Programación Dinámica

Marks Calderón Niquin

Programación Dinámica

- Potente técnica de diseño de algoritmos
- Usado en problemas de Optimización (Max, min)
- Guarda sus soluciones en una tabla para futura utilización, esta se le denomina memorización

Fibonacci

Fibonacci

Fibonacci relationship

$F_1 = 1$

$F_2 = 1$

$F_3 = 1 + 1 = 2$

$F_4 = 2 + 1 = 3$

$F_5 = 3 + 2 = 5$

In general :

$$F_n = F_{n-1} + F_{n-2}$$

or

$$F_{n+1} = F_n + F_{n-1}$$

Fibonacci pseudocodigo

Fibonacci relationship

$F_1 = 1$

$F_2 = 1$

$F_3 = 1 + 1 = 2$

$F_4 = 2 + 1 = 3$

$F_5 = 3 + 2 = 5$

In general :

$$F_n = F_{n-1} + F_{n-2}$$

or

$$F_{n+1} = F_n + F_{n-1}$$

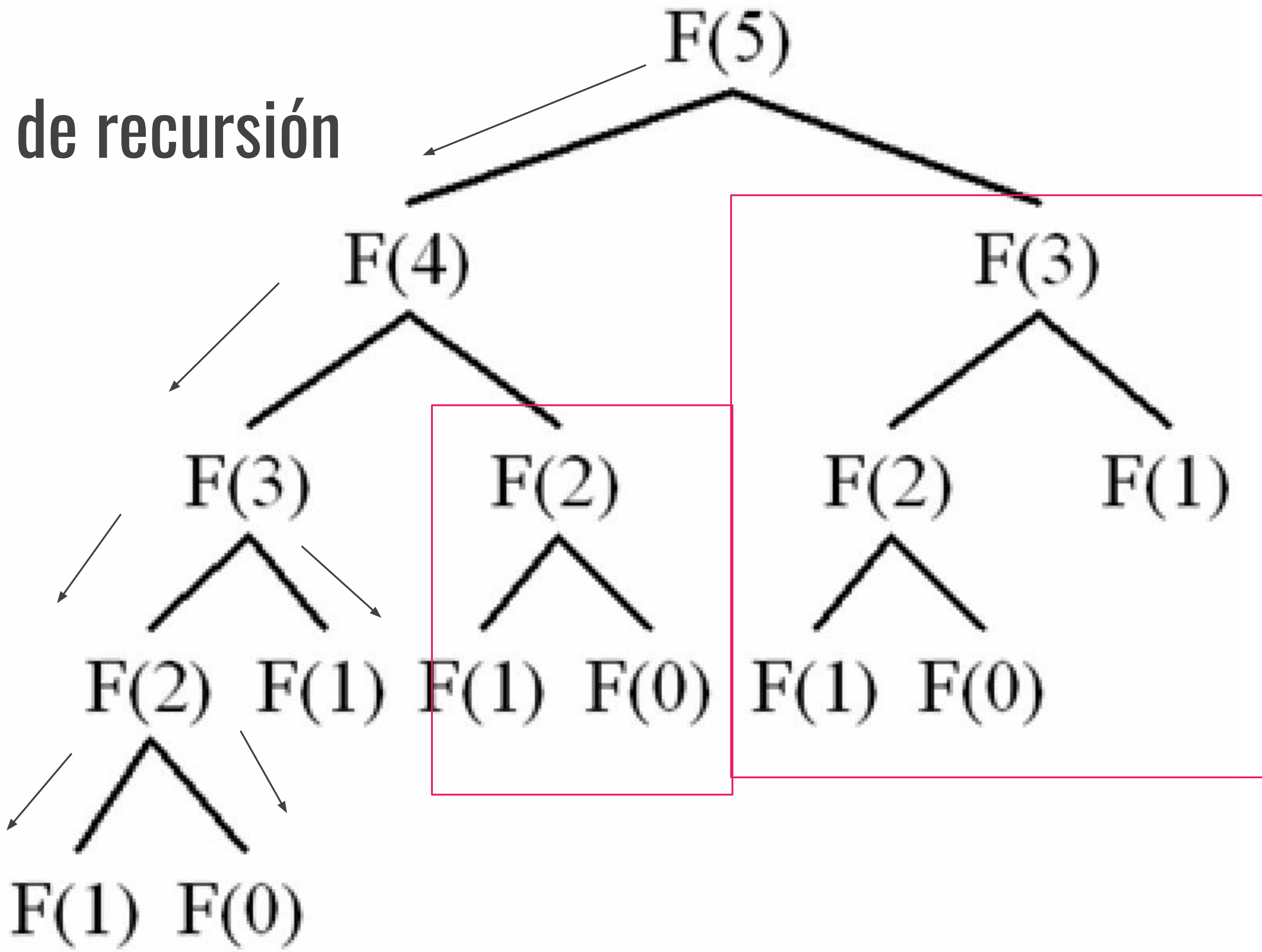
Fibonacci: Bottom-up (Calcula primero)

```
def fib(n):

    # declaracion de arreglo
    f = [0]*(n+1)
    # caso base
    f[1] = 1
    # calculando el fibonacci y almacenando los
valores
    for i in range(2 , n+1):
        f[i] = f[i-1] + f[i-2]
    return f[n]
```

```
# funcion principal
def main():
    n = 9
    print "Numero Fibonacci es " , fib(n)

main()
```

Y si memoriza las soluciones?

Fibonacci: árbol de recursión



Fibonacci: Top-down (verifica el pre-calculo)

```
# Funcion para calcular nth numero fibonacci
def fib(n, lookup):
    # caso base
    if n == 0 or n == 1 :
        lookup[n] = n

    # Si el valor es calculado anteriormente
previamente
    if lookup[n] is None:
        lookup[n] = fib(n-1 , lookup)  + fib(n-2 ,
lookup)

    # return el valor correspondiente al valor de n
    return lookup[n]
```

```
# funcion principal
def main():
    n = 5
    #declaracion de la tabla
    lookup = [None]*(101)
    print( "Numero Fibonacci es ", fib(n, lookup))

if __name__=="__main__":
    main()
    #fuente:
https://www.geeksforgeeks.org/dynamic-progra
mming-set-1/
```

Version en C++

```
#include <iostream>
#include <vector>
using namespace std;

int fib(int n, vector<int>& L){
    if( n == 0 || n == 1)
        return L[n] = n;
    //Si el valor es calculado anteriormente previamente
    if( L[n] == -1)
        L[n] = fib(n-1, L) + fib(n-2, L);
    return L[n];
}
int main()
{
    int n = 6;
    vector<int> lookup(101, -1);
    cout<<fib(n, lookup);
    return 0;
}
```

Version 2.0 C++

```
#include <iostream>
#include <vector>
using namespace std;
int fib(int n, vector<int>& L){
    if( n == 0 || n == 1)
        return L[n] = n;
    //Si el valor es calculado anteriormente previamente
    if( L[n] == -1){
        cout<<"fib "<<n<<endl;
        L[n] = fib(n-1, L) + fib(n-2, L);
    }
    return L[n];
}
int main()
{
    int n = 6;
    vector<int> lookup(101, -1);
    cout<<fib(n, lookup)<<endl;
    for(int i=0; i < lookup.size(); i++)
        cout<<lookup[i]<<" , ";

    cout<<fib(10, lookup)<<endl;
    return 0;
}
```

`vector<vector<int> > L;`

Ejercicios

1. Crear una función recursiva para calcular la potencia dado la base y el exponente, la base y el exponente deben ser valores [1, 8]
   pot(3,6) , pot(3,8)
2. Crear una función recursiva para calcular el factorial
   fact(4) , fact(8)
   :

	1	2	3	4	5	6	7	8
1								
2		4	8					
3								
4								
5								
6								
7								
8								

Programación Dinámica

IDEA:

- Caracterizar la estructura de una solución óptima
- Definir recursivamente el valor de la solución óptima
- Construir la solución óptima en base a la información calculada

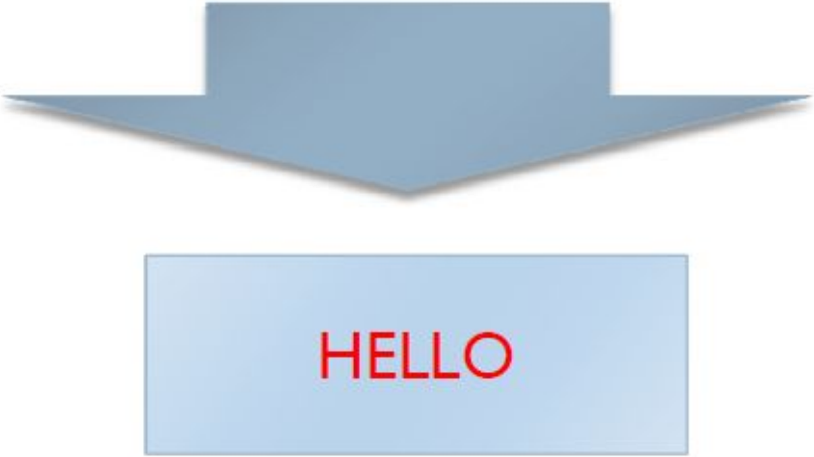
Subsecuencia común más larga

Veamos

IELO

IEGLO
LLO
EGO

HEGLO
HELO
HELLO



IELLO

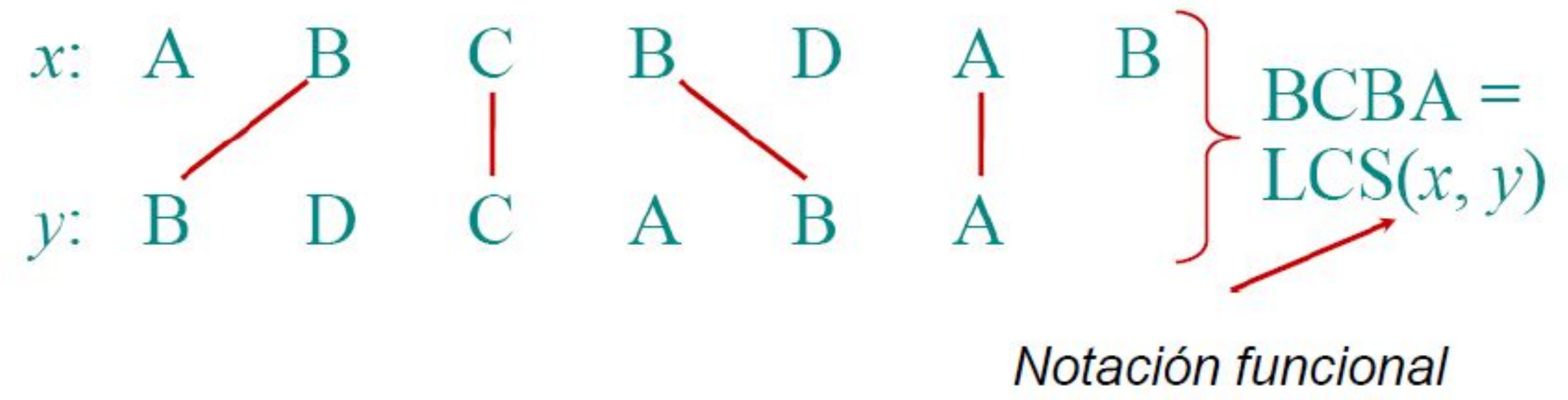
IGO

HEO

Subsecuencia común más larga

IDEA:

- Dadas dos secuencias x[1…m] e y[1…n], encontrar una subsecuencia más larga común a ambas



Algoritmo de fuerza bruta?

Algoritmo de fuerza bruta

Algoritmo de fuerza bruta:

- Revisar cada subsecuencia de x[1..m] para verificar si es también subsecuencia de y[1..n]

Análisis:

- Revisión= Tiempo O(n) por subsecuencia
- 2^m subsecuencia de x
- Tiempo de ejecución del peor caso = O(n*2^m) **exponencial**

Subsecuencia común más larga

String A = “acbaed”

String B = “abcadf”

Longest Common Subsequence (LCS): 4

String A	a	c	b	a	e	d
String B	a	b	c	a	d	f

Subsecuencia común más larga[recursivo]

Caso 1: String A: "ABCD", String B: "AEBD"

LCS("ABCD", "AEBD") = 1 + LCS("ABC", "AEB")

Caso 2: String A: "ABCDE", String B: "AEBDF"

LCS("ABCDE", "AEBDF") = Max(LCS("ABCDE", "AEBD"), LCS("ABCD", "AEBDF"))

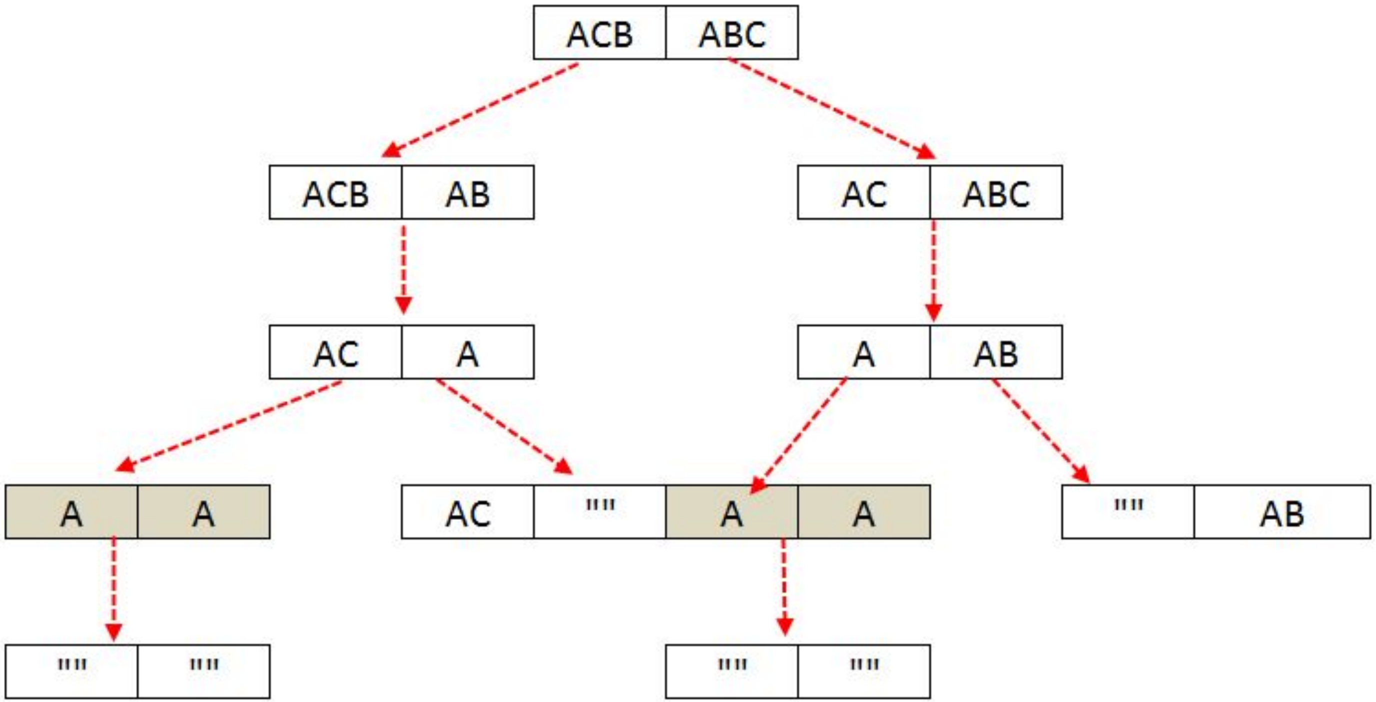
Caso 3: String A= “” , String B = “ABC” o viceversa

LCS(“”, “ABC”) = 0

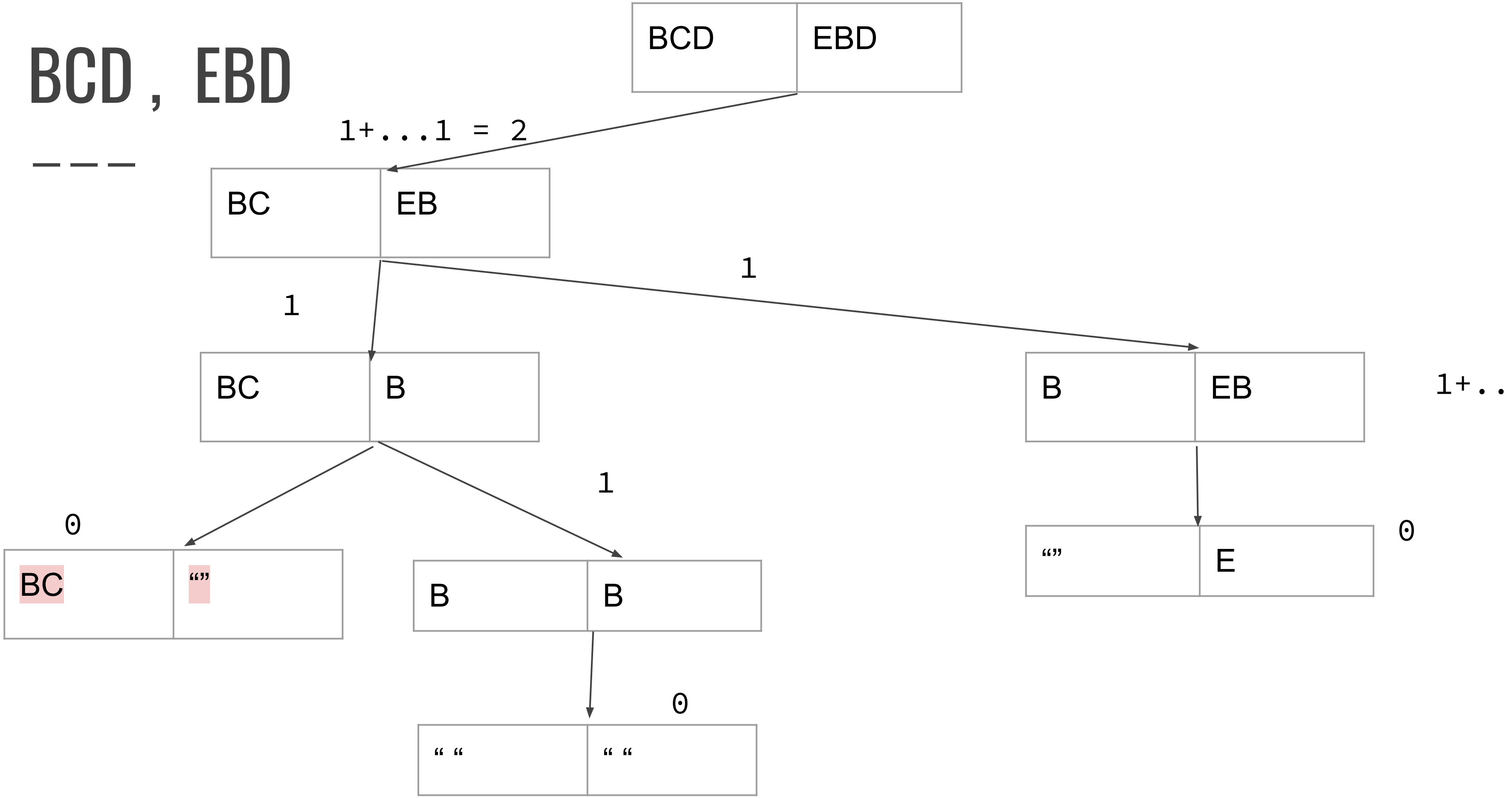
Caso 4: String A= “” , String B = “”

LCS(“”, “”) = 0

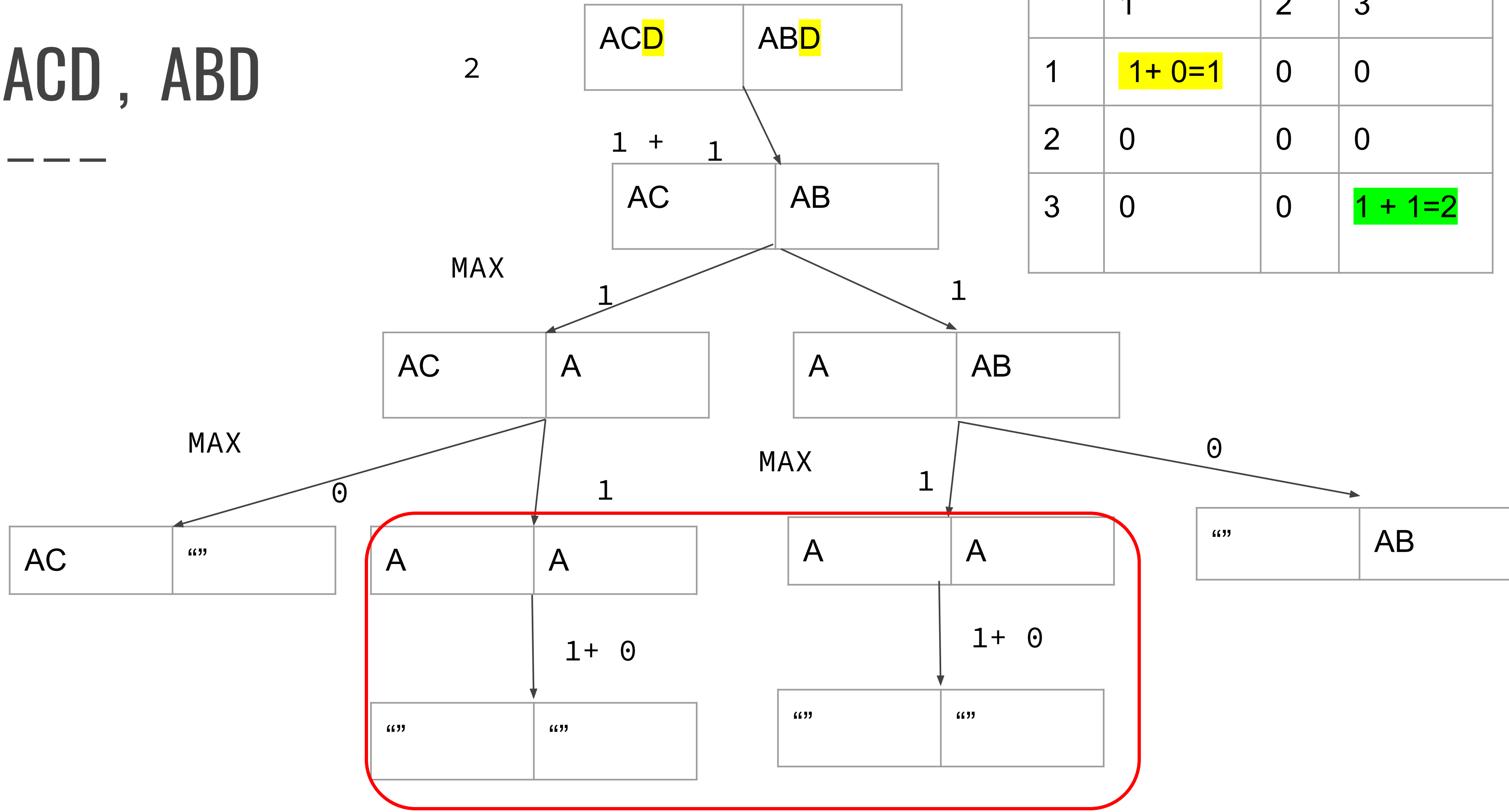
Subsecuencia común más larga[recursivo]



BCD , EBD



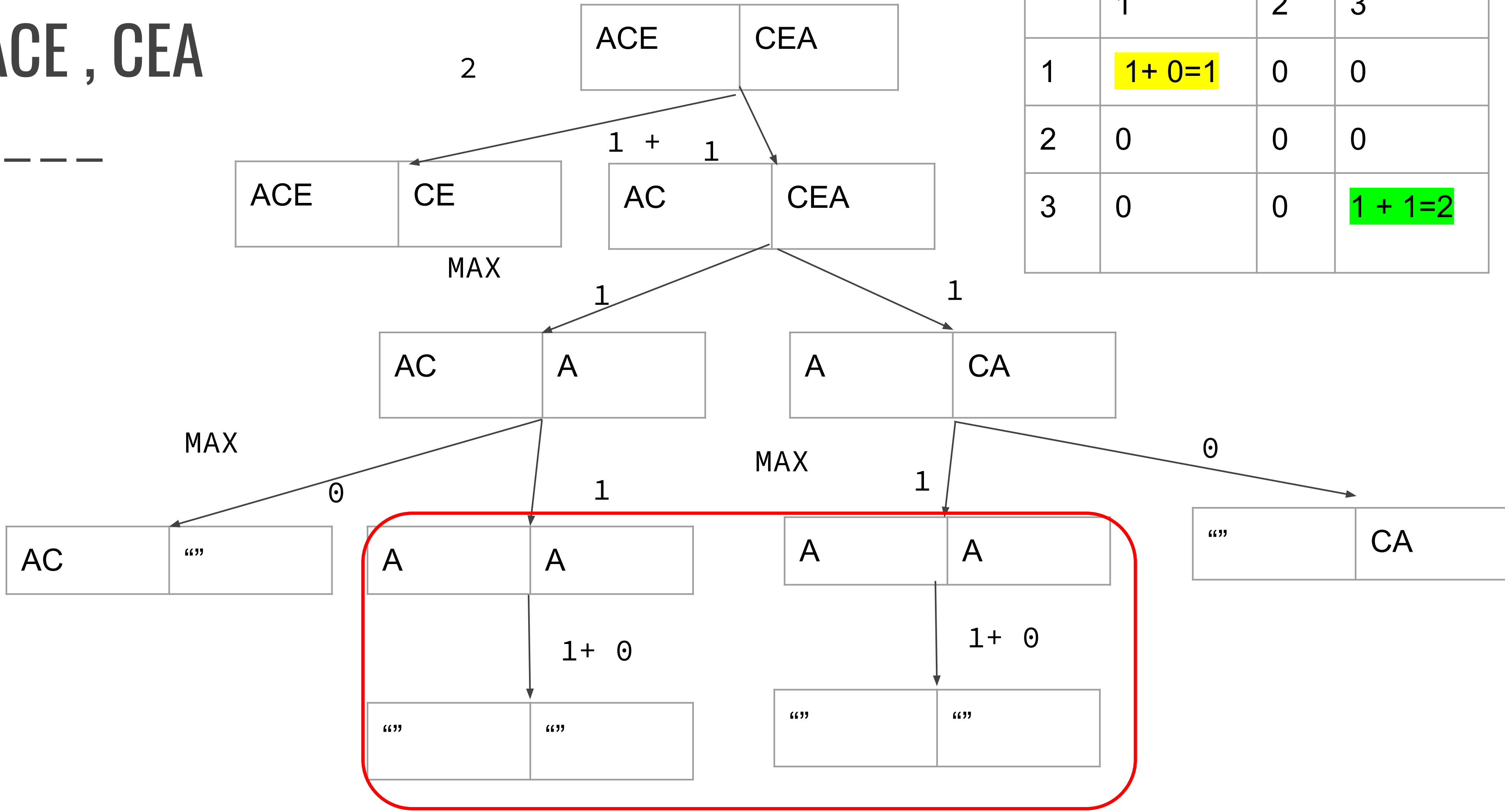
ACD , ABD



	1	2	3
1	1+ 0=1	0	0
2	0	0	0
3	0	0	1 + 1=2

ACE , CEA

_ _ _



	1	2	3
1	1+ 0=1	0	0
2	0	0	0
3	0	0	1 + 1=2

ACE , CEA

Programación Dinámica LCS

- Resolvemos de la forma bottom-up y almacenamos la solución de subproblemas en un arreglo solución, se usa cuando es necesario.
- Si z=LCS(x,y) , entonces cualquier prefijo de z es una LCS de un prefijo de x y un prefijo de y

Resolviendo recursivamente

Algoritmo

```
n = len(x)
m = len(y)
C = n+1xm+1
```

$$
\begin{aligned}
&\mathrm{LCS}(x, y, i, j)\\
&\quad \textbf{if } x[i] = y[j]\\
&\qquad \textbf{then } c[i,j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1\\
&\qquad \textbf{else } c[i,j] \leftarrow \max\{\mathrm{LCS}(x, y, i-1, j),\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathrm{LCS}(x, y, i, j-1)\}
\end{aligned}
$$

Peor caso:

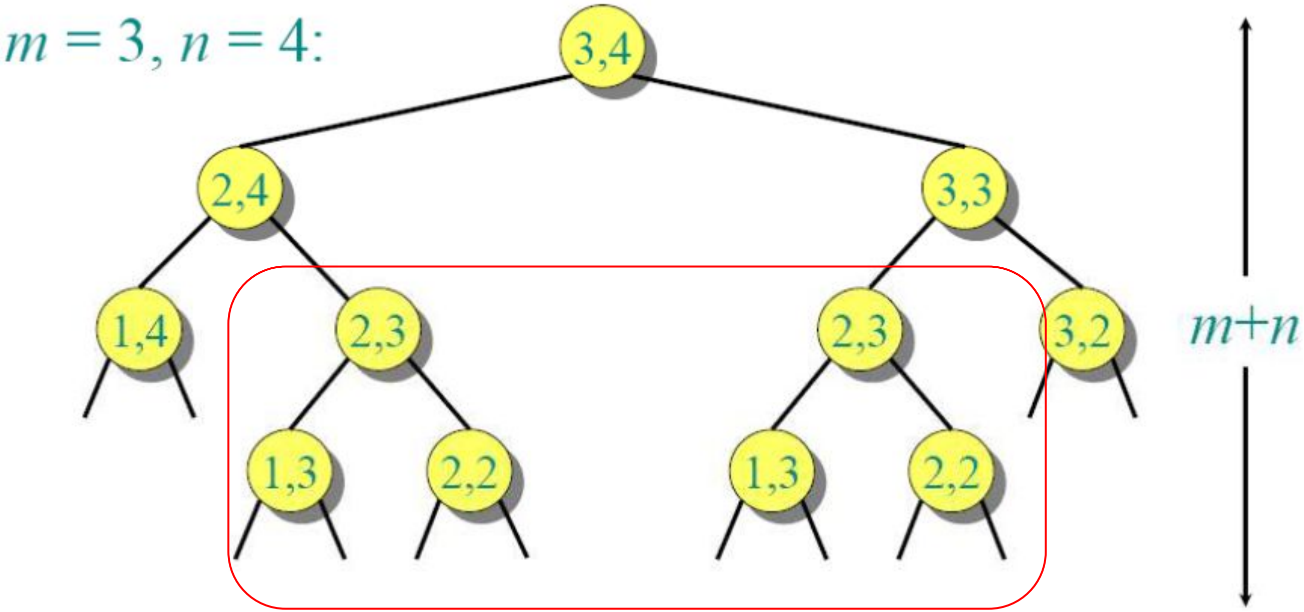
- x[i] != y[j] , el algoritmo evalua dos subproblemas, cada uno con sólo un parámetro decrementado

Implementación LCS recursivo

```
n= len(A)
m= len(B)
C = [[0]*(m+1)]*(n+1)
def LCS ( A,B, i , j):
  if (i<0):
    return 0
  if (j<0):
    return 0

  if (A[i]==B[j]):
    C[i][j] = LCS(A,B,i-1,j-1) + 1
    return C[i][j]
  else:
    return max(LCS(A,B,i-1,j), LCS(A,B,i,j-1))
```

Resolviendo recursivamente

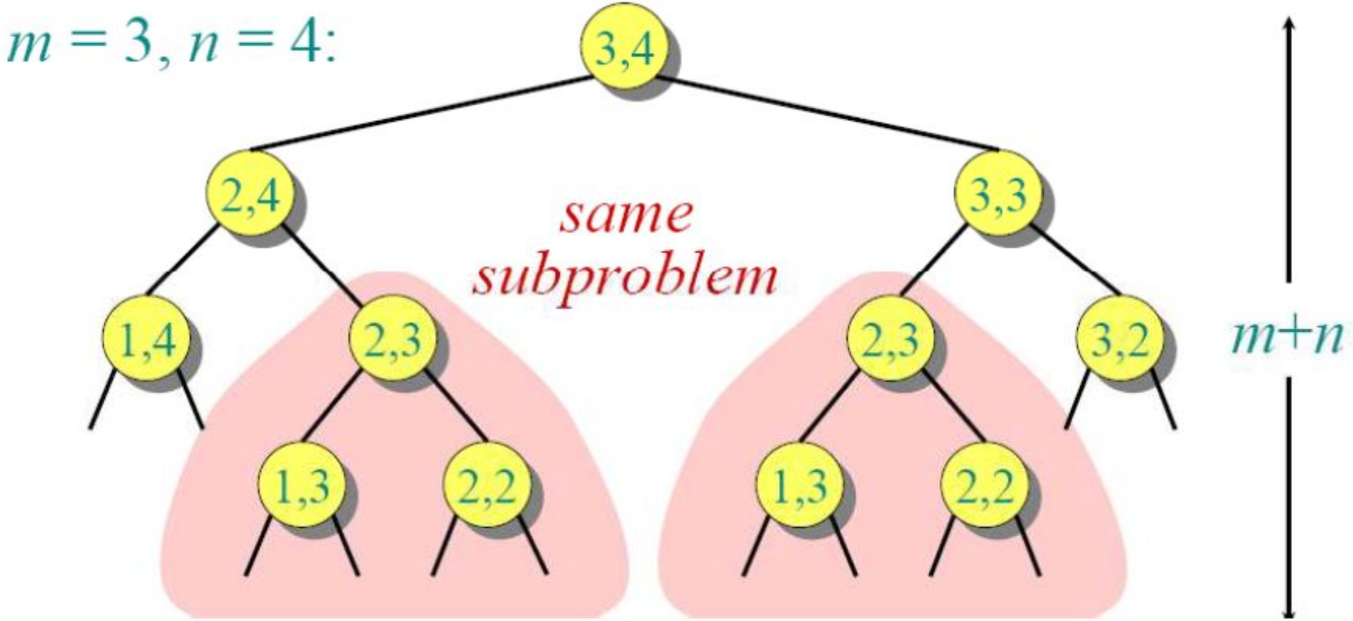
- Árbol de Recursión



- Altura = m +n . Funciona potencialmente exponencial

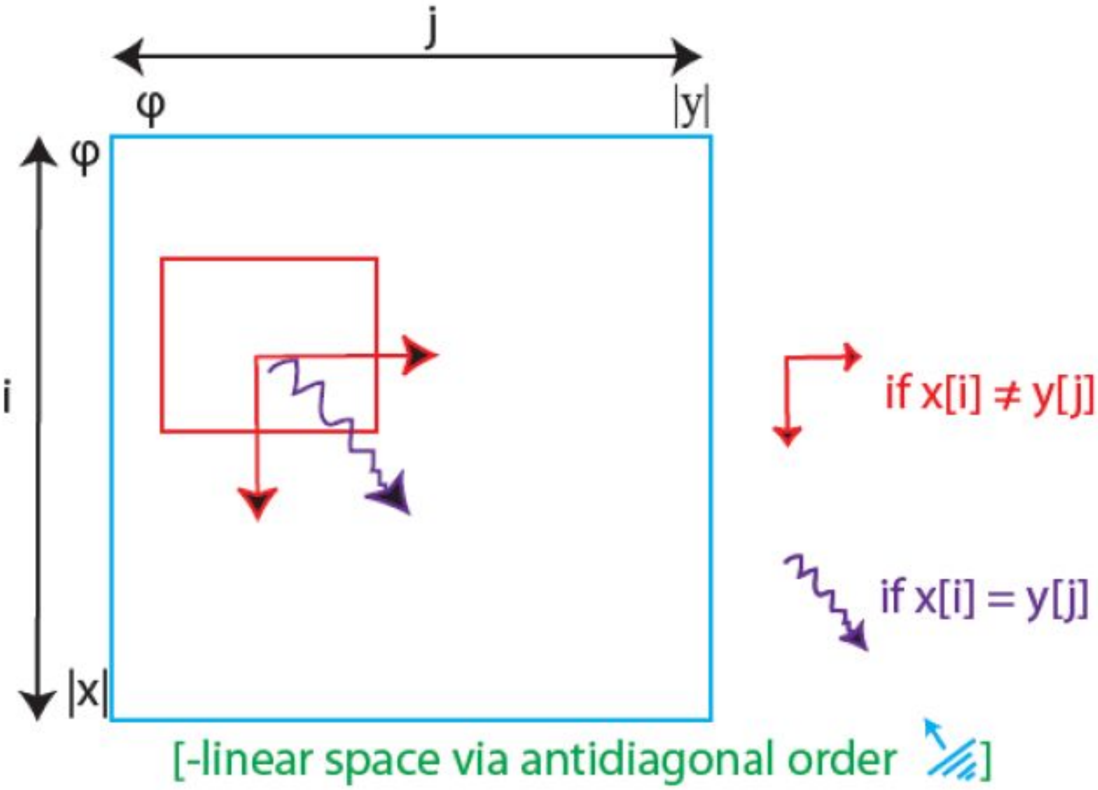
Resolviendo recursivamente

- Árbol de Recursión



- Altura = m +n . Funciona potencialmente exponencial, y se estas resolviendo problemas ya resueltos

Programación dinámica. Tabla



Algoritmo con PD

```
for (int i = 1; i <= A.length; i++) {
        for (int j = 1; j <= B.length; j++) {
                if (A[i - 1] == B[j - 1]) {
                        LCS[i][j] = LCS[i - 1][j - 1] + 1;
                } else {
                        LCS[i][j] = Math.max(LCS[i - 1][j], LCS[i][j - 1]
                }
        }
}
return LCS[A.length][B.length];
```

Algoritmo con PD

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	0+1 = 1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	0 + 1=1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Algoritmo con PD: BCBA

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Algoritmo con PD

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = $\theta(mn)$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Algoritmo con PD

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = ***θ(mn)***

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	$\boxed{3}$	3	3	4
A	0	1	2	2	3	3	$\boxed{4}$	4

Algoritmo con PD

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = ***θ(mn)***

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	$\boxed{3}$	3	3	4
A	0	1	2	2	3	3	$\boxed{4}$	4

Algoritmo con PD

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = $\theta(mn)$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Algoritmo con PD

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = $\theta(mn)$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Algoritmo con PD

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = $\theta(mn)$
  - Reconstruir la LCS mediante trazos hacia atrás.
  - Espacio = $\theta(mn)$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

		S	P	R	I	N	G	T	I	M	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	1	1	1	1	1	1	1	1	1
I	0	0	1	1	2	2	2	2	2	2	2
O	0	0	1	1	2	2	2	2	2	2	2
N	0	0	1	1	2	3	3	3	3	3	3
E	0	0	1	1	2	3	3	3	3	3	4
E	0	0	1	1	2	3	3	3	3	3	4
R	0	0	1	2	2	3	3	3	3	3	4

Carrizales, Agüero

		P	I	O	N	E	E	R
	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0
P	0	1	0	0	0	0	0	0
R	0	1	0	0	0	0	0	1
I	0	1	2	0	0	0	0	1
N	0	1	2	2	1	1	1	1
G	0	1	2	2	2	2	2	2
T	0	1	2	2	2	2	2	2
I	0	1	2	2	2	2	2	2
M	0	1	2	2	2	2	2	2
E	0	1	2	2	2	3	3	3

Ramos, Ruiz

2: horseback, snowflake

Rosales y Mamani

		H	O	R	S	E	B	A	C	K
	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	1	1	1	1	1	1
N	0	0	0	0	1	1	1	1	1	1
O	0	0	1	1	1	1	1	1	1	1
W	0	0	1	1	1	1	1	1	1	1
F	0	0	1	1	1	1	1	1	1	1
L	0	0	1	1	1	1	1	1	1	1
A	0	0	1	1	1	1	1	2	2	2
K	0	0	1	1	1	1	1	2	2	3
E	0	0	1	1	1	2	2	2	2	3

2: horseback, snowflake

Quispe y madueño

	A	B	C	D	E	F	G	H	I	J	K
1			h	o	r	s	e	b	a	c	k
2		0	0	0	0	0	0	0	0	0	0
3	s	0	0	0	0	1	1	1	1	1	1
4	n	0	0	0	0	1	1	1	1	1	1
5	o	0	0	1	1	1	1	1	1	1	1
6	w	0	0	1	1	1	1	1	1	1	1
7	f	0	0	1	1	1	1	1	1	1	1
8	l	0	0	1	1	1	1	1	1	1	1
9	a	0	0	1	1	1	1	1	2	2	2
10	k	0	0	1	1	1	1	1	2	2	3
11	e	0	0	1	1	1	2	2	2	2	3

3: stone, longest

		S	T	O	N	E
	0	0	0	0	0	0
L	0	0	0	0	0	0
O	0	0	0	1	1	1
N	0	0	0	1	2	2
G	0	0	0	1	2	2
E	0	0	0	1	2	3
S	0	1	1	1	2	3
T	0	1	2	2	2	3

Acuña, Rojas

4: maelstrom , becalm

		b	e	c	a	l	m
		0	0	0	0	0	0
m	0	0	0	0	0	0	1
a	0	0	0	0	1	0	1
e	0	0	1	1	1	1	1
l	0	1	1	1	1	2	2
s	0	1	1	1	1	2	2
t	0	1	1	1	1	2	2
r	0	1	1	1	1	2	2
o	0	1	1	1	1	2	2
m	0	1	1	1	1	2	3

Arteaga, Infantes

5: AGACTGTC, TAGTCACG

Grados y Rivera

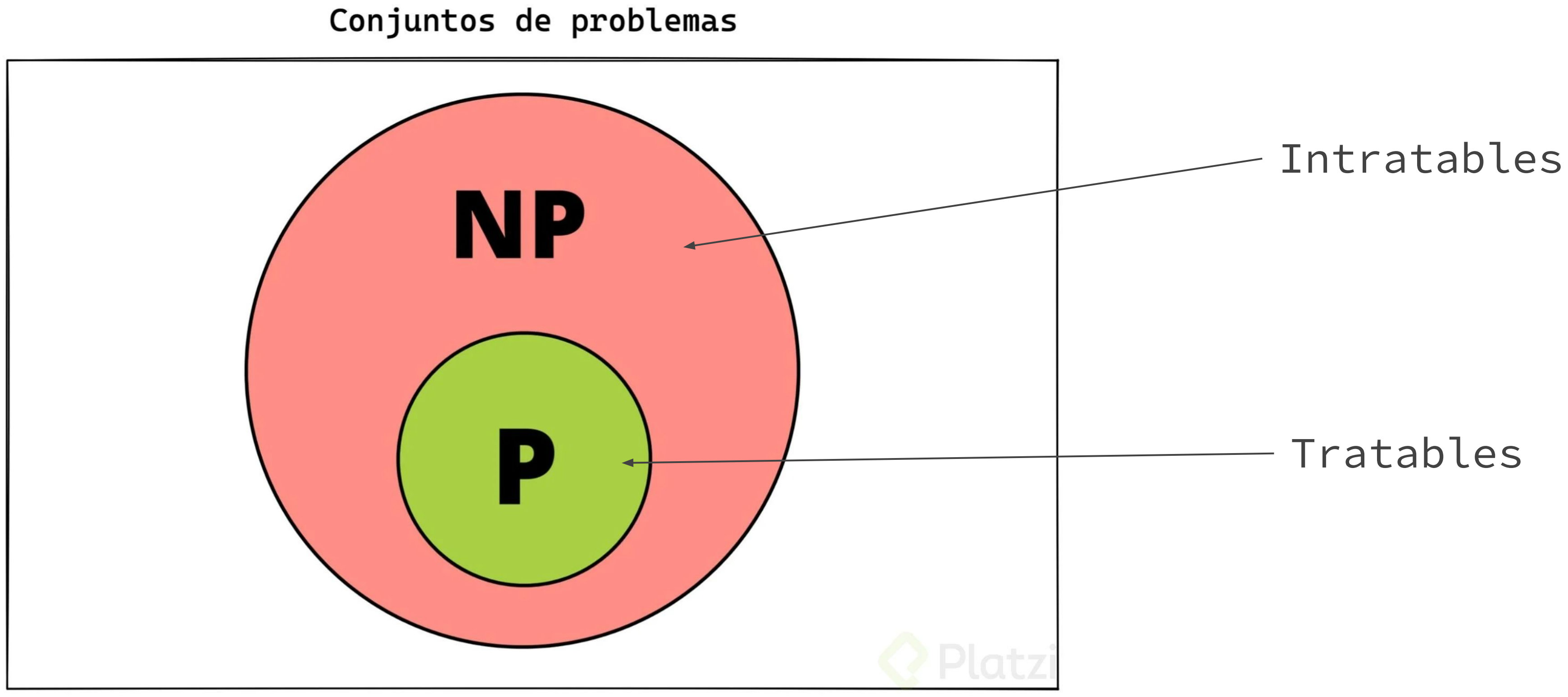
		T	A	G	T	C	A	C	G
	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
G	0	0	1	2	2	2	2	2	2
A	0	0	1	2	2	2	3	3	3
C	0	0	1	2	2	3	3	4	4
T	0	1	1	2	3	3	3	4	4
G	0	1	1	2	3	3	3	4	5
T	0	1	1	2	3	3	3	4	5
C	0	1	1	2	3	4	4	4	5

Formar la cadena

- IDEA:
  - Calcular la tabla bottom – up
  - Tiempo = $\theta(mn)$
  - Reconstruir la LCS mediante trazos hacia atrás.
  - Espacio = $\theta(mn)$

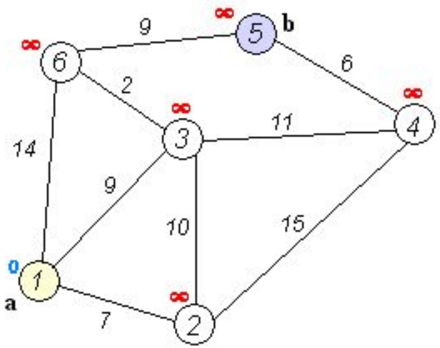
		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Tipos de problemas



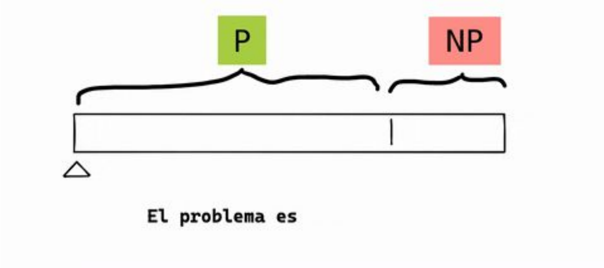
Clase P

- Algoritmos con complejidades: logaritmicos, línea logarítmicos, lineales, cuadráticos, cúbicos, incluso algun polinomico.
- Ejemplo: algoritmo de dijsktra

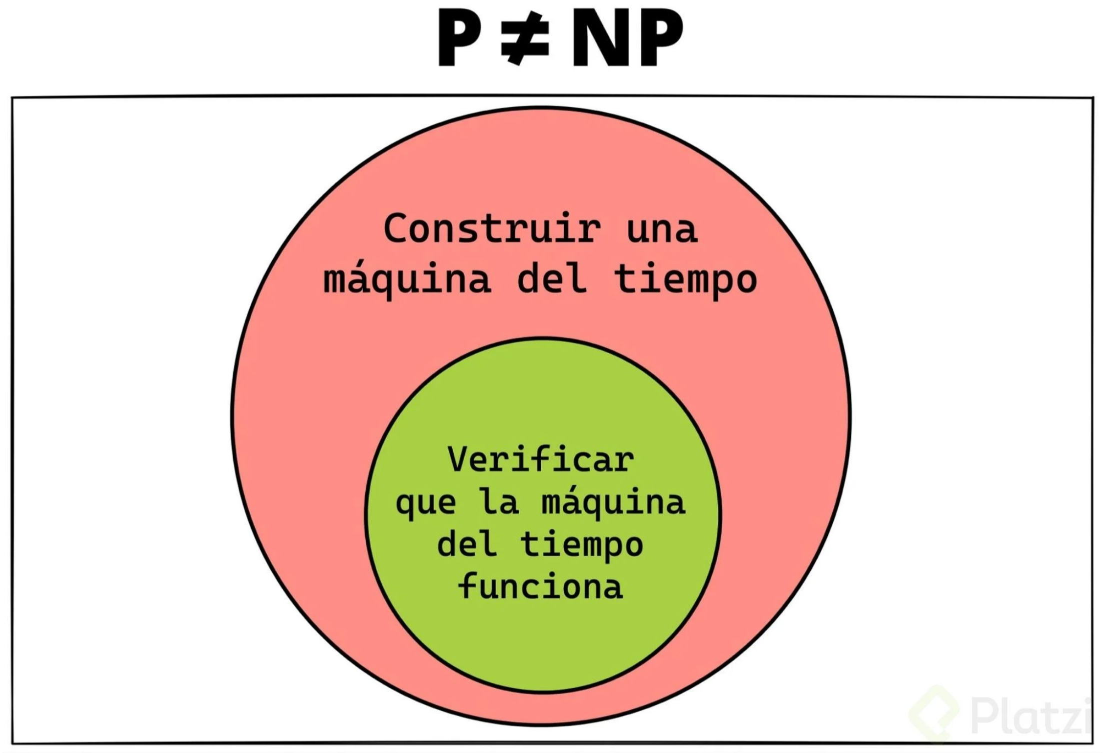


Clase NP

- La clase NP está compuesta por problemas posibles de solucionar hasta los intratables.
- La única forma que tengan un tiempo polinómico es realizando una etapa aleatoria para que la instancia del problema se ajuste al mejor caso.
- No-Determinista significa que a veces tenemos que probar muchas posibles soluciones antes de encontrar la correcta.



P != NP



¿Pueden ser
iguales?