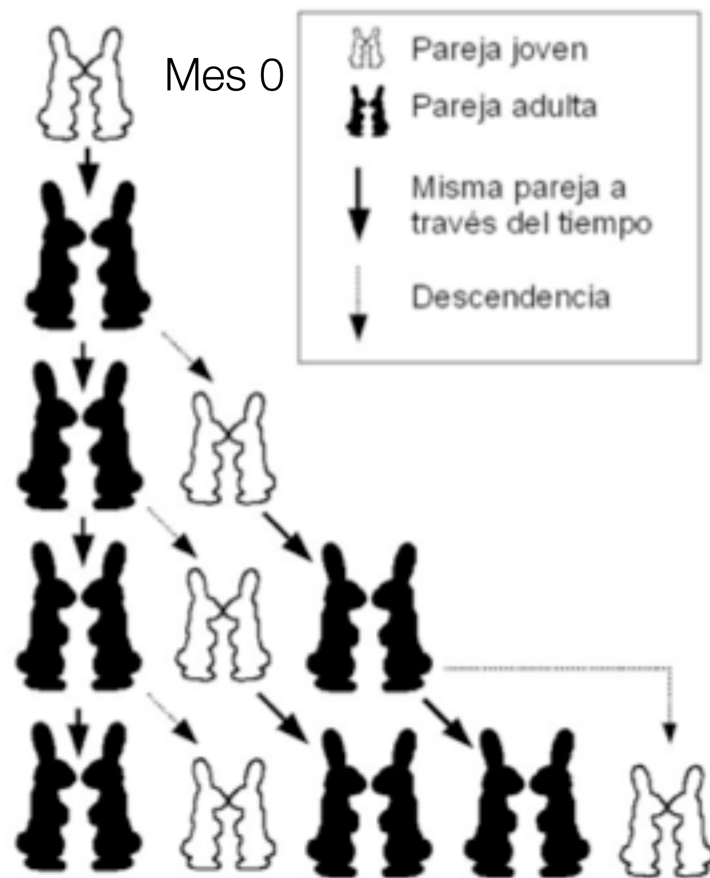
Recursividad, recursividad con memoria y un poco de programación dinámica.

- mat-151

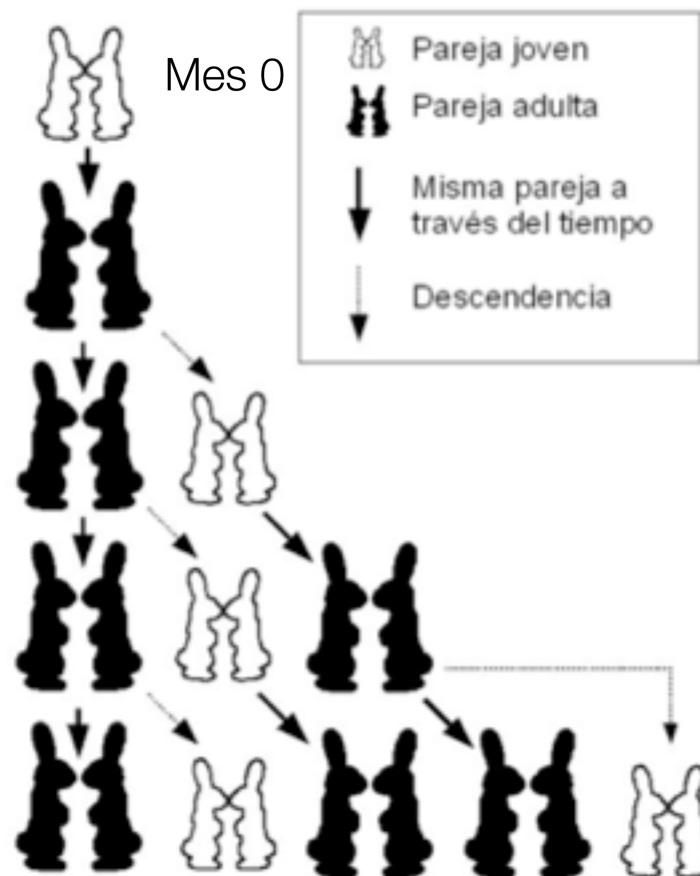
Los conejos y la serie de Fibonacci

La serie de Fibonacci indica cuantas parejas adultas hay en un mes dado



Los conejos y la serie de Fibonacci

La serie de Fibonacci indica cuantas parejas adultas hay en un mes dado



Si en un mes se tienen **a** parejas jóvenes y **b** parejas adultas, al siguiente mes se tendrán **a + b** parejas adultas y **b** parejas jóvenes. Por lo tanto, el número de conejos adultos en un mes **n**, es el número de conejos adultos en el mes **n-1** más el número de conejos jóvenes en el mes **n-1**.

Como el número de conejos jóvenes en el mes **n-1** es el número de conejos adultos en el mes **n-2**, entonces podemos concluir que

Los conejos y la serie de Fibonacci

La serie de Fibonacci indica cuantas parejas adultas hay en un mes dado

Si en un mes se tienen **a** parejas jóvenes y **b** parejas adultas, al siguiente mes se tendrán **a + b** parejas adultas y **b** parejas jóvenes. Por lo tanto, el número de conejos adultos en un mes **n**, es el número de conejos adultos en el mes **n-1** más el número de conejos jóvenes en el mes **n-1**.

Como el número de conejos jóvenes en el mes **n-1** es el número de conejos adultos en el mes **n-2**, entonces podemos concluir que

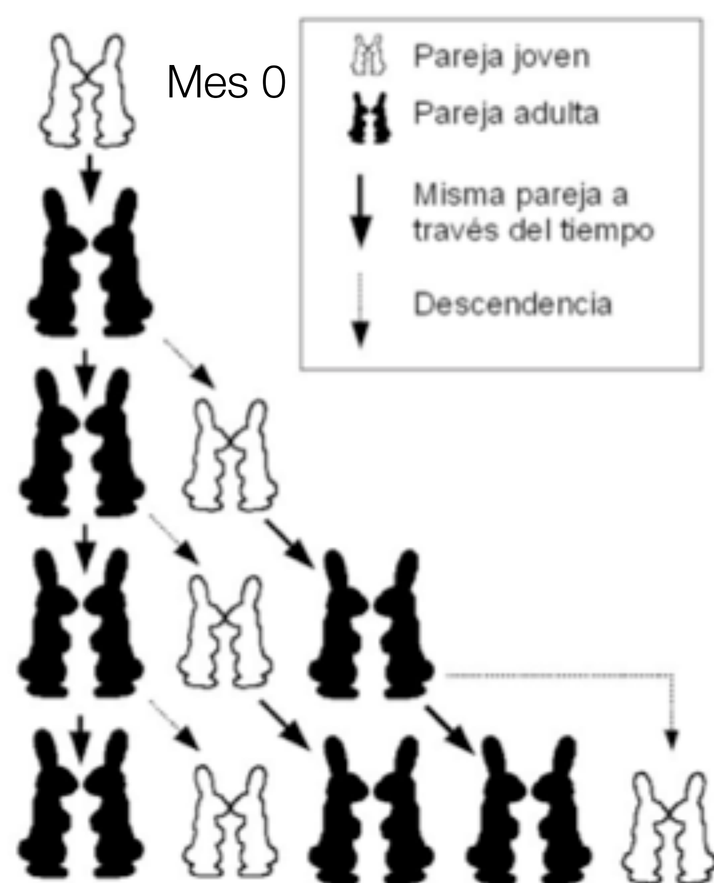
$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

Los conejos y la serie de Fibonacci

La serie de Fibonacci indica cuantas parejas adultas hay en un mes dado



Si en un mes se tienen **a** parejas jóvenes y **b** parejas adultas, al siguiente mes se tendrán **a + b** parejas adultas y **b** parejas jóvenes. Por lo tanto, el número de conejos adultos en un mes **n**, es el número de conejos adultos en el mes **n-1** más el número de conejos jóvenes en el mes **n-1**.

Como el número de conejos jóvenes en el mes **n-1** es el número de conejos adultos en el mes **n-2**, entonces podemos concluir que

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

Serie de números de Fibonacci

Serie de números de Fibonacci

- Version recursiva

```
int F_rec (int i){
    if(i<1) return 0;
    if(i==1) return 1;
    return F_rec(i-1)+ F_rec(i-2);

}
```

Serie de números de Fibonacci

- Version recursiva

```
int F_rec (int i){
    if(i<1) return 0;
    if(i==1) return 1;
    return F_rec(i-1)+ F_rec(i-2);

}
```

¿Cuál es el problema con este código?

Serie de números de Fibonacci (version 2)

- Version recursiva con memoria

```
#define maxF 100
int F_recMem(int i){
    static int knownF[maxF];
    if(knownF[i] !=0 ) return knownF[i];

    int t=i;
    if(i<0) return 0;
    if(i>1) t =  F_recMem (i-1)+F_recMem (i-2);

    return knownF[i] = t;
}
```

Serie de números de Fibonacci ¿Existe una tercera y cuarta versión mejor?

Serie de números de Fibonacci ¿Existe una tercera y cuarta versión mejor?

- Versión iterativa con un vector de almacenamiento

Serie de números de Fibonacci ¿Existe una tercera y cuarta versión mejor?

- Versión iterativa con un vector de almacenamiento

```
int *F ;
...
int F_iter(int i){

    F[0] = 0; F[1] =1;
    for(int k=2;k<=i;k++)
        F[k] = F[k-1] + F[k-2];

    return F[i];
}
```

El Problema de la bolsa del ladrón entero

Supongase que un ladrón se encuentra en una casa con muchas instancias de N objetos diferentes i=1,...N, cada objeto tiene un tamaño $size_i$ y un valor val_i

El ladrón tiene una bolsa de tamaño finito *sizeBag* de tal forma que tiene que decidir cuantos objetos de cada tipo x_i (con x_i entero no negativo) va a robarse para maximizar el valor del robo, es decir, la tarea es:

sujeto a:

El Problema de la bolsa del ladrón entero

Supongase que un ladrón se encuentra en una casa con muchas instancias de *N* objetos diferentes *i*=1,...N, cada objeto tiene un tamaño $size_i$ y un valor val_i

El ladrón tiene una bolsa de tamaño finito *sizeBag* de tal forma que tiene que decidir cuantos objetos de cada tipo x_i (con x_i entero no negativo) va a robarse para maximizar el valor del robo, es decir, la tarea es:

$$\max \sum_{i}^{N} x_i val_i$$

sujeto a:

El Problema de la bolsa del ladrón entero

Supongase que un ladrón se encuentra en una casa con muchas instancias de N objetos diferentes i=1,...N, cada objeto tiene un tamaño $size_i$ y un valor val_i

El ladrón tiene una bolsa de tamaño finito *sizeBag* de tal forma que tiene que decidir cuantos objetos de cada tipo x_i (con x_i entero no negativo) va a robarse para maximizar el valor del robo, es decir, la tarea es:

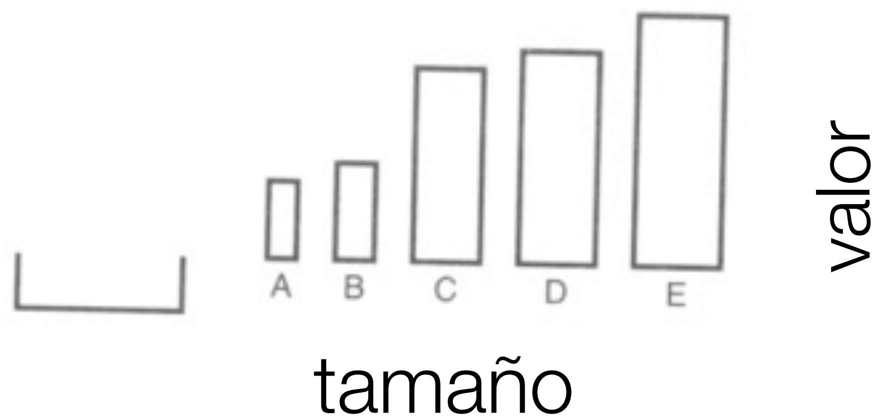
$$\max \sum_i^N x_i val_i$$

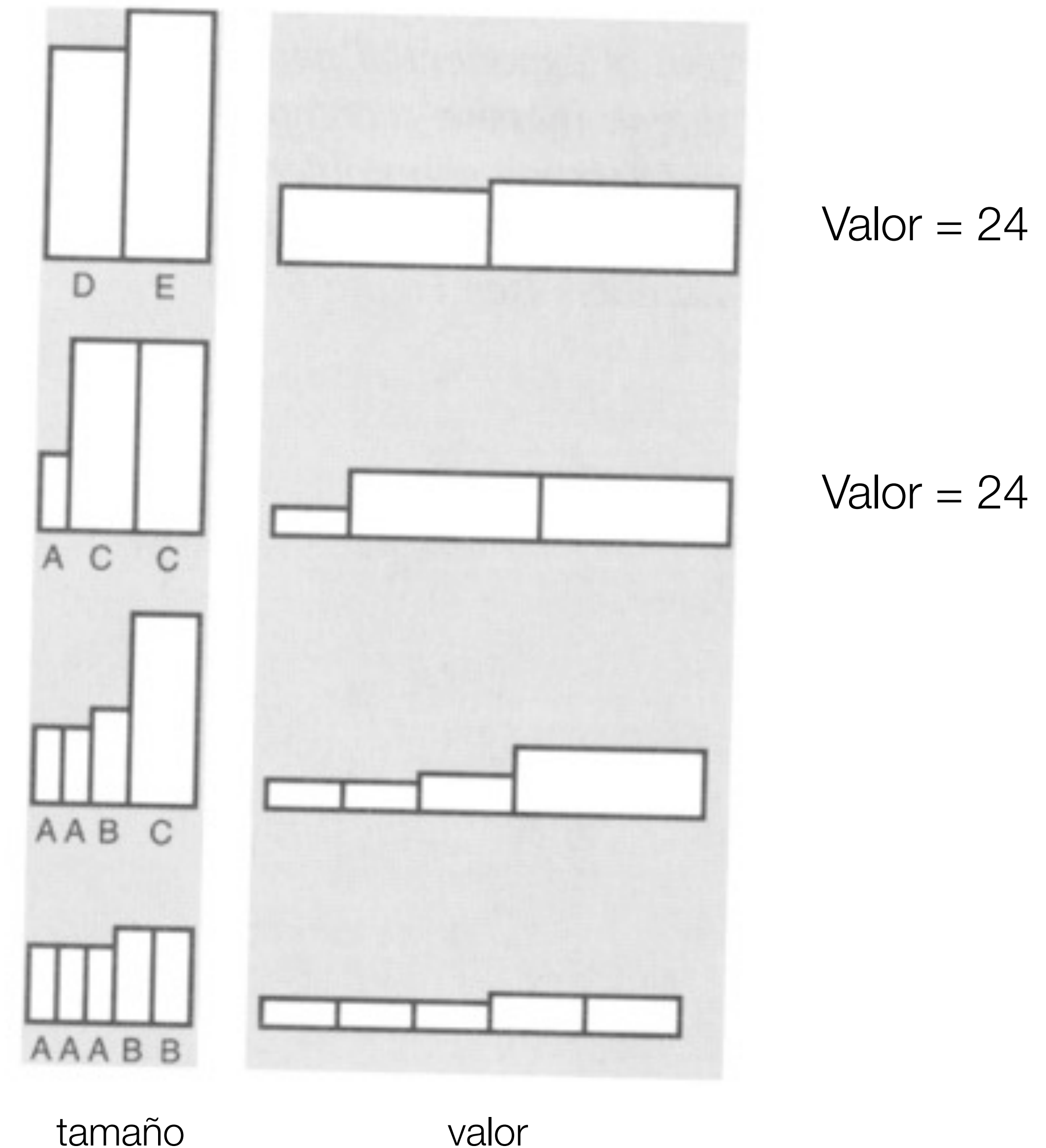
sujeto a:

$$\sum_i^N x_i size_i \leq sizeBag$$

El Problema de la bolsa del ladrón entero, ejemplo

Para una bolsa de tamaño 17





El Problema de la bolsa del ladrón, implementación.

El Problema de la bolsa del ladrón, implementación.

```
typedef struct {char name; int size; int val;} Item;

#define N 5
Item items[N];

int knap(int cap){
    int i,space,max,t;
    int maxi=-1;

    for(i=0,max=0;i<N;i++)
        if( (space = cap-items[i].size)>=0 )
            if( (t = knap(space) + items[i].val )> max )
                {max = t; maxi = i;}
    return max;
}
```

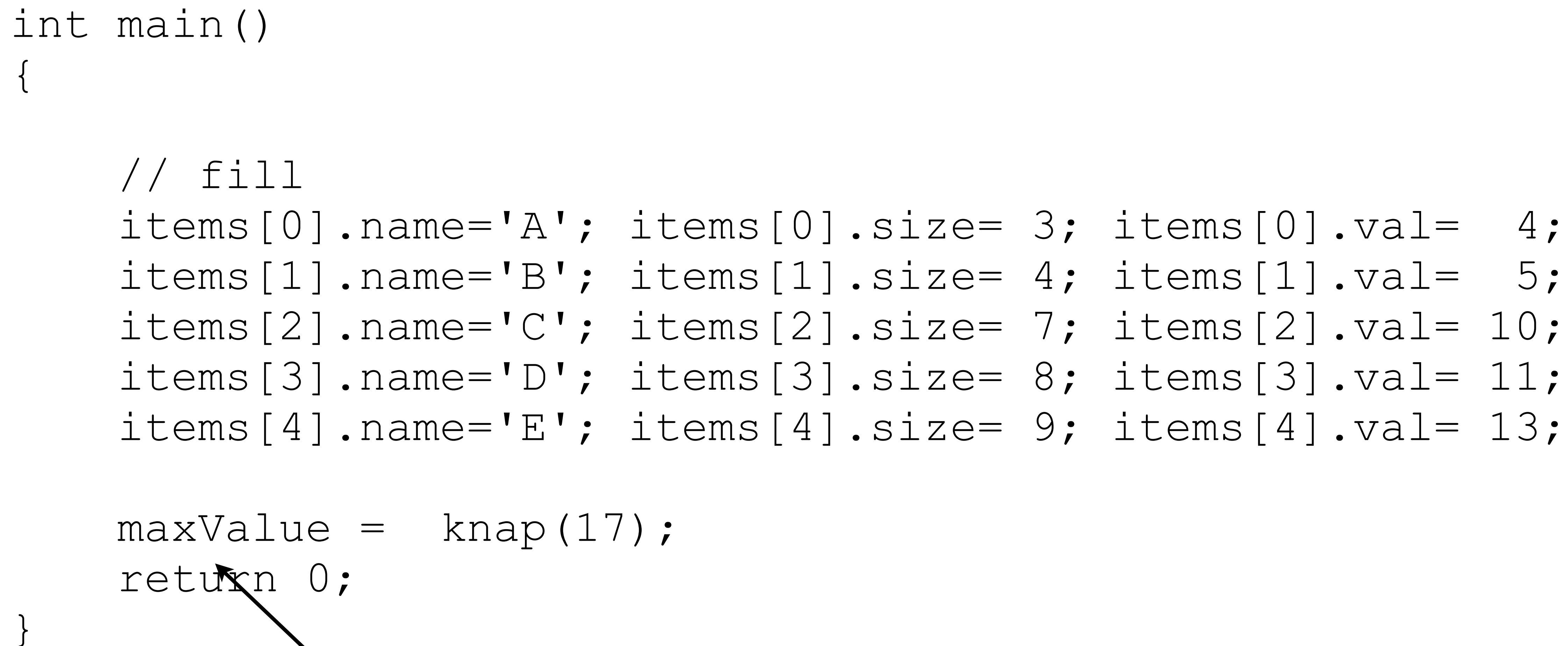
El Problema de la bolsa del ladrón, usando la función:

```
int main()
{

    // fill
    items[0].name='A'; items[0].size= 3; items[0].val=  4;
    items[1].name='B'; items[1].size= 4; items[1].val=  5;
    items[2].name='C'; items[2].size= 7; items[2].val= 10;
    items[3].name='D'; items[3].size= 8; items[3].val= 11;
    items[4].name='E'; items[4].size= 9; items[4].val= 13;

    maxValue =  knap(17);
    return 0;
}
```

El Problema de la bolsa del ladrón, usando la función:



```
int main()
{

    // fill
    items[0].name='A'; items[0].size= 3; items[0].val=  4;
    items[1].name='B'; items[1].size= 4; items[1].val=  5;
    items[2].name='C'; items[2].size= 7; items[2].val= 10;
    items[3].name='D'; items[3].size= 8; items[3].val= 11;
    items[4].name='E'; items[4].size= 9; items[4].val= 13;

    maxValue =  knap(17);
    return 0;
}
```

El Problema de la bolsa del ladrón, usando la función:

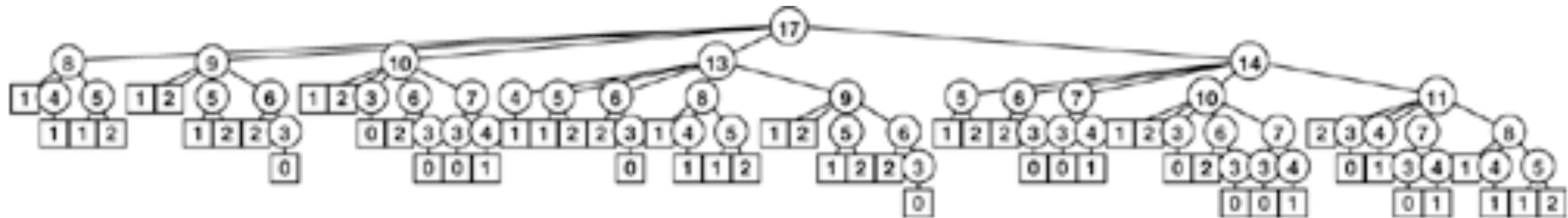
```
int main()
{

    // fill
    items[0].name='A'; items[0].size= 3; items[0].val=  4;
    items[1].name='B'; items[1].size= 4; items[1].val=  5;
    items[2].name='C'; items[2].size= 7; items[2].val= 10;
    items[3].name='D'; items[3].size= 8; items[3].val= 11;
    items[4].name='E'; items[4].size= 9; items[4].val= 13;

    maxValue =  knap(17);
    return 0;
}
```

OJO, esto SOLO nos inica el valor máximo de lo que podemos llevar

Árbol de llamadas recursivas para tamaño 17



	0	1	2	3	4
item	A	B	C	D	E
size	3	4	7	8	9
val	4	5	10	11	13

Expresiones infijas y posfijas y como evaluarlas

- mat-151

Problema: evaluar una expresion infija

- Por ejemplo :

  5 * (((9 + 8) * (4 * 6)) + 7)

La idea es calcular los valores intermedios, por ejemplo 9+8, 4*6, sustituir y regresar a calcular.

¿Que pasa si tenemos la misma expresión en post-fijo “(9+8)” -> “9 8 +”?

5 9 8 + 4 6 * * 7 + *

Problema: evaluar una expresion infija

- Pasando de una expresion posfija a una infija (“a b *” -> “(a*b)”)

```
5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) ( 4 * 6 ) * 7 + *
5 ( ( 9 + 8 ) * ( 4 * 6 ) ) 7 + *
5 ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) *
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )
```

Notar que en la posfija no necesitabamos parentesis

Problema: evaluar una expresion infija

- Pasando de una expresion posfija a una infija (“a b *” -> “(a*b)”   )

```
5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) ( 4 * 6 ) * 7 + *
5 ( ( 9 + 8 ) * ( 4 * 6 ) ) 7 + *
5 ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) *
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )
```

Notar que en la posfija no necesitabamos parentesis

Problema: evaluar una expresion infija

- Pasando de una expresion posfija a una infija (“a b *” -> “(a*b)”)

```
5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) ( 4 * 6 ) * 7 + *
5 ( ( 9 + 8 ) * ( 4 * 6 ) ) 7 + *
5 ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) *
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )
```

Notar que en la posfija no necesitabamos parentesis

Problema: evaluar una expresion infija

- Pasando de una expresion posfija a una infija (“a b *” -> “(a*b)”)

```
5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) ( 4 * 6 ) * 7 + *
5 ( ( 9 + 8 ) * ( 4 * 6 ) ) 7 + *
5 ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) *
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )
```

Notar que en la posfija no necesitabamos parentesis

Problema: evaluar una expresion infija

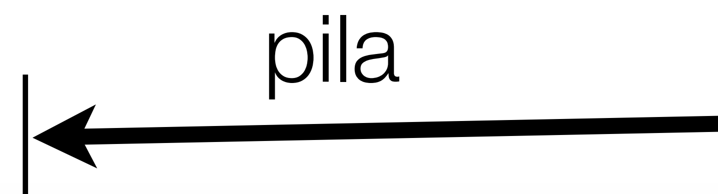
- Pasando de una expresion posfija a una infija (“a b *” -> “(a*b)”   )

```
5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) ( 4 * 6 ) * 7 + *
5 ( ( 9 + 8 ) * ( 4 * 6 ) ) 7 + *
5 ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) *
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )
```

Notar que en la posfija no necesitabamos parentesis

Problema: evaluar una expresion infija (2do paso)

- Paso 2: ¿Cómo evaluamos una expresion posfija?:



De izquierda a derecha (de arriba a abajo):

-Es numero, lo insertamos en una pila

-Es operador, sacamos los dos últimos números y los operamos con él, metemos el resultado a la pila

- El último número en la pila es el resultado total

5	5			
9	5	9		
8	5	9	8	
+	5	17		
4	5	17	4	
6	5	17	4	6
*	5	17	24	
*	5	408		
7	5	408	7	
+	5	415		
*	2075			

Problema: evaluar una expresion infija, (2do paso)

- Como evaluamos una expresión posfija:

```
#include <iostream.h>
#include <string.h>
#include "STACK.cxx"
int main(int argc, char *argv[])
  { char *a = argv[1]; int N = strlen(a);
    STACK<int> save(N);
    for (int i = 0; i < N; i++)
      {
        if (a[i] == '+')
          save.push(save.pop() + save.pop());
        if (a[i] == '*')
          save.push(save.pop() * save.pop());
        if ((a[i] >= '0') && (a[i] <= '9'))
          save.push(0);
        while ((a[i] >= '0') && (a[i] <= '9'))
          save.push(10*save.pop() + (a[i++]-'0'));
      }
    cout << save.pop() << endl;
  }
```

Problema: evaluar una expresion infija, (2do paso)

- Como evaluamos una expresión posfija:

```
#include <iostream.h>
#include <string.h>
#include "STACK.cxx"
int main(int argc, char *argv[])
  { char *a = argv[1]; int N = strlen(a);
    STACK<int> save(N);
    for (int i = 0; i < N; i++)
      {
        if (a[i] == '+')
          save.push(save.pop() + save.pop());
        if (a[i] == '*')
          save.push(save.pop() * save.pop());
        if ((a[i] >= '0') && (a[i] <= '9'))
          save.push(0);
        while ((a[i] >= '0') && (a[i] <= '9'))
          save.push(10*save.pop() + (a[i++]-'0'));
      }
    cout << save.pop() << endl;
  }
```

Problema: evaluar una expresion infija, (1er paso)

- Si tenemos expresiones infijas (naturales) ¿cómo las evaluamos?:

¡Las convertimos de infija a posfija!

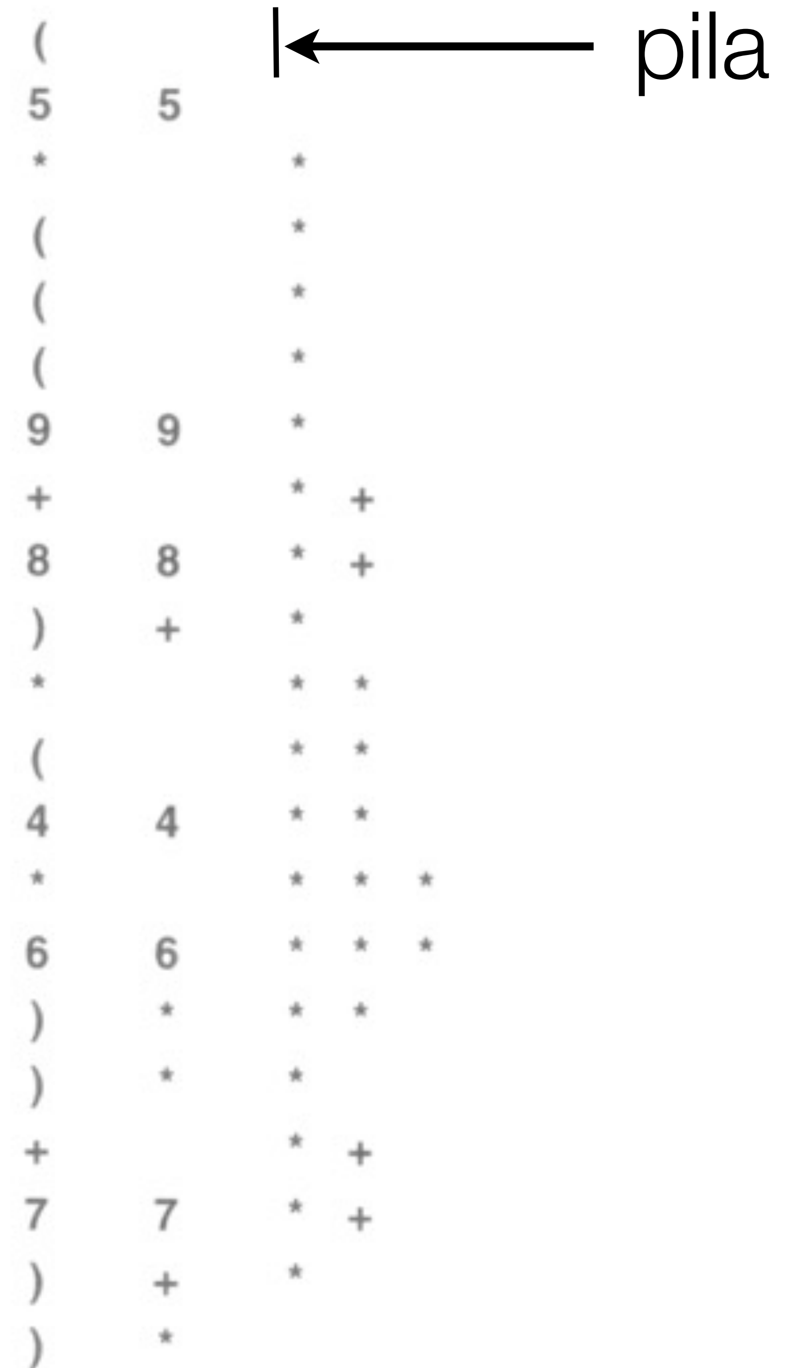
De izquierda a derecha:

-Numero son escritos a la cadena posfija

-Paréntesis izquierdos ignorados

-Operadores son insertados en la pila

-Paréntesis derecho, sacamos operador del tope y lo escribimos a la salida.



Problema: evaluar una expresion infija (1er paso)

- Convirtiendo de infijo a posfijo

```
#include <iostream.h>
#include <string.h>
#include "STACK.cxx"
int main(int argc, char *argv[])
  { char *a = argv[1]; int N = strlen(a);
    STACK        ops(N);
    for (int i = 0; i < N; i++)
      {
        if (a[i] == ’)’)
          cout << ops.pop() << " ";
        if ((a[i] == ’+’) || (a[i] == ’*’))
          ops.push(a[i]);
        if ((a[i] >= ’0’) && (a[i] <= ’9’))
          cout << a[i] << " ";
      }
    cout << endl;
  }
```

Problema: evaluar una expresion infija (1er paso)

- Convirtiendo de infijo a posfijo

```
#include <iostream.h>
#include <string.h>
#include "STACK.cxx"
int main(int argc, char *argv[])
  { char *a = argv[1]; int N = strlen(a);
    STACK<char> ops(N);
    for (int i = 0; i < N; i++)
      {
        if (a[i] == ')')
          cout << ops.pop() << " ";
        if ((a[i] == '+') || (a[i] == '*'))
          ops.push(a[i]);
        if ((a[i] >= '0') && (a[i] <= '9'))
          cout << a[i] << " ";
      }
    cout << endl;
  }
```