Notebook ICPC-UFPS
Semillero de Investigación en Linux y Software Libre

Gerson Yesid Lázaro - Angie Melissa Delgado

2 de septiembre de 2015

Índice

1. Bonus: Input Output

1.1. Scanner

Libreria para recibir las entradas; reemplaza el Scanner original, mejorando su eficiencia.
Contiene los metodos next, nextLine y hasNext. Para recibir datos numericos, hacer casting de next.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.util.StringTokenizer;

public class Main {

    static class Scanner{
        InputStreamReader isr = new
            InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        StringTokenizer st = new StringTokenizer("");
        int espacios = 0;

        public String nextLine() throws IOException{
                if(espacios>0){
                espacios--;
                return "";
            }else if(st.hasMoreTokens()){
                StringBuilder salida = new StringBuilder();
                while(st.hasMoreTokens()){
                    salida.append(st.nextToken());
                    if(st.countTokens()>0){
                        salida.append(" ");
                    }
                }
                return salida.toString();
            }
            return br.readLine();
        }

        public String next() throws IOException{
                espacios=0;
            while (!st.hasMoreTokens() ) {
                st = new StringTokenizer(br.readLine() );
            }
            return st.nextToken();
        }

        public boolean hasNext() throws IOException{
            while (!st.hasMoreTokens()) {
                String linea = br.readLine();
                if (linea == null) {
                    return false;
                }
                if(linea.equals("")){
                        espacios++;
                }
                st = new StringTokenizer(linea);
            }
            return true;
        }
    }
}
```

1.2. printWriter

Utilizar en lugar del System.out.println para mejorar la eficiencia.

```
import java.io.PrintWriter;

PrintWriter so = new PrintWriter(System.out);
so.print("Imprime sin salto de linea");
so.println("Imprime con salto de linea");

//Al finalizar
bw.flush();
```

2. Dynamic Programming

Dados N articulos, cada uno con su propio valor y peso y un tamaño maximo de una mochila, se debe calcular el valor maximo de los elementos que es posible llevar.
Debe seleccionarse un subconjunto de objetos, de tal manera que quepan en la mochila y representen el mayor valor posible.

2.1. Knapsack

```
static int MAX_WEIGHT = 40;//Peso maximo de la mochila
static int MAX_N = 1000; //Numero maximo de objetos
static int N;//Numero de objetos
static int prices[] = new int[MAX_N];//precios de cada
    producto
static int weights[] = new int[MAX_N];//pesos de cada
    producto
static int memo[][]= new int[MAX_N][MAX_WEIGHT];//tabla dp

//El metodo debe llamarse con 0 en el id, y la capacidad de
    la mochila en w
static int knapsack(int id, int w) {
       if (id == N || w == 0) {
              return 0;
       }
       if (memo[id][w] != -1) {
              return memo[id][w];
       }
       if (weights[id] > w){
              memo[id][w] = knapsack(id + 1, w);
       }else{
              memo[id][w] = Math.max(knapsack(id + 1, w),
                  prices[id] + knapsack(id + 1, w -
                  weights[id]));
       }
       return memo[id][w];
}

//Antes de llamar al metodo, todos los campos de la tabla
    memo deben iniciarse a -1
```

2.2. Longest increasing subsequence

Halla la longitud de la subsecuencia creciente mas larga. MAX debe definirse en el tamaño limite del array, n es el tamaño del array. Puede aplicarse también sobre strings, cambiando el parametro int s[] por string s. Si debe ser estrictamente creciente, cambiar el ¡= de s[j] ¡= s[i] por ¡

```
       static int MAX = 1005;
       static int memo[] = new int[MAX];

       static int longestIncreasingSubsequence(int s[]){
              int n = s.length;
              memo[0] = 1;
              int output = 0;
              for (int i = 1; i < n; i++){
                     memo[i] = 1;
                     for (int j = 0; j < i; j++){
                            if (s[j] <= s[i] && memo[i] <
                                memo[j] + 1){
                                   memo[i] = memo[j] + 1;
                            }
                     }
                     if(memo[i] > output){
                            output = memo[i];
                     }
              }
              return output;
       }
```

3. Graphs

3.1. BFS

Algoritmo de búsqueda en anchura en grafos, recibe un nodo inicial s y visita todos los nodos alcanzables desde s. BFS también halla la distancia más corta entre el nodo inicial s y los demás nodos si todas las aristas tienen peso 1.

```
import java.util.ArrayList;
import java.util.Queue;
import java.util.LinkedList;


static int v, e; //vertices, arcos
static int MAX=100005;
```

```
static ArrayList<Integer> ady[] = new ArrayList[MAX];
    //lista de Adyacencia
static long distance[] = new long[MAX];

//M\'etodo para limpiar los valores de las estructuras.
static void init() {
    for (int j = 0; j <= v; j++) {
        distance[j] = -1;
        ady[j] = new ArrayList<Integer>();
    }
}

//Recibe el nodo inicial s
static void bfs(int s){
    Queue<Integer> q=new LinkedList<Integer>();
    q.add(s); //Inserto el nodo inicial
    distance[s]=0;
    int actual, i, next;

    while(!q.isEmpty()){
        actual=q.poll();
        for(i=0; i<ady[actual].size(); i++){
            next=ady[actual].get(i);
            if(distance[next]==-1){
                distance[next]=distance[actual]+1;
                q.add(next);
            }
        }
    }
}
```

3.2. DFS

Algoritmo de búsqueda en profundidad para grafos. Parte de un nodo inicial s visita a todos sus vecinos. DFS puede ser usado para contar la cantidad de componentes conexas en un grafo y puede ser modificado para que retorne información de los nodos dependiendo del problema. Permite hallar ciclos en un grafo.

```
import java.util.ArrayList;

static int v, e; //vertices, arcos
static int MAX=100005;
static ArrayList<Integer> ady[] = new ArrayList[MAX];
static boolean marked[] = new boolean[MAX];

//Limpia las estrucuturas de datos
static void init(){
        for (int j = 0; j <= v; j++) {
        marked[j] = false;
        ady[j] = new ArrayList<Integer>();
    }
}

//Recibe el nodo inicial s
static void dfs(int s){
        marked[s]=true;
        int i, next;

        for(i=0; i<ady[s].size(); i++){
                next=ady[s].get(i);
                if(!marked[next]){
                        dfs(next);
                }
        }
}
```

3.3. Dijkstra's Algorithm

Algoritmo que dado un grafo con pesos no negativos halla la ruta mínima entre un nodo inicial s y todos los demás nodos.

```
import java.util.ArrayList;
import java.util.PriorityQueue;


static int v, e; //vertices, arcos
static int MAX=100005;
```

```
static ArrayList<Node> ady[] = new ArrayList[MAX];
static int marked[] = new int[MAX];
static long distance[] = new long[MAX];
static int prev[] = new int[MAX];

//M\'etodo para limpiar los valores de las estructuras.
//Llamarlo siempre antes de utilizar el m\'etodo dijkstra()
static void init() {
    long max = Long.MAX_VALUE;

    for (int j = 0; j <= v; j++) {
        marked[j] = 0;
        prev[j] = -1;
        distance[j] = max;
        ady[j] = new ArrayList<Node>();
    }
}


//Recibe el nodo inicial s
static void dijkstra(int s) {
    PriorityQueue<Node> pq=new PriorityQueue<Node>();
    pq.add(new Node(s, 0));//se inserta a la cola el nodo
        Inicial.
    distance[s] = 0;
    int actual, j, adjacent;
    long weight;
    Node x;

    while (pq.size() > 0) {
        actual = pq.peek().adjacent;
        if (marked[actual] == 0) {
            marked[actual] = 1;
            for (j = 0; j < ady[actual].size(); j++) {
                adjacent = ady[actual].get(j).adjacent;
                weight = ady[actual].get(j).cost;
                if (marked[adjacent] == 0) {
                    if (distance[adjacent] > distance[actual]
                        + weight) {
                        distance[adjacent] = distance[actual]
                            + weight;
                        prev[adjacent] = actual;
                        pq.add(new Node(adjacent,
                            distance[adjacent]));
                    }
                }
            }
        }
        pq.poll();
    }
}

//Retorna en un String la ruta desde s hasta t
//Recibe el nodo destino t
static String path(int t) {
    String r="";
    while(prev[t]!=-1){
        r="-"+t+r;
        t=prev[t];
    }
    if(t!=-1){
        r=t+r;
    }
    return r;
}


static class Node implements Comparable<Node> {

    public int adjacent;
    public long cost;

    public Node(int ady, long c) {
        this.adjacent = ady;
        this.cost = c;
    }

    @Override
    public int compareTo(Node o) {
        if (this.cost >= o.cost) {
            return 1;
        } else {
```

```
            return -1;
        }
    }
}
```

3.4. Floyd-Warshall's Algorithm

Algoritmo para grafos que halla la distancia mínima entre cualquier par de nodos. Matrix[i][j] guardará la distancia mínima entre el nodo i y el j.
Ajustar los tipos de datos segun el problema.

```
static int v, e; //vertices, arcos
static int MAX=505;
static int matrix[][]=new int [MAX][MAX];

//M\'etodo para limpiar las estructuras de datos
static void init() {
    int i, j;
    for(i=0; i<v; i++){
        for(j=0; j<v; j++){
            matrix[i][j]=-1;
        }
    }
}

static void floydWarshall(){
    int i,j,k, aux;
    k=0;
    while(k<v){
        for(i=0; i<v; i++){
            if(i!=k){
                for(j=0; j<v; j++){
                    if(j!=k){
                        aux=matrix[i][k]+matrix[k][j];
                        if(aux<matrix[i][j] && aux>0){
                            matrix[i][j]=aux;
                        }
                    }
                }
            }
        }
        k++;
    }
}
```

3.5. Kruskal's Algorithm

Algoritmo para hallar el arbol cobertor mínimo de un grafo no dirigido y conexo. Utiliza la técnica de Union-Find(Conjuntos disjuntos) para detectar que aristas generan ciclos.

```
import java.util.ArrayList;
import java.util.Collections;


static int v, e; //vertices, arcos
static int MAX=100005;
static int parent[]= new int [MAX];
static int rank[]= new int [MAX];
static ArrayList<Edge> edges;
static ArrayList<Edge> answer;

//limpiar las estructuras de datos
static void init() {
    edges=new ArrayList<Edge>();
    answer=new ArrayList<Edge>();
    for (int j = 0; j <= v; j++) {
        parent[j] = j;
        rank[j] = 0;
    }
}

//UNION-FIND
static int find(int i){
    if(parent[i]!=i){
        parent[i]=find(parent[i]);
    }
```

```
        return parent[i];
}

static void unionFind(int x, int y){
    int xroot = find(x);
    int yroot = find(y);

    if (rank[xroot] < rank[yroot])
        parent[xroot] = yroot;
    else if (rank[xroot] > rank[yroot])
        parent[yroot] = xroot;

    else{
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}

static void kruskall(){
    Edge actual;
    int aux=0;
    int i=0;
    int x,y;
    Collections.sort(edges);

    while(aux<(v-1)){
        actual=edges.get(i);
        x=find(actual.src);
        y=find(actual.dest);

        if(x!=y){
            answer.add(actual);
            aux++;
            unionFind(x, y);
        }
        i++;
    }
}

static class Edge implements Comparable<Edge> {
    public int src, dest, weight;

    public Edge(int s, int d, int w) {
        this.src = s;
        this.dest = d;
        this.weight=w;
    }

    @Override
    public int compareTo(Edge o) {
        return this.weight-o.weight;
    }
}
```

3.6. Trajan's Algorithm

Algoritmo para hallar los puentes e itsmos en un grafo no dirigido.

```
import java.util.ArrayList;
import java.lang.Math;


static int n, e; //vertices, arcos
static int MAX=1010;
static ArrayList<Integer> ady[]=new ArrayList [MAX];
static boolean marked[]=new boolean [MAX];
static int prev[]=new int [MAX];
static int dfs_low[]=new int [MAX];
static int dfs_num[]=new int [MAX];
static int itsmos[]=new int [MAX];
static ArrayList<Edge> bridges;
static int dfsRoot, rootChildren, cont;

static void init() {
    bridges=new ArrayList<Edge>();
    cont=0;
    int i;
    for(i=0; i<n; i++){
        ady[i]=new ArrayList<Integer>();
```

```
        marked[i]=false;
        prev[i]=-1;
        itsmos[i]=0;
    }
}

static void dfs(int u){
    dfs_low[u]=cont;
    dfs_num[u]=cont;
    cont++;
    marked[u]=true;
    int j, v;

    for(j=0; j<ady[u].size(); j++){
        v=ady[u].get(j);
        if(!marked[v]){
            prev[v]=u;
            //Caso especial
            if(u==dfsRoot){
                rootChildren++;
            }
            dfs(v);
            //PARA ITSMOS
            if(dfs_low[v]>=dfs_num[u]){
                itsmos[u]=1;
            }
            //PARA PUENTES
            if(dfs_low[v]>dfs_num[u]){
                bridges.add(new
                    Edge(Math.min(u,v),Math.max(u,v)));
            }
            dfs_low[u]=Math.min(dfs_low[u], dfs_low[v]);
        }else if(v!=prev[u]){ //Arco que no sea backtrack
            dfs_low[u]=Math.min(dfs_low[u], dfs_num[v]);
        }
    }
}

static class Edge{

    public int src, dest;

    public Edge(int s, int d) {
        this.src = s;
        this.dest = d;
    }
}
```

4. Math

4.1. Binary Exponentiation

Realiza a^b y retorna el resultado módulo c. Si se elimina el módulo c, debe tenerse precaución para no exceder el límite

```
static int binaryExponentiation(int a, int b, int c){
    if (b == 0){
        return 1;
    }
    if (b % 2 == 0){
        int temp = binaryExponentiation(a,b/2, c);
        return (int)(((long)(temp) * temp) % c);
    }else{
        int temp = binaryExponentiation(a, b-1, c);
        return (int)(((long)(temp) * a) % c);
    }
}
```

4.2. Binomial Coefficient

Calcula el coeficiente binomial nCr, entendido como el número de subconjuntos de k elementos escogidos de un conjunto con n elementos.

```
static long binomialCoefficient(long n, long r) {
  if (r < 0 || n < r) {
        return 0;
  }
```

```
  r = Math.min(r, n - r);
  long ans = 1;
  for (int i = 1; i <= r; i++) {
    ans = ans * (n - i + 1) / i;
  }
  return ans;
}
```

4.3. Catalan Number

Guarda en el array Catalan Numbers los numeros de Catalan hasta MAX.

```
static int MAX = 30;
static long catalanNumbers[] = new long[MAX+1];

static void catalan(){
       catalanNumbers[0] = 1;
       for(int i = 1; i <= MAX; i++){
              catalanNumbers[i] =
                  (long)(catalanNumbers[i-1]*((double)(2*((2
                  * i)- 1))/(i + 1)));
       }
}
```

4.4. Euler Totient

Función totient o indicatriz (ϕ) de Euler. Para cada posición n del array result retorna el número de enteros positivos menores o iguales a n que son coprimos con n (Coprimos: MCD=1)

```
static void totient(int n, int resultados[]){
       boolean aux[]=new boolean[n];
       for(int i=0; i<n; i++) {
              resultados[i]=i;
       }
       for(int i=2; i<n; i++){
              if(!aux[i]) {
                     for(int j=i; j<n ; j+=i){
                            aux[j]=true;
                            resultados[j]=
                                resultados[j]-(resultados[j]/i)
                                ;
                     }
              aux[i] = false;
              }
       }
}
```

4.5. Gaussian Elimination

Resuelve sistemas de ecuaciones lineales por eliminación Gaussiana. matrix contiene los valores de la matriz cuadrada y result los resultados de las ecuaciones. Retorna un vector con el valor de las n incongnitas. Los resultados pueden necesitar redondeo.

```
import java.util.ArrayList;

static int MAX = 100;
static int n = 3;
static double matrix[][] = new double[MAX][MAX];
static double result[] = new double[MAX];

static ArrayList<Double> gauss() {

       ArrayList<Double> ans = new ArrayList<Double>();
       for(int i=0;i<n;i++){
              ans.add(0.0);
       }
       double temp;
       for (int i = 0; i < n; i++) {
       int pivot = i;
           for (int j = i + 1; j < n; j++) {
              temp = Math.abs(matrix[j][i]) -
                  Math.abs(matrix[pivot][i]);
              if (temp > 0.000001) {
```

```
                pivot = j;
            }
        }
        double temp2[] = new double[n];
        System.arraycopy(matrix[i],0,temp2,0,n);
        System.arraycopy(matrix[pivot],0,matrix[i],0,n);
        System.arraycopy(temp2,0,matrix[pivot],0,n);
        temp = result[i];
        result[i] = result[pivot];
        result[pivot] = temp;

        if (!(Math.abs(matrix[i][i]) < 0.000001)) {

            for (int k = i + 1; k < n; k++) {
                temp = -matrix[k][i] / matrix[i][i];
                matrix[k][i] = 0;
                for (int l = i + 1; l < n; l++) {
                    matrix[k][l] += matrix[i][l] *
                        temp;
                }
                result[k] += result[i] * temp;
            }
        }
    }
    for (int m = n - 1; m >= 0; m--) {
    temp = result[m];
    for (int i = n - 1; i > m; i--) {
            temp -= ans.get(i) * matrix[m][i];
    }
    ans.set(m,temp / matrix[m][m]);
    }
    return ans;
}
```

4.6. Greatest common divisor

Calcula el máximo común divisor entre a y b mediante el algoritmo de Euclides

```
int mcd(int a, int b) {
        int aux;
        while(b!=0){
                a %= b;
                aux = b;
                b = a;
                a = aux;
        }
        return a;
}
```

4.7. Lowest Common multiple

Cálculo del mínimo común múltiplo usando el máximo común divisor REQUIERE mcd(a,b)

```
int mcm(int a, int b) {
        return a*b/mcd(a,b);
}
```

4.8. Prime Factorization

Guarda en primeFactors la lista de factores primos del value de menor a mayor. IMPORTANTE: Debe ejecutarse primero la criba de Eratostenes. La criba debe existir al menos hasta la raiz cuadrada de value (se recomienda dejar un poco de excedente).

```
import java.util.ArrayList;

static ArrayList<Long> primeFactors = new ArrayList<Long>();

static void calculatePrimeFactors(long value){
        primeFactors.clear();
        long temp = value;
        int factor;
        for (int i = 0; (long)primes.get(i) * primes.get(i)
            <= value; ++i){
```

```
                factor = primes.get(i);
                while (temp % factor == 0){
                        primeFactors.add((long)factor);
                        temp /= factor;
                }
        }
        if (temp != 1) {
                primeFactors.add(temp);
        }
}
```

4.9. Sieve of Eratosthenes

Guarda en primes los números primos menores a MAX

```
import java.util.ArrayList;

static int MAX = 10000000;
static ArrayList<Integer> primes = new ArrayList<Integer>();
static boolean sieve[] = new boolean[MAX+5];

static void calculatePrimes() {
  sieve[0] = sieve[1] = true;
  int i;
  for (i = 2; i * i <= MAX; ++i) {
    if (!sieve[i]) {
      primes.add(i);
      for (int j = i * i; j <= MAX; j += i)
        sieve[j] = true;
    }
  }
  for(; i <= MAX; i++){
        if (!sieve[i]) {
      primes.add(i);
    }
  }
}
```

5. String

5.1. KMP's Algorithm

Encuentra si el string pattern se encuentra en el string cadena.

```
import java.util.ArrayList;

static ArrayList<Integer> table(String pattern){
        int m=pattern.length();
        ArrayList<Integer> border = new ArrayList<Integer>();
        border.add(0);
        int temp;
        for(int i=1; i<m; ++i){
                border.add(border.get(i-1));
                temp = border.get(i);
                while(temp>0 &&
                    pattern.charAt(i)!=pattern.charAt(temp)){
                        if(temp <= i+1){
                                border.set(i,border.get(temp-1));
                                temp = border.get(i);
                        }
                }
                if(pattern.charAt(i) == pattern.charAt(temp)){
                        border.set(i,temp+1);
                }
        }
        return border;
}

static boolean kmp(String cadena, String pattern){
        int n=cadena.length();
        int m=pattern.length();
        ArrayList<Integer> tab=table(pattern);
        int seen=0;

        for(int i=0; i<n; i++){
                while(seen>0 &&
                    cadena.charAt(i)!=pattern.charAt(seen)){
                        seen=tab.get(seen-1);
                }
```

```
                if(cadena.charAt(i)==pattern.charAt(seen))
                        seen++;
                if(seen==m){
                        return true;
                }
        }
        return false;
}
```

6. Tips and formulas

6.1. Catalan Number

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

Primeros 30 números de Catalán:

n	C_n
0	1
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1.430
9	4.862
10	16.796
11	58.786
12	208.012
13	742.900
14	2.674.440
15	9.694.845
16	35.357.670
17	129.644.790
18	477.638. 700
19	1.767.263.190
20	6.564.120.420
21	24.466.267.020
22	91.482.563.640
23	343.059.613.650
24	1.289.904.147.324
25	4.861.946.401.452
26	18.367.353.072.152
27	69.533.550.916.004
28	263.747.951.750.360
29	1.002.242.216.651.368
30	3.814.986.502.092.304

6.2. Euclidean Distance

Fórmula para calcular la distancia Euclideana entre dos puntos en el plano cartesiano (x,y).
Extendible a 3 dimensiones

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

6.3. Permutation and combination

Combinación (Coeficiente Binomial): Número de subconjuntos de k elementos escogidos de un conjunto con n elementos

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

Combinación con repetición: Número de grupos formados por n elementos, partiendo de m tipos de elementos.

$$CR_m^n = \binom{m+n-1}{n} = \frac{(m+n-1)!}{n!(m-1)!}$$

Permutación: Número de formas de agrupar n elementos, donde importa el orden y sin repetir elementos

$$P_n = n!$$

Elegir r elementos de n posibles con repetición

$$n^r$$

Permutaciones con repetición: Se tienen n elementos donde el primer elemento se repite a veces , el segundo b veces , el tercero c veces, ...

$$PR_n^{a,b,c...} = \frac{P_n}{a!b!c!...}$$

Permutaciones sin repetición: Número de formas de agrupar r elementos de n disponibles, sin repetir elementos

$$\frac{n!}{(n-r)!}$$

6.4. Time Complexities

Aproximación del mayor número n de datos que pueden procesarse para cada una de las complejidades algoritmicas. Tomar esta tabla solo como referencia.

Complexity	n
$O(n!)$	11
$O(n^5)$	50
$O(2^n * n^2)$	18
$O(2^n * n)$	22
$O(n^4)$	100
$O(n^3)$	500
$O(n^2 \log_2 n)$	1.000
$O(n^2)$	10.000
$O(n \log_2 n)$	10^6
$O(n)$	10^8
$O(\sqrt{n})$	10^{16}
$O(\log_2 n)$	-
$O(1)$	-

6.5. mod: properties

1. (a % b) % b = a % b (Propiedad neutro)
2. (ab) % c = ((a % c)(b % c)) % c (Propiedad asociativa en multiplicación)

3. $(a + b) \,\%\, c = ((a \,\%\, c) + (b \,\%\, c)) \,\%\, c$ (Propiedad asociativa en suma)