6° Edición

Training Camp

Argentina 2015

PROGRAMACIÓN DINÁMICA

Nicolás Fochesatto – Universidad Nacional del Sur

Sucesión de Fibonacci

- Sucesión infinita de números naturales, de forma que: $f_0 = 1$ y $f_1 = 1$, definiéndose el resto de la sucesión como: $f_{n+2} = f_n + f_{n+1}$, para todo $n \geq 0$

1 1 2 3 5 8 13

- ¿Cómo podríamos calcular el elemento 100 de la sucesión?

Algoritmo Recursivo

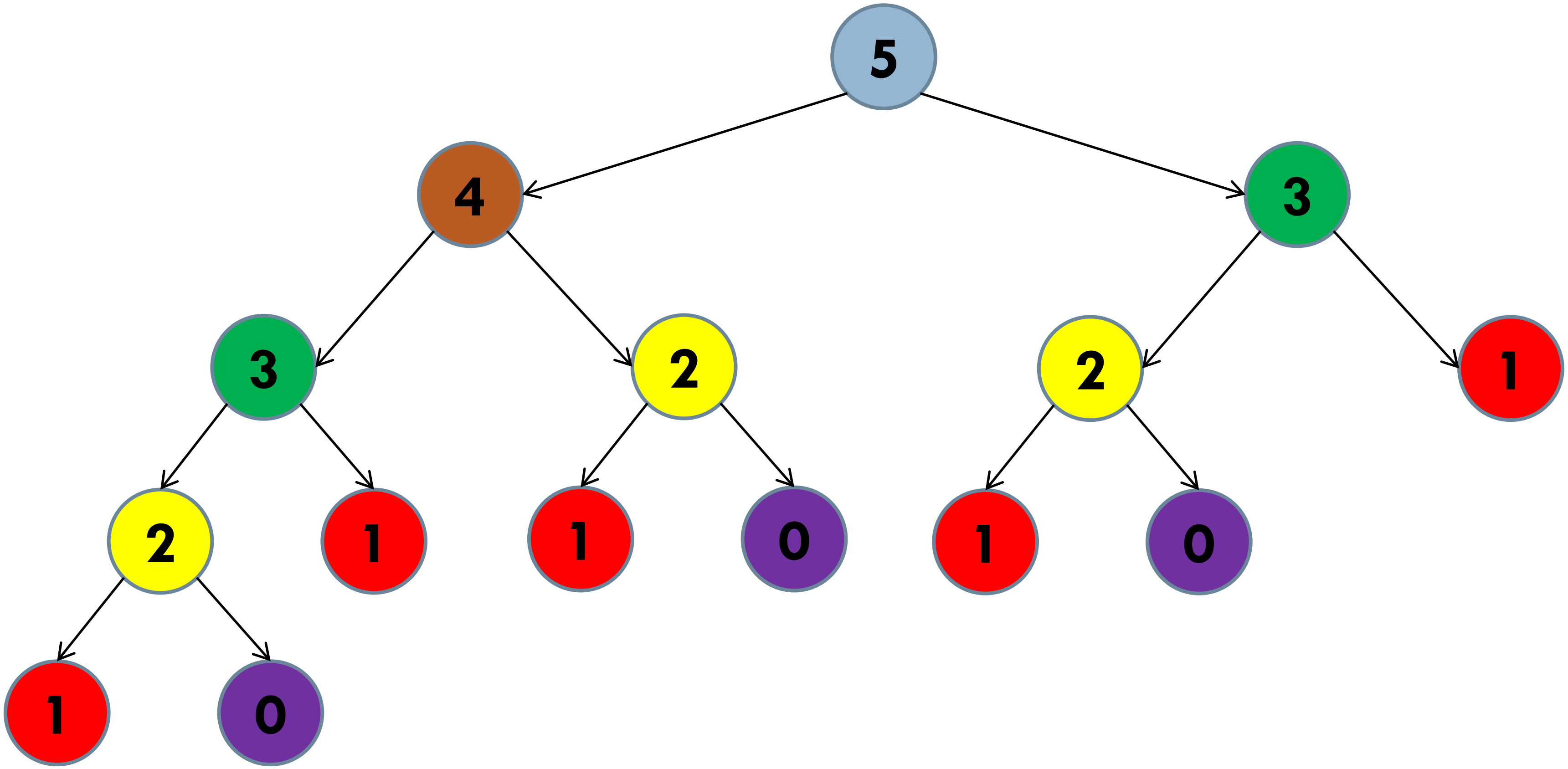
- Un algoritmo recursivo es aquel que expresa la solución de un problema en términos de una o más llamadas a sí mismo.

- Eventualmente deberán alcanzarse uno o más casos de resolución trivial, que llamaremos casos bases, que no requieren de la solución recursiva.

Algoritmo Recursivo para Fibonacci


```
FUNCTION Fibo(n) // n entero
    IF(n==0 or n==1)
        RETURN 1;
    ELSE
        RETURN Fibo(n-1)+Fibo(n-2);
END
```

Algoritmo Recursivo para Fibonacci



Algoritmo Recursivo para Fibonacci

- El tiempo de ejecución de este algoritmo crece exponencialmente con n.
- Esta solución es ineficiente: se está llamando a la misma función con el mismo parámetro más de una vez.
- El algoritmo carece de memoria: repite acciones ya realizadas en el pasado.

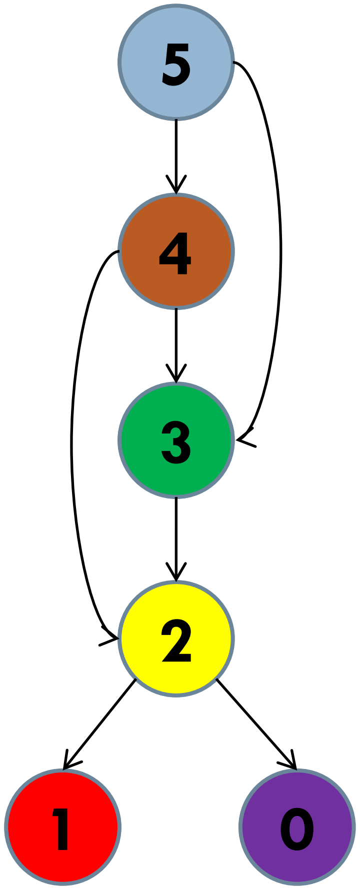
Programación Dinámica

- La programación dinámica consiste en:
  - Dividir un problema en subproblemas más pequeños;
  - Resolver cada subproblema hasta llegar a un caso base.

Hasta acá se asemeja mucho a una solución recursiva, pero…

  - Se guarda el resultado de cada instancia del problema la primera vez que se calcula.

Programación Dinámica



Programación Dinámica

- La definición anterior se conoce como Top-Down con Memoization.
  - Se sigue el orden natural de la recursión, pero se determina si cierta instancia fue o no previamente calculada.

- La alternativa es la solución Bottom-Up.
  - Se construye la solución desde los casos bases, siguiendo la dirección opuesta al árbol visto anteriormente. Cada vez que se quiere calcular un problema, se sabe que se tienen los subproblemas calculados.

DP para Fibonacci

- Solución Top – Down con memoization.

```
arreglo DP[n];
DP[0] = DP [1] = 1;
FOR (i = 2 to n-1)
   DP[i]=-1;
FUNCTION Fibo(n)
   IF(DP[n] != -1 )
         RETURN DP[n];
   ELSE
         DP[n] = Fibo(n-1)+Fibo(n-2);
         RETURN DP[n];
END
```

DP para Fibonacci

- **Solución Bottom - Up**

```
FUNCTION Fibo(n)
   arreglo DP[n];
   DP[0] = DP [1] = 1;
   FOR (i = 2 to n-1)
     DP[i]= DP[i-1] + DP[i-2];
  RETURN DP[n];
END
```

Top-Down VS Bottom-Up

- Ambos métodos obtienen un tiempo de ejecución de orden polinomial – O(n) para el caso de Fibonacci.
- Método Top-Down:
  - más conveniente cuando el espacio de subproblemas es mucho mayor que el realmente necesario para la solución – se calculan sólo las subinstancias necesarias.
  - No es necesario preocuparse por el orden en que se calculan las subinstancias

Top-Down VS Bottom-Up

- Método Bottom-Up
  - Es conveniente si el árbol a resolver es muy profundo – al no acumularse operaciones pendientes en el Stack.
  - No utiliza llamadas a funciones, lo que lo hace más rápido.

Otros problemas

- Existen algunos problemas clásicos que pueden resolverse en forma sencilla con DP, por ejemplo:
  - Factorial de un número;
  - Números Combinatorios;

Por otro lado, puede necesitarse resolver un problema de optimización – hallar el mayor o menor valor que cumple cierta condición. Los algoritmos DP pueden funcionar bien bajo ciertas condiciones.

Solapamiento de subproblemas

- Ocurre cuando el algoritmo recursivo visita más de una vez los mismos subproblemas.

- Por ejemplo para el caso de los números combinatorios:

$$C_2^3 = C_1^2 + C_2^2 \text{, donde } C_1^2 = C_0^1 + C_0^2 \text{ y } C_2^2 = C_1^1 + C_1^2$$

- Se puede apreciar como C_1^2 es necesario para calcular tanto C_2^2 como C_2^3.

Principio de Optimalidad

- Un problema requiere tener Subestructura Óptima para poder ser optimizado por DP.
  - La solución óptimal al problema contiene soluciones óptimales a los subproblemas

- ¡Esto no vale para cualquier problema!

Principio de Optimalidad

- Sea G = (V, E) un grafo dirigido:
  - Camino más corto entre cualquier par de nodos – dado cualquier par de nodos *u*, *v*, encontrar el camino de menor cantidad de arcos que los une.
  - Camino simple más largo – encontrar el camino simple entre cualquier par de nodos *u*, *v* que contenga la mayor cantidad de vértices.

Reconstrucción de una solución óptima

- Se puede querer no sólo saber CUÁL es el valor óptimo, sino CÓMO llegar a ese valor – para eso se debe reconstruir la solución óptima.
- En cada paso se toman decisiones sobre como dividir al problema en los subproblemas cuyas soluciones darán la solución óptima.
- Es necesario guardar en cada estado toda la información necesaria sobre la decisión óptima.

Ejemplos Clásicos

- Subset sum: Dados n números enteros positivos, decidir si se puede obtener una suma S usando algunos de ellos (a lo sumo una vez cada uno).

- Knapsack o problema de la mochila: Dados objetos con peso y valor, queremos meter el máximo valor posible en una mochila que soporta hasta P de peso.

Algunos Problemas

- http://codeforces.com/problemset/problem/225/C
- http://codeforces.com/problemset/problem/163/A
- http://goo.gl/ARNe7
- http://goo.gl/BJtPZ

Dinámica en rangos

- Son aquellos problemas cuyos estados consisten en subintervalos de un rango original.

- La decisión siempre consiste en qué índice del intervalo hacer la división en subrangos para obtener la solución óptima.

Multiplicación de matrices

- Dadas n matrices $\{A_1, A_2, \ldots, A_n\}$, encontrar la forma de realizar el producto, entre todas ellas, que requiera menor cantidad de producto escalares.

- Orden de la matriz A_i es $p_{i-1} \times p_i$.

- El costo (en cantidad de cálculos) de multiplicar dos matrices A_i y A_{i+1} es $p_{i-1} \cdot p_i \cdot p_{i+1}$

Multiplicación de matrices

- Rango genérico: Suponemos querer multiplicar en el rango: $\{A_i, A_{i+1}, \ldots, A_j\}$, $1 \leq i \leq j \leq n \ldots$ (¿Por qué no hacerlo en un intervalo $\{A_1, A_2, \ldots, A_j\}$?)

- Subrangos: Dividimos los rangos en un índice k, de forma que los subrangos son $\{A_i, A_{i+1}, \ldots, A_k\}$ y $\{A_{k+1}, A_{k+2}, \ldots, A_j\}$

Multiplicación de matrices

- La cantidad de operaciones para esta solución serán la cantidad de operaciones para el primer subrango, más las del segundo, más las operaciones necesarias para multiplicar las matrices resultantes.

$$oper(i,j) = oper(i,k) + oper(k+1,j) + p_{i-1} \cdot p_k \cdot p_j$$

Multiplicación de matrices

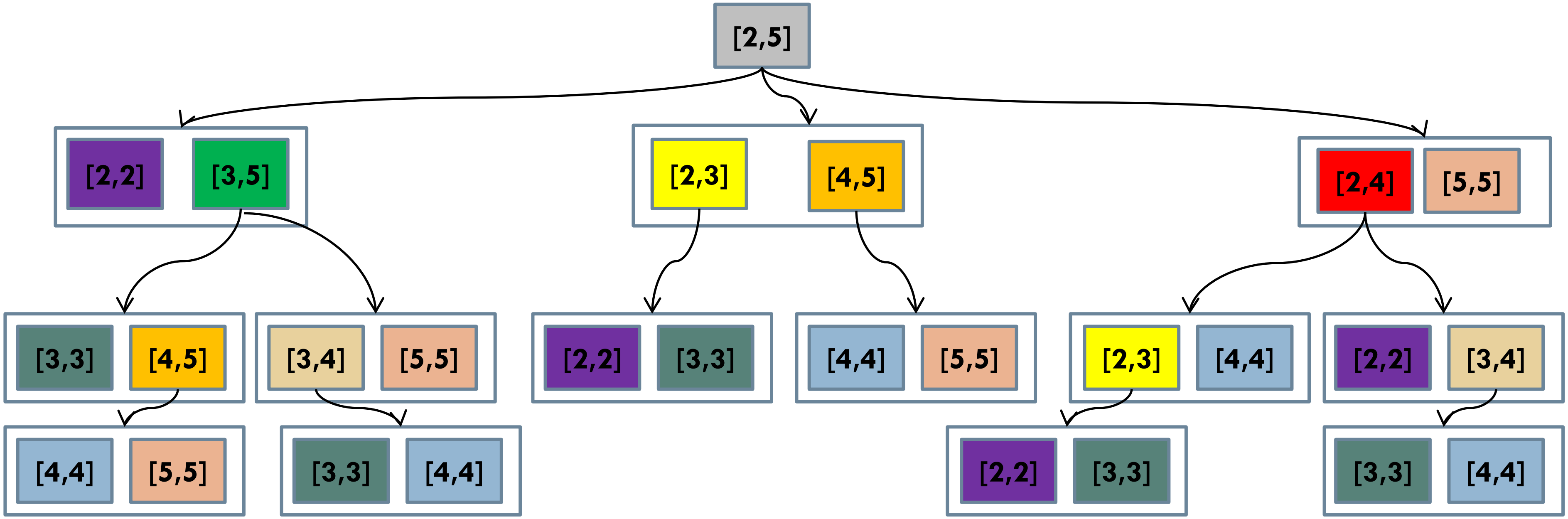
- Principio de optimalidad
  - Vamos a demostrar que una solución óptima de $\boldsymbol{oper}(\boldsymbol{i}, \boldsymbol{j})$ requiere soluciones óptimas de $\boldsymbol{oper}(\boldsymbol{i}, \boldsymbol{k})$ y $\boldsymbol{oper}(\boldsymbol{k}+1, \boldsymbol{j})$
  - Demostramos por el absurdo.

- Solución recursiva

$$\boldsymbol{m}[\boldsymbol{i}, \boldsymbol{j}] \begin{cases} 0 & \boldsymbol{si} \quad \mathrm{i} = \mathrm{j} \\ \min\limits_{\boldsymbol{i} \leq \boldsymbol{k} \leq \boldsymbol{j}} (\boldsymbol{m}[\boldsymbol{i}, \boldsymbol{k}] + \boldsymbol{m}[\boldsymbol{k}+1, \boldsymbol{j}] + \boldsymbol{p}_{\boldsymbol{i}-1} \cdot \boldsymbol{p}_{\boldsymbol{k}} \cdot \boldsymbol{p}_{\boldsymbol{j}}) & \boldsymbol{si} \quad \mathrm{i} \neq \mathrm{j} \end{cases}$$

Multiplicación de matrices

- Solapamiento de soluciones
  - La solución recursiva implica repetir varias veces las mismas subinstancias.



Multiplicación de matrices

- Solución DP Bottom-up

```
MatrixMult (vector p, integer n) // p[n+1]
  arreglo m[n][n];
  FOR i = 1 a n
     m[i,i]=0;
  FOR i = (n-1) a 1
     FOR j = (i+1) a n
          m[i,j] = infinito;
          FOR k = i a j-1
               q = m[i,k]+ m[k+1,j] + p[i-1]·p[k]·p[j];
               IF(q < m[i,j])
                    m[i,j] = q;
  RETURN m
END
```

Multiplicación de matrices

- Solución DP Top-Down

```
Arreglo m[n][n];
FOR i = 1 a n
  FOR j = 1 a n
       m[i,j] = infinito;
MatrixMult (arreglo m, vector p, 1, n);
```

Multiplicación de matrices

- Solución DP Top-Down

```
MatrixMult (arreglo m, vector p, i, j) // p[n+1]
   IF(m[i,j] < infinito)
         RETURN m[i,j];
   IF(i == j)
         m[i,j]=0;
   ELSE
         FOR k = i a j-1
            q =  MatrixMult(m,p,i,k)+ MatrixMult(m,p,k+1,j)
                 + p[i-1]·p[k]·p[j];
            IF(q < m[i,j])
                 m[i,j] = q ;
   RETURN m[i,j]
END
```

Multiplicación de matrices

- Reconstruir Solución Óptima
  - El algoritmo anterior sólo devuelve la cantidad mínima de operaciones.
  - Ambos métodos se pueden modificar fácilmente para guardar la información necesaria.
  - Basta con agregar un arreglo sol[n][n] que guarde la información del índice k óptimo para cada rango (i,j);

```
IF(q < m[i,j])
     m[i,j] = q;
     sol[i,j] = k;
```

Multiplicación de matrices

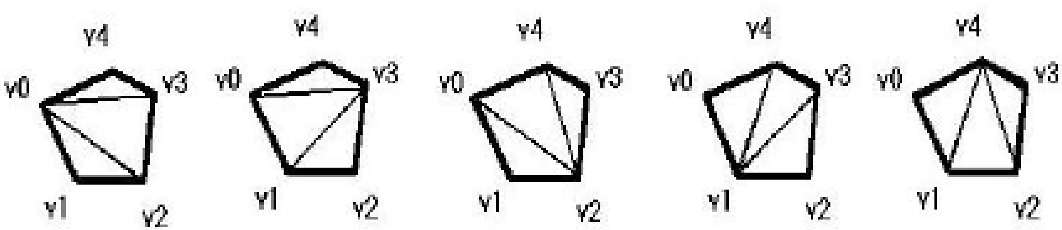
- Reconstruir Solución Óptima
  - Una vez que se tiene la tabla con los k óptimos, se procede a reconstruir la solución

```
IMPRIMIR-SOL-OPT(arreglos s, enteros i, j)
    IF(i==j)
        print "A";
        print i;
   ELSE
        print "(";
        IMPRIMIR-SOL-OPT(s,i,s[i,j]);
        IMPRIMIR-SOL-OPT(s,s[i,j]+1,j);
        print ")";
END
```

Triangulación optimal de polígonos

- Se tiene un polígono convexo de $\boldsymbol{n}$ lados y vértices $\boldsymbol{v}_{\boldsymbol{i}}$, $0 \leq \boldsymbol{i} \leq \boldsymbol{n}-1$, siendo el lado i el formado por los vértices $\boldsymbol{v}_{\boldsymbol{i}-1}$ y $\boldsymbol{v}_{\boldsymbol{i}}$.

- Se quiere encontrar una triangularización optimal de acuerdo a algún criterio dado.



Otros ejemplos

- ABB óptimo
  - Árbol de Búsqueda Binario óptimo: Dada una secuencia de valores ordenados $v_1 < v_2 < \ldots < v_n$, compute un Árbol Binario de Búsqueda que minimice la cantidad de comparaciones necesarias (profundidad) esperadas para encontrar un elemento – por ejemplo si se quisiera programar un traductor de texto palabra por palabra.
- Parenteseado:
  - Dada una cadena de caracteres {, }, [,], (y), de longitud par , dar la mínima cantidad de reemplazos que deben realizarse al string para que quede bien parenteseado.

Otros ejemplos

- Longest Increasing Subsequence
  - Dada una secuencia de números, encontrar la subsecuencia de números que aparezcan de manera creciente y que sea lo más larga posible.

- http://www.spoj.com/problems/SUPPER/
- http://www.spoj.com/problems/LIS2/

Máscara de Bits

- Método útil cuando se quiere seleccionar un subconjunto de un conjunto de elementos.

- Consiste en representar a los subconjuntos de un conjunto de *n* elemento por medio de la representación binaria de los números que van desde 0 hasta $2^n - 1$

Máscara de Bits

- Supongamos que tenemos el conjunto {1,3,8}, en total hay 8 formas de elegir subconjuntos de ese conjunto, como se representa a continuación:

Número	Máscara	Subconjunto	Número	Máscara	Subconjunto
0	000	{ }	4	100	{1}
1	001	{8}	5	101	{1,8}
2	010	{3}	6	110	{1,3}
3	011	{3,8}	7	111	{1,3,8}

Máscara de Bits

- Operadores Bitwise – comparaciones bit a bit

Bit 1	Bit 2	AND(&)	OR(|)	XOR(^)
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

- Shifting:
  - A derecha $(x >> i)$, "elimina" de la representación binaria de x los últimos i dígitos.
  - A izquierda $(x << i)$, "agrega" a la representación binaria de x un total de i dígitos 0 al final.

Aplicaciones

- Iteraciones sobre subconjuntos
  - A veces se requiere, para calcular una función en un conjunto, calcularla previamente sobre sus subconjuntos.
- Iteraciones sobre superconjuntos
  - De la misma forma a veces se requiere, para calcular una función en un subconjunto, calcularla previamente en sus superconjuntos.

Problema de los peces

- Hay $\boldsymbol{n}$ ($\boldsymbol{n \leq 18}$) peces en el mar. Cada minuto se encuentran dos peces al azar (la probabilidad es uniforme para todo par de peces). Sea $\boldsymbol{p}[\boldsymbol{i}][\boldsymbol{j}]$ la probabilidad de que, dado un encuentro entre el pez $\boldsymbol{i}$ y el pez $\boldsymbol{j}$, el pez $\boldsymbol{i}$ se coma al $\boldsymbol{j}$ (y viceversa para $\boldsymbol{p}[\boldsymbol{j}][\boldsymbol{i}]$). Sabemos que $\boldsymbol{p}[\boldsymbol{i}][\boldsymbol{j}] + \boldsymbol{p}[\boldsymbol{j}][\boldsymbol{i}] = 1$ y que $\boldsymbol{p}[\boldsymbol{i}][\boldsymbol{i}] = 0$.

  ¿Cual es la probabilidad de que sobreviva el pez 0?

Problema de los peces

- Función: Dado un subconjunto (que representaremos con la variable “mask”), nos devuelve la probabilidad de que el pez “0” sobreviva en ese subconjunto.
- Casos base:
  - Cuando el conjunto este compuesto por un pez la función devolverá 1 si ese pez es el “0”, y 0 en cualquier otro caso.
  - Cuando un subconjunto no contenga al pez “0”, la función devolverá 0.

Problema de los peces

- Recursión:
  - De cada conjunto se pueden calcular sus subconjuntos, estos serán los estados del problema – los subconjuntos posibles se crean suponiendo que uno de los peces fue comido (uno de los "1" de la máscara se vuelve "0");
  - Cada subconjunto tiene una probabilidad de aparición de acuerdo a las probabilidad de que el pez que se quitó haya sido comido.
  - La probabilidad de que el pez "0" sobreviva en el conjunto será la probabilidad de que sobreviva en el subconjunto, multiplicada por la probabilidad de pasar del conjunto al subconjunto

Problema de los peces


```
Variables globales
  arreglo dp[1<<18]; // inicializado en
                              -1
  arreglo p[18][18];
  entero n; // número de peces
```

Problema de los peces


```
FUNCTION f(entero mask)
  IF(dp[mask]>=0) RETURN dp[mask];
  vivos = 0; // cant. peces vivos en mask
  FOR i = 0 to (n-1)
     IF((mask>>i)%2==1) vivos = vivos+1;
  pares = (vivos*(vivos-1))/2;//pares en mask
  IF (vivos==1) // caso base
     IF(mask=1) dp[mask]=1;
     ELSE dp[mask]=0;
     RETURN dp[mask];
```

Problema de los peces


```
dp[mask]=0;
FOR i = 0 to (n-1)
  FOR j = 0 to (i-1)
      IF((mask>>i)%2==1 AND (mask>>j)%2==1)
            IF(i!=0 AND j!=0)
                  dp[mask]= dp[mask] +
                        (f(mask^(1<<i))*p[j][i]+
                        f(mask^(1<<j))*p[i][j])/pares;
            ELSE IF(i==0) dp[mask]= dp[mask] +
                        f(mask^(1<<j))*p[i][j])/pares;
            ELSE IF(j==0) dp[mask]= dp[mask] +
                        f(mask^(1<<i))*p[j][i])/pares;
RETURN dp[mask]
```

Problema de los peces

- Supongamos ahora que queremos calcular la probabilidad de supervivencia de todos los peces.
  - Podríamos repetir el algoritmo anterior n veces…
  - O hacer algo más eficiente.
- Vamos a calcular, dado un subconjunto, la probabilidad de haber llegado al mismo a través de los conjuntos de los que puede venir (sus superconjuntos);

Problema de los peces

- Nuestra solución será la función calculada en todos los subconjuntos que sólo contengan 1 pez.

- Caso base: ahora el único caso base será que la función en el conjunto inicial (el que contiene todo los peces) debe devolver 1 (ya que siempre vamos a iniciar desde ese conjunto).

Problema de los peces


```
dp[(1<<n)-1] = 1;
FUNCTION f(entero mask)
  IF(dp[mask]>=0) RETURN dp[mask];
  vivos = 1; // cant. peces vivos en mask
  FOR i = 0 to (n-1)
      IF((mask>>i)%2==1) vivos = vivos+1;
  pares = (vivos*(vivos-1))/2;//pares en mask
```

Problema de los peces


```
dp[mask]=0;
FOR i = 0 to (n-1)
  FOR j = 0 to (n-1)
      IF((mask&(1<<i))!=0 AND (mask&(1<<j))==0)
            dp[mask]= dp[mask] +
                  (f(mask|(1<<j))*p[i][j])/pares;

RETURN dp[mask]
```

Otro ejemplo

- En una clase hay $2n$ alumnos ($n \leq 8$) y tienen que hacer trabajos prácticos en grupos de a 2. El i-ésimo alumno vive en el punto (x_i , y_i) de la ciudad. El profesor sabe que los alumnos se reúnen para hacer los trabajos prácticos en la casa de uno de los dos miembros del grupo. Por eso decide que la suma de las distancias (rectas) entre compañeros de grupo debe ser mínima. Dar esta distancia.

Referencias

- Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.
- TopCoder Tutorial: https://goo.gl/J2JkUh
- Codechef Tutorial: https://goo.gl/9b1JBw
- Bit Masking: https://goo.gl/1YSf4l