer to (Node of T)

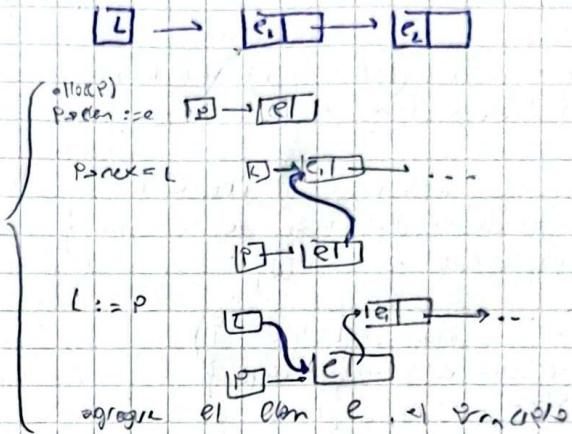
alloc(p)

p → elem := e

p → next := L

L := p

end proc



② fun is_empty(L:int of T) ret b:boolean

b := L=null

end fun

{-PRE: not is_empty(L)-}

③ fun head (L:int of T) ret e:T

e := L → elem

end fun

④ Proc tail (Input, L:int of T)

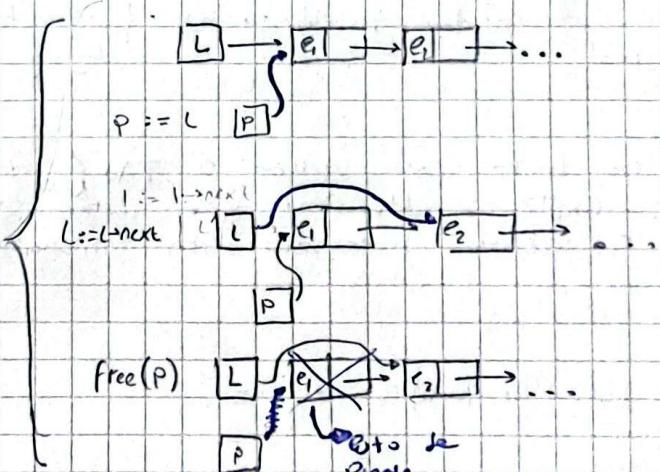
Var p: pointer to (Node of T)

p := L

L := L → next

free(p)

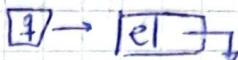
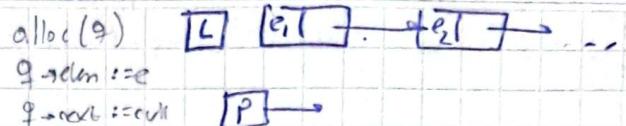
end Proc



```

• proc addr (in/out L: List of T, in e:T)
  Var p, q : pointer to (Node of T)
  alloc (q)
  q->elem := e
  q->next := null
  if (is-empty (L))
    then p := L
    do (p->next != null)
      p := p->next
    od
    p->next := q
  else L := q
  fi
end proc

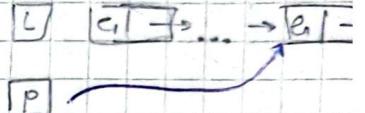
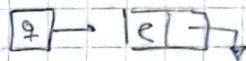
```



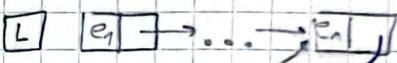
p := L



si no se cumple
a null, rov al
siguiente do (p->next != null)
p := p->next
od



p->next := q



• fun length (L: List of T) ret: nat

```

Var p := pointer to (Node of T)
n := 0
p := L
do (p != null)
  n := n + 1
  p := p->next
od
end fun

```

Otro problema que se puede resolver con un TAD o el del parámetros desbalanceados, es decir diseñar un algoritmo que tome una expresión, o arreglo de carácter y devuelva verdadero si la expresión tiene los paréntesis correctamente balanceados y falso en caso contrario.

podrá ir sumando 1 si encuentra un '(' y restar 1 si ve un ')' pero no se acuerda, lo que hay a utilizar es un contador.

El contador tendrá 4 operaciones

- inicializar
- incrementar
- comprobar si el valor es cero
- decrementar

} solo los enteros pueden ser un tipo válido, por eso los contadores

spec Counter where

constructors

```

fun init(): ret: Counter,
  {
  }
```

proc incr(in/out C: Counter)

destroy

proc destroy(in/out C: Counter)

operations

```

fun isInit('C:Counter) ret: b:Bool
```

proc decr(in/out C: Counter)

{PRE: not isInit(C)}

la idea es la misma, iniciamos un contador y recorremos el arreglo de caracteres, si llega a un paréntesis se ejecuta un bucle que abre incrementando el contador, si se encuentra uno que cierra lo decrementamos, al finaliza el recorrido si el contador es nulo, entonces estamos la cadena está balanceada

fun matching_parenes(a:array[1..N] of char) ret b:Boolean

var i:Nat

var c:Counter

b := true

init (c)

i := 1

do $i \leq N \wedge b \rightarrow$ if $a[i] = '(' \rightarrow inc(c)$

$a[i] = ')' \wedge is_init(c) \rightarrow b := false$

$a[i] = ')' \wedge \neg is_init(c) \rightarrow dec(c)$

fi

$i := i + 1$

od

$b := b \wedge is_init(c)$

destroy(c)

end fun

Implementación del TAD Contador

Implement Counter where

fun is_init(c:Counter) ret b:Boolean

$b := (c = 0)$

end fun

No falt que todos los op son O(1)

type Counter = nat

Proc init (out c:Counter)

$c := 0$

end proc

{-PRE: $\neg is_init(c) -}$

proc dec(in/out c:Counter)

$c := c - 1$

end proc

proc inc (in/out c:Counter)

$c := c + 1$

end proc

var c destroy (in/out c:Counter)

skip

end proc

Pila y colas

Decorar la diferencia entre especificación e implementación, podemos ver mitre, implementar una para una forma especificación.

① PILA

Generalización del problema de una Cadena balanceada, si abres no debi cerrar, si cierras si no tienes corchetes y llaves, entonces ya no alcanzarán un tipo contador (si es múltiplo pierde), a que la cadena $(2 + [5 \times 3]) + 1$ el resultado, sin embargo, si se trabajan los símbolos de forma independiente, estos balanceados

No alcanza, también hay que saber en qué orden cerrar los delimitadores, o sea que sentido de abrieron, mejor dicho, el que va a cerrar fue el último en abrirse

No hace falta una lista/inicio de que delimitadores están abiertos, y en qué orden,

una solución posible es la siguiente:

Recorrer la cadena de izquierda a derecha, y agregar la obligación de cerrar un parentesis (o corchete o llave) cada vez que se encuentra uno que abre, luego revisar la obligación de cerrar (corchete que tengan el mismo > que la proxima obligación sea el delimitador que esta entrando) cada vez que se llega un delimitador que cierra.

Para finalmente comprobar que este condicione con obligaciones, este varia

- El delimitador hace falta **algo**, que me permita }
 • Iniciar la recorrida }
 • Agregar una obligación }
 • Comprobar si quedan obligaciones }
 • Estimar la proxima/primer obligación }
 • Quitar una obligación }
- Esto es lo que se conoce como **PILA**

Una pila se define por las 5 operaciones

1. Colocar en vacio
2. añadir elemento (apilar)
3. Comprobar si pila vacia
4. obtener el primer elemento (si no esta vacia)
5. quitar el primer elemento (desapilar)

} Con estos 2 op ganes el conjunto
de op. basicos, por los cuales son los
Constructores.

Spec stack of T where

operations

fun is_empty(s:Stack of T) ret b:Boolean

Constructors:

fun empty_stack() ret s:Stack of T

{ - PRE: $\neg \text{is_empty}(s)$ - }

Proc Push(in/out s:Stack of T, in e:T)

fun top(s:Stack of T) ret e:T

{ - PRE: $\neg \text{is_empty}(s)$ - }

Proc Pop(in/out s:Stack of T)

Algoritmos de control de delimitadores balanceados

fun matching_delimiters(a:array[1..n] of Char) ret b:boolean

- Var i:nat

- Var p: Stack of Char

b := true

p := empty_stack()

i := 1

do i <= n ^ b → if left(a[i]) → push(p, mta(a[i]))

right(a[i]) ^ (i->empty(p)) v top(p) ≠ a[i] → b := false

right(a[i]) ^ right(p) ^ top(p) = a[i] → pop(p)

otherwise → skip

fi

i := i + 1

od

b := b ^ !i->empty(p)

aleatorio(p)

end fun

~~Otro algoritmo avisa que $\text{match}('[') = \{'$, lo mismo con '**'**' > ']'~~

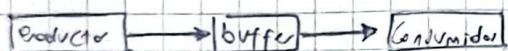
El algoritmo avisa la existencia de funciones

- match $\text{tp match}('[') = \{'$, lo mismo con '**'**' > ']'
- left que retorna verdadero si se trata de un delimitador de apertura (abre en el lado contrario)
- right opuesta a left

(B) Colas

Simplifica que ciertos datos deben transferirse de una unidad a otra, contactar con un agente que produce los datos y otro que los utiliza o consume, a este relación se le conoce como Productor-Consumidor

Para amortiguar el impacto respecto a la diferencia de velocidades, se puede introducir un buffer entre ellos, este es espacio de almacenamiento temporal que el productor produce y lo envía en el mismo orden a medida que son solicitados.



Algunos ejemplos:

- ① La uso de un buffer no debe afectar el orden en el que llegan los datos al Consumidor
- ② El propósito es que tanto Productor y Consumidor tengan funciones a su velocidad, sin la necesidad de sincronizarse
- ③ El buffer está inicialmente vacío
- ④ A medida que se agregan datos por parte del Productor, estos se van almacenando
- ⑤ Cuando se necesita enviar un dato al Consumidor, habrá que ver que el buffer no sea vacío y en caso sea enviará el primer dato que le llegó al buffer y se lo eliminará del mismo

Ej decir es **algo**, con lo que presta:

- inicializarlo como vacío
- agregar o eliminar un dato
- comprobar si es vacío
- examinar el primer dato (el más viejo)
- quitar o deselecciónar un dato

Esto es lo que se conoce como **cola**

El primer dato en llegar es el primero que se enviará, por eso a la cola se la conoce como **cola FIFO (first in - first out)**

nuevamente al TAD se le define las dos operaciones

- 1 ~~constructor~~ inicializar (in/out) en vacío
- 2 Encolar un nuevo elemento
- 3 Consultar si está vacío.
- 4 Examinar el primer elemento (si no está vacío)
- 5 Decolar el primer elemento (si no está vacío)

} nuevamente estas 2 operaciones me
governan todos los demás, por eso son
los constructores

Spec Queue of T where

Constructors

```
fun empty_queue() ret q: Queue of T
proc enqueue (in/out q: Queue of T, in e:T)
```

Operations

```
. fun is_empty_queue (q: Queue of T) ret b:Bool
```

{-PRE: \neg is_empty_queue(q) -?}

```
fun first (q : Queue of T) ret e:T
```

{-PRE: \neg is_empty_queue(q) -?}

```
proc dequeue (in/out q: Queue of T)
```

algoritmo de transferencia de datos a través de un buffer

proc Buffer ()

Var d: data

Var q: Queue of data

empty_queue (q)

do (not-finish ())

if there_is_product () \rightarrow d := get_product()

enqueue (q, d)

there_is demand () \rightarrow is_empty_queue (q) \rightarrow d := first (q)

consume (d)

dequeue (q)

fi

od

end proc

Mientras, se activa la ejecución de múltiples funciones

- finish , que devuelve verdadero solo cuando el servicio de Productor/Consumidor estén finalizados

- there_is_product }, que hace referencia a la oferta y demanda respectivamente

- there_is_demand },

- get_product , para obtener el ^{dato} producto desde el Productor

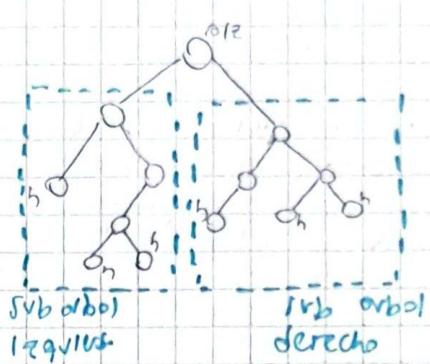
- consume , para enviar el dato hacia el Consumidor

árboles binarios de búsqueda

se trata de una estructura de datos recursiva, tenemos nodos que almacenan un elemento y das subestructuras que a su vez podrán contener otros nodos o el vacío

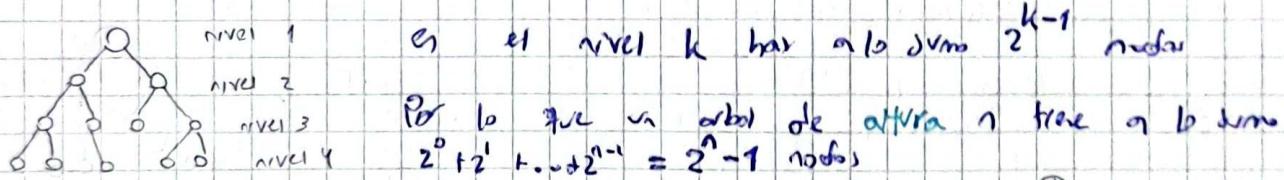
Componentes:

árbol vacío
agregar un elemento



algunas definiciones y terminología

- Un solitario nodo es un árbol en vacío
- Un árbol tiene raíz, subárboles izquierdo y derecho
- A los subárboles también se les conoce como hijos
- Todo nodo es el padre de sus hijos
- Una hoja o follón nodo es un nodo sin hijos vacíos
- Se define el nivel como la "distancia" al nodo raíz + 1



(*) operación

definiendo la altura como el máximo nivel de un nodo

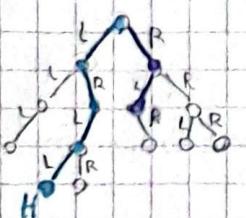
También definimos un camino como una secuencia de nodos

- (*) Un árbol se dice balanceado si cada nodo tiene 2 o ningún hijo, en estos árboles la altura $c_i \log_2(n)$ siendo n la cantidad de nodos

Camino y recorridos

Podemos recorrer un árbol indicando el camino desde la raíz

Ejemplos



el camino LRLR nos lleva hasta la hoja K

el camino RLL no es un camino válido

Especificación del TAD: árbol binario

tree Direction = enumrate
 left
 right
 end enumrate

spec tree of T where

Construcciones

fun Empty-tree() ret t:Tree of T

fun node(tl:Tree of T, tr:Tree of T, e:T) Ret t:Tree of T

Operaciones

fun is_empty-tree(t:Tree of T) ret b:Bool

{-devería True si el árbol es vacío -}

fun root(t:Tree of T) ret e:T

{-debería el elemento en la raíz de t -}

fun left(t:Tree of T) ret tl:Tree of T

{-debería el subárbol izquierdo de t -}

{-PRE: not is_empty-tree(t) -}

fun right(t:Tree of T) ret tr:Tree of T

{-debería el subárbol derecho de t -}

{-PRE: not is_empty-tree(t) -}

fun height(t:Tree of T) ret n:Nat

{-debería la distancia entre la raíz y la hoja más alta -}

fun is_Path(t:Tree of T, p:Path) ret b:Bool

{-devería True si p es un camino válido en t -}

fun subtree_at(t:Tree of T, p:Path) ret ts:Tree of T

~~{-debería el subárbol que se obtiene al recorrer p en t -}~~

fun element_at(t:Tree of T, p:Path) ret e:T

{-debería el elemento que se encuentra al recorrer p en t -}

{-PRE: is_Path(t, p) -}

La manera usual de implementar árboles binarios, es a través de punteros, de forma similar a lo que se hizo con listas enlazadas. Definimos un tipo Node como un tipo que un elemento, y dos punteros a sí mismo.

Implement Tree of T where

tree Node = tuple

left : Pointer to (Node of T)

right : Pointer to (Node of T)

value : T

end tuple

tree Tree of T = Pointer to (Node of T)

fun Empty-tree() ret t:Tree of T

t := null

end fun

fun node(tl:Tree of T, e:T, tr:Tree of T) ret t:Tree of T

alloc(t)

t → value := e

t → left := tl

t → right := tr

end fun

árboles binarios de búsqueda

Son árboles binarios donde la información está ordenada de forma particular

Todos los elementos en el árbol deben ser menor o igual que su padre si este es el nodo izquierdo, y todos hijos derechos deben ser mayores o iguales que su padre.

Esto quiere decir que los valores almacenados en el subárbol izquierdo deben ser menores al nodo raíz, los valores en el lado derecho deben ser mayores que el elemento en la raíz.

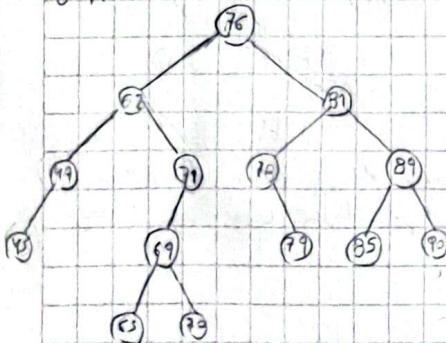
No solo eso, si no que estas condiciones deben cumplirse para todos los subárboles.

Si se cumplen estas condiciones, se dice que el árbol es un árbol binario de búsqueda (ABB).

Por lo tanto al hora de añadir un elemento a un ABB debes asegurarte que se sigan cumpliendo las condiciones, debes realizar el siguiente procedimiento:

- Si es menor, el procedimiento es trivial, el árbol existía solo con el nodo a agregar, y los subárboles vacíos.
- Caso contrario compara el elemento e con la raíz del árbol
 - Si es menor que la raíz, se lo agrega al subárbol izquierdo
 - Si es mayor que la raíz, se lo agrega al subárbol derecho

Ejemplo



Parte 3 Técnicas de resolución de problemas

Algoritmos voraces

1) Técnica más simple de resolución de problemas, normalmente se trata de problemas que requieren ~~que~~ Cierta optimización, algunos ejemplos pueden ser:

- Combinar más carros
- Problema de la mochila
- Costo mínimo (árbol de expansión)
- Problema de la moneda

Los algoritmos voraces buscan la solución vía iteración "paso a paso" de ahí su nombre en inglés **greedy** (puesto que solo van de una moneda a la otra, no tienen en cuenta que sea óptima)

Lo que tienen hecho en cada paso es elegir una componente de la solución final que parezca la más apropiada hasta ese punto. ~~así sucesivamente~~
No revisa si la elección fue la correcta, confía en que las fases previas conducen a la solución óptima. Por este motivo múltiples problemas No admiten una solución greedy/voraz

Problemas que sí admiten una solución voraz

- ① Problema de la moneda (caso patrón)
- ② Problema de la mochila (caso patrón)
- ③ Camino de costo mínimo en un grafo
- ④ Problema del árbol de expansión/árbol generador mínimo

Componentes comunes de un algoritmo voraz

- ① Se plantea un problema a resolver de forma óptima
- ② Una solución estará conformada por un conjunto de candidatos
- ③ Tendremos 3 tipos de candidatos → No considerados
 - ↳ Incorporados
 - ↳ Descartados
- ④ Existe forma de ver si los candidatos ya incorporados, completan una solución del problema (independientemente si ésta es una solución óptima o no)
- ⑤ Un conjunto de candidatos puede ser factible, es decir si este conjunto es solución (óptima o no) del problema
- ⑥ También tendremos una función de elección de candidatos

Entonces invertiremos la lógica: no tendré candidatos, luego de forma iterativa iremos formando algunos candidatos (de acuerdo a la función de selección) considerando que este candidato haga a una solución factible, y de nuevo este procedimiento hasta llegar a una solución

función voraz (C : set of "candidatos") ret S: "Solución a construir"

```

S := "solución vacía"
do S := "no es solución" → c := "selección" de C ←
  eliminar (C, c)
  if "agregar c a S es factible" →
    "agregó c a S"
  od
end fun
  
```

lo mas importante es el criterio de Selección

P

② Problema de la moneda

Tenemos un contenedor frágil de monedas de cada una de las siguientes denominaciones $1\frac{1}{2}, \frac{1}{4}, \frac{1}{10}, \frac{1}{20}, \frac{1}{100}$. Debe pagar cierta cantidad de forma exacta, ~~minimizando~~ la cantidad de monedas utilizadas.

Debemos elegir un criterio de selección, que no indique que moneda tomar a continuación el único que nos viene a la mente es tomar la moneda de mayor denominación y repetir este proceso con el monto restante.

fun Combi (m:Nat, C:set of Nat) ret S:Nat

Version 1)

Var c, resto:Nat

Var Coax := copy-set(C)

resto := m

do resto > 0 →

c := "máximo elemento de Coax"

eliminar (Coax, c)

S := S + (resto, div, c)

← agregar la moneda c, todos los veces que entre

resto := resto mod c

← restar el valor de agregar varias veces la moneda

od

destruir-set (Coax)

end fun

fun Combi (m:Nat, C:set of Nat) ret S:Nat

Version 2)

Var c, resto :Nat

Var Coax := copy-set(C)

S := 0

resto := m

do (γ is_expt_set (Coax)) →

c := "máximo elemento de Coax" ← Selección

if c > resto then

← Chequeo factibilidad

eliminar (Coax, c)

else

← agrega a la solución

resto := resto - c

S := S + 1

fi

od

destruir-set (Coax)

end fun

El siguiente algoritmo devolverá los valores de las ~~monedas~~ ^{monedas} o, la solución final

fun Combi (m:Nat, C:set of Nat) ret S:List of Nat

Var c, resto :Nat

Var Coax := set of Nat

S := empty-list()

Coax := copy-set(C)

resto := m

do (γ is_expt-set (Coax)) →

c := "máximo elemento de Coax"

if c > resto then

eliminar (Coax, c)

else

resto := resto - c

addl (S, c)

fi

destruir-set (Coax)

La primera versión tiene un problema, para algunos sistemas conjuntos de decisiones quede que el ciclo no termine o que se quiera obtener el máximo de un conjunto vacío. En otras palabras el algoritmo no siempre garantiza una solución óptima.

Los conjuntos para los cuales se puede encontrar una solución óptima se llaman "sistemas canónicos".

El problema de la mochila es un problema de PL. Entera.

$$\begin{array}{l} \min N_1 + N_2 + \dots + N_n \\ \text{s.t. } P_1 N_1 + P_2 N_2 + \dots + P_n N_n = M \\ M = \text{Monto a cubrir} \end{array}$$

con N_i : cantidad de monedas del tipo i
 P_i : denominación de la moneda del tipo i

b) Problema de la mochila

Se posee una mochila de capacidad W , y tenemos n objetos de valor V_1, \dots, V_n y pesos W_1, \dots, W_n . Se busca escoger una los objetos tales que se maximice el valor total de lo cargado en la mochila.

Tiene muchas posibles formas de escoger un objeto

- El más valioso
 - El menor peso
 - El más valioso respecto a su peso
- } Ninguno de estos criterios garantiza una solución óptima

Este problema no admite una solución óptima, aunque se simplifica mucho si se permite fraccionar los objetos.

Antes de pasar a la implementación, definiremos el tipo de datos de los objetos en la mochila.

tipo objeto = tuple

```
    id: Nat
    value: Float
    weight: Float
end tuple
```

tipo Obj-Mochila = tuple

```
    obj: objeto
    fract: Float
end tuple
```

fun Mochila(W: float, C: set of objeto) ret L: list of obj-mochila

var O-m: Obj-Mochila

var resto: float

var Caux: Set of objeto

S := emptry-set()

resto := W

do (resto > 0) →

O-m.objeto := select. obj(Caux)

if O-m.objeto.weight ≤ resto then

O-m.fract := 1

resto := resto - O-m.objeto.weight

else

O-m.fract := resto / O-m.weight

resto :=

fi

f;

oddr(S, O-m)

eliminar(Caux, O-m, obj)

od od

destruir-set(Caux)

end fun

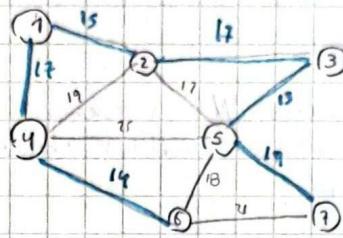
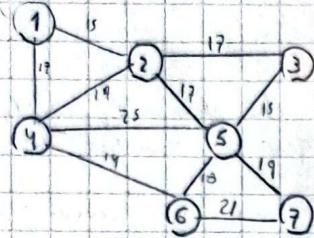
Algoritmos Voraces sobre grafos

(A) Árbol generador de costo minimo \equiv Spanning Tree

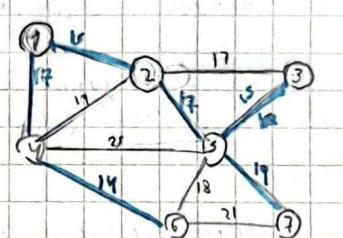
Sea $G = (V, A)$ un grafo conexo, no dirigido con un costo no negativo asociado a cada arista

Se dice que T es un **árbol generador** si $T \subseteq A$, es decir el grafo (V, T) es conexo y acíclico, el costo de este árbol sea la suma de los costos de sus aristas, se buble el conjunto de aristas que minimice el costo.

Ejemplos:



ambas son
árboles de
expansión

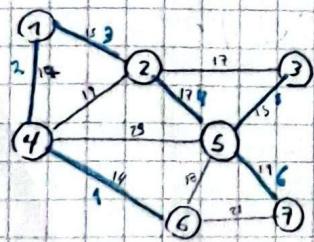


Particularmente bonitas
tienen el mismo costo

Hay 2 algoritmos importantes para encontrar un árbol generador a partir de un grafo: Prim y Kruskal.

Prim: Parte de un vértice origen, y se va expandiendo, a cada paso se une el tendido ya existente con algunos de los vértices aun no alcanzados, seleccionando la arista de menor costo.

Kruskal: Se divide al grafo en distintas componentes (originalmente unidas por un vértice) y se van uniendo los vértices, donde en cada paso se selecciona la arista de menor costo capaz de unir componentes.



Podemos representar los grafos con una tupla con 2 componentes, uno para vértices y otro para aristas

type vertex = Nat

type edge = tuple

V_1 : Vertex

V_2 : Vertex

Cost : Nat

end tuple

type graph = tuple

vertices : set of vertex

edges : set of edge

end tuple

C: Todos los vértices
no conectados

nodo
initial

fun Prim(G:Graph, k:Vertex) ret T: Set of Edge

Var e: Edge

Vd C: Set of vertex

C := GPr-set(G.vertices)

elim(C, k)

T := emp.set()

do (not isEmpset(C)) →

C := "selecciona la arista de costo mínimo tq c.v₁ ∈ C & c.v₂ ∉ C ó c.v₂ ∈ C & c.v₁ ∉ C"

if member(c.v₁, C) then

elim(C, c.v₁)

else

!prim(C, c.v₁)

fi

add(T, c)

od

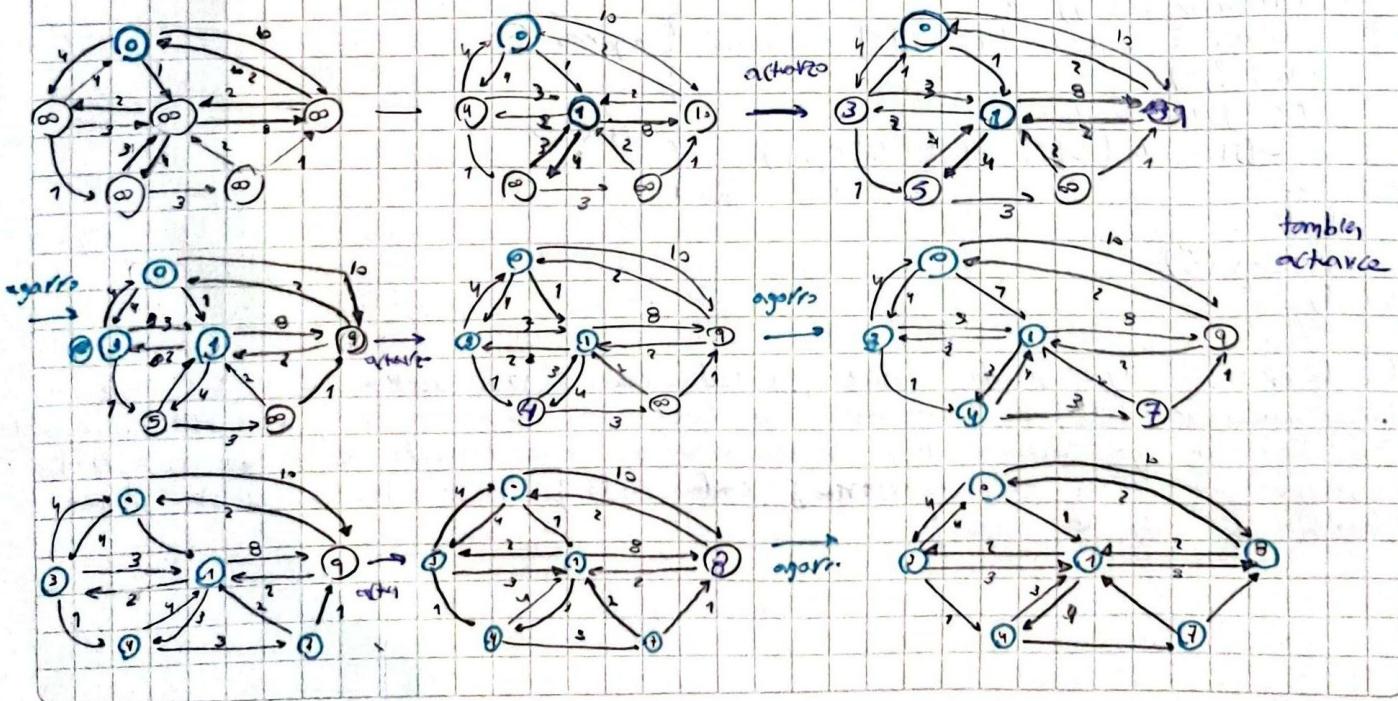
end fun

B) Camino de costo mínimo

Sea $G = (V, A)$ un grafo dirigido con costos no negativos en sus aristas \geq sea $v \in V$ verlo que de bucle es obtener el camino de menor costo desde v hacia cada uno de los demás vértices

El algoritmo de Dijkstra realiza una secuencia de n pasos, donde n es el número de vértices/nodos, donde en cada paso "aprende" el camino de menor costo desde el nodo v , hacia otro vértice, a este nuevo vértice se lo marca.

Idea: Tratemos de verlo a través del tiempo, en cada paso los vértices marcados actualizan el camino de menor costo desde el nodo v , mientras que para los vértices no yet marcados mantienen el costo del camino pintado de menor costo desde el nodo v , el camino pintado es el que sobriza al vértice dentro del marco, todos los otros marcados



El Algoritmo

Si el grafo tiene un conjunto de vértices $V = \{1, \dots, n\}$ y que los costos estén en una matriz $L: \mathbb{R}^{n \times n}$, es decir $L: \text{array}[1..n, 1..n]$ of Real donde $L[i, j]$ contiene el costo de la arista que va desde el vértice i al vértice j , en caso de no haber dicha arista $L[i, j] = \infty$, y $L[i, i] = 0$, o sea

$$L[i, j] = \begin{cases} 0 & i=j \\ \infty & (i, j) \notin A \\ \epsilon_{ij} & (i, j) \in A \end{cases}$$

el algoritmo también funciona para grafos
no dirigidos donde $L[i, j] = L[j, i]$
 $\forall i, j \in V$

la versión que veremos del algoritmo en vez de hallar el costo mínimo desde V hasta cada nodo halla solo el costo de dicho camino, es decir que halla el costo del camino de costo mínimo desde V hasta los demás nodos.

El resultado retorna en un arreglo $D: \text{array}[1..n]$ of Nat con $D[i]$ el costo de desde el nodo V al nodo i .

El conjunto C es el conjunto de los vértices hacia los que todavía desconoces el mejor camino de menor costo.

fun Dijkstra ($L: \text{array}[1..n, 1..n]$ of Nat, $V: \text{Nat}$) ret $D: \text{array}[1..n]$ of Nat

```
var C: Nat
var C: set of Nat
for i := 1 to n do
    add (C, i)
    od
    erm (C, V)
    for j := 1 to n do
        D[j] = L[V, j]
        od
    do ( $\neg$  is-empty-set (C))  $\rightarrow$ 
        c := "elijo el elemento de C qd D[c] sea minimo"
        add (C, c)
        erm (C, c)
        for j in C do
            D[j] = min (D[j], D[c] + L[c, j])
        od
    od
    destros_set (C)
end fun
```

La implementación sigue siendo voraz, en cada paso elijo el vértice c al que puedo llegar con menor costo, y actualizo el costo para llegar a los demás vértices habiendo "aprendido" el mejor camino, dicha actualización se hace tomando el mínimo entre lo que estaba ir antes hasta ~~c~~ y cada nodo j y lo que me cueste si voy desde el nodo c hasta j .

Backtracking

Anteriormente le vimos algoritmos voraces, este tipo evita en ir armado la solución, donde en cada paso toma un criterio de decisión para elegir que candidato se agrega a la solución, dicho criterio debe ser bueno, y una vez elegido el candidato, no se revisa si la solución es correcta ni elige, avanza que no siempre la solución voraz llegaba a una solución óptima, y es aquí cuando entra el Backtracking.

La idea del backtracking es elegir un candidato y ver si este me lleva a la sol. óptima, en caso de que no, vuelve atrás y prueba otro candidato.

A este proceso de "volver atrás" se le conoce como Backtrack,

Nota: en la práctica lo que se hace es considerar todas las posibilidades e intentar cada una de ellas para saber cuál fue la solución óptima, por este motivo los algoritmos de Backtracking son de Fuerza Bruta y sumamente lentos.

Diferencia con algoritmos greedy

- Si hay solución óptima, la encuentra
- Ineficientes
- No hay un buen criterio de selección (fuerza bruta)

(A) Problema de la moneda con Backtracking

Sean d_1, \dots, d_n las denominaciones de las monedas donde se desea pagar una suma k de moneda exacta utilizando la menor cantidad de monedas.

Idea del algoritmo: Recursivamente, se probando una moneda, si la denominación es factible (el resto dividido es menor o igual a K) entonces la agrega a la solución, finalmente toma el número de todos las denominaciones.

```

fun Cambio(K:Nat, C:set of Nat) ret S:Nat
    var c:Nat
    var Caux : set of Nat
    if k = 0 then S := 0
    else if is_empty(C) then S := 100
    else
        Caux := copy_set(C)
        c := get(C)
        Caux := Caux - {c}
        if c ≤ k then
            S := min (Cambio(k-c, C)+1, Cambio(k, Caux))
        else
            S := Cambio(k, Caux)
    fi
    f1
    kb_destroy(Caux)
end fun

```

④aca está la magia, toma el número entre
 ①) si tomas con la moneda c , es decir la denominación total incluyendo la c , y la cantidad de monedas sumo a 1
 $(\text{Cambio}(k-c, C)+1)$. y ② la solución sin tener en cuenta a la moneda c
 $(\text{Cambio}(k, Caux))$

algoritmo actualizado ↘

Aclaraciones

- El primer caso del if, corresponde a cuando ya se llega a pagar el total, en dicha situación no serán necesarias más monedas.
- El segundo caso es cuando probé devolver tantas denominaciones que ya no puedo pagar el monto, ~~lo que significa que el problema no es factible~~.
- Tercer caso, si la denominación es factible para pagar el monto C , en cuyo caso intentaría otra vez, ~~utilizando~~ viendo un arreglo para las denominaciones, el parámetro i , me dice hasta que denominación utilizar.

fun $\text{combi}(\text{d}: \text{array}[1..n] \text{ of Nat}, k: \text{Nat}) \text{ ret } \text{Nat}$

```
if  $k=0$  then  $r:=0$ 
else if  $i=0$  then  $S:=\infty$ 
else if  $d[i] > k$  then  $R:=\text{combi}(\text{d}, k, i-1)$ 
else  $r:=\min(\text{combi}(\text{d}, k-d[i], i), \text{combi}(\text{d}, k, i-1))$ 
fi
end fun
```

Definición recursiva

Además que lo más interesante de estos algoritmos con backtracking es la forma recursiva, o más sencillamente definirla matemáticamente.

$\text{combi}(i, j) \triangleq$ menor nro de monedas necesarias para pagar el monto j de forma exacta, con las denominaciones d_1, \dots, d_i

$$\text{combi}(i, j) = \begin{cases} 0 & j=0 \\ \infty & j>0 \wedge i=0 \\ \text{combi}(i-1, j) & d_i > j \geq 0 \wedge i > 0 \\ \min(\text{combi}(i-1, j), 1 + \text{combi}(i, j-d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

Caso ① $j=k=0$ (el monto a pagar es 0)

Caso ② $j>0 \wedge i=0$ monto a pagar > nro de r no tengo monedas, (Problema no factible)

Caso ③ $d_i > j > 0 \wedge i > 0$, el monto d_i es menor que lo que tengo que pagar, devuelvo la moneda.

Caso ④ $j \geq d_i > 0 \wedge i > 0$ lo que de d_i hasta ahora, el monto $j - ...$

B) Problema de la mochila con Backtracking

Teneros una mochila con capacidad W , tenemos n objetos de valor V_1, \dots, V_n con su peso w_1, \dots, w_n , se busca realizar la combinación de objetos que maximizan el valor total.

Por simplicidad, no nos interesa la combinación de objetos V_1, \dots, V_n solo nos interesa el valor total que estos nos proporcionan.

definimos una forma recursiva $m(i, j)$: Siendo $i \geq$ cantidad de obj. objectos $> j$ da ~~que~~ la capacidad de la mochila.

$m(i,j)$: "mayor valor alcanzable, viendo los objetos $1, 2, \dots, i$, sin exceder la capacidad $j"$

$$m(i,j) = \begin{cases} 0 & j=0 \\ 0 & j>0 \wedge i=0 \\ m(i-1,j) & W_i > j \geq 0 \wedge i>0 \\ \max(m(i-1,j), V_i + m(i-1, j-W_i)) & j > W_i \geq 0 \wedge i>0 \end{cases}$$

Notar: la función es muy similar a la del problema de la moneda, esto es gg el anterior en un caso particular de este. Con donde $W_1, \dots, W_n = -1$

⑤ Problema de camino de costo mínimo

Dado un grafo dirigido $G = (V, A)$ con costos no negativos en sus aristas, se busca encontrar, para cada par de vértices v, w , el camino de menor costo que los une.
 Se asume $V = \{1, \dots, n\}$

Simplificando el problema

① Solo se interesa el costo de los caminos, no los arcos del mismo

Generalizando el problema

② Son $1 \leq i, j \leq n \wedge 0 \leq k \leq n$

③ Definimos $C(k, i, j)$: "menor costo posible para un recorrido desde i hasta j , cuyos vértices intermedios están en el conjunto $\{1, \dots, k\}$ "

④ La solución del problema original sea $C(n, \text{origen}, \text{destino})$

$$C(k, i, j) = \begin{cases} L[i, j] & k=0 \\ \min(C(k-1, i, j), C(k-1, i, k) + C(k-1, k, j)) & k \geq 1 \end{cases}$$

donde $L[i, j]$ es el costo de la arista i a j ($\text{anfint. sr } (i, j) \in A$)

[Conclusiones generales]

- En cada subproblema el algoritmo calcula como resultado de acuerdo a la decisión de agregar un condensado a la solución y se guarda con la mejor
 - En el problema de la moneda decide si utilizar o no la decisión i -ésima
 - En el problema de la mochila "no" o "no" es la decisión i -ésima
- En el problema de camino min. costo decide si quitar o no por el vértice k
- En general suelen ser subproblemas muy repetitivos, por ejemplo

Por el problema de la moeda, si se quiere pagar el monto 90 con monedas de denominación 1, 5, 10

combro (3, 90) libra a combro (2, 90) y a combro (2, 80) costo 1

combro (2, 80) libra a combro (1, 80) y a combro (2, 80) costo 1

combro (2, 80) libra a combro (2, 80) y a combro (3, 70) costo 1

algoritmo exponencial

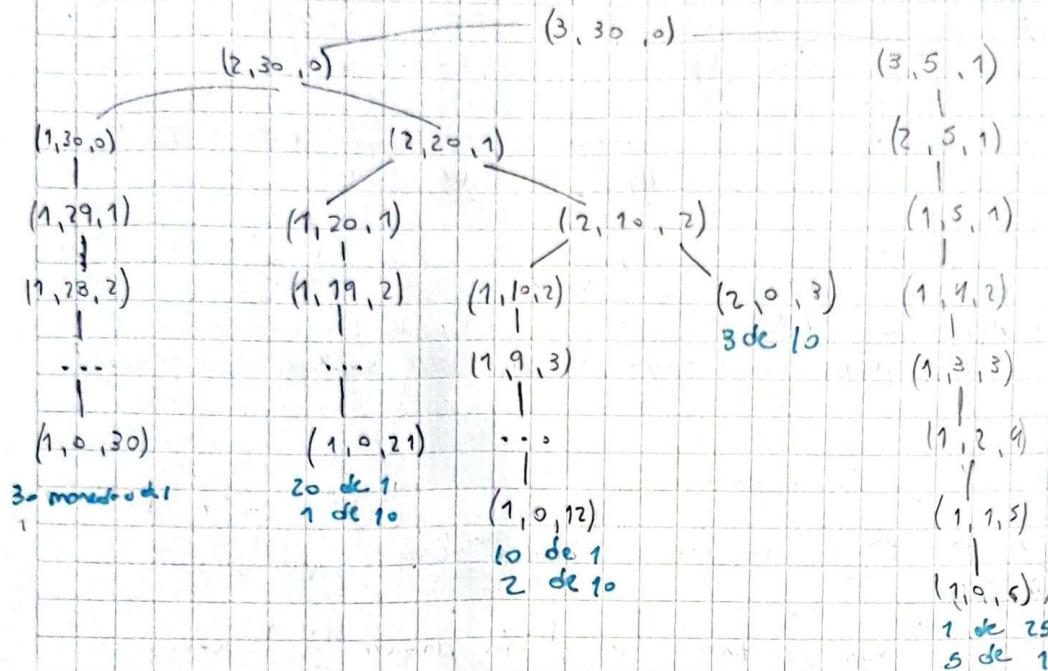
Nota: se libra 2 veces a combro (2, 80)

Podemos dibujar en un arbol, donde cada vértice es la función cambio(i, j). El vértice (i, j, x) nos indica el ~~camino más corto~~ cambio(i, j) donde el resultado hasta ese momento es x .

Ejemplo

$$k = 30$$

$$d_1 = 1, d_2 = 10, d_3 = 25$$



en general desde el vértice (i, j, x)

① si $i, j > 0$ y $d_i < j \Rightarrow \exists!$ orilla al vértice $(i-1, j, x)$ que no entra la moneda más grande

② $j \leq d_i$ existen 2 orillas $((i-1, j, x))$ (\wedge no usa esa moneda) $((i, j-d_i, x+1))$ (\wedge considera la moneda en consideración)

③ los n/ k son el nodo $((n, k, 0))$

④ los hojas son $(1, 0, x)$

Programación Dinámica

Recordar la función del problema de la mochila, y de la mochila, a través de Backtracking

- Cambio (i, j): menor nro de monedas necesarias para pagar un monto j considerando los monedas d_1, \dots, d_n

$$\text{cambio}(i, j) = \begin{cases} 0 & i=0 \\ \infty & j>0 \wedge i=0 \\ \min\{\text{cambio}(i-1, j), 1 + \text{cambio}(i, j-d_i)\} & d_i > j > 0 \wedge i > 0 \\ & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

- Mochila (i, j): "mejor" valor contable sin exceder la capacidad j , con objetos $1, 2, \dots, i$.

$$\text{mochila}(i, j) = \begin{cases} 0 & j=0 \\ 0 & j>0 \wedge i=0 \\ \max\{\text{mochila}(i-1, j), V_i + \text{mochila}(i-1, j-w_i)\} & w_i > j > 0 \wedge i > 0 \\ & j \geq w_i > 0 \wedge i > 0 \end{cases}$$

- DLM
 - (vuelo) Son: de donde a donde

↳ vía 1 → Indice
 ↳ vía 2 → Número pasajero

- En que orden la lleno

Con Programación Dinámica: Podremos transformar una definición recursiva en definición iterativa, pero se deberá tener una tabla de valores, en esta tabla ítems útiles guardados (sustitutos parciales) para evitar recorrer caminos.

Ejemplo : Generación de Fibonacci

$$f_n = \begin{cases} n & n \leq 1 \\ f_{n-1} + f_{n-2} & n > 1 \end{cases}$$

f-PRE: $n > 1$

fun fibo (n:nat) ret r:nat este algoritmo es
 var f: array[0..n] of Nat linear

- $f[0] := 0$
 $f[1] := 1$
 for $i := 2$ to n do
 $f[i] := f[i-1] + f[i-2]$
 od
 $r := f[n]$

end fun

Volvemos al problema de la moneda

requeriremos una tabla bidimensional, en una dimensión tendremos el parámetro i referida al parámetro j , notar también que a la función comb(i,j) puede suministrarse el primer parámetro comb(i-1,j) o usar el valor de comb(i,j-d)

Por lo que para llenar el valor de la tabla necesitaremos que la tabla esté bien definida es:

- ① todas las casillas de la misma fila
- ② la fila anterior respecto a la fila i la columna

$i, j-d, i, j$

i, j
 i, j

Por lo que sellaremos la tabla de izq a derecha y de abajo hacia arriba

```
for comb (d:array[1..n] of nat, k:nat) ret r:nat
    var comb [0..n, 0 .. k] of Nat
    for j:=1 to k do comb[0,j]:=∞ od
    for i:=0 to n do comb[i,0]:=0 od
    for i:=1 to n do
        for j:=1 to k do
            if d[i]>j then comb[i,j]:=comb[i-1,j]
            else
                comb[i,j]:=min{comb[i-1,j], 1+comb[i-j-d][i]}
            fi
        od
    od
    r:=comb[n,k]
end fun
```

||En la primera fila son ∞
 ||En la primera columna son 0

p. vaciar
 p. BT
 p. pd

de forma análoga podemos plantear el problema de la mochila

Tendremos una línea dimensiones para los valores de los objetos a considerar (de 0 a n) y otra para los valores de los pesos (0 a W)

fun mochila(V:arr[1..n] of Valor, W:arr[1..n] of Nat, W:Nat) ret r:Valor

```

var moch: arr[0..n, 0..W] of Valor
for i:=0 to n do moch[i, 0] := 0 od
for j:=0 to W do moch[0, j] := 0 od
for i:=1 to n do
  for j:=1 to W do
    if w[i] > j then moch[i, j] := moch[i-1, j]
    else moch[i, j] := max{moch[i-1, j], v[i] + moch[i-1, j-w[i]]}
  fi
od
od
r := moch(n, W)
end fun

```

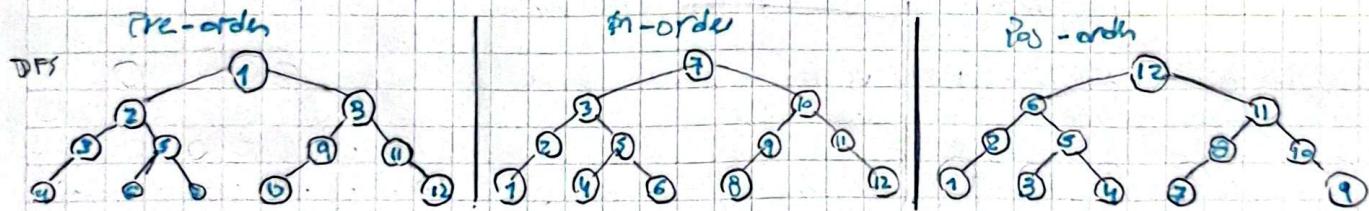
Rcorridos de grafos

Recorrer un grafo significa procesar las vértices del mismo de una forma organizada de forma tal que no se salga

- Procesar todos los vértices
- No dejar ningún vértice sin procesar

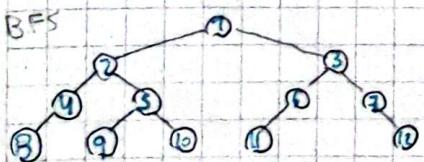
Un caso sencillo es el de el arbol binario, que tiene 3 formas de recorrerse

- **Preorden:** Se visita la raíz, luego el subarbol izquierdo y finalmente el derecho
- **In-orden:** Se visita el subarbol izquierdo, luego la raíz y finalmente el subarbol derecho
- **Pos-orden:** Recorremos el subarbol izquierdo, luego el derecho y al final el nodo es la raíz



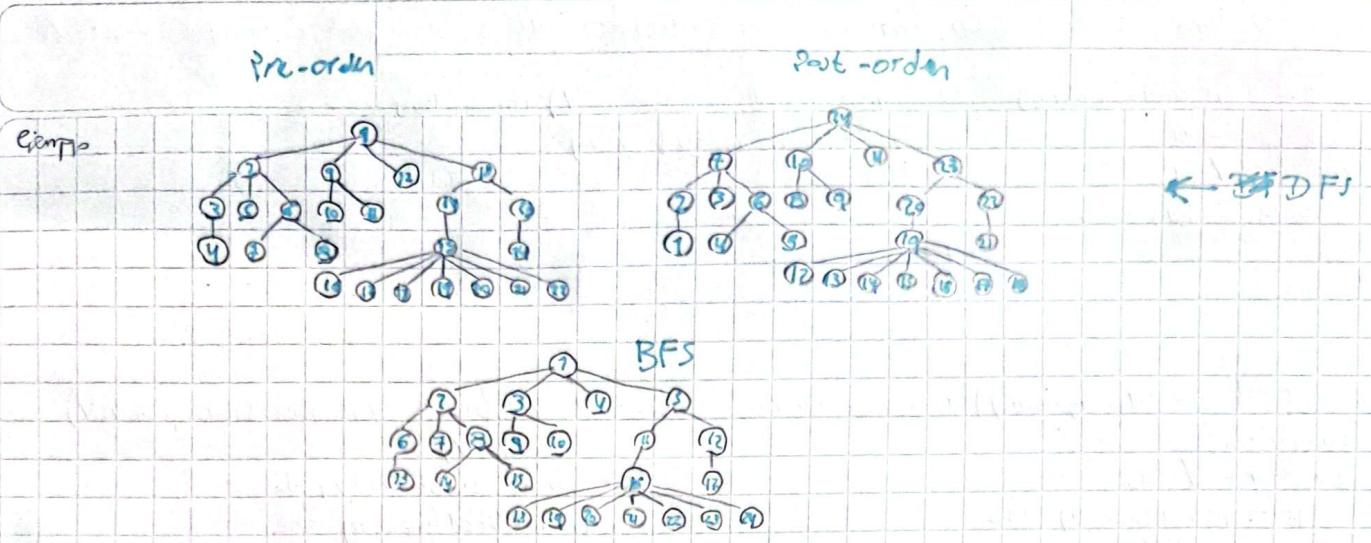
Existen otras formas de recorrer árboles, por ejemplo

BFS



los primeros 3, recorre un arbol en profundidad, esto lo hace en q lo archo

Ahora veremos recorridos en árboles finitos, es decir, árboles, cada nodo tiene una cantidad finita (pues variable) de hijos, es decir caso 4 recorrido in-orden no tiene sentido, el que nosotros hemos hecho la raíz se tiene más de 2 hijos



al visitar un nodo, se lo marca con un numero positivo

```
type tmark = tuple
  ord: array[V] of Nat
  cont: nat
end tuple
```

```
proc init(out: mark; tmark)
  mark.cont := 0
end proc
```

```
proc visit(inout mark: tmark, in v: V)
  mark.cont := mark.cont + 1
  mark.ord[v] := mark.cont
end proc
```

a) Variables que dà grafos. Nos dadas por su raiz & una función children que devuelve los hijos de cada vértice

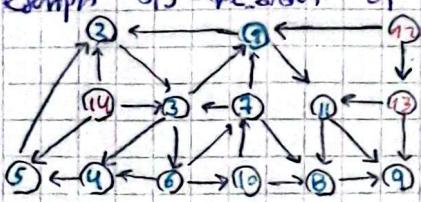
```
fn pre_order(G=(V, root, children)) ret mark:tmark
  init(mark)
  pre_traverse(G, mark, root)
end fn
```

```
proc pre_traverse(in G, inout mark:tmark, in v: V)
  visit(mark, v)
  for w ∈ children(v) do
    pre_traverse(G, mark, w)
  od
end proc
```

```
fn pos_order(G=(V, root, children)) ret mark:tmark
  init(mark)
  pos_traverse(G, mark, root)
end fn
```

```
proc pos_traverse(in G, inout mark:tmark, in v: V)
  for w ∈ children(v) do
    pos_traverse(G, mark, w)
  od
  visit(mark, v)
end proc
```

Ejemplo: dfs - post-order en un grafo



Ahora podemos tener ciclos. Por eso es necesario ver si un vértice ya fue visitado.

```

proc init (fout mark:tmark)    fv: visited (mark:tmark, v:V) ret b: Bool
    mark cont := 0
    for v ∈ V do
        mark.ord[v] := 0
    od
end proc

```

```

fun dfs ((G = (V, neighbours)) ret mark:tmark
    init(mark)
    for v ∈ V do
        if ¬ visited (mark, v) then
            dfs (G, mark, v)
        fi
    od
end fm

```

```

proc dftsearch (in G, inout mark:tmark, in v:V)
    visit (mark, v)
    for w ∈ neighbours (v) do
        if ¬ visited (mark, w) then
            dftsearch (G, mark, w)
        fi
    od
end proc

```

Podemos ordenar la DFS iterativa, si utilizamos una cola en lugar de una pila (a la tecnic

```

proc bfSearch (in G, inout mark:tmark, in v:V)
    var q: queue of V
    empty(q)
    visit (mark, v)
    enqueue (q, v)
    while ¬ isEmpty (q) do
        if existe w ∈ neighbours (first(q)) tal que ¬ visited (mark, w) then
            visit (mark, w)
            enqueue (q, w)
        else
            degrev (q)
        fi
    od
end proc

```

Procedimiento Principal

```

fun bfs (G = (V, neighbours)) ret mark:tmark
    init(mark)
    for v ∈ V do
        if ¬ visited (mark, v) then
            bfSearch (G, mark, v)
        fi
    od
end fun

```

