Introducción a la Programación Dinámica

Jorge Humberto Terán Pomier

May 21, 2013

Contents

1 Principios de la Programación Dinámica

La programación dinámica es un método que en general resuelve problemas de optimización, que involucran hacer una secuencia de decisiones determinandas, para cada decisión, subproblemas que pueden resolverse de una forma similar, tal que una solución óptima del problema original pueda hallarse de las soluciones óptimas de los subproblemas. Este método se basa en los principios de optimalidad de Bellman.

Una política óptima tiene la propiedad de que cualquiera que sea el estado inicial y decisión inicial, son las decisiones restantes las que deben constituir una política óptima con respecto a la situación resultante de la primera decisión. Más sucintamente, este principio afirma que las políticas óptimas tienen subpolíticas óptimas. Que el principio válido sigue de la observación: si una política tiene una subpolítica que no es óptima, el reemplazo de la política por una subpolítica óptima mejorará la política original.

Para que la programación dinámica sea computacionalmente eficiente deben existir subproblemas talque un subproblema de uno es un subproblema de otro. En esta situación solo se debe hallar la solución a un subproblema y reusarse tan seguido como sea necesario.

La programación dinámica es un proceso de decisión, recursivo por lo que es es necesario definir los siguientes aspectos:

Estado .- Definimos estado, como la información sobre las decisiones realizadas hasta un momento específico.

Función objetivo.- La función objetivo es una función del estado y es el beneficio o costo óptimo resultante de realizar una secuencia de decisiones asociadas con el estado.

Función de recompensa.- La función recompensa, es el beneficio o costo que se puede atribuir a la siguiente decisión hecha en el estado.

Función de transformación.- La función de transformación o de transición, especifica el siguiente estado que resulta de hacer una decisión en el estado.

Operador.- El operador es una operación binaria, por lo general adición o multiplicación o minimizar / maximizar, que nos permite combinar la rentabilidad de decisiones separadas. Esta operación debe ser asociativa, si los rendimientos de decisiones han de ser independientes del orden en el que se hacen.

Condición base.- Dado que la programación dinámica es un proceso recursivo hay que especificar las condiciones que hacen que la recursión termine. Generalmente los valores son cero o infinito.

Cuando un problema se puede resolver expresando la solución con una ecuación de recurrencia es muy probable que se pueda resolver usando métodos de la Programación Dinámica.

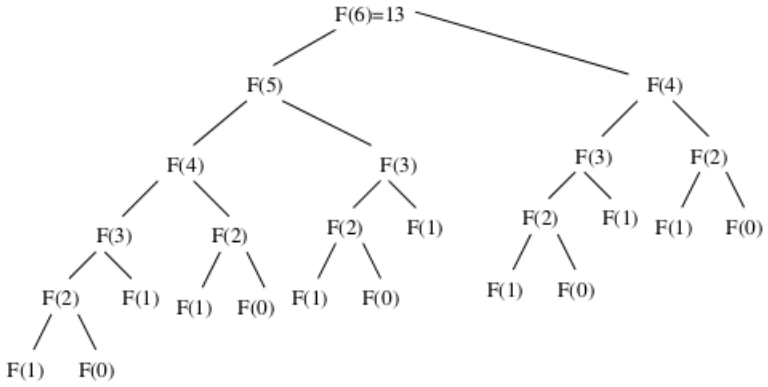


Figure 1: Arbol de recursión, [3]

Veamos un ejemplo. La secuencia de Fibonacci es mediante la suma de los dos términos anteriores. Los primeros números de la secuencia son :$1, 1, 2, 3, 5, 8, 13, 21, ...$ Formalmente la expresamos con la

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{en otros casos} \end{cases}$$

Como se ve en la definición los casos base (casos que terminan la recursión) se dan cuando $n = 0$ y cuando $n = 1$. Codificando en esta solución tenemos:

```
static long Fib_r(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return (Fib_r(n - 1) + Fib_r(n - 2));
    }
```

Al probar la recursión llamamos a $Fib_r(n)$ y vemos que los resultados son correctos. Ahora cuando probamos con valores de n mayores a 40, vemos que el tiempo de ejecución es demasiado largo. Analicemos el árbol generado en la solución recursiva presentada. Tomando $n = 13$ vemos

Como se ve en el gráfico $f(4)$ se calcula en ambos lados, $f(2)$ se calcula 5 veces, así podemos ver que muchos valores se vuelve a calcular muchas veces. Es aquí donde los principios de la Programación dinámica nos ayudan a resolver este problema de tiempo.

Cuanto tiempo toma hallar $f(n)$? Como se ve en el árbol de recursión en cada nodo existen dos caminos que se abren, lo que nos hace pensar que la complejidad es proporcional a 2^n.

Para hallar el tiempo de proceso observemos la solución de la ecuación recursiva [4].

$$f(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

de donde:

$$\frac{f_n}{f_{n+1}} \approx \frac{(1+\sqrt{5})}{2} = 1.61803$$

Esto significa que $f_n > 1.61^n$ por lo que se tienen $f_n > 1.61^n$ llamadas de procedimientos, lo que nos lleva a concluir que es un algoritmo exponencial.

1.1 Guardar los valores calculados

Caching es una técnica para guardar los valores calculados con anterioridad y utilizarlos posteriormente. Para evitar recalcular los mismos valores repetidamente creamos un vector para almacenar el valor de $f(n)$ ya calculados.

```
    static long Fib_r(int n) {
        if (f[n] == 0 && n > 0)
            f[n] = Fib_r(n - 1) + Fib_r(n - 2);
        return f[n];
    }
}
```

En este problema hemos creado un vector f donde guardamos los valores anteriores. Cuando un valor $f(n) =$ tenemos un n que no ha sido hallado. Los casos base $f(1) = 0, f(1) = 1$ se los inicializa el momento de definir el vector.

¿Cuál es la complejidad de este algoritmo? Si hacemos un árbol para ver como se evalúa la recursión veremos que una vez que se tiene un valor anterior termina eliminando todas las ramas de este valor. Esto nos da un tiempo proporcional a n. Esto equivale a decir un tiempo lineal.

Hemos convertido el tiempo polinomial a un tiempo lineal.

El método general de explícitamente guardar los resultados de llamadas recursivas para evitar el recálculo provee un camino para obtener lo máximo de la programación dinámica. Sin embargo en algoritmos como quicksort no tiene sentido porque en las llamadas se tienen diferentes parámetros.

Utilizar caching tiene sentido cuando el espacio que ocupan diferentes parámetros es pequeño y podemos disponer de espacio de almacenamiento extra. En nuestro caso solo almacenamos un entero entre 0 y n por los que hay solo $O(n)$ valores que guardar en el cache.

Un gasto lineal de memoria es una buena compensación para reducir el tiempo exponencial.

1.2 Eliminando la recursión

Si analizamos el problema anterior vemos que solo es necesario sumar dos valores anteriores con lo que podemos simplemente hacer un programa iterativo como se muestra:

```
        f[0]=0;
        f[1]=1;
        for (int i = 2; i <= n; i++)
            f[i] = f[i - 1] + f[i - 2];
    }
```

En este código se ve más claramente que $O(n) = n$ que muestra que el tiempo de proceso es lineal.

1.3 Eliminando los cálculos intermedios

Un análisis detallado nos muestra que no es necesario guardar todos los valores intermedios. Esto porque la recurrencia solo depende de los dos valores anteriores. Para reducir la memoria utilizada solo es necesario definir dos variables temporales para almacenar los valores anteriores.

```
    static long Fib_r(int n) {
        long anterior1 = 1, anterior2 = 0, proximo = 0;
        for (int i = 2; i <= n; i++) {
            proximo = anterior1 + anterior2;
            anterior2 = anterior1;
            anterior1 = proximo;
        }
        return (anterior2 + anterior1);
    }
}
```

En este problema es más visible la complejidad lineal. Es un solo ciclo que va desde 1 hasta n que claramente tiene $O(n) = n$.

1.4 Programas mencionados en el texto

1.4.1 Fibonacci solución recursivo

```
/*
 * Solucion recursiva al problema de Fibonacci
 * Autor Jorge Teran
```

```
 */

public class FibRecursivo {

    public static void main(String[] args) {
        System.out.println(Fib_r(45));

    }

    static long Fib_r(int n) {
        if (n == 0)
            return 0;
        if (n == 1)
            return 1;
        return (Fib_r(n - 1) + Fib_r(n - 2));
    }
}
```

1.4.2 Fibonacci solución con caching

```
/*
 * Solucion con caching al problema de Fibonacci
 * Autor Jorge Teran
 */

public class FibCaching {
    static long[] f = new long[100];

    public static void main(String[] args) {
        f[0] = 0;
        f[1] = 1;
        System.out.println(Fib_r(45));

    }

    static long Fib_r(int n) {
        if (f[n] == 0 && n > 0)
            f[n] = Fib_r(n - 1) + Fib_r(n - 2);
        return f[n];
    }
}
```

1.4.3 Fibonacci solución no recursivo

```
/*
```

```
 * Solucion no recursiva,  al problema de Fibonacci
 * Autor Jorge Teran
 */

public class FiBNoRecursivo {
    static long[] f = new long[100];

    public static void main(String[] args) {

        System.out.println(Fib_r(45));

    }

    static long Fib_r(int n) {
        f[0] = 0;
        f[1] = 1;
        for (int i = 2; i <= n; i++)
            f[i] = f[i - 1] + f[i - 2];
        return f[n];
    }
}
```

1.4.4 Fibonacci solución utilizando solo dos variables

```
/*
 * Solucion utilizando dos variables, problema de Fibonacci
 * Autor Jorge Teran
 */
public class FibGuardaDos {
    public static void main(String[] args) {

        System.out.println(Fib_r(45));

    }

    static long Fib_r(int n) {
        long back1 = 1, back2 = 0, next = 0;
        for (int i = 2; i <= n; i++) {
            next = back1 + back2;
            back2 = back1;
            back1 = next;
        }
        return (back2 + back1);
    }
}
```

2 Coeficientes binomiales

Los coeficientes binomiales son otro ejemplo de eliminar la recursión especificando el orden de evaluación. Los coeficientes binomiales son una forma de hallar las combinaciones e indicar el número de maneras de escoger k elementos de n posibilidades. Y se escriben como:

$$\left(\begin{array}{c} n \\ k \end{array} \right)$$

Por ejemplo si tenemos 3 elementos abc las formas diferentes de escoger 2 elementos es ab, ac, bc. matemáticamente se calculan con la formula:

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

Calcular el factorial implica realizar n multiplicaciones. Recordemos que $n! = n(n-1)(n-2)...1$. Para eliminar el proceso de multiplicar, que generará números muy grandes y reducir el tiempo, porque en la multiplicación la demora es mucho mayor que el de la suma, recurriremos a la siguiente ecuación [4]:

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ k \end{array} \right)$$

2.1 Solución recursiva

Como ya hemos convertido el problema inicial a una recursión podemos aplicar los principios aprendidos.

Ninguna recursión está completa si no conocemos el caso base. ¿Cuáles coeficientes binomiales conocemos sin calcularlos? Eventualmente el lado derecho nos llevará a $\left(\begin{array}{c} n-1 \\ 0 \end{array} \right)$. Cuántas formas existen para encontrar 0 elementos de un conjunto? Exactamente 1 el conjunto vacío.

Si no es convincente también podemos tomar $\left(\begin{array}{c} m \\ 1 \end{array} \right) = m$ como caso base. El término de la derecha llegará hasta $\left(\begin{array}{c} k \\ k \end{array} \right) = 1$.

Para escribir un programa que resuelva el requerimiento de hallar los coeficientes binomiales, observamos que hay dos valores que cambian. Esto lleva a deducir que se requiere una matriz de dos dimensiones para almacenar los valores anteriores.

El programa que mostramos tiene una matriz $x[][]$ para almacenar los cálculos anteriores.

```
public static long CoefBin(int n, int k) {
```

```
        if (x[n][k] > 0)
            return x[n][k];
        if (k == 0) {
            x[n][k] = 1;
            return 1;
        }
        if (k == 1) {
            x[n][k] = n;
            return n;
        }
        if (k == n) {
            x[n][k] = 1;
            return 1;
        }
        x[n][k] = CoefBin(n - 1, k - 1) + CoefBin(n - 1, k);
        return x[n][k];
    }
```

2.2 Eliminando la recursión

Ahora podemos eliminar la recursión. Para comenzar se inicializa la matriz con los casos base.

¿Qué muestran las condiciones de los casos base?

$\binom{n}{m}$	0	1	2	3	4	5
0	A					
1	B	G				
2	C	2	H			
3	D	3	3	I		
4	E	4	6	4	J	
5	F	5	10	10	5	K

Los valores de la $A - K$ son los valores iniciales

$\binom{n}{m}$	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Los valores iniciales de la $A - K$ son uno, y corresponden al caso base:

```
public static void triángulo(int[][] x) {
```

```
  // Invariante cada linea debe comenzar con uno
  for (int i=0;i<x.length;i++)
      x[i][0]=1;
  // Invariante cada linea debe terminar con uno
    for (int i=0;i<x.length;i++)
      x[i][i]=1;
    for (int i=1;i<x.length;i++)
  // Invariante cada linea debe sumar 2**i
      for (int j=1;j<=i;j++)
        x[i][j]=x[i-1][j-1]+x[i-1][j];
    }
  }
```

Una vez que se tiene calculada la matriz x para hallar n tomados de k es suficiente tomar el valor $x[n][k]$.

Como se ve la complejidad es este algoritmo es $O(n) = n^2$. Lo que muestra que hemos reducido la complejidad del problema de 2^n a n^2.

2.3 Programas mencionados en el texto

2.3.1 Coeficientes binomiales solución recursiva

```
import java.util.Arrays;

/*
 * Solucion recursiva al problema de los coeficientes binomiales
 * Autor Jorge Teran
 */

public class CoefBinomialRecursivo {
    static int n = 22;

    public static void main(String[] args) {
        System.out.println(CoefBin(20, 15));
    }

    public static long CoefBin(int n, int k) {
        if (k == 0) {
            return 1;
        }
        if (k == 1) {
            return n;
        }
```

```
        if (k == n) {
            return 1;
        }
        return CoefBin(n - 1, k - 1) + CoefBin(n - 1, k);
    }
}
```

2.3.2 Coeficientes binomiales con caching

```
import java.util.Arrays;

/*
 * Solucion con caching al problema de los coeficientes binomiales
 * Autor Jorge Teran
 */

public class CoefBinomialCaching {
    static int n = 22;
    static long[][] x = new long[n][n];

    public static void main(String[] args) {
        System.out.println(CoefBin(20, 15));
        for (long[] i : x)
            System.out.println(Arrays.toString(i));
    }

    public static long CoefBin(int n, int k) {
        if (x[n][k] > 0)
            return x[n][k];
        if (k == 0) {
            x[n][k] = 1;
            return 1;
        }
        if (k == 1) {
            x[n][k] = n;
            return n;
        }
        if (k == n) {
            x[n][k] = 1;
            return 1;
        }
        x[n][k] = CoefBin(n - 1, k - 1) + CoefBin(n - 1, k);
        return x[n][k];
    }
}
```

2.3.3 Coeficientes binomiales solución

```
import java.util.Arrays;

/*
 * Solucion iterativa al problema de los coeficientes binomiales
 * Autor Jorge Teran
 */

public class CoefBinomial {
    public static void main(String[] args) {
        int[][] x = new int[6][6];
        CoefBin(x);
        for (int[] i : x)
            System.out.println(Arrays.toString(i));
    }

    public static void CoefBin(int[][] x) {
        // Invariante cada linea debe comenzar con uno
        for (int i = 0; i < x.length; i++)
            x[i][0] = 1;
        // Invariante cada linea debe  terminar con uno
        for (int i = 0; i < x.length; i++)
            x[i][i] = 1;
        for (int i = 1; i < x.length; i++)
            // Invariante cada linea debe sumar 2**i
            for (int j = 1; j <= i; j++)
                x[i][j] = x[i - 1][j - 1] + x[i - 1][j];
    }
}
```

2.4 Ejercicio - Pagando con Monedas

Supongamos que vivimos en un país donde las siguientes monedas estan disponibles: 1 peso (100 centavos), 25 centavos, 10 centavos, 5 centavos y 1 centavo.

El problema radica en pagar a un cliente en monedas utilizando el menor número de monedas. Por ejemplo si queremos pagar 289 centavos la mejor solución es pagar con 2 monedas de a peso, 3 monedas de 25, una de 10, 4 de un centavo.

Una de las ideas que se nos vienen a la mente es dar primero las monedas de mayor valor, luego las de siguiente valor y así sucesivamente, tal como se vió en el ejemplo.

Que pasará si las monedas que tenemos disponibles son $6, 4, 1$ y debemos pagar un valor de 8. Con esta idea primero escogemos una moneda de 6 y luego dos monedas de 1. con un total de

3 monedas. Claramente se ve que hay una solución menor. Se requieren solo dos de 4.

2.4.1 Solución

Para construir la solución utilizando programación dinámica crearemos una matriz $c[n][p]$ con una fila por cada denominación y una columna por cada monto a pagar desde de 0 hasta p. En esta matriz $c(i, j)$ se representará el número mínimo de monedas requerido para pagar un monto $(0 \leq j \leq p)$ con denominaciones $(1 \leq i \leq n)$.

Veamos los casos base:

- Para todo valor de i, si requerimos pagar 0 la respuesta es 0
- Para todo valor de j si hay una moneda la respuesta es j

No incluir ninguna moneda del tipo $t(i)$, supone que el valor de $c(i, j)$ va a coincidir con el de $c(i-1, j)$, y por tanto $c(i, j) = c(i-1, j)$.

De incluirla, entonces, al incluir la moneda del tipo $t(i)$, el número de monedas global coincide con el número óptimo de monedas para una cantidad $(j - t(i))$ más esta moneda $t(i)$, es decir podemos expresar $c(i, j)$, en este caso, como $c(i, j) = 1 + c(i, j - t[i])$.

Dado que tenemos que minimizar el número de monedas debemos escoger

$$min(c(i-1, j), 1 + c(i, j - t[i]))$$

La ecuación de recurrencia para este problema sería:

$$c[i][j] = \begin{cases} j & \text{si } i = 0 \\ 0 & \text{si } j = 0 \\ c[i-1][j] & \text{si } j < t[i] \\ c[i][j] = Math.min(c[i-1][j], 1 + c[i][j - t[i]]); & \text{en otros casos} \end{cases}$$

La solución es planteada en el programa siguiente:

```
int c[][] = new int[t.length][valor + 1];
//inicializar el caso base
for (int i = 0; i < t.length; i++)
     c[i][0] = 0;
for (int j = 0; j <= valor; j++)
     c[0][j] = j;
```

```
//hallar el minimo
for (int i = 1; i < t.length; i++) {
   for (int j = 1; j <= valor; j++) {
      if (j < t[i])
         c[i][j] = c[i - 1][j];
       else
c[i][j] = Math.min(c[i - 1][j], 1 + c[i][j - t[i]]);
           }
        }
```

La respuesta quedará en el último valor. Con este análisis se obtiene una solución eficiente. La solución por fuerza bruta requeriría hallar todas las posibles combinaciones de monedas que sumen el monto solicitado. Este problema claramente exponencial ha sido reducido a una solución polinomial.

2.5 Programas mencionados en el texto

2.5.1 Solución al ejercicio de las monedas

```
import java.util.Arrays;

public class Monedas {

    public static void main(String[] args) {
        // int t[] = { 1,2,3 };
         // int t[] = { 1,4,6 };
        //int t[] = { 1, 3, 8, 16, 22, 57, 103, 526 };
        int t[] = { 1, 5, 10, 25, 100 };
        int valor = 85;
        int c[][] = new int[t.length][valor + 1];
        for (int i = 0; i < t.length; i++)
            c[i][0] = 0;
        for (int j = 0; j <= valor; j++)
            c[0][j] = j;

        for (int i = 1; i < t.length; i++) {
            for (int j = 1; j <= valor; j++) {
                if (j < t[i])
                    c[i][j] = c[i - 1][j];
                else
                    c[i][j] = Math.min(c[i - 1][j], 1 + c[i][j
                            - t[i]]);
            }
        }
```

```
        System.out.println(c[t.length - 1][valor]);
        for (int[] i : c)
            System.out.println(Arrays.toString(i));
    }
}
```

3 Corte de troncos

Supongamos que una empresa maderera quiere cortar troncos, en tamaños de longitud diferente [1]. Supongamos además que conocemos el precio p_i para cada corte de una longitud dada. Por ejemplo:

longitud i	1	2	3	4	5	6	7	8	9	10
precio p_i	1	5	8	9	10	17	17	20	24	30

Con estos datos, consideremos el caso de cortar un tronco de longitud 4 para obtener la máxima ganancia posible.Todos los cortes posibles son:

Numero de Cortes	Tamaño de los cortes	Precio
1	4	9
2	$2+2$	$5+5=10$
3	$1+3$	$1+8=9$
4	$3+1$	$9+1=9$
5	$1+1+1+1$	$1+1+1+1=4$
6	$1+1+2$	$1+1+5=7$
7	$1+2+1$	$1+5+1=7$
8	$2+1+1$	$5+1+1=7$

Como ve el mejor precio que podemos obtener es de 10 que se obtiene cortando los troncos por la mitad haciendo dos troncos de 2.

En general un tronco de tamaño n se puede cortar de 2^{n-1} formas distintas. Si realizamos un programa que halle estas 2^{n-1} diferentes particiones el tiempo es proporcional a 2^n. Este tiempo es inaceptable para la mayor parte de los problemas por lo que buscaremos una solución más eficiente.

Se puede pedir que los troncos se corten en un orden determinado. Esto puede reducir el número de formas distintas. Sin embargo sigue siendo una solución poco eficiente. Estas formas de dividir, se denominan funciones de partición.

Para $i = 1, 2, 3, ..., n-1$ la descomposición en partes la haremos utilizando la notación de la suma. Por ejemplo un corte de longitud $7 = 2+2+3$ nos indica que un tronco de longitud 7

se corto en tres troncos dos de longitud 2 y uno de longitud 3. Si la solución óptima corta el tronco en k partes, para algún $1 \leq k \leq n$ entonces la descomposición óptima del tronco es:

$$n = i_1 + i_2 + i_3.....i_k$$

después de cortar el tronco en partes de longitud $i_1, i_2, ..., i_k$. Es corte proveerá las utilidades

$$r_n = p_{i_1} + p_{i_2} + p_{i_3}....., p_{i_k}$$

En forma general podemos para $n \geq 1$ podemos decir que los valores r_n en términos de los cortes más pequeños es:

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1}, r_1)$$

En forma más simple se puede expresar como

$$r_n = max(p_i + r_{n-i}$$

El primer argumento p_n corresponde al caso de no hacer cortes y vender el tronco de longitud n en el tamaño actual. Los otros elementos corresponden al máximo rédito, al cortar en piezas de tamaño i y $n - i$ partes, para $i = 1, 2, 3..., n - 1$. Como no conocemos cual i optimiza el problema debemos considerar todos los valores de i y escoger el que maximice las ganancias.

Vea que el problema original es de tamaño n, resolveremos problemas del mismo tipo pero de tamaño menor. Una vez que hacemos la primera partición podemos considerar a las dos partes como dos instancias independientes del mismo problema. La solución óptima será el máximo de las dos soluciones.

Se dice que el problema de cortar troncos muestra una ***subestructura óptima***: soluciones óptimas al problema incorporan soluciones óptimas a los subproblemas relacionados, que se pueden resolver en forma independiente.

Expresando la solución en forma recursiva podemos ver que consiste en cortar una pieza de longitud i a la izquierda y $n - 1$ al lado derecho. El caso base se da cuando $n = 0$ cuyo resultado es 0.

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ r_n = max(p_i + r_{n-1} & \text{para } 1 \leq i \leq n \end{cases}$$

Esto indica que solo lo que queda, exceptuando la primera parte, puede ser subdivida. Esto nos da una versión más simple de las ecuaciones anteriores. La implementación de la solución es:

```
static int Resolver(int n) {
    if (n == 0)
        return 0;
    int q = Integer.MIN_VALUE;
    for (int i = 1; i <=n; i++) {
        q = Math.max(q, precios[i] + Resolver(n - i));
    }
    return q;
}
```

Analicemos la complejidad de esta solución

El programa hace n iteraciones donde se realiza la llamada recursiva.

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

Resolviendo la recurrencia podemos hallar que el tiempo asintótico es 2^n. El problema es que esta solución está hallando todas las 2^{n-1} formas de cortar para escoger la que proporciona el valor máximo.

3.1 Solución utilizando Programación Dinámica

Ahora mostraremos como convertir este problema en un algoritmo eficiente utilizando programación dinámica, utilizando los principios que se vió sobre la programación dinámica.

El método de la programación dinámica funciona como sigue. Habiendo observado que una solución recursiva es ineficiente porque resuelve un mismo subproblema repetidamente, arreglamos para que se resuelva una sola vez, guardando su solución. Si nos referimos a este subproblema posteriormente, solo tomamos la solución en lugar de recalcularla. La programación dinámica utiliza memoria adicional para ahorrar tiempo de recálculo. Con esta técnica una solución con tiempo de ejecución exponencial puede convertirse en tiempo de ejecución polinomial. El tiempo de ejecución es polinomial cuando cada uno de los subproblemas es polinomial en la entrada y puede resolverse en tiempo polinomial.

Existen dos métodos de programación dinámica, que se explicarán con este ejemplo.

El primer método es del *tipo top-down con memorización.* En este método escribimos la solución en forma recursiva y luego la modificamos para guardar los resultados de cada subproblema. El procedimiento ahora revisa si tiene un resultado para este subproblema en cuyo caso da la respuesta inmediatamente ahorrando cálculos futuros. Si no conoce el resultado entonces procede de la forma normal. Decimos que el procedimiento recursivo recuerda que resultados ha calculado previamente.

El segundo método el *botton-up*. Este método depende de alguna noción natural del tamaño de un subproblema. Esto es resolver un subproblema depende de resolver subproblemas más pequeños. Ordenamos los subproblemas por tamaño y comenzamos a resolver desde el más pequeño. Cuando resolvemos algún subproblema ya hemos resulto los subproblemas más pequeños de los cuales depende su solución. Cuando encontramos un subproblema se resuelve una sola vez, dado que ya se resolvieron todos los problemas que son prerequisito.

Los dos métodos se ejecutan en tiempos similares, excepto que el recursivo tiene una constante de tiempo mayor. Veamos como se construyen las soluciones utilizando estos métodos.

3.2 Solución top-down con memorización

Analicemos el programa recursivo. En el podemos ver que cuando $n = 0$ el resultado es cero. Este es el caso que hace que la recursión termine. Para almacenar los valores anteriores definimos un vector que denominamos *memoria* en el que vamos almacenando todos los valores ya calculado.

Ahora la recursión termina cuando existe un valor ya calculados esto es $memoria[n] \geq 0$. En otros casos continuamos como el programa recursivo. Luego antes de devolver el resultado lo guardamos en nuestro vector de resultados ya calculados.

La implementación de la solución es:

```
static int Resolver(int n) {
    if (memoria[n] >= 0)
        return memoria[n];
    int q = 0;
    if (n == 0)
        q = 0;
    else {
        q = Integer.MIN_VALUE;
        for (int i = 1; i <= n; i++) {
            q = Math.max(q, precios[i] + Resolver(n - i));
        }
    }
    memoria[n] = q;
    return memoria[n];
```

3.3 Solución bottom-up

Para esta solución es necesario primero resolver un subproblema de tamaño i más pequeño que un problema de tamaño j. En esta solución se utiliza el arreglo *memoria* para guardar los resultados de subproblemas anteriores. Comienza inicializando el valor r[0] en cero, dado

que este corte no da ninguna utilidad. Luego resuelve el problema para tamaños $1, , 2, 3, ..., n$, incrementando el tamaño cada vez.

La complejidad es proporcional a n^2, y es fácil de ver dado que tiene dos ciclos anidados. En la solución recursiva no se ve claramente que su complejidad es proporcional a n^2 debido a que retornan los valores que ya están calculados.

La implementación de la solución es:

```
static int Resolver(int n) {
     for (int j = 1; j <= n; j++) {
         int q = Integer.MIN_VALUE;
         for (int i = 1; i <= j; i++) {
             q = Math.max(q, precios[i] + memoria[j - i]);
         }
         memoria[j] = q;
     }
     return memoria[n];
 }
```

3.4 Reconstruyendo la solución

No solo es suficiente conocer cual es la utilidad máxima sino también queremos conocer el detalle de los cortes. Para esto se hace necesario guardar los subíndices de las sub soluciones óptimas.

Para esto creamos un vector s en el cual guardamos el índice de cada solución óptima. Luego en un proceso de que parte del ultimo valor podemos ir hallando los valores que nos llevaron al resultado.

Volvamos al ejemplo, los valores del vector *memoria* y el vector s para $n = 10$ son:

longitud i	0	1	2	3	4	5	6	7	8	9	10
precio p_i	0	1	5	8	9	10	17	17	20	24	30
memoria[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

Para reconstruir la secuencia de cortes comenzamos del último valor que está en la solución de cortes óptimos. Se imprime este corte y se resta el tamaño cortado para obtener el siguiente corte óptimo.

La implementación de la solución es:

```
   static void Reconstruir(int n) {
        System.out.println("Reconstruyendo");
        while (n > 0) {
            System.out.print(s[n] + " ");
            n = n - s[n];
        }
        System.out.println();
    }
```

3.5 Programas mencionados en el texto

3.5.1 Solución recursiva al problema cortar troncos

```
/*
*Solucion recursiva al problema de cortar troncos
*Autor Jorge Teran
*/
public class TroncosRecursivo {
    // static int[] longitud = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    static int[] precios = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30 };

    public static void main(String[] args) {

        System.out.println(Resolver(4)); // debe ser 10
        System.out.println(Resolver(8)); // debe ser 22
    }

    static int Resolver(int n) {
        if (n == 0)
            return 0;
        int q = Integer.MIN_VALUE;
        for (int i = 1; i <=n; i++) {
            q = Math.max(q, precios[i] + Resolver(n - i));
        }
        return q;
    }
}
```

3.5.2 Solución con caching al problema cortar troncos

```
import java.util.Arrays;
/*
*Solucion con caching al problema de cortar troncos
*Autor Jorge Teran
```

```
*/
public class TroncosMemoria {
    // static int[] longitud = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    static int[] precios = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30 };
    static int lt = precios.length;
    static int[] memoria = new int[lt];

    public static void main(String[] args) {
        for (int i = 0; i < lt; i++)
            memoria[i] = Integer.MIN_VALUE;
        System.out.println(Resolver(4)); // debe ser 10
        System.out.println(Resolver(8)); // debe ser 22
    }

    static int Resolver(int n) {
        if (memoria[n] >= 0)
            return memoria[n];
        int q = 0;
        if (n == 0)
            q = 0;
        else {
            q = Integer.MIN_VALUE;
            for (int i = 1; i <= n; i++) {
                q = Math.max(q, precios[i] + Resolver(n - i));
            }
        }
        memoria[n] = q;
        // System.out.println(Arrays.toString(memoria));
        return memoria[n];
    }
}
```

3.5.3 Solución no recursiva al problema cortar troncos

```
import java.util.Arrays;
/*
*Solucion no recursiva al problema de cortar troncos
*Autor Jorge Teran
*/

public class TroncosBottomUp {
    // static int[] longitud = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    static int[] precios = { 0,1, 5, 8, 9, 10, 17, 17, 20, 24, 30 };
    static int lt = precios.length;
    static int[] memoria = new int[lt];
```

```
    public static void main(String[] args) {
        System.out.println(Resolver(4)); // debe ser 10
        memoria = new int[lt];
        System.out.println(Resolver(8)); // debe ser 22
    }

    static int Resolver(int n) {
        for (int j = 1; j <= n; j++) {
            int q = Integer.MIN_VALUE;
            for (int i = 1; i <= j; i++) {
                q = Math.max(q, precios[i] + memoria[j - i]);
            }
            memoria[j] = q;
        }
        // System.out.println(Arrays.toString(memoria));
        return memoria[n];
    }
}
```

3.5.4 Reconstrucción del problema cortar troncos

```
import java.util.Arrays;
/*
*Recontruyendo la solucion problema de cortar troncos
*Autor Jorge Teran
*/

public class TroncosReconstruir {
    // static int[] longitud = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    static int[] precios = { 1, 5, 8, 9, 10, 17, 17, 20, 24, 30, 0 };
    static int lt = precios.length;
    static int[] memoria = new int[lt];
    static int[] s = new int[lt];

    public static void main(String[] args) {
        int r = Resolver(4);
        System.out.println(r); // debe ser 10
        Reconstruir(4);
        memoria = new int[lt];
        s = new int[lt];
        r = Resolver(8);
        System.out.println(r); // debe ser 22
        Reconstruir(8);
    }
```

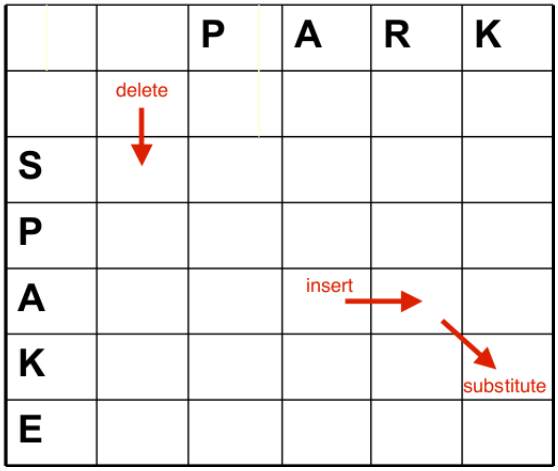
```
    static int Resolver(int n) {

        for (int j = 1; j <= n; j++) {
            int q = Integer.MIN_VALUE;
            for (int i = 0; i < j; i++) {
                if (q < precios[i] + memoria[j - i - 1]) {
                    q = precios[i] + memoria[j - i - 1];
                    s[j] = i + 1;
                }
            }
            memoria[j] = q;
        }
        // System.out.println(Arrays.toString(memoria));
        // System.out.println(Arrays.toString(s));
        return memoria[n];
    }

    static void Reconstruir(int n) {
        System.out.println("Reconstruyendo");
        while (n > 0) {
            System.out.print(s[n] + " ");
            n = n - s[n];
        }
        System.out.println();
    }
}
```

4 Distancia de edición

Se llama *Distancia de Levenshtein* o *Distancia de edición* al número mínimo de operaciones requeridas para transformar una cadena en otra. Se entiende por operación una inserción, eliminación o substitución de un carácter. Esta distancia recibe ese nombre en honor al científico ruso Vladimir Levenshtein, quien se ocupara de esta distancia en 1965. Es útil en programas relacionados al genoma humano, plagio, corrección de ortografía, etc.. Veamos algunos ejemplos de transformaciones [2].

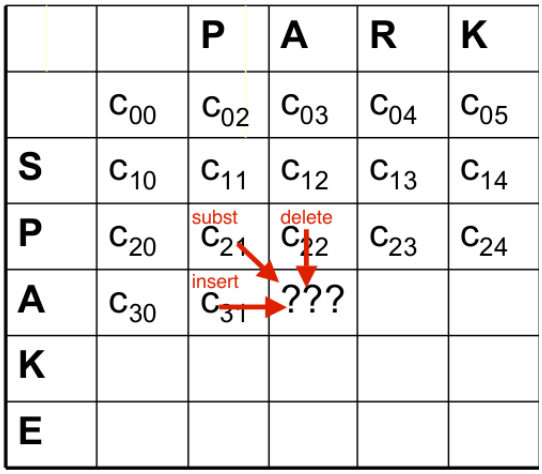
```
casa --> cala (sustitución de 's' por 'l')
Juan --> Juana (inserción de 'a' al final)
calla --> calle (sustitución de 'a' por 'e')
```

Veamos el proceso de cambiar $spake -- > park$:




```
spake --> park
spake --> pake (eliminar "s")
pake --> parke (insertar "r")
parke --> park (eliminar  "e")
```

Veamos como es el proceso general:



En este gráfico podemos ver el proceso de inducción. Para encontrar coincidencias aproximadas en una cadena hay que, primero definir una función de costo que nos diga la distancia entre dos cadenas. Minimizar la distancia, también minimiza el costo de los cambios para convertir una cadena en otra.

- Substitución consiste en cambiar un único caracter del patrón s a un caracter diferente en el texto t como el cambio de *cala* a *casa*.
- Inserción consiste en insertar un único carácter en el patrón s para ayudarlo a coincidir con el texto t, como el cambio de *Juan* a *Juana*.

- Eliminación consiste en eliminar un único caracter en el patrón s para ayudarlo a coincidir con el texto t, como el cambio de *casa* a *asa*.

Podemos darnos cuenta que el costo de remplazar es 0 cuando son iguales porque no es necesario hacer nada, 1 en otros casos porque remplazamos un solo caracter.

Para conseguir que la cuestión sobre la semejanza de cadenas sea una pregunta con sentido, es necesario que fijemos el costo de cada una de estas operaciones de transformación. El costo de insertar o borrar sera 1, podemos darnos cuenta que el costo de remplazar es 0 cuando son iguales porque no es necesario hacer nada, 1 en otros casos porque remplazamos un solo caracter.

Los casos base que terminan la recursión se dan en los extremos. Cuanto tenemos 0 caracteres en una de las dos cadenas se tiene que insertar todos los carateres de la otra. Y en en el otro sentido será exactamente igual.

Con esta información podemos plantear la ecuación de recurrencia:

$$D(i,j) = \begin{cases} j & \text{si i=0} \\ i & \text{si j=0} \\ minimo(D(i-1,j) + textcostoborrar, & \\ D(i,j-1) + textcostoinsertarr & \\ D(i-1,j-1) + textcostoremplazar) & \end{cases}$$

Ahora que tenemos la ecuación de recurrencia podemos escribir el programa con una solución recursiva directamente:

```
static int string_compare(String s, String t, int i, int j) {
    int k, costoMin;
    int opt[] = new int[3];
    if (i == 0)
        return j * insdel(' ');
    if (j == 0)
        return i * insdel(' ');
    opt[EMPAREJA] = string_compare(s, t, i - 1, j - 1)
            + empareja(s.charAt(i), t.charAt(j));
    opt[INSERTAR] = string_compare(s, t, i, j - 1) + insdel(t.charAt(j));
    opt[BORRAR] = string_compare(s, t, i - 1, j) + insdel(s.charAt(i));
    costoMin = opt[EMPAREJA];
    for (k = INSERTAR; k <= BORRAR; k++)
        if (opt[k] < costoMin) {
            costoMin = opt[k];
        }
    return costoMin;
```

```
    }
```

La primera parte del programa lee los datos del teclado y llama a la implementación. Hemos creado un vector de tamaño 3 para almacenar los costos de emparejar, insertar y de borrar. Luego se escoge el menor de estos para devolver el óptimo de los subproblemas. Como ve, cada caso es una implementación directa de la ecuación de recurrencia.

¿Cuánto demora este programa? Al hacer las pruebas vemos que con cadenas de pocos caracteres demora excesivamente. ¿A qué se debe? Analizando la complejidad vemos que existen tres ramificaciones recursivas para obtener un valor, que significa una complejidad del orden de $O(n) = 3^n$. Además que los valores se calculan múltiples veces Siendo un algoritmo con un tiempo de proceso claramente exponencial ya tenemos la respuesta. Hay que aplicar los principios de la programación dinámica para obtener una solución en tiempo eficiente.

4.1 Solución con Programación Dinámica

Para implementar una solución hay plantear el pricipio de optimalidad.

$$D(n) = minimo \begin{cases} = D(i-1,j) + \ costo\ de\ borrar \\ = D(i,j-1) + \ costo\ de\ insertar \\ = D(i-1,j-1) + costo\ remplazo;\ cero\ si\ igualan \end{cases}$$

Las condiciones iniciales de este problema son:

- Las filas y columnas de la matriz representan cada una, una cadena.
- Insertar i caracteres cuesta i
- `m[0][i] = i;`
- borrar i caracteres cuesta i
- `m[i][0] = i;`

Ahora hay que recorrer la matriz de la programación dinámica hallando los costos intermedios

```
for (i = 1; i < s.length(); i++)
    for (j = 1; j < t.length(); j++) {
        opt[EMPAREJA] = m[i - 1][j - 1] + empareja(s.charAt(i), t.charAt(j));
        opt[INSERTAR] = m[i][j - 1] + insdel(t.charAt(j));
        opt[BORRAR] = m[i - 1][j] + insdel(s.charAt(i));
        m[i][j] = opt[EMPAREJA];
        for (k = INSERTAR; k <= BORRAR; k++)
            if (opt[k] < m[i][j]) {
```

```
                m[i][j] = opt[k];
            }
    }
return (m[s.length()-1][t.length()-1]);
```

¿Dónde está la solución?. La solución está en la última posición.

```
return (m[s.length()-1][t.length()-1]);
```

Ejemplo de una ejecución:

```
    p a r k
  0 1 2 3 4
s 1 1 2 3 4
p 2 1 2 3 4
a 3 2 1 2 3
k 4 3 2 2 2
e 0 1 2 3 3
```

Preguntas Finales. ¿Cómo es su complejidad? Analizando el código vemos que dos ciclos anidados hacen todo el trabajo. Esto significa que su complejidad es proporcional a n^2. Una vez más reducción de una complejidad exponencial a una complejidad polinomial.¿Dónde se aplica? Se aplica en correctores de ortografía, detección de plagio, procesos de lenguaje natural, secuencias ADN, etc. Finalmente Que hacer para tener el detalle de las acciones realizadas? Hay que guardar todas las decisiones óptimas y recorrer en orden inverso.

4.2 Reconstruir las acciones realizadas

No solo queremos hallar el valor mínimo, también queremos conocer todas las operaciones necesarias para hallar éste valor.

Para ejemplificar una solución general que puede aplicarse a diversos tipos de problemas, creamos una clase *cell* para que almacene el costo y el padre de cada una de las celdas seleccionadas:

```
static class cell {
    int costo, padre;
}
```

Corregimos el programa para que almacene en el atributo *costo* los valores del costo, y el valor del padre en el atributo *padre*.

La tabla almacenada del proceso de hallar la solución óptima es:

```
    p a r k
  0 1 2 3 4
s 1 1 2 3 4
p 2 1 2 3 4
a 3 2 1 2 3
k 4 3 2 2 2
e 0 1 2 3 3
```

Ahora podemos reconstruir el programa recursivamente. El caso base está en la posición $(0, 0)$ que tiene el valor -1. Comenzamos la reconstrucción desde el ultimo valor, repitiendo volviendo a ver si empareja, si hay que borrar, o insertar

```
static void reconstruct_path(String s, String t, int i, int j) {
    if (m[i][j].padre == -1) //fin cuando es m(0,0)
        return;
     if (m[i][j].padre == EMPAREJA) {
        reconstruct_path(s, t, i - 1, j - 1);
        empareja_out(s, t, i, j);
        return;
    }
    if (m[i][j].padre == INSERTAR) {
        reconstruct_path(s, t, i, j - 1);
        insertar_out(t, j);
        return;
    }
    if (m[i][j].padre == BORRAR) {
        reconstruct_path(s, t, i - 1, j);
        borrar_out(s, i);
        return;
    }
}
```

4.3 Ejercicios

4.3.1 Ejercicio Palindrome

Un palíndrome es una cadena que se lee igual de izquierda a derecha como de derecha a izquierda. Por ejemplo, I, GAG y MADAM son palíndromos, pero ADAM no lo es. En este sentido, consideramos también la cadena vacía como un palíndrome.

De cualquier cadena no palindrómica, siempre se puede quitar algunas letras, y obtener una subsecuencia palindrómica. Por ejemplo, dada la cadena de ADAM, se elimina la letra M y obtener un palíndromo ADA.

Escriba un programa para determinar el palíndromo de mayor longitud que se puede obtener a partir de una cadena.

Entrada

La primera línea de entrada contiene un número entero ($T \leq 60$). Cada una de las siguientes T líneas es una cadena, cuya longitud es siempre menor que 1000.

Salida

Para cada cadena de entrada, el programa debe imprimir la longitud del mayor palíndrome se puede obtener mediante la eliminación de cero o más caracteres de la misma.

Ejemplos de entrada	**Ejemplos de salida**
2 ADAM MADAM	

4.3.2 Solución

Primero fijamos el caso base. Observamos que hay dos casos: el primero cuando la cadena es de longitud par y el segundo cuando es de longitud impar.

Cuando la cadena de longitud impar y $(l = r)$ entonces $len(l, r) = 1$.

Cuando la cadena de longitud par si $(l + 1 = r)$ y $a[r] = a[l]$ entonces $len(l, r) = 2$ en otros casos es 1.

Si ambos caracteres son iguales $a[r] = a[l]$ entonces $len(l, r) = 2 + len(l + 1, r - 1)$.

Caso contrario incrementar o decrementar un lado que nos da la siguiente ecuación de recurrencia $len(l, r) = max(len(r, l - 1), len(r - 1, l))$

La codificación del programa queda como sigue:

```
static int longestPalin(int l, int r) {
```

```
        if (l == r)
            return 1;
        if (l + 1 == r) {
            if (s.charAt(l) == s.charAt(r))
                return 2;
            else
                return 1;
        }
        if (s.charAt(l) == s.charAt(r))
            return dp[l][r] = 2 + longestPalin(l + 1, r - 1);
        else
            return dp[l][r] = Math.max(longestPalin(l, r - 1), longestPalin(l + 1, r)
    }
```

4.4 Programas mencionados en el texto

4.4.1 Solución recursiva al problema distancia de edición

```
import java.util.Scanner;
/*
 * Solucion recursiva al problema Distancia de Edicion
 * Autor Jorge Teran
 */

public class DistanciaRecursivo {

    static final int EMPAREJA = 0;
    static final int INSERTAR = 1;
    static final int BORRAR = 2;
    public static void main(String[] args) {
        String s = new String();
        String t = new String();
        Scanner sc = new Scanner(System.in);
        System.out.println("Ingrese String s");
        s = sc.nextLine();
        System.out.println("Ingrese String t");
        t = sc.nextLine();
        s = " " + s;
        t = " " + t;
        System.out.printf("Mejor Costo = %d \n",
                string_compare(s, t, s.length() - 1, t.length() - 1));
    }
    static int string_compare(String s, String t, int i, int j) {
        int k, costoMin;
```

```
        int opt[] = new int[3];
        if (i == 0)
            return j * insdel(' ');
        if (j == 0)
            return i * insdel(' ');
        opt[EMPAREJA] = string_compare(s, t, i - 1, j - 1)
                + empareja(s.charAt(i), t.charAt(j));
        opt[INSERTAR] = string_compare(s, t, i, j - 1) + insdel(t.charAt(j
        opt[BORRAR] = string_compare(s, t, i - 1, j) + insdel(s.charAt(i));
        costoMin = opt[EMPAREJA];
        for (k = INSERTAR; k <= BORRAR; k++)
            if (opt[k] < costoMin) {
                costoMin = opt[k];
            }
        return costoMin;
    }
    static int insdel(char c) {
        return 1;
    }

    static int empareja(char c, char d) {
        if (c == d)
            return 0;
        return 1;
    }
}
```

4.4.2 Solución no recursiva al problema distancia de edición

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * Solucion no recursiva al problema Distancia de Edicion
 * Autor Jorge Teran
 */

public class DistanciaNoRecursivo {
    static final int EMPAREJA = 0;
    static final int INSERTAR = 1;
    static final int BORRAR = 2;
    static int [][]m;
    public static void main(String[] args) {
        String s = new String();
        String t = new String();
        Scanner sc = new Scanner(System.in);
```

```
        System.out.println("Input String s");
        s = sc.nextLine();
        System.out.println("Input String t");
        t = sc.nextLine();
        s = " " + s;
        t = " " + t;
        System.out.printf("Mejor Costo = %d \n",
                string_compare(s, t));
    }

    static int string_compare(String s, String t) {
        int i = 0, j = 0, k = 0; /* contadores */
        int opt[] = new int[3]; /* costo de las tres opciones */
        m = new int[s.length()+ 1][t.length() + 1]; //matriz de PD
        for (i = 0; i < s.length(); i++) {
            row_init(i);
        }
        for (i = 0; i < t.length(); i++) {
            column_init(i);
        }
        for (i = 1; i < s.length(); i++)
            for (j = 1; j < t.length(); j++) {
                opt[EMPAREJA] = m[i - 1][j - 1] + empareja(s.charAt(i), t.c
                opt[INSERTAR] = m[i][j - 1] + insdel(t.charAt(j));
                opt[BORRAR] = m[i - 1][j] + insdel(s.charAt(i));
                m[i][j] = opt[EMPAREJA];
                for (k = INSERTAR; k <= BORRAR; k++)
                    if (opt[k] < m[i][j]) {
                        m[i][j] = opt[k];
                    }
            }
       // printMatrix(s,t);
        return (m[s.length()-1][t.length()-1]);
    }

    static int insdel(char c) {
        return 1;
    }

    static int empareja(char c, char d) {
        if (c == d)
            return 0;
        return 1;
    }
```

```
    static void row_init(int i) {
        m[0][i] = i;
    }

    static void column_init(int i) {
        m[i][0] = i;
    }

    static void printMatrix(String t, String s) {
        char[] s1 = s.toCharArray();
        char[] t1 = t.toCharArray();
        for (int i = 0; i < s1.length; i++) {
            System.out.print(" "+s1[i]);
        }
        System.out.println();
        for (int i = 0; i < t.length(); i++) {
            System.out.print(t1[i]+" ");
            for (int j = 0; j < s.length(); j++) {
                System.out.print(m[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

4.4.3 Reconstruir la solución 1 del problema distancia de edición

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * Solucion recursiva al problema Distancia de Edicion
 * cada celda tiene costo y predecesor (padre)
 * probamos colocando el costo
 * El codigo original esta en el libro de Skiena y Revilla
 * Autor Jorge Teran
 */
public class DistanciaBack1 {
    static final int EMPAREJA = 0;
    static final int INSERTAR = 1;
    static final int DELETE = 2;
    static cell[][] m = new cell[100][100]; // matriz de PD

    public static void main(String[] args) {
        String s = new String();
        String t = new String();
```

```
        Scanner sc = new Scanner(System.in);
        System.out.println("Ingrese String s");
        s = sc.nextLine();
        System.out.println("Ingrese String t");
        t = sc.nextLine();
        s = " " + s;
        t = " " + t;
        System.out.printf("Mejor Costo = %d \n", string_compare(s, t));
    }

    static int string_compare(String s, String t) {
        int i = 0, j = 0, k = 0; /*contadores */
        int opt[] = new int[3]; /* costo de las tres opciones */

        for (int l = 0; l < 100; l++)
            for (int l2 = 0; l2 < 100; l2++)
                m[l][l2] = new cell();

        for (i = 0; i < s.length(); i++) {
            row_init(i);
        }
        for (i = 0; i < t.length(); i++) {
            column_init(i);
        }
        for (i = 1; i < s.length(); i++)
            for (j = 1; j < t.length(); j++) {
                opt[EMPAREJA] = m[i - 1][j - 1].costo
                        + empareja(s.charAt(i), t.charAt(j));
                opt[INSERTAR] = m[i][j - 1].costo + insdel(t.charAt(j));
                opt[DELETE] = m[i - 1][j].costo + insdel(s.charAt(i));
                m[i][j].costo = opt[EMPAREJA];
                for (k = INSERTAR; k <= DELETE; k++)
                    if (opt[k] < m[i][j].costo) {
                        m[i][j].costo = opt[k];
                    }
            }
         printMatrix(s,t);
        return (m[s.length() - 1][t.length() - 1].costo);
    }

    static int insdel(char c) {
        return 1;
    }

    static int empareja(char c, char d) {
```

```
        if (c == d)
            return 0;
        return 1;
    }

    static void row_init(int i) {
        m[0][i].costo = i;
    }

    static void column_init(int i) {
        m[i][0].costo = i;
    }

    static void printMatrix(String t, String s) {
        char[] s1 = s.toCharArray();
        char[] t1 = t.toCharArray();
        for (int i = 0; i < s1.length; i++) {
            System.out.print(" "+s1[i]);
        }
        System.out.println();
        for (int i = 0; i < t.length(); i++) {
            System.out.print(t1[i]+" ");
            for (int j = 0; j < s.length(); j++) {
                System.out.print(m[i][j].costo + " ");
            }
            System.out.println();
        }
    }

    static class cell {
        int costo, padre;
    }
}
```

4.4.4 Reconstruir la solución 2 del problema distancia de edición

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * Solucion recursiva al problema Distancia de Edicion
 * cada celda tiene costo y predecesor (padre)
 * Agregamos el predecesor y el metodo para imprimir
 * El codigo original esta en el libro de Skiena y Revilla
 * Autor Jorge Teran
 */
```

```
public class DistanciaBack2 {
    static final int EMPAREJA = 0;
    static final int INSERTAR = 1;
    static final int BORRAR = 2;
    static cell[][] m = new cell[100][100]; // matriz de PD

    public static void main(String[] args) {
        String s = new String();
        String t = new String();
        Scanner sc = new Scanner(System.in);
        System.out.println("Ingrese String s");
        s = sc.nextLine();
        System.out.println("Ingrese String t");
        t = sc.nextLine();
        s = " " + s;
        t = " " + t;
        System.out.printf("Mejor costo = %d \n", string_compare(s, t));
        reconstruct_path(s, t, s.length() - 1, t.length() - 1);
    }

    static int string_compare(String s, String t) {
        int i = 0, j = 0, k = 0; /*contadores */
        int opt[] = new int[3]; /* costo de las tres opciones */

        for (int l = 0; l < 100; l++)
            for (int l2 = 0; l2 < 100; l2++)
                m[l][l2] = new cell();

        for (i = 0; i < s.length(); i++) {
            row_init(i);
        }
        for (i = 0; i < t.length(); i++) {
            column_init(i);
        }

        for (i = 1; i < s.length(); i++)
            for (j = 1; j < t.length(); j++) {
                opt[EMPAREJA] = m[i - 1][j - 1].costo
                        + empareja(s.charAt(i), t.charAt(j));
                opt[INSERTAR] = m[i][j - 1].costo + insdel(t.charAt(j));
                opt[BORRAR] = m[i - 1][j].costo + insdel(s.charAt(i));
                m[i][j].costo = opt[EMPAREJA];
                for (k = INSERTAR; k <= BORRAR; k++)
                    if (opt[k] < m[i][j].costo) {
                        m[i][j].costo = opt[k];
```

```
                    m[i][j].padre = k;
                }
            }
        printMatrix(s, t);
        return (m[s.length() - 1][t.length() - 1].costo);
    }

    static int insdel(char c) {
        return 1;
    }

    static int empareja(char c, char d) {
        if (c == d)
            return 0;
        return 1;
    }

    static void row_init(int i) {
        m[0][i].costo = i;
        // Inicializar los padres
        // fila =insertar
        if (i > 0)
            m[0][i].padre = INSERTAR;
        else
            m[0][i].padre = -1;
    }

    static void column_init(int i) {
        m[i][0].costo = i;
        // Inicializar los padres
        // columna = borrar
        if (i > 0)
            m[i][0].padre = BORRAR;
        else
            m[0][i].padre = -1;
    }

    static void printMatrix(String t, String s) {
        char[] s1 = s.toCharArray();
        char[] t1 = t.toCharArray();
        for (int i = 0; i < s1.length; i++) {
            System.out.print(" "+s1[i]);
        }
        System.out.println();
        for (int i = 0; i < t.length(); i++) {
```

```
            System.out.print(t1[i]+" ");
            for (int j = 0; j < s.length(); j++) {
                System.out.print(m[i][j].padre + " ");
            }
            System.out.println();          }
    }

    static void reconstruct_path(String s, String t, int i, int j) {
        if (m[i][j].padre == -1) //fin cuando es m(0,0)
            return;

        if (m[i][j].padre == EMPAREJA) {
            reconstruct_path(s, t, i - 1, j - 1);
            empareja_out(s, t, i, j);
            return;
        }
        if (m[i][j].padre == INSERTAR) {
            reconstruct_path(s, t, i, j - 1);
            insertar_out(t, j);
            return;
        }
        if (m[i][j].padre == BORRAR) {
            reconstruct_path(s, t, i - 1, j);
            borrar_out(s, i);
            return;
        }
    }

    static void empareja_out(String s, String t, int i, int j) {
        if (s.charAt(i) == t.charAt(j))
            System.out.printf("M");
        else
            System.out.printf("S");
    }

    static void insertar_out(String t, int j) {
        System.out.printf("I");
    }

    static void borrar_out(String s, int i) {
        System.out.printf("D");
    }

    static class cell {
        int costo, padre;
```

```
    }
}
```

4.4.5 Solución al ejercicio palindrome

```
import java.util.Scanner;
/*
 * Solucion al Ejercicio palindrome MÃ¡ximo
 * Autor Jorge Teran
 */
public class Palindrome {

    static int dp[][];;
    static String s;
    public static void main(String[] args) {
        int t;
        Scanner lee = new Scanner(System.in);
        t= lee.nextInt();
        while (t-->0) {
            s= lee.next();
            dp = new int[s.length()+1][s.length()+1];
            System.out.println(longestPalin(0, s.length() - 1));
        }
    }
    static int longestPalin(int l, int r) {
        if (l == r)
            return 1;
        if (l + 1 == r) {
            if (s.charAt(l) == s.charAt(r))
                return 2;
            else
                return 1;
        }
        if (s.charAt(l) == s.charAt(r))
            return dp[l][r] = 2 + longestPalin(l + 1, r - 1);
        else
            return dp[l][r] = Math.max(longestPalin(l, r - 1), longestPalin
    }
}
/*
3
ADA
ADAM
MADAM
```

*/

5 La subsecuencia común más grande

El problema de la subsecuencia común mas grande (LCS por sus siglas en inglés: longest common subsequence) es un problema combinatorio que es una medición de similitud y surge naturalmente en distintas aplicaciones prácticas, como búsqueda de patrones en moléculas de ADN, comparación de archivos, etc.

Dadas dos secuencias $A[1...M]$ y $B[1...N]$ diremos que $C[1...P]$ es una subsecuencia de B si la primera puede obtenerse a partir de la segunda borrando 0 o más letras conservando la posición relativa de las letras restantes.

Por ejemplo: *ad* es una subsecuencia de *abcde*.

Dados $A =$ "*abcde*" $B =$ "*bcdeadg*" hallar la subsecuencia más larga entre A y B se escribe como LCS(A,B). Subsecuencias comunes de A y B tenemos de varias longitudes. De longitud 2 tenemos *ad*, de longitud tres se tiene *bcd* y *bce*.

5.1 Solución por fuerza bruta

Supongamos que $A = bcdeadg$ todas las posibles secuencias son: $b, bc, c, bcd.....bcdeadg$ Cuántas subsecuencias existen? Podemos ver que existen 2^n subconjuntos que podemos formar. Podemos hacer lo mismo con $B = bcd$, luego hallamos las secuencias iguales y escogemos la más larga. Cuánto vale la complejidad? Tenemos 2^n posibles conjuntos que comparar esto nos dice que la complejidad será del orden de $O(n) = 2^n$ Es bueno porque produce la solución. ¿Será rápido? obviamente no pensemos por ejemplo en una cadena de 30 de caracteres longitud, esto requerirá comparar 1073741824 subconjuntos, que como se dará cuenta es mucho tiempo de proceso.

5.2 Solución utilizando programación dinámica

Para revolver éste problema utilizando programación dinámica, seguiremos con los siguientes pasos:

- Expresar el problema como un problema de búsqueda
- Espacio de búsqueda: secuencias comunes de A y B
- Función Objetivo: hallar la subcadena más larga

Para las soluciones de programación dinámica es muy conveniente expresar los problemas como un procedimiento recursivo. En nuestro caso:

```
buscar (S){
 dividir S en sub espacios S_1,S_2...S_m
 encontrar el mejor en cada Si
 Devolver el mejor de los mejores
}
```

Si podemos expresar como hallar el mejor S_i recursivamente tendremos una solución

Como tenemos 2 secuencias requeriremos una tabla (LCS) de dos dimensiones, Para almacenar los resultados previos. Después escribimos un procedimiento no recursivo que llene las entradas (i,j) de LCS asumiendo que otras entradas ya se llenaron. El tiempo de será igual al tamaño de la tabla multiplicado el tiempo de llenar cada entrada.

El problema inicial lo podemos dividir en sub problema con los siguientes criterios:

- $S = S_1 \cup S_2$
- S_1 son todas las secuencias que comienzan con $A[1]$
- S_2 el resto, comienzan con cualquier otra letra

¿Como se busca S_1?

- R es el elemento más largo de S y es $R[1....p]$
- $R[1] = A[1]$
- $R[1]$ debe aparecer en B
- $B[k] = R[1] = A[1]$
- $R[2..p]$ es una subsecuencia de $R[2...m]$ y $B[k+1...n]$

Sea $LCS(i,j)$ la subsecuencia común más larga de $A[1...i]$ y $B[1...j]$ Entonces los casos base son $LCS(0,j) = 0$ y $LCS(i,0) = 0$. Luego, $LCS(i,j) = 1 + LCS(i+1, j+1)$ si $A[i] = B[j]$, y el caso general $LCS(i,j) = max(LCS(i+1,j), LCS(i,j+1))$

Caracterizamos la ecuación de recurrencia

$$LCS(i,j) = \begin{cases} = 0 \text{ si } i = 0, j = 0 \\ = 1 + LCS(i+1, j+1) \text{ si } i, j > 0 \text{ y } A_i = B_j \\ = max(LCS(i+1,j), LCS(i,j+1)) \text{si } i, j > 0 \text{ y } A_i \neq B_j \end{cases}$$

5.3 Construyendo el código

Definimos nuestra matriz de programación dinámica $LCS(1...m, 1...n)$ y comencemos de atrás para adelante

```
for (int i = M - 1; i >= 0; i--) {
    for (int j = N - 1; j >= 0; j--) {
        if (a.charAt(i) == b.charAt(j))
            lcs[i][j] = lcs[i + 1][j + 1] + 1;
        else
            lcs[i][j] = Math.max(lcs[i + 1][j], lcs[i][j + 1]);
    }
}
```

El resultado estará en la primera posición de la matriz $LCS[0][0]$. ¿Cual es el tiempo de proceso? Como ve se tienen dos ciclos anidados que nos muestran que el tiempo de proceso es proporcional a n^2.

5.4 Reconstruyendo la solución

No solamente es necesario hallar el valor óptimo sino que es necesario reconstruir el el camino, esto es decir cual es la subsecuencia que tiene la longitud máxima. Para esto recorremos la matriz partiendo del valor óptimo y de acuerdo a la definición cuando igualan los valores de las dos cadenas imprimimos el valor. Para saber si debemos ir comparando los caracteres de la primera o segunda cadena usamos los resultados en la matriz que indica cual es el próximo valor a considerar. El programa es el siguiente:

```
int i = 0, j = 0;
   while (i < M && j < N) {
       if (a.charAt(i) == b.charAt(j)) {
           System.out.print(a.charAt(i));
           i++;
           j++;
       } else if (lcs[i + 1][j] >= lcs[i][j + 1])
           i++;
       else
           j++;
   }
   System.out.println();
```

Por ejemplo las cadenas $A = abcdef$ y $B = acfx$ producen la siguiente tabla:

```
[3, 2, 1, 0, 0]
[2, 2, 1, 0, 0]
[2, 2, 1, 0, 0]
[1, 1, 1, 0, 0]
[1, 1, 1, 0, 0]
[1, 1, 1, 0, 0]
[0, 0, 0, 0, 0]
LCS es: 3
```

Partiendo de los valores de las cadenas buscamos la primera ocurrencia donde tienen un caracter en común, es a y esta en la posición cero. Se imprime. Luego hay que comparar los valores $lcs[i+1][j] >= lcs[i][j+1]$ y vemos que son iguales lo que hará que se incremente i, o j dando la próxima posición de la cadena a. Este proceso finalmente en tiempo lineal imprimirá la secuencia buscada.

5.5 Programas mencionados en el texto

5.5.1 Solución al problema de la subsecuencia más larga

```
import java.util.Arrays;

public class Monedas {

    public static void main(String[] args) {
        // int t[] = { 1,2,3 };
         // int t[] = { 1,4,6 };
        //int t[] = { 1, 3, 8, 16, 22, 57, 103, 526 };
        int t[] = { 1, 5, 10, 25, 100 };
        int valor = 85;
        int c[][] = new int[t.length][valor + 1];
        for (int i = 0; i < t.length; i++)
            c[i][0] = 0;
        for (int j = 0; j <= valor; j++)
            c[0][j] = j;

        for (int i = 1; i < t.length; i++) {
            for (int j = 1; j <= valor; j++) {
                if (j < t[i])
                    c[i][j] = c[i - 1][j];
                else
                    c[i][j] = Math.min(c[i - 1][j], 1 + c[i][j
                            - t[i]]);
            }
        }
```

```
        System.out.println(c[t.length - 1][valor]);
        for (int[] i : c)
            System.out.println(Arrays.toString(i));
    }
}
```

6 Ejercicios Resueltos

Estos ejercicios pueden encontrarse en www.topcoder.com [5].

6.1 Recaudando fondos

En un pequeño pueblo todas las casas se han construido en círculo alrededor de un pozo de agua. Le han asignado la tarea de recaudar donativos para la restauración.

Cada residente del pueblo, ubicado al contorno del pozo, está deseoso de donar una cierta cantidad de dinero. Sin embargo nadie está de acuerdo en contribuir a un fondo en el cual sus vecinos que se encuentra en la puerta contigua, hayan contribuido. Los vecinos que viven en la puerta contigua siempre se listan consecutivamente. La última casa también es vecina de la primera.

Halle el máximo dinero que puede ser recaudado.

Por ejemplo, dada la secuencia de donativos $10, 3, 2, 5, 7, 8$, el valor máximo se obtiene con $10 + 2 + 7$. Es mejor $10 + 5 + 8$, sin embargo el 8 y 10 corresponden a vecinos. El valor máximo que se puede recaudar es 19.

Entrada

La entrada consiste de varios casos de prueba. La primera línea contiene un número ($2 \leq N \leq 40$) que representa el número de personas. Luego siguen ($0 \leq d_i \leq 1000$) valores separados por espacio representando el valor que cada vecino puede donar. Los casos de prueba terminan cuando no hay más datos.

Salida

Por cada caso de prueba escriba en una línea la máxima donación que puede obtener.

Ejemplos de entrada	**Ejemplos de salida**
6 10 3 2 5 7 8 2 11 15 7 7 7 7 7 7 7 7 10 1 2 3 4 5 1 2 3 4 5 40 94 40 49 65 21 21 106 80 92 81 679 4 61 6 237 12 72 74 29 95 265 35 47 1 61 397 52 72 37 51 1 81 45 435 7 36 57 86 81 72	19 15 21 16 2926

6.1.1 Estrategia de Solución

Primero analizamos las características del problema y vemos que hay dos secuencias que se deben analizar. La primera que es cuando comienzo de la primera posición tomando en cuenta los datos contiguos pertenecen a sus vecinos. La segunda cuando considero los datos que comienzan en su vecino es decir la segunda posición. El máximo de ambos da el resultado óptimo.

Veamos la primera secuencia. Para iniciar el vector de programación dinámica (pd) anotaremos los valores del caso base. Estos son el primer valor y el segundo, o sea, *pd[0]= al aporte del primer vecino* y *pd[1]= aporte del segundo vecino*. En el caso de que solo existieran dos vecinos la respuesta es el máximo de ambos.

El tercer valor que es $pd[2]$ es el valor máximo es el valor entre $pd[2] + pd[2-2]$ y $pd[2-1]$. Como los dos valores iniciales son los valores base debemos comenzar a iterar desde el segundo valor.

Para la segunda secuencia fijamos los valores base *pd[1]= al aporte del segundo vecino* y *pd[2]= aporte del tercer vecino*. De la misma forma debemos hacer $pd[2] + pd[2-2]$ y $pd[2-1]$ hasta llegar a $n-2$.

¿Por qué va de dos en dos? Esto porque no podemos sumar con su vecino.

¿Por qué va de uno en uno? Porque si decidimos saltar un valor tomaremos el valor siguiente.

Consideremos los siguientes valores $10, 3, 2, 5, 7, 8$ las secuencias generarían los siguiente valores: $[10, 3, 12, 12, 19, 0]$ y $[0, 3, 2, 8, 9, 16]$ Dando como respuesta el valor máximo que es 19

6.1.2 Programa que revuelve el problema

```
import java.util.*;

public class Vecino {
    static int d[] = { 10, 3, 2, 5, 7, 8 };
    //static int d[] = { 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 };
    //static int d[] = { 1,10,1,1,10,1 };
    static int[] dp1=new int[d.length];
    static int[] dp2=new int[d.length];

    public static void main(String[] args) {
        dp1[0]=d[0];
        dp1[1]=d[1];
        int s=d.length;
        for(int i=2;i<s-1;i++){
            dp1[i]=Math.max(dp1[i-1],(dp1[i-2]+d[i]));
        }
        System.out.println(Arrays.toString(dp1));
        int ans1=dp1[s-2];

        dp2[1]=d[1];
        dp2[2]=d[2];
        for(int i=3;i<s;i++){
            dp2[i]=Math.max(dp2[i-1],(dp2[i-2]+d[i]));
        }
        System.out.println(Arrays.toString(dp2));
        int ans2=dp2[s-1];

        System.out.println(Math.max(ans1,ans2));
    }
}
```

6.2 Secuencias ZigZag

Una secuencia de números se denomina *zig-zag* si la diferencia de números sucesivos alterna entre números negativos y positivos. La primera diferencia si existe puede ser positiva o negativa.

Una secuencia con dos o menos números es la secuencia trivial.

Por ejemplo la secuencia $(1, 7, 4, 9, 2, 5)$ es *zig-zag* porque las diferencias $(6, -3, 5, -7, 3)$ son alternadamente positivas y negativas. En contraste la secuencia $(1, 4, 7, 2, 5)$ no es una secuencia *zig-zag* porque la primera diferencia es positiva, La secuencia $(1, 7, 4, 5, 5)$ tampoco es una secuencia *zig-zag* porque la última diferencia es cero.

Dada una secuencia de enteros, devuelva la longitud máxima de la subsecuencia *zig-zag* que se puede formar. Una subsecuencia se obtiene borrando algunos elementos (tal vez cero) de la secuencia original, dejando los elementos restantes en su orden original.

Entrada

La entrada consiste de varios casos de prueba. La primera línea de cada caso de prueba tiene el numero ($0 \leq N \leq 50$ de elementos de la secuencia. La segunda línea de cada caso de prueba contiene los ($0 \leq N_i \leq 1000$) números de la secuencia separados por un espacio. La entrada termina cuando no hay más datos.

Salida

Para cada caso de prueba imprima en una línea la máxima subsecuencia zigzag que se pueda formar.

Ejemplos de entrada	**Ejemplos de salida**
6 1 7 4 9 2 5 10 1 17 5 10 13 15 10 5 16 8 1 44 9 1 2 3 4 5 6 7 8 9 19 70 55 13 2 99 2 80 80 80 80 100 19 7 5 5 5 1000 32 32 50 374 40 854 203 203 156 362 279 812 955 600 947 978 46 100 953 670 862 568 188 67 669 810 704 52 861 49 640 370 908 477 245 413 109 659 401 483 308 609 120 249 22 176 279 23 22 617 462 459 244	6 7 1 2 8 36

6.2.1 Estrategias de solución

Para resolver este problema mostraremos dos estrategias.

Para esquematizar la primera estrategia, vemos que existen dos secuencias una que comienza con una diferencia negativa y otra con una diferencia positiva. Para estos dos casos vamos a crear dos vectores para almacenar las soluciones intermedias. Para cada una de estas soluciones hay que ir recorriendo el vector para las probables secuencias, donde las soluciones se encuentran de la siguiente forma: $dp[0][i] = Math.max(dp[1][j] + 1, dp[0][i])$;. Finalmente se halla el máximo de las dos secuencias que es la solución final.

La segunda estrategia es la de llevar la cuenta del signo en un vector separado. Esto hace que se requiera un vector de signos y un vector de resultados intermedios. Existen los siguientes casos:

1. $signo == 0$ Al principio se puede escoger diferencia positiva o negativa
2. $(signo < 0)\&\&(a[i] - a[j] > 0)$ El valor previo fue negativo así que requerimos una diferencia positiva
3. $(signo > 0)\&\&(a[i] - a[j] < 0)$ El valor previo fue positivo, requerimos una diferencia negativa

Una vez encontrado el elemento, almacenar el signo para la próxima iteración

Para cada i, debemos averiguar si el siguiente elemento forma una secuencia, puede ser el que esté a la derecha de i o el que esté en el índice 0. Por esto es necesario recorrer todos los elementos desde el índice $i - 1$ hasta el índice 0.

6.2.2 Programa para la estrategia 1

```
import java.util.Arrays;


public class Zigzag2 {

    public static void main(String[] args) {
        //int [] a={ 1, 17, 5, 10, 13, 15, 10, 5, 16, 8 };
        int a[] = {396, 549, 22, 819, 611, 972, 730, 638, 978, 342, 566, 51
        int n=a.length;
        System.out.println(n);
        int dp[][]= new int[2][n];
        Arrays.fill(dp[0], 1);
        Arrays.fill(dp[1], 1);
```

```
        for (int i = 0; i < n; i++) {
            for(int j=0;j<=i-1;j++){
                if (a[i] - a[j] > 0)
                    dp[0][i] = Math.max(dp[1][j] + 1, dp[0][i]);
                else if (a[i] - a[j] < 0)
                    dp[1][i] = Math.max(dp[0][j] + 1, dp[1][i]);

            }
        }
        System.out.println(Arrays.toString(dp[0]));
        System.out.println(Arrays.toString(dp[1]));
    }
}
```

6.2.3 Programa para la estrategia 2

```
import java.util.Arrays;


public class Zigzag {

    public static void main(String[] args) {
        int [] a={ 1, 17, 5, 10, 13, 15, 10, 5, 16, 8 };
        //int a[] = {396, 549, 22, 819, 611, 972, 730, 638, 978, 342, 566,
        int SIZE=a.length;
        int signos[]= new int[SIZE];
        int dp[]= new int[SIZE];
        Arrays.fill(dp, 1); // caso base;
        for(int i=1;i<SIZE;i++){
            for(int j = i-1; j>=0; j--){
                int signo = signos[j];
                if( (signo == 0) || ( (signo < 0) && (a[i] - a[j] > 0) ) ||

                    dp[i] = dp[j] + 1;
                    signos[i] = a[i] - a[j];
                    break;
                }
            }
        }

        System.out.println(dp[SIZE-1]);
        System.out.println(Arrays.toString(dp));
    }
}
```

6.3 La Subsecuencia ascendente más larga

La secuencia ascendente más larga se define como la secuencia más larga que podemos hallar quitando algunos elementos. Para solucionar este problema también utilizaremos programación dinámica. Por ejemplo sea la secuencia: $(10, 22, 9, 33, 21, 50, 41, 60, 80)$. La secuencia ascendente más larga que podemos encontrar es de longitud 6 y es $(10, 22, 33, 50, 60, 80)$. Si vemos la secuencia $(6, 3, 5, 2, 7, 8, 1)$ tenemos las siguientes secuencias:

- Subsecuencia no acendente: $(5, 2, 1)$
- Subsecuencia ascendente: $(3, 5, 9)$
- Subsecuencia ascendente: $(2, 7, 8)$
- La subsecuencia ascendente más larga: $(3, 5, 7, 8)$

6.3.1 Estrategia de Solución

Para resolver este problema construimos un vector de programación dinámica donde almacenaremos las soluciones intermedias. Para cada elemento del vector es necesario comparar todos los elementos para ver si hay un nuevo máximo. Cada vez que analizamos un nuevo elemento incrementamos en uno el valor anterior en el vector de programación dinámica.

Como cada elemento del vector de dp tiene el máximo hasta ese elemento es necesario buscar el valor máximo de este, que será la solución.

6.3.2 Programa que resuelve el problema

```
import java.util.Arrays;

public class Longest {

    public static void main(String[] args) {
        //int a[] = { 10, 22, 9, 33, 21, 50, 41, 60, 80 };
        int a[]={6, 3, 5, 2, 7, 8, 1};
        int n = a.length;
        int dp[] = new int[n];
        int max = 0,i,j;
        for (i = 0; i < n; i++) {
            max = 0;
            for (j = 0; j < i; j++) {
                if(a[j]<a[i]&&dp[j]>max)
                    max = dp[j];
```

```
                }
                dp[j] = max + 1;
            }
            max=0;
            for (int j2 = 0; j2 < dp.length; j2++) {
                if (dp[j2]>max)
                    max=dp[j2];

            }
            System.out.println(max);
            System.out.println(Arrays.toString(dp));
        }
}
```

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3th edition, 2009.

[2] Steven Skiena and Miguel A. Revilla. *Programming challenges: the programming contest training manual*. Springer-Verlag, 1st edition, 2003.

[3] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 2nd edition, 2010.

[4] Jorge Teran Pomier. *Fundamentos de programación*. Ed. Apolo, 2 edition, 2010.

[5] Topcoder. Ejercicios tomados traducidos de www.topcoder.com @ONLINE, May 2013.