Introducción

Calcular números de Fibonacci

Si utilizamos una algoritmo recursivo «directo» para calcular los números de la sucesión de Fibonacci, estamos en problemas pues se necesitaría una cantidad exponencial de tiempo para que el algoritmo nos diera el valor del Fibonacci F_n.

Puedes verificar esto corriendo el siguiente código. Aproximadamente para $n=37$ y mayores ya empieza a tardarse demasido.

```
def rec_fibo(n):
    if n==0:
        return(0)
    if n==1:
        return(1)
    if n>1:
        return(rec_fibo(n-1)+rec_fibo(n-2))

rec_fibo(37)
```

```
24157817
```

El algoritmo tarda tanto cuando está implementado así pues cada vez que necesita un Fibonacci, lo vuelve a calcular recursivamente y esto hace que se hagan demasidas cuentas.

Una mejor idea es ir almacenando cosas que ya hemos calculado. Esto lo podemos hacer de varias formas, por ejemplo, podemos almacenar los valores de Fibonacci que vayamos necesitando en un arreglo. Esto lo podemos hacer de dos formas distintas:

- Una primera es pensar de arriba hacia abajo (**memoización**): nos preguntamos primero por F_n y de ahí, qué para calcularlo, y de ahí que necesitamos para calcular esos valores, etc. Cuando hayamos calculado ya algo, lo almacenamos.
- Otra es pensar de abajo hacia arriba (**tabulación**): sabemos F_0 y F_1, y de ahí calculamos F_2, y luego F_3 y así sucesivamente (y los almacenando). Continuamos hasta llegar a nuestro objetivo.

Hagamos implementaciones pensando en estas dos maneras de ir almacenando la información.

```
F=[0,1]

def mem_fibonacci(n,F):
    if len(F)>n:
        return(F[n])
    else:
        new_fibonacci=mem_fibonacci(n-1,F)+mem_fibonacci(n-2,F)
        F.append(new_fibonacci)
        return(mem_fibonacci(n-1,F)+mem_fibonacci(n-2,F))

for j in range(10):
    print(mem_fibonacci(j,F))
```

```
0
1
1
2
3
5
8
13
21
34
```

Este algoritmo sigue siendo recursivo, pero va almacenando con una estrategia de memoización, lo cual le permite correr mucho más rápido, en tiempo $O(n)$.

rápido, en tiempo $O(n)$.

Si ahora hacemos una implementación con tabulación, también vamos memorizando los valores de Fibonacci, pero ahora explícitamente desde abajo hacia arriba. La implementación se vería así:

```
def tab_fib(n):
    F=[0,1]
    for j in range(n-1):
        F.append(F[-1]+F[-2])
    return(F[n])

for j in range(10):
    print(tab_fib(j))

tab_fib(1000)
```

```
0
1
1
2
3
5
8
13
21
34
```

```
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

Esta implementación también corre en tiempo $O(n)$, aunque va calculando los números de Fibonacci de manera un poco diferente a la anterior.

Tanto la implementación de arriba por memoización, como esta por tabulación toman $O(n)$ espacio, pues en la lista F se van almacenando todo los números de Fibonacci.

Podemos mejorar esto. En vez de acordarnos de todo, todo lo que vamos calculando, basta con acordarnos de lo mínimo necesario para ir calculando los números de Fibonacci, es decir, en cada momento sólo recordamos los últimos dos. Una implementación de esto se vería así:

```
def fib(n):
    a=0
    b=1
    for j in range(n):
        b,a=a+b,b
    return(a)

for j in range(10):
    print(fib(j))
```

```
0
1
1
2
3
5
8
13
21
34
```

En esta implementación se usa la idea de acordarnos de cosas muy ligeramente, pues sólo nos acordamos de a y b, y entonces no es necesario usar $O(n)$ espacio, sino simplemente $O(1)$ espacio y sigue corriendo en tiempo $O(n)$.

Encontrar coeficientes binomiales

Cuando queremos calcular coeficientes binomiales tenemos una situación similar pues podemos calcularlos de manera recursiva, usando que $\binom{n}{0}=\binom{n}{n}=1$, que $\binom{0}{0}=1$ y que se cumple la identidad de Pascal:

\[\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}.\]

Una implementación directa de la fórmula de Pascal para convertirla en un algoritmo recursivo de nuevo tiene el problema de que muchos subproblemas se calculan repetidamente. Para evitar esto, podemos utilizar alguna de las técnicas de memorización vistas arriba. Por ejemplo, a continuación se muestra una implementación que usa tabulación.

```
def tab_cb(n,k):
    TP=[]
    for nivel in range(n+1):
        renglon=[]
        for elemento in range(nivel+1):
            if elemento==0 or elemento==nivel:
                renglon.append(1)
            else:
                renglon.append(TP[-1][elemento-1]+TP[-1][elemento])
        TP.append(renglon)
    return(TP[n][k])

for j in range(51):
    print(tab_cb(50,j))
```

```
1
50
1225
19600
230300
2118760
15890700
99884400
536878650
2505433700
10272278170
37353738800
121399651100
354860518600
937845656300
2250829575120
4923689695575
9847379391150
18053528883775
30405943383200
47129212243960
67327446062800
88749815264600
108043253365600
121548660036300
126410606437752
121548660036300
108043253365600
88749815264600
67327446062800
47129212243960
30405943383200
18053528883775
9847379391150
4923689695575
2250829575120
937845656300
354860518600
121399651100
37353738800
10272278170
2505433700
536878650
99884400
15890700
2118760
230300
19600
1225
50
1
```

Subsecuencias crecientes máximas

Dada una secuencia de números \(a_1,a_2,a_3,a_4,\ldots,a_n\) una sub-secuencia consiste de tomar algunos de ellos de izquierda a derecha.

Ejemplo. Si tenemos a la secuencia

\[10,15,8,4,2,3,1,9,12,21,5,7,\]
una posible subsecuencia es

una posible subsecuencia es
\[8,4,1,9,5.\]

Una secuencia es creciente cuando sus elementos de izquierda a derecha están en orden creciente.

Ejemplo. La secuencia

\[10,15,18,21,35\]
está en orden creciente. La secuencia
\[10,15,35,21,18\]
no está en orden creciente pues \(35\) está a la izquierda de \(18\) pero \(35>18\).

Problema. Dada una secuencia \(a_1,\ldots,a_n\), encontrar la subsecuencia creciente más grande.

Para plantear este problema de manera recursiva, nos conviene mucho más plantear un problema un poquito más general.

Problema. Dada una secuencia \(a_1,\ldots,a_n\) y un índice \(j\), encontrar la subsecuencia creciente más grande que termina en la posición \(j\).

Este es un mejor problema, pues se puede resolver en términos recursivos. La subsecuencia más grande que termina en la posición \(j\) es de tamaño \(1\) si \(a_j\) es menor que todos los anteriores y si es mayor que alguno de los anteriores entonces es uno más que la subsecuencia más grande anterior y que se pueda extender con \(a_j\).

Implementemos esto con programación dinámica, almacenando las respuestas de \(1\) a \(n\).

```
secuencia=[10,15,8,4,2,3,1,9,12,21,5,7]
auxiliar=[]
padres=[]

for j in range(len(secuencia)):
    nuevo=1
    padre=-1
    for k in range(j):
        if secuencia[k]<secuencia[j] and auxiliar[k]+1>nuevo:
            nuevo=auxiliar[k]+1
            padre=k
    auxiliar.append(nuevo)
    padres.append(padre)

print(auxiliar)
print(padres)

mayor_long=0
mayor_long_ind=-1
for j in range(len(auxiliar)):
    if auxiliar[j]>mayor_long:
        mayor_long_ind=j
        mayor_long=auxiliar[j]

print('La subsecuencia más grande termina en el índice {} y es de longitud {}'.format(mayor_long_ind,mayor_long))

subsecuencia=[]
indice=mayor_long_ind

while indice!=-1:
    subsecuencia=[secuencia[indice]]+subsecuencia
    indice=padres[indice]

print('Una posible subsecuencia más grande es {}'.format(subsecuencia))
```

```
[1, 2, 1, 1, 1, 2, 1, 3, 4, 5, 3, 4]
[-1, 0, -1, -1, -1, 4, -1, 5, 7, 8, 5, 10]
La subsecuencia más grande termina en el índice 9 y es de longitud 5
Una posible subsecuencia más grande es [2, 3, 9, 12, 21]
```

Reto. Hacer un algoritmo que encuentre todas las subsecuencias crecientes de longitud máxima.

Particiones de trabajo justas

Tarea moral

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Problema
2. Problema
3. Problema
4. Problema
5. Problema

Anterior
Más ejemplos de backtrack

Siguiente
Ideas probabilistas en diseño de algoritmos

Por Leonardo Ignacio Martínez Sandoval