Recursividad II (2015-2016)

A - Juego

- Al adoquín 0 se llega de una forma (ya se está ahí)
- Al adoquín 1 se llega de una forma (salto de longitud 1)
- Al adoquín n se puede llegar desde el $n-1$ (salto de longitud 1) o desde el $n-2$ (salto salto de longitud 2)
- $\therefore NumeroDeCaminos(N) = Fibonacci(N)$

B - LCS

Definamos $LCS(s1, i, s2, j)$ como el LCS entre los *subtrings* $s1[i...n]$ y $s2[j...m]$ donde n y m son las longitudes de $s1$ y $s2$ respectivamente

- Si $i = n \vee j = m$ se está calculando el LCS entre la cadena vacía y otra cadena y se sabe que el LCS es la propia cadena vacía (Caso Base).
- Si $s1[i] = s2[j]$ entonces se puede afirmar que el caracter $s[i]$ ($s[j]$) está presente en al menos una subsecuencia común y que si a toda subsecuencia común que exista "a la derecha" de i y j se le concatena al inicio el caracter $s[i]$ ($s[j]$) se obtiene una nueva subsecuencia común de longitud mayor en una unidad. Por tanto existe una subsecuencia común que comienza en la posición i y j de cada cadena.
- Si $s1[i] \neq s2[j]$ entonces ninguna subsecuencia común comienza al mismo tiempo en la posición i y j de los *strings* analizados. Las nuevas subsecuencias pueden formarse entre los *substrings* $\big(\ s1[i...n]\ ,\ s2[j+1...m]\ \big)$ y $\big(\ s1[i+1...n]\ ,\ s2[j...m]\ \big)$.
- El objetivo es quedarse de todas las subsecuencias encontradas con la de mayor longitud. Por tanto:

$$LCS(s1, i, s2, j) = \begin{cases} \text{“ ”} & \text{si } i = n \vee j = m \\ s1[i] + LCS(s1, i+1, s2, j+1) & \text{si } s1[i] = s2[j] \\ Max(LCS(s1, i, s2, j+1), LCS(s1, i+1, s2, j)) & \text{si } s1[i] \neq s2[j] \end{cases}$$

Concluyéndose que en el problema nos piden calcular $LCS(s1, 0, s2, 0)$

C - Fito en el Bastión

En este problema básicamente se nos pide un algoritmo de ordenación que sea estable (que se mantenga el orden relativo de los elementos iguales) y eficiente. Inmediatamente se debe pensar en *MergeSort*.

El primer problema que nos enfrentamos es que se debe ordenar un conjunto de elementos (los nombres) por el orden de otro conjunto (las alturas). Esto se puede resolver de varias maneras. Una de ellas es mantener dos *arrays* uno con los nombres y otro con las alturas e ir ejecutando el algoritmo de ordenación sobre ellos en paralelo, o sea, se utilizan los valores del *array* de alturas para hacer las comparaciones y determinar orden y cada vez que sea necesario hacer un intercambio, se realiza en los dos *arrays*. Otra forma (utilizada en el código de abajo) es crear una clase (o usar cualquier otra estructura) que contenga dos propiedades (Nombre y Altura) y hacer el array a ordenar del tipo de esta clase, nuevamente solo comparar por las alturas.

La estabilidad del *MergeSort* nos garantiza que Fito sea justo con aquellos que llegaron primero y sean de la misma altura. Recordar que Fito siempre va al inicio.

Problema C

```
using System;

namespace RecursividadII
{
    public class C
    {
        public class Persona
        {
            public Persona(string n, int h)
            {
                Nombre = n;
                H = h;
            }
            public string Nombre { get; set; }
            public int H { get; set; }
        }

        static void Main()
        {
            int N = int.Parse(Console.ReadLine());

            Console.ReadLine();

            if(N > 1)
            {
                Persona[] p = new Persona[N - 1];
                for(int i = 0; i < N - 1; i++)
                {
                    string[] line = Console.ReadLine().Split();
                    string n = line[0];
                    int h = int.Parse(line[1]);
                    p[i] = new Persona(n, h);
                }

                MergeSort(p);

                Console.WriteLine("Fito");

                for(int i = 0; i < N - 1; i++)
                    Console.WriteLine(p[i].Nombre);
            }
            else
            {
                Console.Write("Fito");
            }
        }
```

```
        public static void MergeSort(Persona[] arr)
        {
            MergeSort(arr, 0, arr.Length - 1, new Persona[arr.Length]);
        }

        private static void MergeSort(Persona[] arr, int left, int right, Persona[] tmp)
        {
            if (right > left)
            {
                int m = (left + right)/2;

                MergeSort(arr, left, m, tmp);
                MergeSort(arr, m + 1, right, tmp);

                Merge(arr, left, right, tmp);
            }
        }

        private static void Merge(Persona[] arr, int left, int right, Persona[] tmp)
        {
            int m = (left + right)/2;

            int p = 0;
            int k = left;
            int l = m + 1;

            while (k <= m && l <= right)
            {
                if (arr[k].H <= arr[l].H)
                    tmp[p++] = arr[k++];
                else
                    tmp[p++] = arr[l++];
            }

            while (k <= m)
                tmp[p++] = arr[k++];

            while (l <= right)
                tmp[p++] = arr[l++];

            for (int i = 0; i < p; i++)
                arr[left + i] = tmp[i];
        }
    }
}
```

D - Cambio de Monedas

Sea $C(N, S, i)$ el número de forma de dar cambio para N monedas, contándose con infinita cantidad de monedas de las denominaciones $\{S[0], S[1], ..., S[i]\}$

- Si la solución no contiene a $S[i]$ entonces podemos resolver el problema para N monedas y denominaciones $\{S[0], S[1], ..., S[i-1]\}$, o sea, $C(N, S, i-1)$
- Si la solución contiene a $S[i]$ entonces la solución contiene al menos un $S[i]$, por lo que se está solucionando el problema para $N - S[i]$ monedas y todas las denominaciones que existen en esta instancia: $\{S[0], S[1], ..., S[i]\}$, o sea, $C(N - S[i], S, i)$
- Si $N = 0$ entonces $C(N, S, i) = 1$ ya que existe una sola manera de dar cambio: no dar ninguna moneda en lo absoluto (Caso Base I)
- Si $N < 0$ entonces $C(N, S, i) = 0$ ya que es una cantidad negativa de dinero (Caso Base II)
- Si $N > 0 \wedge m < 0$ entonces $C(N, S, i) = 0$ ya que no se cuenta con denominaciones para dar cambio alguno (Caso Base III)

Por tanto:

$$C(N, S, i) = \begin{cases} 0 & \text{si } N < 0 \vee i < 0 \\ 1 & \text{si } N = 0 \\ C(N, S, i-1) + C(N - S[i], S, i) & \text{eoc} \end{cases}$$

Concluyéndose que en el problema nos piden calcular $C(N, S, m)$ donde m es el total de denominaciones ($m = |S|$)

E - Palabras de Dyck

Si se cambian las 'X' por '(' y las 'Y' por ')' el problema se transforma encontrar la cantidad de cadenas de paréntesis balanceados de longitud $2n$ (n abiertos y n cerrados) ya que no es difícil demostrar que una cadena es una palabra de Dyck si y solo si es una cadena de paréntesis balanceados asumiendo 'X' como '(' y 'Y' como ')'

El problema de encontrar la cantidad de cadenas de parentesis balanceados es conocido y se resuelve recursivamente de la siguiente forma:

$$P(N) = \begin{cases} 1 & \text{si } N = 1 \\ \sum_{k=1}^{N} P(k)P(N-k) & \text{si } N > 1 \end{cases}$$

F - Rectángulo de mayor área en un histograma

Podemos usar una estrategia Divide y Vencerás para resolver este problema. La idea es encontrar la posición del mínimo del *array* dado y una vez obtenido esto, la mayor área es el máximo de los tres valores siguientes:

- Máxima área a la izquierda del valor mínimo, sin incluirlo (Llamado recursivo)
- Máxima área a la derecha del valor mínimo, sin incluirlo (Llamado recursivo)
- Número de barras multiplicado por el valor mínimo (mayor rectángulo con altura igual al valor mínimo)

Problema F

```
using System;

namespace RecursividadII
{
    public class F
    {
        public static int PosMin(int[] a, int l, int r)
        {
            int min = int.MaxValue;
            int idx = -1;

            for(int i = l; i <= r; i++)
            {
                if(a[i] < min)
                {
                    min = a[i];
                    idx = i;
                }
            }

            return idx;
        }

        public static int MaxArea(int[] h)
        {
            return MaxArea(h, 0, h.Length-1);
        }

        public static int MaxArea(int[] h, int l, int r)
        {
            if(l > r) return 0;
            if(l == r) return h[l];

            int min = PosMin(h, l, r);

            int maxL = MaxArea(h, l, min - 1);
            int maxR = MaxArea(h, min + 1, r);
            int barra = (r - l + 1) * h[min];

            return Math.Max(maxL, Math.Max(maxR, barra));
        }

        public static void Main()
        {
            string[] line = Console.ReadLine().Split();
            int[] h = new int[line.Length];
            for(int i = 0; i < line.Length; i++)
                h[i] = int.Parse(line[i]);

            Console.WriteLine(MaxArea(h));
        }
    }
}
```

G - Máxima Suma Contigua

Usando una estrategia Divide y Vencerás se puede resolver este problema usando el siguiente algoritmo:

1. Dividir el *array* en dos mitades
2. Devolver el máximo de estos tres valores:
   - Suma máxima contigua del *subarray* de la izquierda (llamado recursivo)
   - Suma máxima contigua del *subarray* de la derecha (llamado recursivo)
   - Suma máxima contigua de los *subarrays* que cruzan el punto medio

Problema G

```
using System;

namespace RecursividadII
{
    public class G
    {
        public static int MaxSum(int[] a)
        {
            return MaxSum(a, 0, a.Length -1);
        }

        public static int SumaCruzada(int[] arr, int l, int m, int h)
        {
            int sum = 0;
            int ls = int.MinValue;
            for (int i = m; i >= l; i--)
            {
                sum = sum + arr[i];
                if (sum > ls)  ls = sum;
            }

            sum = 0;
            int rs = int.MinValue;
            for (int i = m+1; i <= h; i++)
            {
                sum = sum + arr[i];
                if (sum > rs)  rs = sum;
            }
            return ls + rs;
        }

        public static int MaxSum(int[] a, int l, int r)
        {
            if(l == r) return a[l];

            int m = l + (r-l)/2;

            int maxL = MaxSum(a, l, m);
            int maxR = MaxSum(a, m + 1, r);
            int cruz = SumaCruzada(a, l, m, r);

            return Math.Max(maxL, Math.Max(maxR, cruz));
        }

        public static void Main()
        {
            string[] line = Console.ReadLine().Split();
            int[] a = new int[line.Length];
            for(int i = 0; i < line.Length; i++)
                a[i] = int.Parse(line[i]);

            Console.WriteLine(MaxSum(a));
        }
    }
}
```

H - Rellenando el tablero

Este problema puede ser resuelto con Divide y Vencerás siguiendo la siguiente idea (ojo: es solo una idea, en el código final hay que lidiar con otras cosas):

- Sea $Rellenar(N, P)$ donde N es el tamaño del cuadrado a analizar y P la posición del hueco (o casilla que no se puede usar)
- Caso Base: $N = 2$, Un cuadrado de 2x2 es sencillo rellenar con las fichas del juego, solo hay que poner una ocupando las tres casillas que no son el hueco
- Dividir el tablero en cuatro subcuadrados de lado $N/2$. Notar que la casilla con el hueco queda solo en uno de estos cuadrados, por lo que quedan tres sin hueco. Poner una ficha en el centro de la división teniendo el cuidado de ponerla en los subcuadrados sin el hueco. De esta forma ahora se tienen cuatro cuadrados de tamaño $N/2$ con una casilla que no se puede usar.
- Sean P, $P1$, $P2$, $P3$ las casillas marcadas, entonces resolver recursivamente los cuatro subproblemas
  - $Rellenar(N/2, P)$
  - $Rellenar(N/2, P1)$
  - $Rellenar(N/2, P2)$
  - $Rellenar(N/2, P3)$

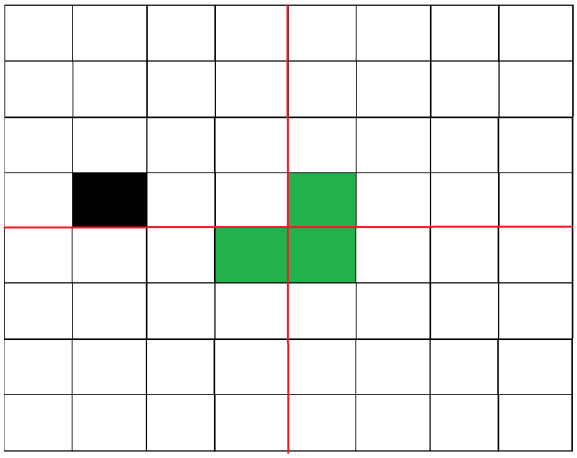

Figura 1: Luego de poner la primera pieza

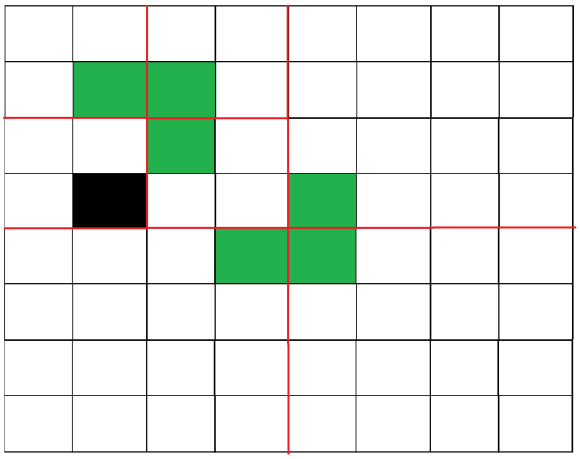

Figura 2: Llamado recursivo al primer subcuadrado

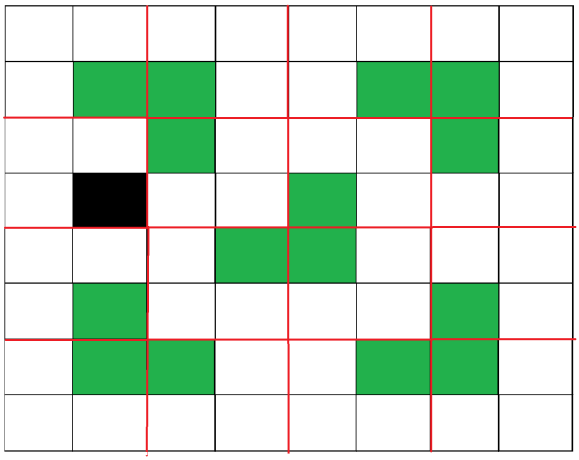

Figura 3: Primer paso en los 4 subcuadrados

Problema H

```
using System;

namespace RecursividadII
{
    class H
    {
        /*  (f, c): esquina superior izquierda del cuadrado a analizar
            l: lado del cuadrado a analizar
            (fh, ch): posicion del hueco o casilla que no se puede usar
            ans: matriz respuesta
            idx: indice de la ficha que se va usar
            retorna: El indice de la ultima ficha que se uso */
        static int Rellena(int f, int c, int l, int fh, int ch, int[,] ans, int idx)
        {
            //Cuadrado de 2x2
            if (l == 2)
            {
                for (int i = 0; i < l; i++)
                    for (int j = 0; j < l; j++)
                        if (f + i == fh && c + j == ch)
                            continue;
                        else
                            ans[f + i, c + j] = idx;
                return idx;
            }
            else
            {
                int curIdx = idx;
                l /= 2;

                //Cuadrado superior izquierdo
                if (f <= fh && fh < f + l && c <= ch && ch < c + l)
                    //(fh, ch) esta en este cuadrado, llamado recursivo
                    idx = Rellena(f, c, l, fh, ch, ans, idx + 1);
                else
                {
                    //(fh, ch) no esta en este cuadrado,
                    //marcar casilla que toca,
                    //computar esquina superior izquierda, llamado recursivo
                    ans[f + l - 1, c + l - 1] = curIdx;
                    idx = Rellena(f, c, l, f + l - 1, c + l - 1, ans, idx + 1);
                }

                //Cuadrado superior derecho
                if (f <= fh && fh < f + l && c + l <= ch && ch < c + 2 * l)
                    idx = Rellena(f, c + l, l, fh, ch, ans, idx + 1);
                else
                {
                    ans[f + l - 1, c + l] = curIdx;
                    idx = Rellena(f, c + l, l, f + l - 1, c + l, ans, idx + 1);
                }

                //Cuadrado inferior izquierdo
                if (f + l <= fh && fh < f + 2 * l && c <= ch && ch < c + l)
                    idx = Rellena(f + l, c, l, fh, ch, ans, idx + 1);
                else
                {
                    ans[f + l, c + l - 1] = curIdx;
                    idx = Rellena(f + l, c, l, f + l, c + l - 1, ans, idx + 1);
                }

                //Cuadrado inferior derecho
                if (f + l <= fh && fh < f + 2 * l && c + l <= ch && ch < c + 2 * l)
                    idx = Rellena(f + l, c + l, l, fh, ch, ans, idx + 1);
                else
                {
                    ans[f + l, c + l] = curIdx;
                    idx = Rellena(f + l, c + l, l, f + l, c + l, ans, idx + 1);
                }
            }
            return idx;
        }
```

```
        public static void Main()
        {
            string[] line = Console.ReadLine().Split();
            int N = int.Parse(line[0]);
            int F = int.Parse(line[1]);
            int C = int.Parse(line[2]);

            int[,] ans = new int[N, N];

            Rellena(0, 0, N, F, C, ans, 1);

            for (int i = 0; i < N; i++)
            {
                if (i > 0)
                    Console.WriteLine();

                for (int j = 0; j < N; j++)
                {
                    if (j > 0)
                        Console.Write(" ");

                    Console.Write("{0}", ans[i, j] == 0 ? "X" : ans[i, j].ToString());
                }
            }
        }
    }
}
```