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

Gerson Yesid Lázaro - Angie Melissa Delgado

8 de noviembre de 2015

Índice

1. Bonus: Input Output

1.1. scanf y printf

```
#include <cstdio>

scanf("%d",&value); //int
scanf("%ld",&value); //long y long int
scanf("%c",&value); //char
scanf("%f",&value); //float
scanf("%lf",&value); //double
scanf("%s",&value); //char*
scanf("%lld",&value); //long long int
scanf("%x",&value); //int hexadecimal
scanf("%o",&value); //int octal
```

2. Dynamic Programming

2.1. Knapsack

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.

```
#include <algorithm>

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

//El metodo debe llamarse con 0 en el id, y la capacidad de
    la mochila en w
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] = max(knapsack(id + 1, w),
                    prices[id] + knapsack(id + 1, w -
                    weights[id]));
        }
```

```
        return memo[id][w];
}

//La tabla memo debe iniciar en -1
memset(memo, -1, sizeof memo);
```

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 $\leq$ de "s[j] $\leq$ s[i]"por $<$.

```
const int MAX = 1005;
int memo[MAX];

int longestIncreasingSubsequence(int s[], int n){
        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. Graph

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.

```
int v, e; //vertices, arcos
int MAX=100005;
vector<int> ady[MAX]; //lista de Adyacencia
long long distance[MAX];

static void init() {
    for (int j = 0; j <= v; j++) {
        distance[j] = -1;
        ady[j].clear();
    }
}

static void bfs(int s){
    queue<int> q;
    q.push(s); //Inserto el nodo inicial
    distance[s]=0;
    int actual, i, next;

    while(q.size()>0){
        actual=q.front();
        q.pop();
        for(i=0; i<ady[actual].size(); i++){
            next=ady[actual][i];
            if(distance[next]==-1){
                distance[next]=distance[actual]+1;
                q.push(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.

```
int v, e; //vertices, arcos
int MAX=100005;
vector<int> ady[MAX];
int marked[MAX];

void init(){
       for (int j = 0; j <= v; j++) {
      marked[j] = 0;
      ady[j].clear();
   }
}

static void dfs(int s){
       marked[s]=1;
       int i, next;

       for(i=0; i<ady[s].size(); i++){
              next=ady[s][i];
              if(marked[next]==0){
                     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.

```
int v,e;
vector<Node> ady[100001];
int marked[100001];
long long distance[100001];
int prev[100001];

class cmp
{
public:
    bool operator()(Node n1,Node n2)
    {
      if(n1.second>n2.second)
      return true;
      else
      return false;
    }
};

void init(){
    long long max=LLONG_MAX;
    for(int j=0; j<=v; j++){
        ady[j].clear();
        marked[j]=0;
        prev[j]=-1;
        distance[j]=max;
    }
}

void dijkstra(int s){
    priority_queue< Node , vector<Node> , cmp > pq;
    pq.push(Node(s, 0));//se inserta a la cola el nodo
        Inicial.
    distance[s]=0;
    int actual, j, adjacent;
    long long weight;

    while(!pq.empty()){
        actual=pq.top().first;
        pq.pop();
        if(marked[actual]==0){
            marked[actual]=1;
            for(j=0; j<ady[actual].size(); j++){
```

```
                adjacent=ady[actual][j].first;
                weight=ady[actual][j].second;
                if(marked[adjacent]==0){
                    if(distance[adjacent] > distance[actual] +
                        weight){
                        distance[adjacent] = distance[actual]
                            + weight;
                        prev[adjacent]=actual;
                        pq.push(Node(adjacent,
                            distance[adjacent]));
                    }
                }
            }
        }
    }
}
```

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.

```
#define Node pair<int,long long> //(Vertice adyacente, peso)

int v,e;
int matrix[505][505];

void floydWarshall(){
    int k=0;
    int aux, i ,j;

    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.
Para hallar los 2 arboles cobertores minimos, se debe ejecutar el algoritmo v-1 veces, en cada una de ellas dscartar una de las aristas previamente elegidas en el arbol.

```
struct Edge{
    int origen, destino, peso;

    bool operator!=(const Edge& rhs) const{
        if(rhs.origen!=origen || rhs.destino!=destino ||
            rhs.peso!=peso){
            return true;
        }
        return false;
    }
};

int v,e;
int MAX=10001;
int parent[MAX];
int rank[MAX];
Edge edges[MAX];
Edge answer[MAX];

void init(){
```

```
    for(int i=0; i<v; i++){
        parent[i]=i;
        rank[i]=0;
    }
}

int cmp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->peso > b1->peso;
}

int find(int i){
    if(parent[i]!=i){
        parent[i]=find(parent[i]);
    }
    return parent[i];
}

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

    // Attach smaller rank tree under root of high rank tree
    // (Union by rank)
    if (rank[xroot] < rank[yroot])
        parent[xroot] = yroot;
    else if (rank[xroot] > rank[yroot])
        parent[yroot] = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else{
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}

void kruskall(){
    Edge actual;
    int aux=0;
    int i=0;
    int x,y;
    qsort(edges, e, sizeof(edges[0]), cmp);

    while(aux<v-1){
        actual=edges[i];
        x=find(actual.origen);
        y=find(actual.destino);

        if(x!=y){
            answer[aux]=actual;
            aux++;
            unionFind(x, y);
        }
        i++;
    }
}
```

3.6. Tarjan's Algorithm

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

```
vector<int> ady[1010];
int marked[1010];
int prev[1010];
int dfs_low[1010];
int dfs_num[1010];
int itsmos[1010];
int n, e;
int dfsRoot,rootChildren,cont;
vector<pair<int,int>> bridges;

void init(){
    bridges.clear();
    cont=0;
    int i;
    for(i=0; i<n; i++){
        ady[i].clear();
```

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

void dfs(int u){
    dfs_low[u]=dfs_num[u]=cont;
    cont++;
    marked[u]=1; //esto no estaba >.<
    int j, v;

    for(j=0; j<ady[u].size(); j++){
        v=ady[u][j];
        if(marked[v]==0){
            prev[v]=u;
            //para el caso especial
            if(u==dfsRoot){
                rootChildren++;
            }
            dfs(v);
            //PARA ITSMOS
            if(dfs_low[v]>=dfs_num[u]){
                itsmos[u]=1;
            }
            //PARA bridges
            if(dfs_low[v]>dfs_num[u]){
                bridges.push_back(make_pair(min(u,v),max(u,v)));
            }
            dfs_low[u]=min(dfs_low[u], dfs_low[v]);
        }else if(v!=prev[u]){ //Arco que no sea backtrack
            dfs_low[u]=min(dfs_low[u], dfs_num[v]);
        }
    }
}
```

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

```
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 ((long long)(temp) * temp) % c;
    }else{
        int temp = binaryExponentiation(a, b-1, c);
        return ((long 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.

```
long long binomialCoefficient(long long n, long long r) {
  if (r < 0 || n < r) {
        return 0;
  }
  r = min(r, n - r);
  long 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.

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

void catalan(){
       catalanNumbers[0] = 1;
       for(int i = 1; i <= MAX; i++){
              catalanNumbers[i] = (long
                  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)

```
#include <string.h>

const int MAX = 100;
int result[MAX];

void totient () {
       bool temp[MAX];
       int i,j;
       memset(temp,1,sizeof(temp));
       for(i = 0; i < MAX; i++) {
              result[i] = i;
       }
       for(i = 2; i < MAX; i++){
              if(temp[i]) {
                     for(j = i; j < MAX ; j += i){
                            temp[j] = false;
                            result[j] = result[j] -
                                (result[j]/i) ;
                     }
                     temp[i] = true ;
              }
       }
}
```

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.

```
#include <vector>
#include <algorithm>
#include <limits>
#include <cmath>

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

vector<double> gauss() {

       vector<double> ans(n, 0);
       double temp;
       for (int i = 0; i < n; i++) {
       int pivot = i;
           for (int j = i + 1; j < n; j++) {
               temp = fabs(matrix[j][i]) -
                   fabs(matrix[pivot][i]);
               if (temp > numeric_limits<double>::epsilon()) {
                      pivot = j;
               }
           }
```

```
            swap(matrix[i], matrix[pivot]);
            swap(result[i], result[pivot]);

            if (!(fabs(matrix[i][i]) <
                numeric_limits<double>::epsilon())) {

                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[i] * matrix[m][i];
        }
        ans[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

Calculo 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).

```
#include <vector>

vector <long long> primeFactors;

void calculatePrimeFactors(long long value){
        primeFactors.clear();
        long long temp = value;
        int factor;
        for (int i = 0; (long long)primes[i] * primes[i] <=
            value; ++i){
                factor = primes[i];
                while (temp % factor == 0){
                        primeFactors.push_back(factor);
                        temp /= factor;
                }
        }
        if (temp != 1) {
```

```
            primeFactors.push_back(temp);
        }
}
```

4.9. Sieve of Eratosthenes

Guarda en primes los números primos menores o iguales a MAX

```
#include <vector>

const int MAX = 10000000;
vector<int> primes;
bool sieve[MAX+5];

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

5. String

5.1. KMP's Algorithm

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

```
#include <vector>

vector<int> table(string pattern){
        int m=pattern.size();
        vector<int> border(m);
        border[0]=0;

        for(int i=1; i<m; ++i){
                border[i]=border[i-1];
                while(border[i]>0 &&
                    pattern[i]!=pattern[border[i]]){
                        border[i]=border[border[i]-1];
                }
                if(pattern[i] == pattern[border[i]]){
                        border[i]++;
                }
        }
        return border;
}

bool kmp(string cadena, string pattern){
        int n=cadena.size();
        int m=pattern.size();
        vector<int> tab=table(pattern);
        int seen=0;

        for(int i=0; i<n; i++){
                while(seen>0 && cadena[i]!=pattern[seen]){
                        seen=tab[seen-1];
                }
                if(cadena[i]==pattern[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. Tabla ASCII

6.4. 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

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

6.5. 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}
$O(\log_2 n)$	-
$O(1)$	-

6.6. 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)