Ayudantía 8 - Algoritmos y Complejidad
Programación Dinamica

Sassy Complexes

1. Repaso

1.1. Las definiciones recursivas son algoritmos recursivos

Para calcular un número de Fibonacci, por definicón formal hacemos una recursión, para esta recursión no podemos usar la estrategia voraz de descartar soluciones pues aquí estamos obligados a revisar todo el árbol.

Listing 1: Fibonacci Recursivo

```
1 int fib(int n){
2     if(n < 2){
3         return n;
4     }else{
5         return fib(n-1) +  fib(n-2)
6     }
7 }
```

Cuanto se tarda el señor Fibonacci?

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1$$

$$T(n) = 2 \cdot F_{n+1} - 1$$

Tiene *tight bound*:

$$\Theta\left(\phi^n\right) \text{ con } \phi = \frac{\sqrt{(5)}+1}{2} \approx 1{,}618033$$

Este algoritmo es lentisimo.

1.2. Del BackTracking a la Memo(r)izacion: Overlapping substructure

Se habra dado cuenta de que hay calculos que se vuelven a hacer varias veces, por la naturaleza recursiva de los números de fibonacci, podemos acelerar esto guardando los calculos en un diccionario y leyendolos de ahi, en vez de volver a hacer el calculo otra vez, esto se llama *Memoizacion*.

Listing 2: Fibonacci Recursivo con Memoizacion

```python
def fib(n):
    F = dict()
    if n < 2: return n
    else:
        if n not in F:
            F[n] = fib(n-1) + fib(n-2)
        return F[n]
```

Con esto estamos reduciendo el tiempo de calculo de la lesera, pero cuanto? notese que estamos calculando F(i) al menos una vez para cada i, ahora esto tiene complejidad O(n).

1.3. De la Memoizacion a la Programación Dinamica

COmo estamos evaluando cada F(i) una unica vez, es como hacer una iteración, cuando el diccionario se llene, podemos iterar sobre el.

Listing 3: Fibonacci Iterativo

```cpp
#include <vector>

int fib(int n){
    std::vector<int> F(n+1);
    F[0] = 0;
    F[1] = 1;
    for(std::size_t i = 2; i <= n; i++){
        F[i] = F[i-1] + F[i-2]
    }
    return F[n];
}
```

Complejidad Temporal? **O(n) sumas**.
Complejidad espacial? **O(n) enteros**.
Sin llamadas recursivas!

1.4. Memoizacion sin memoria

No es necesario recordar todo el diccionario, por que al final solo estamos usando los ultimos dos:

Listing 4: Fibonacci Iterativo con memoria a corto plazo

```
1   int fib(int n){
2       int prev = 1;
3       int curr = 0;
4       for(std::size_t i = 1; i <= n; i++){
5           int next = curr + prev;
6           prev = curr;
7           curr = next;
8       }
9       return curr;
10  }
```

1.4.1. Bonus: No puedes darle mas potencia?

Observe:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} prev \\ curr \end{bmatrix} = \begin{bmatrix} curr \\ prev + curr \end{bmatrix}$$

La iteración de arrba la escribimos como una matriz :)

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}$$

Elevar una matriz a la potencia n toma $O(\log n)$, y multiplicar las matrices 2×2. Esto nos da complejidad $O(\log n)$ multiplicaciones enteras. Admirelo un rato....(bueno, no tanto, los números empiezan a crecer muy rapido hasta un punto en que ya no se puede hacer aritmetica entera en tiempo constante (long long ints, entre otras leseras mas grandes que 64 bits, toma un rato))

2. Demostrando

Para demostrar correctitud uno primero demuestra factibilidad y luego optimalidad. Demostrar factibilidad es trivial por que se obvia el hecho de que el algoritmo retorna una solucion posible en el contexto del problema (ejemplo: si estoy avanzando por un camino, que el camino esté conectado, no voy teletransportandome). Demostrar optimalidad para los voraces es importante por que los voraces generalmente no son óptimos, el simple hecho de eliminar posibles soluciones hace que uno empieze a tener miedo y se ponga a demostrar cosas, pero con dinamica eso no pasa por que aca estamos revisando el problema completo, y por el solo hecho de revisar el problema completo ya hace al algoritmo óptimo. Revisamos todas la soluciones posibles y nos quedamos con la mejor, muy óptimo.
Aun así un algoritmo de programación dinamica cumple con:

Overlapping Subproblems Diferentes ramas de la recursion se pueden reciclar

Optimal Substructure Las soluciones optimas a los subproblemas componen la solución óptima al problema grande.

Polynomial Subproblems El numero de problemas es “pequeño” y se puede evaluar en tiempo polinomial.

También esta la manera Algoco de demostrar:

Inductive Substructure Resolver una rama de la recursion es independiente de las demas (que se pueda reciclar es otra cosa)

Optimal Substructure Las soluciones optimas a los subproblemas componen la solución óptima al problema grande.

Complete Choice Revisamos todas las ramas, nos quedamos con la mejor de todas las soluciones.

Usted nunca va a demostrar formalmente un algoritmo de dinamica en este ramo, asuma que es óptimo solo por que hay Complete Choice!!

3. Ejercicios

3.1. Ejemplo Clasico: Longest Increasing Subsequence (LIS)

Encuentre el largo de la subsecuencia mas larga de una secuencia dada, tal que los elementos de la subsecuencia esten ordenados y en orden incremental.

Ejemplo:

$$10,22,9,33,21,50,41,60$$

Tiene posibles subsecuencias:

$$\{10\},\{10,22\},\{10,9,33\},\{33,21,60\},\text{etc}$$

De las cuales, incrementales son:

$$\{10\},\{9,33,41\},\{33,41,60\},\{33,50,60\},\text{etc}$$

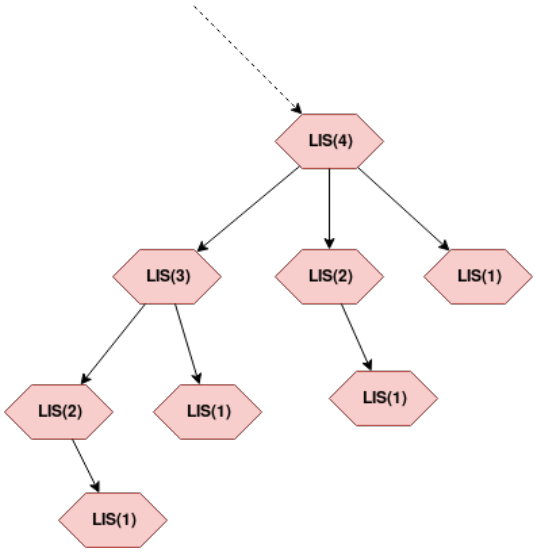
La subsecuencia incremental mas larga es:

$$\{10,22,33,50,60\},\{10,22,33,41,60\}\text{Claramente no hay una unica solución}$$

Todas son de largo 5, el largo mas largo.

3.1.1. Proponga un algoritmo:

Considere una posible solucion recursiva dependiente del largo de la entrada, la subsecuencia mas larga de la secuencia de largo N es la subsecuencia mas larga de una posible secuencia de un largo menor $N-1, N-2, N-3, \ldots$ mas 1, que es el hecho de incluir el elemento que estamos verificando en el resto de la subsecuencia que podria o no incluir otros elementos. Poniendo esto en forma de árbol:



Algoritmo: Notemos que si tomamos el primer elemento de la secuencia, tenemos que agregarle la subsecuencia mas larga de la cola de la secuencia, siempre y cuando los elementos sean mas grandes que el que tomamos, es decir, estamos buscando $LIS(i, j)$, que es el largo mas largo de la subsecuencia incremental de $A[j..n]$, la cola de la secuencia, despues y mas grandes que $A[i]$. Hagamos trampa y agreguemos un $A[0] = -\text{ínf}$, corremos el resto de la secuencia, por lo tanto, el LIS global será $LIS(0,1)$, osea:

$$LIS(i,j) = \begin{cases} 0 & \text{IF } j > n \\ LIS(i, j+1), & \text{IF } A[i] \geq A[j] \\ \text{máx}\{LIS(i, j+1), 1 + LIS(j, j+1)\} & \end{cases}$$

- El primer caso nos indica que estamos fuera del arreglo, por lo tanto no hay secuencia (recordemos hicimos trampa y pusimos el A[0] como $-$ínf)
- El segundo caso nos indica que la subsecuencia mas larga es la subsecuencia mayor a $A[i]$ pero sin $A[j]$ por que no era incremental
- El tercer caso es el caso general, buscamos la subsecuencia mas larga tal que es la de la cola, todos los elementos mayores sin incluirme, o la de la cola incluyendome

Orden de ejecución: Observe como la formula de la recurrencia implicitamente le dice como tiene que llenar la tabla, agregamos una columna extra llena de ceros por si no se dio cuenta, y vamos a ir ocrriendo desde derecha a izquierda, de arriba hacia abajo

Tabla: Creamos una tabla DP de tamaño $N+1 \times N+1$, con N el largo de la secuencia de entrada, mas 1 por el infinito que agregamos tramposamente, mas 1 por el caso base

	−ínf	10	22	9	33	21	50	41	60	UwU
−ínf										0
10										0
22										0
9										0
33										0
21										0
50										0
41										0
60										0

Cuadro 1: Inicialización, aun no echamos a correr el algorimo

```
1  import numpy as np
2  def lis(A):
3      A = np.concatenate([ [-np.inf], A] )
4      n = len(A)
5      DP = np.zeros((n+1, n+1), dtype=int)
6      print(DP)
7      for j in range(n-1, 0, -1): # zero base arrays, por eso el n-1
8          for i in range(0, j+1):
9              print(i, j)
10             if A[i] >= A[j]:
11                 DP[i, j] = DP[i, j+1]
12             else:
13                 DP[i, j] = np.max([ DP[i, j+1], 1 + DP[j, j+1] ])
14             print(DP)
15     return DP[0, 1]
```

3.2. Von Brand Coins Strikes Again **Este ejercicio no ha sido comprobado**

En el lejano pais de "*Horstlandia*", donde la maldad es infinita, utilizan el sistema "*VonBrand Coins*" con denominaciones $\{1,4,9,25\}$. Proponga un algoritmo para entregar el cambio usando la menor cantidad de monedas posibles. Asuma que tiene infinitas monedas disponibles. Por ahora solo preocupese de encontrar el set minimo.

Respuesta:

Puede seguir algunos pasos si quiere:

Defina subproblemas: Esto es una recursion, ¿cuales son los subproblemas que intenta resolver?

R: Tenemos varios casos base, si la cantidad de dinero que queremos pasar no es igual a alguna moneda, agregamos una moneda.

$$\begin{array}{l} 1 \text{ If } N == Divisa \\ 1+\texttt{numCoins}(N-Divisa) \\ \vdots \end{array}$$

Escriba la recurrencia: Tiene sus subproblemas, ahora escriba el problema grande.

R:

$$OPT(n) = \begin{cases} 1, & \text{if } n \in COINS \\ 1 + \text{mín} \begin{Bmatrix} OPT(n-1) \\ OPT(n-4) \\ OPT(n-9) \\ \vdots \end{Bmatrix} & \text{otherwise} \end{cases}$$

Demuestra que tu recursion es correcta: Caso por caso, la **RECURRENCIA** tiene que dar el resultado correcto.

R: Si tomamos el set de los *Von Brand Coins* $\{25, 9, 4, 1\}$, probemos con $n = 37$

$$\begin{array}{l} OPT(37) = \\ \qquad \text{mín}\{OPT(36), OPT(33), OPT(28), OPT(12)\} \\ \qquad \vdots \\ \qquad = 4 \end{array}$$

Te armas el árbol y vas "ruteando"

Demuestra que tu algoritmo evalua la recurrencia: Paso por paso, el programa va llenando la tabla de la misma forma que quedo escrita en la recurrencia.

R: Aprovechamos de hacer el algoritmo:

```
1   import numpy as np
2   def dpMakeChange(Coins, n):
3       DP = np.zeros(n+1, dtype=int)
4       for cents in range(n+1):
5           coinCount = cents
6           for j in [c for c in Coins if c <= cents]:
7               if DP[cents-j] + 1 < coinCount:
8                   coinCount = DP[cents-j]+1
9           DP[cents] = coinCount
10      return DP[n]
```

Esto llena la tabla así:

0	0	0	0	0	0
0	1	0	0	0	0
0	1	2	0	0	0
0	1	2	3	0	0
0	1	2	3	1	0
0	1	2	3	1	2

Demuestra que el algoritmo es correcto: Mas que nada mostrar que el "RETURN C[0].º lo que sea, es realmente lo que queremos.

R: DP[n] en nuestro caso

3.3. Dynamic Robot Motion Programing **Este ejercicio no ha sido comprobado**

Imagine una grilla bidimensional de $n \times k$(si, una matriz, otra vez ¬¬) sobre la que se mueve un robot llamado "Robi". Robi empieza en la casilla $[0,0]$ y tiene como objetivo llegar a la casilla $[n,k]$. El robot se mueve vertical y horizontalmente una casilla a la vez. Sobre el tablero se han arrojado monedas al azar (como maximo una por celda). Usted es muy avaro y obliga a Robi a recolectar la mayor cantidad de monedas mientras se mueve desde una esquina a la otra, pero también es perezoso asi que va a escribir un programa para calcular el recorrido.

	0	1	2	3	4
0				$	
1		$			
2				$	
3		$			

Respuesta:

Sea $\mathtt{F}(i,j)$ el número de monedas que llevamos recolectadas hasta la casilla $[i,j]$. Como el robot se mueve en movimientos horizontales y verticales por casillas adyacentes (Robi no sabe saltar), tenemos que elegir entre el recorrido que viene desde $\mathtt{F}(i-1,j)$(desde la izquierda) ó $\mathtt{F}(i,j-1)$(desde arriba):

$$\mathtt{F}(i,j) = c_{i,j} + \max\{\mathtt{F}(i-1,j), \mathtt{F}(i,j-1)\}$$

Donde $c_{i,j}$ es el dinero que nos aporta la moneda (0 si no hay moneda, 1 si la hay (por que todas las monedas son iguales aquí)). Nos definimos la tabla:

$$\begin{vmatrix} F(0,0) & \cdots & F(n-1,0) & F(n,0) \\ F(0,1) & \cdots & F(n-1,1) & F(n,1) \\ \vdots & \ddots & \vdots & \vdots \\ F(0,k-1) & \cdots & F(n-1,k-1) & F(n,k-1) \\ F(0,k) & \cdots & F(n-1,k) & F(n,k) \end{vmatrix}$$

Esto nos guarda la cantidad de monedas obtenidas, para recuperar el recorrido necesitamos guardar informacion de "de donde viene el F" que tomamos, no lo vamos a hacer por que en-realidad es otro problema.

```python
#! /bin/env python3
import numpy as np
def robi(C):
    # Matriz que lleva la cuenta
    M = np.zeros(C.shape, dtype=int)
    # Conteo de monedas
    for r in range(C.shape[0]):
        for c in range(C.shape[1]):
            M[r, c] = np.max([M[r-1, c], M[r, c-1]]) + C[r,c]
    return M

# Tiramos monedas sobre un tablero de 5x6
tablero = np.random.randint(2, size=(6, 5), dtype=int)
print("Tablero:\n", tablero)
robi(tablero)
```

3.4. Edit Distance **Este ejercicio no ha sido comprobado**

¿Cuantas insercciones de letras, borrados y substituciones se necesitan para transformar una palabra en otra? `FOOD` → `MONEY`

`FOOD` → `MOOD` → `MOND` → `MONED` → `MONEY`

4, el costo de ir desde la comida al dinero es 4.

```
F  O  O     D
M  O  N  E  Y
```

Entonces, Edit Distance tendra dos argumentos, cada palabra de largo m y n respectivamente. $Edit(A[1..m], B[1..n])$

La edit distance entre A y B es el minimo de:

$$\texttt{Edit}(A[1..m], B[1..n]) = \text{mín}\left\{\begin{array}{c} \texttt{Edit}(A[1..m-1], B[1..n]) + 1 \\ \texttt{Edit}(A[1..m], B[1..n-1]) + 1 \\ \texttt{Edit}(A[1..m-1], B[1..n-1]) + [A[m] \neq B[n]] \end{array}\right\}$$

Agregando los casos base (Ir de una string completa a una vacia (n o m borrados) o de una vacia a una llena (n o m agregados))

$$\texttt{Edit}(i,j) = \text{mín}\left\{\begin{array}{ll} i & \text{If } j = 0 \\ j & \text{If } i = 0 \\ \text{mín}\left\{\begin{array}{c} \texttt{Edit}(i-1, j) + 1 \\ \texttt{Edit}(i, j-1) + 1 \\ \texttt{Edit}(i-1, j-1) + \left[A[i] \neq B[j]\right] \end{array}\right\} & \text{otherwise} \end{array}\right\}$$

3.5. Estatua mas grande **Este ejercicio no ha sido comprobado**

Un país remoto planea celebrar el cumpleaños de su líder supremo colocando en la capital, una estatua gigante en conmemoración de su persona. No se escatimará en gastos, el único requisito es que la estatua tiene que tener una base cuadrada lo más grande posible. Aunque se puede destruir cualquier infraestructura para colocar la estatua, la ciudad ya tiene otras estatuas más pequeñas que sería una ofensa destruir, y por lo tanto deben permanecer en su lugar. Los ingenieros han representado la ciudad con una grid binaria, donde las celdas marcadas con 1 indican la presencia de una estatua y las marcadas con 0 indican que el terreno está vacío o bien que lo que hay en ese lugar no es importante y se puede demoler en pos del gran líder, sin embargo, no logran ponerse de acuerdo en el lugar óptimo en donde poner la estatua.

Respuesta: Iremos llenando una matriz del mismo tamaño que la ciudad, indicando el lado del cuadrado mas grande que se forma desde el $(0,0)$ al (i,j). La solución al problema es el maximo numero que se pueda encontrar en la matriz.
Los subproblemas son:

$$p(i,j) = \begin{cases} 0 & \text{IF } M[i][j] = True \\ 1 & \text{IF } (i = 0 \vee j = 0) \wedge M[i][j] = False \\ 1 + \min\{p(i-1,j), p(i,j-1), p(i-1,j-1)\} & \text{otherwise} \end{cases}$$

Y el problema general:

$$P(n,m) = \max_{0 \le i \le n, 0 \le j \le m} \{p(i,j)\}$$

4. Coeficientes binomiales

Recuerde la definición del coeficiente binomial:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Que tiene algunas propiedades que siempre se nos olvidan:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$
$$\binom{n}{0} = 1$$
$$\binom{k}{k} = 1$$

Con $n, k \in \mathbb{N}$:

1. Describa las desventajas de usar la definición para calcular $\binom{n}{k}$
2. Determine la complejidad de calcular $\binom{n}{k}$ por definición.
3. Describa un algoritmo *"eficiente"* para calcular $\binom{n}{k}$
4. Determine la complejidad de su algoritmo eficiente para calcular $\binom{n}{k}$

Respuesta:

1. La multiplicacion de factoriales rapidamente causa **overflow** (17! excede un `int32`)
2. $X!$ realiza $X-1$ multiplicaciones, tenemos:
   - $n-1$ multiplciaciones en el numerador
   - 1 resta; $n-k$ y k multiplicaciones, y una multiplicacion mas en el denominador ($1+n-k+k+1=n+2$)
   - 1 division

   Con un total de $\Theta(2n+3)$
3. Vamos por paso:

   a) Nos definimos la recurrencia desde las propiedades mencionadas:

   $$\begin{aligned}\mathtt{CB}(n,k) &= \mathtt{CB}(n-1,k-1)+\mathtt{CB}(n-1,k)\ \forall n>k\\ \mathtt{CB}(s,0) &= 1 \forall 0\le s\le n\\ \mathtt{CB}(s,s) &= 1 \forall 0\le s\le k\end{aligned}$$

   b) De la recurrencia nos generamos la tabla:

   $$\begin{vmatrix} \mathtt{CB}(0,0) & \mathtt{CB}(0,1) & \cdots & \mathtt{CB}(0,k)\\ \mathtt{CB}(1,0) & \mathtt{CB}(1,1) & \cdots & \mathtt{CB}(1,k)\\ \vdots & \vdots & \ddots & \vdots\\ \mathtt{CB}(n-1,0) & \mathtt{CB}(n-1,k-1) & \cdots & \mathtt{CB}(n-1,k)\\ \mathtt{CB}(n,0) & \mathtt{CB}(n,1) & \cdots & \mathtt{CB}(n,k)\end{vmatrix}$$

   c) Implementado en el siguiente codigo:

   Listing 5: ¿Que tiene de malo este código?

```python
#! /bin/env python3
import numpy as np
def binom(n, k):
    assert(n >= k) # Comprueba que estamos definidos
    bc = np.zeros((n+1,k+1)) # Inicializa la tabla
    # Casos base
    bc[:,0] = 1
    np.fill_diagonal(bc, 1)
    # El calculo....
    for i in range(1, n+1):
        for j in range(1, k+1):
            bc[i, j] = bc[i-1, j-1] + bc[i-1, j]
    return bc[n, k]
```

4. Un for de k iteraciones dentro de un for de n iteraciones: $\Theta(n\times k)$

5. Resumen

No Confundir Programación Dinamica con lo dinamico de algunos lenguajes como Scheme o Lisp ni con el acto de programar programas. **Dinamico** viene del dinamismo de llenar la tabla según los valores de la misma tabla y **Programación** viene de poner cosas en una tabla, como cuando se ve la programación de la t.v. satelital.
Los algoritmos de programación dinamica siguen condiciones similares a las de los voraces: Inductive Substructure, Optimal Substructure y Complete Choice, esto es, se revisa el problema completo.
Resuelva los problemas recursivamente y quedese con la mejor combinación.