Instituto Politécnico Nacional

Escuela Superior de Cómputo

Sesión 9a: Programación Dinámica II

Integrantes:

Escutia López Arturo

12/DIC/2016

Análisis de Algoritmos

Problemas NP Y P

Normalmente cuando hablamos sobre algoritmos, lo que nos interesa saber es que tanto tiempo requiere ejecutar dicho algoritmo. Gran parte de los algoritmos más comunes tienen tiempos de ejecución de tamaño N^2 o N veces logaritmo de N o similares. Una expresión matemática de éste estilo que involucra N´s ,N^2´s o N´s elevadas a ciertas potencias son denominados como polinomiales , es por eso que se les conoce como problemas P .

P es un conjunto de problemas cuyo tiempo de ejecución son polinomiales. Ahora para darnos una idea sobre los problemas NP consideremos lo siguiente.

Supongamos que un algoritmo cuya complejidad es O(N) le toma un tiempo de ejecución de un segundo por 100 elementos, un algoritmo que tiene una complejidad de O(N^3) le tomaría aproximadamente 3 horas, pero un algoritmo cuya complejidad es O(2^N) le tomaría alrededor de 300 quintillones de años.

Los problemas NP (nondeterministic polynomial time) es el conjunto de problemas cuya solución requiere de tiempos exponenciales para ser resueltos. El problema más común asociado a los problemas NP, es el representar un número muy grande en factor de números primos.

Si cualquier problema NP-completo se encuentra contenido en P, entonces se verificaría que P = NP. Desafortunadamente, se ha demostrado que muchos problemas importantes son NP-completos y no se conoce la existencia de ningún algoritmo rápido para ellos.

Algunos ejemplos de problemas np:

- Problema de satisfacibilidad booleana (SAT)
- Problema de la mochila (knapsack)
- Problema del ciclo hamiltoniano
- Problema del vendedor viajero
- Problema de la clique

1.-

```
Void MatrixC(array size[1 .. n])
        solution[size.length+1][size.length+1]
        cost[size.length+1][size.length+1]
        FOR L = 1 TO N DO                                   <-O(n)
            FOR i = 0 TO N − L + do                         <-O(n)*O(n-L)
                j = i + L ;
                cost[i, j] = infinity;
                FOR k = i TO j -1 DO                          <- O(n)*O(n-L)*O(i+L)
                    w= cost[i, k] + cost[k + 1, j] + size[i ] size[k+1] p[j+1];
                    IF (w< cost[i, j])
                        cost[i, j] <- w;
                        solution[i, j] = k;
                    END IF
                END FOR
            END FOR
        END FOR
    END MatrixC
```

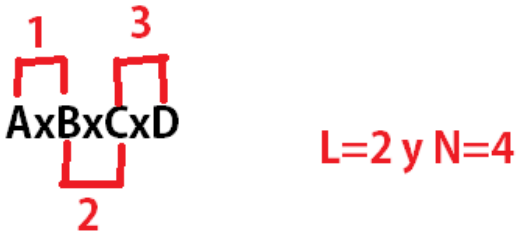
Complexity O(n^3)

```
private void matrixC(int size[]){
    int n=size.length-1;
    int[][] Cost=new int[n][n];
    int solution[][]=new int[n][n];
    int j,w;
    for(int L=1;L<n;L++){
        for(int i=0;i<n-L;i++){
            j=i+L;
            Cost[i][j]=Integer.MAX_VALUE;
            for(int k=i;k<j;k++){
                w=Cost[i][k]+Cost[k+1][j]+size[i]*size[k+1]*size[j+1];
                if(w<Cost[i][j]){
                    Cost[i][j]=w;
                    solution[i][j]=k;
                }
            }
        }
    }

    PrintS(solution,0,solution.length-1);
    Line=Line+" Costo:"+Cost[0][Cost.length-1];
    WFile();
    //System.out.println(" Costo:"+Cost[0][Cost.length-1]);
}
```

Ésta primer función recibe primeramente el arreglo con las dimensiones de las matrices, creamos dos matrices, una que guardara los costos de valores ya precalculados para ser consultados, y otra donde nos facilitara imprimir la solución. Dada n matrices A1,A2,AN desde 1<=i<=N la dimensión de Ai de nxm se define como size[i-1]xsize[i]. Primeramente nos encontramos con un ciclo for que empieza desde 1 hasta N donde N es el total de matrices recibidas, este primer for lo que nos va a permitir definir es la longitud o el número de matrices a multiplicar,es decir si L=2 representa el caso para las multiplicación entre dos

matrices,si L=3 para multiplicar 3 matrices y así sucesivamente. El siguente for es para delimitar el numero posibles combinaciones de multiplicaciones de longitud L, por ejemplo



Para ver cuántas veces es posible multiplicar dos matrices en este caso son 3 para 4 matrices.

En este otro ejemplo el número de matrices a multiplicar es de 3, por lo cual 3 son las posibles subsequencias a escoger de entre las 5 matrices. El ultimo for nos permite delimitar la última matriz que está incluida en la multiplicación de longitud L y realizar el diferente tipo de asociaciones de esta subsequencia, retomando el ejemplo anterior para AxBxC se puede asociar ya sea Ax(BxC) ó (AxB)xC. Ahí entra en acción la matriz de costos para solo consultar el valor de cosas ya calculadas como AxB o AxC. Y se guarda el índice k en el otro arreglo.

```
private void PrintS(int[][]s,int i,int j) {
    if(i==j){
        Line=Line+"A"+i+" ";
        //System.out.print("A"+i+" ");
    }else{
        Line=Line+" ( ";
        //System.out.print("( ");
        PrintS(s,i,s[i][j]);
        PrintS(s,s[i][j]+1,j);
        Line=Line+" ) ";
        //System.out.print(" ) ");
    }
}
```

En esta función es la que nos permite imprimir la manera de multiplicar dependiendo del valor en S(i,j) es donde determina hasta donde asociar, si i==j

imprime esa Matriz ya que no es posible asociarla más, sino que siga asociando y agregando ().Por ejemplo para 6 matrices A1*A2*A3*A4*A5*A6

PrintS(s,1,6) donde s[1,6]=3 (A1 A2 A3 A4 A5 A6)

PrintS(s,1,3) donde s[1,3]=1 (A1 A2 A3)

PrintS(s,1,1) i=j por lo tanto (A1 A2 A3)

PrintS(s,2,3) donde s[2,3]=2 (A1 (A2 A3))

PrintS(s,2,2) i=j por lo tanto (A1 (A2 A3))

PrintS(s,3,3) i=j por lo tanto (A1 (A2 A3))

Este mismo proceso sucede en la otra mitad faltante desde A4 hasta A6, el mismo procedimiento de asociamiento, y así es como se obtiene el resultado final ((A1(A2 A3))((A4 A5) A6))

2.-

```
Allignment(String s1,String s2)
        for i=1 to length(A)        <-O(N)
          for j=1 to length(B)      <- O(M)*O(N)
           Match ← F(i-1,j-1) + S(Ai, Bj)          <-O(1)*O(N*M)
           GAP1← F(i-1, j) + d                     <-O(1)*O(N*M)
           GAP2 ← F(i, j-1) + d                    <-O(1)*O(N*M)
           matrix(i,j) ← max(Match, GAP1, GAP2)        <-O(1)*O(N*M)
        end for
        end for
        while (i > 0 or j > 0)      <-O(N)
        {
          if (i > 0 and j > 0 and F(i,j) == F(i-1,j-1) + S(Ai, Bj))
          {
            AlignmentA ← Ai + AlignmentA
            AlignmentB ← Bj + AlignmentB
            i ← i - 1
            j ← j - 1
          }
          else if (i > 0 and F(i,j) == F(i-1,j) + d)
          {
            AlignmentA ← Ai + AlignmentA
            AlignmentB ← "-" + AlignmentB
            i ← i - 1
          }
          else
          {
            AlignmentA ← "-" + AlignmentA
            AlignmentB ← Bj + AlignmentB
            j ← j - 1
```

}

end allignment

Complejidad O(N*M) pero si n=m O(n^2)

```
int n=min(s1.length(),s2.length())+1;
int m=max(s1.length(),s2.length())+1;
int match,gap1,gap2,s;
matrix=new int[n][m];

for(int i=1;i<n;i++){
    for(int j=1;j<m;j++){
        if(s1.charAt(i-1)==s2.charAt(j-1))
            s=1;
        else
            s=0;
        match=matrix[i-1][j-1]+s;
        gap1=matrix[i-1][j];
        gap2=matrix[i][j-1];
        matrix[i][j]=max(match,max(gap1,gap2));
    }
}
```

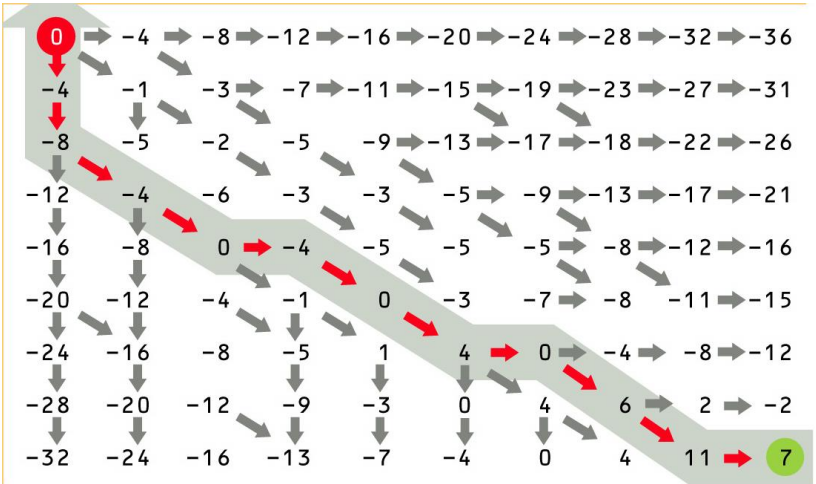
Como sabemos el algoritmos de Needleman-Wunsh nos permite mediante una matriz ir calculando las coincidencias, penalizaciones por espacios y no coincidencias en la primera y segunda palabra para obtener mediante una diagonal la mejor alineación posible, aquí es lo que sea realiza, se crea la matriz de nxm donde n es del tamaño una cadena y m de la otra. y se procede a llenar fila por fila, si existe una coincidencia se da cierto peso, en este caso de uno y en caso de agregar un espacio o de no haber una coincidencia se penaliza en este caso pues no sumamos ningún valor. Los valores se calculan mediante la posición actual sus vecinos superior,superior izquiero y su vecino inmediato izquierdo, y se le asigna a esa casilla el máximo, ya que lo que queremos es conseguir la secuencia con el menor número de penalizaciones hasta llegar a la casilla matrix[n][m]

```
while(i>0 && j>0){
    if(s1.charAt(i-1)==s2.charAt(j-1))
        s=1;
    else
        s=0;
    if(i>0 && j>0 && matrix[i][j]==matrix[i-1][j-1]+s){
        align_A=s1.charAt(i-1)+align_A;
        align_B=s2.charAt(j-1)+align_B;
        i=i-1;
        j=j-1;
    }else if(i>0 && matrix[i][j]==matrix[i-1][j]){
        align_A=s1.charAt(i-1)+align_A;
        align_B="-"+align_B;
        i=i-1;
    }else{
        align_A="-"+align_A;
        align_B=s2.charAt(j-1)+align_B;
        j=j-1;
    }

}
while(j>0){
    align_A="-"+align_A;
    align_B=s2.charAt(j-1)+align_B;
    j=j-1;
}

System.out.println(align_B);
System.out.println(align_A);
```

Aquí tiene la misma base que la parte anterior solo que partiendo de la última casilla matrix[n][m] iremos retrocediendo hasta matrix[0][0] mediante la ruta que se creó al calcular el matrix[n][m]. En caso de que nos movamos en diagonal existe una coincidencia y se imprime dicho carácter en ambas cadenas, cada movimiento por fila le agregamos un espacio a la segunda cadena y en caso de un movimiento por columna es una espacio en la primera cadena. Solamente queda rellenar los espacios en blanco en caso de que las cadenas sean de longitudes diferentes.

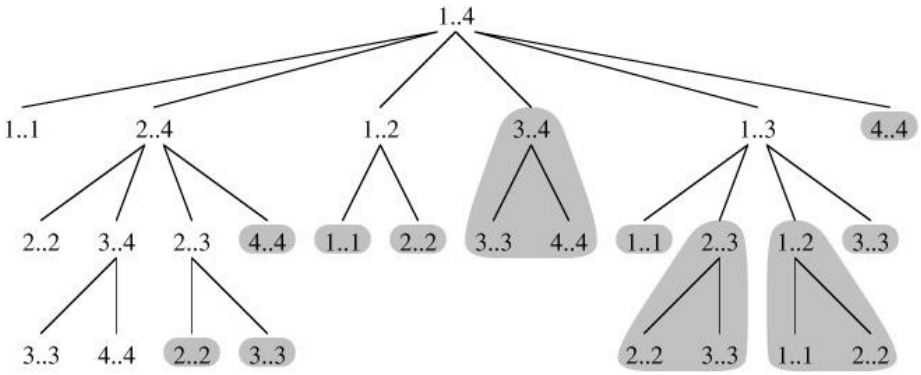


3.-

```
Int Mcost(int i, int j) {
     If(cost[i][j]!=-1) return cost[i][j]
     if (i = = j) m[i, i] = 0;
     else {
               m[i, j] = infinity;
               for k = i to j − 1 do {
                    cost=Rec-Matrix-Chain( i, k) + Rec-Matrix-Chain( k + 1, j) + p[i − 1]*p[k]*p[j];
                    if (cost<m[i, j]) then
                         m[i, j]<-cost
                         Solution[i][j]<-k
                    }
               }
      return m[i,j];
}
```

```
private int Mcost(int i,int j){
    if(cost[i][j]!=-1)
        return cost[i][j];
    else{
        if(i==j)
           cost[i][j]=0;
        else{
            cost[i][j]=Integer.MAX_VALUE;
            for(int k=i;k<j;k++){
                w=Mcost(i,k)+Mcost(k+1,j)+p[i-1]*p[k]*p[j];
                if(w<cost[i][j]){
                    cost[i][j]=w;
                    solution[i][j]=k;
                }
            }
        }
    }
    return cost[i][j];
}
```

Esta primera función es realizar todas las posibles combinaciones desde i hasta k y luego desde k+1 hasta j de manera recursiva y calculamos los pesos ,si los pesos ya fueron calculados previamente se retorna solamente el valor sino se calcula y se almacena y en dado caso que i=j no se calcula nada ya que no una matriz no se está multiplicando pos si misma, de una manera más visual se apreciaría de la siguiente forma

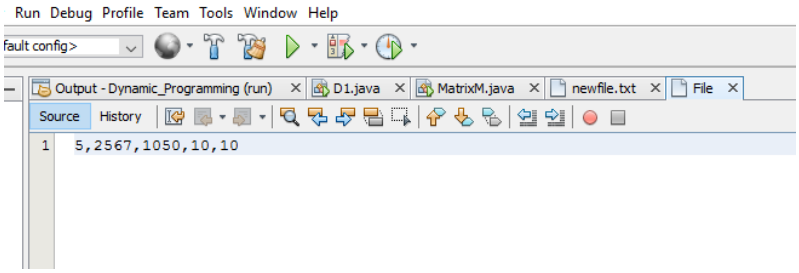


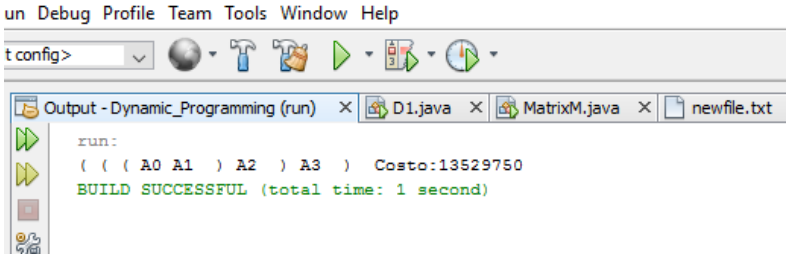
La parte sombreada indica aquellos costos que se repiten por lo cual ya no es necesario calcular nuevamente ya que fueron almacenados solamente se imprime la solución aplicada en el programa anterior

```
private void PrintS(int[][]s,int i,int j) {
    if(i==j){
        Line=Line+"A"+i+" ";
        //System.out.print("A"+i+" ");
    }else{
        Line=Line+" ( ";
        //System.out.print("( ");
        PrintS(s,i,s[i][j]);
        PrintS(s,s[i][j]+1,j);
        Line=Line+" ) ";
        //System.out.print(" ) ");
    }
}
```

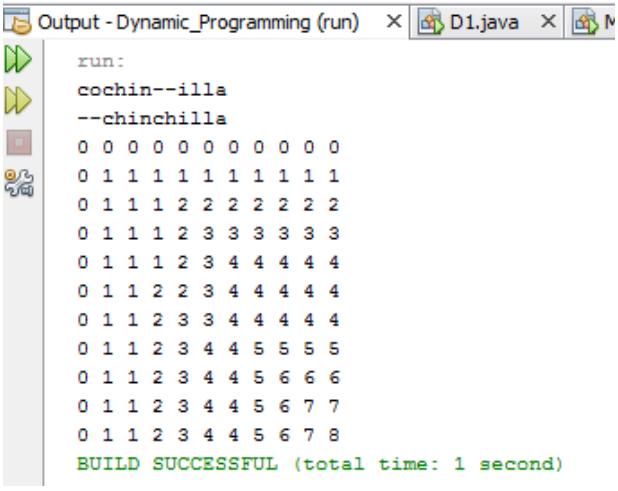
Pruebas

1.-

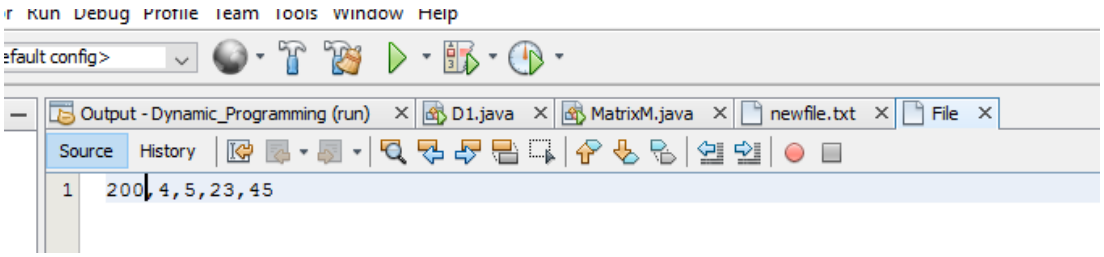


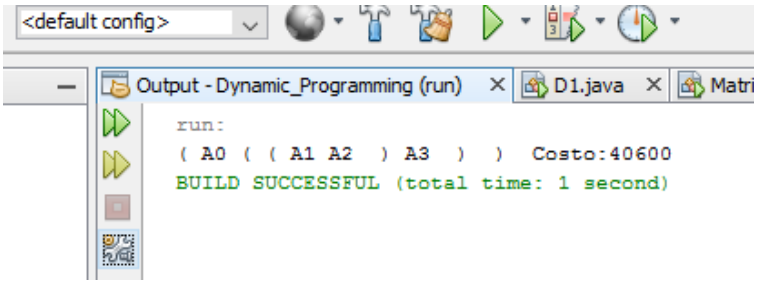


2.-



3.-


Run Debug Profile Team Tools Window Help
efault config>
Output - Dynamic_Programming (run)
D1.java
MatrixM.java
newfile.txt
File
Source
History
1
200 4,5,23,45


<default config>
Output - Dynamic_Programming (run)
D1.java
Matri
run:
(A0 ((A1 A2) A3)) Costo:40600
BUILD SUCCESSFUL (total time: 1 second)