Team Notebook: UPF Programming Force

Contents

1 Basic

1.1 Auxiliar Comparer

```
// returns true if the first argument goes before the second argument
// in the strict weak ordering it defines, and false otherwise.
struct classcomp {
   bool operator() (const int& lhs, const int& rhs) const
   {return lhs > rhs;}
};
```

```
int main() {
    set<int> set1;
    set<int, classcomp> set2;
    set1.insert(26); set1.insert(93); set1.insert(42); // 26, 42, 93
    set2.insert(26); set2.insert(93); set2.insert(42); // 93, 42, 26

    for (auto it=set1.begin(); it!=set1.end(); ++it) cout << *it << " ";
    cout << "\n";
    for (auto it=set2.begin(); it!=set2.end(); ++it) cout << *it << " ";
}
```

1.2 Libraries

algorithm	heap, sort	map	map<S, T>
cfloat	DBL_MAX	queue	priority_queue
cmath	pow, sqrt	set	set<S>
cstdlib	abs, rand	sstream	istringstream, ostringstream
iostream	cin, cout	string	string
iomanip	setprecision	utility	pair<S, T>
list	list<T>	vector	vector<T>

1.3 Macros

```
#define X first
#define Y second
#define LI long long
#define MP make_pair
#define PB push_back
#define SZ size()
#define SQ(a) ((a)*(a))
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))
#define FOR(i,x,y) for(int i=(int)x; i<(int)y; i++)
#define RFOR(i,x,y) for(int i=(int)x; i>(int)y; i--)
#define SORT(a) sort(a.begin(), a.end())
#define RSORT(a) sort(a.rbegin(), a.rend())
#define IN(a,pos,c) insert(a.begin()+pos,1,c)
#define DEL(a,pos,cant) erase(a.begin()+pos,cant)
```

1.4 Permutations

```
int N = 3;
int a[] = {1,2,3};
do {
    for (int i = 0; i < N; ++i) cout << a[i] << " ";
    cout << "\n";
}
while (next_permutation(a, a + N));
```

1.5 Precision cout

```
cout.setf(ios::fixed);
cout.precision(8);
```

2 Data Structures

2.1 Big Numbers

```
#include <cassert>
#define BASE 1000000000

struct big {
    vector<int> V;
    big(): V(1, 0) {}
    big(int n): V(1, n) {} // supone n < 1000000000 !!!
    big(const big &b): V(b.V) {}

    bool operator==(const big &b) const { return V==b.V; }
    int &operator[](int i) { return V[i]; }
    int operator[](int i) const { return V[i]; }
    int size() const { return V.SZ; }
    void resize(int i) { V.resize(i); }

    bool operator<(const big &b) const {
        for (int i = b.SZ-1; SZ == b.SZ && i >= 0; i--)
            if (V[i] == b[i]) continue;
            else return (V[i] < b[i]);
        return (SZ < b.SZ);
    }
```

```
    void add_digit(int l) {
        if (l > 0) V.PB(l);
    }
};

inline big suma(const big &a, const big &b, int k) {
    LI l = 0;
    int size = MAX(a.SZ, b.SZ+k);
    big c; c.resize(size);
    for (int i = 0; i < size; ++i) {
        l += i < a.SZ ? a[i] : 0;
        l += (k <= i && i < k + b.SZ) ? b[i-k] : 0;
        c[i] = l%BASE;
        l /= BASE;
    }
    c.add_digit(int(l));
    return c;
}

inline big operator+(const big &a, const big &b) {
    return suma(a, b, 0);
}
inline big operator+(const big &a, int b) {return a+big(b);}
inline big operator+(int b, const big &a) {return a+big(b);}

inline big operator-(const big &a, const big &b) {
    assert(b < a || a == b);
    LI l = 0, m = 0;
    big c; c.resize(a.SZ);
    for (int i = 0; i < a.SZ; ++i) {
        l += a[i];
        l -= i < b.SZ ? b[i] + m : m;
        if (l < 0) { l += BASE; m = 1; }
        else m = 0;
        c[i] = l%BASE;
        l /= BASE;
    }
    if (c[c.SZ-1] == 0 && c.SZ > 1) c.resize(c.SZ-1);
    return c;
}
inline big operator-(const big &a, int b) {return a-big(b);}

inline big operator*(const big &a, int b) {
    if (b == 0) return big(0);
    big c; c.resize(a.SZ);
    LI l = 0;
    for (int i = 0; i < a.SZ; ++i) {
        l += (LI)b*a[i];
        c[i] = l%BASE;
        l /= BASE;
    }
    c.add_digit(int(l));
    return c;
}
inline big operator*(int b, const big &a) {return a*b;}
inline big operator*(const big &a, const big &b) {
    big res;
    for (int i = 0; i < b.SZ; ++i)
        res = suma(res, a*b[i], i);
    return res;
}

inline void divmod(const big &a, int b, big &div, int &mod) {
    div.resize(a.SZ);
    LI l = 0;
    for (int i = a.SZ-1; i >= 0; --i) {
        l *= BASE;
        l += a[i];
        div[i] = l/b;
        l %= b;
    }
    if (div[div.SZ-1] == 0 && div.SZ > 1) div.resize(div.SZ-1);
    mod=int(l);
}

inline big operator/(const big &a, int b) {
    big div; int mod;
    divmod(a, b, div, mod);
    return div;
}

inline int operator%(const big &a, int b) {
    big div; int mod;
    divmod(a, b, div, mod);
    return mod;
}

inline istream &operator>>(istream &is, big &b) {
    string s;
```

```
    if (is >> s) {
        b.resize((s.SZ - 1)/9 + 1);
        for (int n = s.SZ, k = 0; n > 0; n -= 9, k++) {
            b[k] = 0;
            for (int i = MAX(n-9, 0); i < n; i++)
                b[k] = 10*b[k] + s[i]-'0';
        }
    }
    return is;
}

inline ostream &operator<<(ostream &os, const big &b) {
    os << b[b.SZ - 1];
    for (int k = b.SZ-2; k >= 0; k--)
        os << setw(9) << setfill('0') << b[k];
    return os;
}

void p10519() { //10519: calcula 2+2+4+6+8+10+...+2*n
    for (big n; cin >> n; ) {
        if (n == big(0)) cout << 1 << endl;
        else cout << 2 + n*(n-1) << endl;
    }
}

int main(){
    p10519();
}
```

2.2 Binary Indexed Tree

```
/* Binary indexed tree. Supports cumulative sum queries in O(log n) */
#define N (1<<18)
LL bit[N];

void add(LL* bit,int x,int val) {
    for(; x<N; x+=x&-x)
        bit[x]+=val;
}

LL query(LL* bit,int x) {
    LL res=0;
    for(;x;x-=x&-x)
        res+=bit[x];
    return res;
}
```

2.3 Square Root Trick

```
/* Partitions an array in sqrt(n) blocks of size sqrt(n) to support
 * O(sqrt(n)) range sum queries, O(sqrt(n)) range sum updates, and O(1)
 * point updates */
void update(LL *S, LL *A, int i, int k, int x) {
    S[i/k] = S[i/k] - A[i] + x;
    A[i] = x;
}

LL query(LL *S, LL *A, int lo, int hi, int k) {
        int sum=0, i=lo;
        while((i+1)%k != 0 && i <= hi)
                sum += A[i++];
        while(i+k <= hi)
                sum += S[i/k], i += k;
        while(i <= hi)
                sum += A[i++];
        return sum;
}
```

3 Dynamic Programming

3.1 Change Making Problem

```
int N = 8; // numero de monedas
int m[] = {1,2,5,10,20,50,100,200}; // monedas
int A[100001]; // vector de resultados

int main() {
    int C; // monto C <= 100000
    cin >> C;
    A[0] = 0;
    for (int i = 1; i <= C; i++) {
        A[i] = 1000000;
        for (int j = 0; j < N && m[j] <= i; j++)
```

```
        A[i] = MIN(A[i], A[i-m[j]] + 1);
    }
    cout << A[C] << endl;
}
```

3.2 Edit Distance (Damerau-Levenshtein)

`m[0][_] = m[_][0] = 0`. `m[i][j]` is the edit distance of `s1[0..i]` and `s2[0..j]`. `m[i][j] = min(operations)`, where operations are:

- Add character, subtract character `m[i-1][j] + 1`, `m[i][j-1] + 1` respectively
- Change character `m[i-1][j-1] + (if s1[i] == s2[j] then 1 else 0)`
- Swap last two characters `m[i-2][j-2] + (if s1[i-1] == s2[j] && s1[i] == s2[j-1] then 1)`.

3.3 Knapsack Problem

```
int N = 8; // numero de objetos N <= 1000
int v[] = {1,6,7,1,8,3,7,5}; // valor de objetos
int p[] = {5,3,7,1,8,2,7,3}; // peso de objetos
int A[1001][1001]; // matriz de resultados

int main() {
    int C = 7; // capacidad C <= 1000

    for (int j = 0; j <= C; j++)
        A[0][j] = 0;
    for (int i = 1; i <= N; i++) {
        A[i][0] = 0;
        for (int j = 1; j <= C; j++) {
            A[i][j] = A[i-1][j];
            if (p[i-1] <= j) {
                int r = A[i-1][j-p[i-1]] + v[i-1];
                A[i][j] = MAX(A[i][j], r);
            }
        }
    }
    cout << A[N][C] << endl; // output: 12
}
```

3.4 Longest Common Subsequence

```
table[_][0] = 0;
for(int i=1; i<n+1; i++) {
    table[i][0] = 0;
    for(int j=1; j<n+1; j++) {
        if(x[i-1] == y[j-1])
            table[i][j] = table[i-1][j-1] + 1;
        else
            table[i][j] = max(table[i-1][j], table[i][j-1]);
    }
}
```

3.5 Longest Increasing Subsequence

```
// O(n^2)

for(int i=0; i<N; i++) {
    inc[i] = 1;
    for(int j=0; j<N; j++) {
        if(seq[j] < seq[i]) {
            int v = inc[j] + 1;
            if(v > inc[i])
                inc[i] = v;
        }
    }
        if(inc[i] > max)
            max = inc[i];
}

// O(n log n)

ind[0] = 0;
ind_sz = 1;
while(scanf("%d", &seq[seq_sz++]) == 1) {
    /* Add next element if it's bigger than the current last */
    int i = seq_sz-1;
    if (seq[ind[ind_sz-1]] < seq[i]) {
        predecessor[i] = ind[ind_sz-1];
        ind[ind_sz++] = i;
        continue;
    }
    /* bsearch to find element immediately bigger */
```

```
    int u = 0, v = ind_sz-1;
    while(u < v) {
        int c = (u + v) / 2;
        if (seq[ind[c]] < seq[i])
            u = c+1;
        else
            v = c;
    }
    /* Update b if new value is smaller then previously referenced value
     */
    if (seq[i] < seq[ind[u]]) {
        if (u > 0)
            predecessor[i] = ind[u-1];
        ind[u] = i;
    }
}
```

3.6 Maximum Subarray Sum (Kadane)

```
/* We show the 2D version here. the 1D version is the code block
separated by a newline. You can keep track of where the sequence
starts and ends by messing with the max_here and max assignments
respectively. Use > max_here to keep longer subsequences, >= max_here
to keep shorter ones. Take into account circular arrays by adding the
sum of all elements and the max of the array with sign changed. */
max = mat[0][0];
for(i=0; i<N; i++) {
    memset(aux, 0, sizeof(aux));
    for(k=i; k<N; k++) {
        for(j=0; j<N; j++)
            aux[j] += mat[k][j];

        max_here = aux[0];
        if(max_here > max)
            max = max_here;
        for(j=1; j<N; j++) {
            max_here += aux[j];
            if(aux[j] > max_here)
                max_here = aux[j];
            if(max_here > max)
                max = max_here;
        }
    }
```

```
}
```

3.7 TSP

```
// TSP in O(n^2 * 2^n). Subset is bitmask, Cost is cost.
// tsp_memoize[subset][j] stores the shortest path starting at node -1,
// including the nodes in the subset and finishing at node j.
// This is for the TSP with N+1 nodes. We pick the first one arbitrarily.
Cost distances[N][N], tsp_memoize[1 << (N+1)][N];
const Cost sentinel=-0x3f3f3f3f;
#define TSP(subset, i) (tsp_memoize[subset][i] == sentinel ? \
                                            tsp(subset, i) :
                                                tsp_memoize[subset][i])

Cost tsp(const Subset subset, const int i) {
        Subset without = subset ^ (1 << i);
        Cost minimum = numeric_limits<Cost>::max();
        for(int j=0; j<n_nodes; j++) {
                if(j==i || (without & (1 << j)) == 0)
                        continue;
                Cost v = TSP(without, j);
                v += distances[i][j];
                if(v < minimum)
                        minimum = v;
        }
        return tsp_memoize[subset][i] = minimum;
}

/* fill tsp_memoize with sentinel */
tsp_memoize[1<<i][i] = distance /* from -1 to i */
for(int i=0; i<n_nodes; i++)
        tsp(0xffff >> (16 - n_nodes), i) /* + distance from i to -1 */;
```

4 Geometry

4.1 Convex Hull

```
typedef int T;   // posiblemente cambiar a double
typedef pair<T,T> P;
T xp(P p, P q, P r) {
```

```
    return (q.X-p.X)*(r.Y-p.Y) - (r.X-p.X)*(q.Y-p.Y);
}
struct Vect {
    P p, q; T dist;
    Vect(P &a, P &b) {
        p = a; q = b;
        dist = SQ(a.X - b.X) + SQ(a.Y - b.Y);
    }
    bool operator<(const Vect &v) const {
        T t = xp(p, q, v.p);
        return t < 0 || t == 0 && dist < v.dist;
    }
};

vector<P> convexhull(vector<P> v) { // v.SZ >= 2
    sort(v.begin(), v.end());
    vector<Vect> u;
    for (int i = 1; i < (int)v.SZ; i++)
        u.PB(Vect(v[i], v[0]));
    sort(u.begin(), u.end());
    vector<P> w(v.SZ, v[0]);
    int j = 1; w[1] = u[0].p;
    for (int i = 1; i < (int)u.SZ; i++) {
        T t = xp(w[j-1], w[j], u[i].p);
        for (j--; t < 0 && j > 0; j--)
            t = xp(w[j-1], w[j], u[i].p);
        j += t > 0 ? 2 : 1;
        w[j] = u[i].p;
    }
    w.resize(j+1);
    return w;
}

int main() {
    vector<P> v;
    v.PB(MP(0, 1));
    v.PB(MP(1, 2));
    v.PB(MP(3, 2));
    v.PB(MP(2, 1));
    v.PB(MP(3, 1));
    v.PB(MP(6, 3));
    v.PB(MP(7, 0));
    vector<P> w = convexhull(v);
} // resultado: (0,1) (7,0) (6,3) (1,2)
```

4.2 Line Intersection

Intersection between two lines: here is the system solved. Swap all xs and ys to avoid dividing by zero if $p_x = 0$.

$$s = \frac{P_y - Q_y + \frac{p_y}{p_x}(Q_x - P_x)}{q_y - \frac{p_y}{p_x}q_x}$$

$$x = Q_x + q_x s;\ y = Q_y + q_y s$$

$$t = \frac{Q_x - P_x + q_x * s}{p_x}$$

5 Graphs

5.1 Bellman-Ford (Shortest Path with Negative Weights)

```
// Complexity: E * V - Input: directed graph
typedef pair<pair<int,int>,int> P; // par de nodos + coste
int N;                             // numero de nodos
vector<P> v;                       // representacion aristas

int bellmanford(int a, int b) {
    vector<int> d(N, 1000000000);
    d[a] = 0;
    for (int i = 1; i < N; i++)
        for (int j = 0; j < (int)v.SZ; j++)
            if (d[v[j].X.X] < 1000000000 && d[v[j].X.X] + v[j].Y <
                d[v[j].X.Y])
                d[v[j].X.Y] = d[v[j].X.X] + v[j].Y;
    for (int j = 0; j < (int)v.SZ; j++)
        if (d[v[j].X.X] < 1000000000 && d[v[j].X.X] + v[j].Y < d[v[j].X.Y])
            return -1000000000; // existe ciclo negativo
    return d[b];
}

int main(){
        N=8;
        v.PB(MP(MP(0, 1), +2)); v.PB(MP(MP(1, 2), -1)); v.PB(MP(MP(1, 3),
            +1));
        v.PB(MP(MP(2, 3), +1)); v.PB(MP(MP(6, 4), -1)); v.PB(MP(MP(4, 5),
            -1));
        v.PB(MP(MP(5, 6), -1));

        // min distance, negative cycle, unreachable
```

```
    cout << bellmanford(0, 3) << " " << bellmanford(4, 6) << " "
         << bellmanford(0, 7) << endl;
}
```

5.2 Bron-Kerbosch

```
#define U unsigned int
typedef vector<short int> V;

vector<vector<U> > graf; // vertices/aristas del grafo
U numv, kmax; // # conjuntos/tamano grupo independiente

int evalua(V &vec) {
   for (int n = 0; n < vec.size(); n++)
      if (vec[n] == 1) return n;
   return -1;
}

void Bron_i_Kerbosch() {
   vector<U> v;
   U i, j, aux, k = 0, bandera = 2;
   vector<V> I, Ve, Va;
   I.PB(V()); Ve.PB(V()); Va.PB(V());
   for (i = 0; i < numv; i++) {
      I[0].PB(0); // conjunto vacio
      Ve[0].PB(0); // conjunto vacio
      Va[0].PB(1); // contiene todos
   }
   while(true) {
      switch(bandera) {
      case 2: // paso 2
         v.PB(evalua(Va[k]));
         I.PB(V(I[k].begin(), I[k].end()));
         Va.PB(V(Va[k].begin(), Va[k].end()));
         Ve.PB(V(Ve[k].begin(), Ve[k].end()));
         aux = graf[v[k]].size();
         I[k+1][v[k]] = 1; Va[k+1][v[k]] = 0;
         for (i = 0; i < aux; i++) {
            j = graf[v[k]][i]; Ve[k+1][j] = Va[k+1][j] = 0;
         }
         k = k + 1; bandera = 3;
         break;
      /**********************************************/
      case 3: // paso 3
         for (i = 0, bandera = 4; i < numv; i++) {
            if (Ve[k][i] == 1) {
               aux = graf[i].size();
               for (j = 0; j < aux; j++)
                  if (Va[k][graf[i][j]] == 1)
                     break;
               if (j == aux) { i = numv; bandera = 5; }
            }
         }
         break;
      /**********************************************/
      case 4: // paso 4
         if (evalua(Ve[k]) == -1 && evalua(Va[k]) == -1) {
            for (int n = 0; n < numv; n++)
               if (I[k][n] == 1) cout<< n << " ";
            cout << endl;
            if (k > kmax) kmax = k;
            bandera = 5;
         }
         else bandera = 2; // ir a paso 2
      break;
      /**********************************************/
      case 5: // paso 5
         k = k - 1; v.pop_back(); I[k].clear();
         I[k].assign(I[k+1].begin(), I[k+1].end());
         I[k][v[k]] = 0; I.pop_back(); Ve.pop_back();
         Va.pop_back(); Ve[k][v[k]] = 1; Va[k][v[k]] = 0;
         if (k == 0) {
            if (evalua(Va[0]) == -1) return;
            bandera = 2; // ir a paso 2
         }
         else bandera = 3; // ir a paso 3
      break;
      }
   }
}

int main() {
   U idx, i; stringstream ss; string linea;
   while (cin >> numv) {
      getline(cin, linea);
      for (i = 0; i < numv; i++) { // Lectura del grafo
         // vertices adjacentes al i-esimo vertice
         vector<U> bb; graf.PB(bb);
```

```
            getline(cin, linea);
            ss << linea;
            while (ss >> idx) graf[i].PB(idx);
            ss.clear();
        }
        // Llamada al algoritmo
        kmax = 0;
        cout << "Conjuntos independientes: "<< endl;
        if (numv > 0)
            Bron_i_Kerbosch();
        cout << "kmax: " << kmax << endl;
        // Limpieza variables
        for (i = 0; i < numv; i++) graf[i].clear();
        graf.clear();
    }
}
```

5.3 Dijkstra (Shortest Path)

```
// Complexity: ElogV - Input: undirected graph
typedef int V;        // tipo de costes
typedef pair<V,int> P; // par de (coste,nodo)
typedef set<P> S;     // conjunto de pares

int N;                // numero de nodos
vector<P> A[10001];   // listas adyacencia (coste,nodo)

// int prec[201]; // predecesores (nodes from s to t)
// another way to obtain a path (above all, if there is
// more than one, consists in using BFS from the target
// and add to the queue those nodes that lead to the
// minimum cost in the preceeding node)

V dijkstra(int s, int t) {
    S m;                        // cola de prioridad
    vector<V> z(N, 1000000000); // distancias iniciales
    z[s] = 0;                   // distancia a s es 0
    m.insert(MP(0, s));         // insertar (0,s) en m
    while (m.SZ > 0) {
        P p = *m.begin(); // p=(coste,nodo) con menor coste
        m.erase(m.begin()); // elimina este par de m
        if (p.Y == t) return p.X; // cuando nodo es t, acaba
        // para cada nodo adjacente al nodo p.Y
        for (int i = 0; i < (int)A[p.Y].SZ; i++) {
            // q = (coste hasta nodo adjacente, nodo adjacente)
            P q(p.X + A[p.Y][i].X, A[p.Y][i].Y);
            // si q.X es la menor distancia hasta q.Y
            if (q.X < z[q.Y]) {
                m.erase(MP(z[q.Y], q.Y)); // borrar anterior
                m.insert(q);           // insertar q
                z[q.Y] = q.X;          // actualizar distancia
                            // prec[q.Y] = p.Y;    // actualizar
                                predecesores
            }
        }
    }
    return -1;
}

int main() {
    N = 6;            // solucion 0-1-2-4-3-5, coste 11
    A[0].PB(MP(2, 1)); // arista (0, 1) con coste 2
    A[0].PB(MP(5, 2)); // arista (0, 2) con coste 5
    A[1].PB(MP(2, 2)); // arista (1, 2) con coste 2
    A[1].PB(MP(7, 3)); // arista (1, 3) con coste 7
    A[2].PB(MP(2, 4)); // arista (2, 4) con coste 2
    A[3].PB(MP(3, 5)); // arista (3, 5) con coste 3
    A[4].PB(MP(2, 3)); // arista (4, 3) con coste 2
    A[4].PB(MP(8, 5)); // arista (4, 5) con coste 8
    cout << dijkstra(0, 5) << endl;
}
```

5.4 Floyd-Warshall (All Pairs Shortest Path)

```
// Complexity: n^3
// A: matriz n*n de adyacencia con costes
// ausencia de arista representada por un numero grande
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = MIN(A[i][j], A[i][k] + A[k][j]);
```

5.5 Kruskal (Minimum Spanning Tree)

```
// Complexity: ElogV - Input: undirected graph
typedef vector<pair<int,pair<int,int> > > V;

int N, mf[2000]; // numero de nodos N <= 2000
V v;             // vector de aristas
                 // (coste, (nodo1, nodo2))

// vector< pair<long, int> > K[3001]; // kruskal tree

int set(int n) { // conjunto conexo de n
    if (mf[n] == n) return n;
    else mf[n] = set(mf[n]); return mf[n];
}

int kruskal() {
    int a, b, sum = 0;
    sort(v.begin(), v.end());
    for (int i = 0; i < N; i++)
        mf[i] = i; // inicializar conjuntos conexos
    for (int i = 0; i < (int)v.SZ; i++) {
        a = set(v[i].Y.X), b = set(v[i].Y.Y);
        if (a != b) { // si conjuntos son diferentes
            mf[b] = a; // unificar los conjuntos
            sum += v[i].X; // agregar coste de arista
                       // K[v[i].Y.X].PB(MP(v[i].X, v[i].Y.Y));
                       // K[v[i].Y.Y].PB(MP(v[i].X, v[i].Y.X));
        }
    }
    return sum;
}

int main() {
    N = 5; // solucion 13 (0,3),(1,2),(2,3),(3,4)
    v.PB(MP(4,MP(0,1))); // arista (0,1) coste 4
    v.PB(MP(4,MP(0,2))); // arista (0,2) coste 4
    v.PB(MP(3,MP(0,3))); // arista (0,3) coste 3
    v.PB(MP(6,MP(0,4))); // arista (0,4) coste 6
    v.PB(MP(3,MP(1,2))); // arista (1,2) coste 3
    v.PB(MP(7,MP(1,4))); // arista (1,4) coste 7
    v.PB(MP(2,MP(2,3))); // arista (2,3) coste 2
    v.PB(MP(5,MP(3,4))); // arista (3,4) coste 5
    cout << kruskal() << endl;
}
```

5.6 Maximum Bipartite Matching

```
/* Input: VVI with 1 if connected, 0 if not. mr and mc have the matches
 * for each side. Complexity: E * V.
 * From Stanford University's notebook. */
typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < w[i].size(); j++) {
        if (w[i][j] && !seen[j]) {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;

    for (int i = 0; i < w.size(); i++) {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}
```

5.7 Min Cost Max Flow

```
/* From Stanford University's notebook.
 * To perform minimum weighted bipartite matching:
 *  - Capacity between nodes = 1 (cost whatever given by the problem)
 *  - Capacity from source = 1 and cost = 0
 *  - Capacity to sink = 1 and cost = 0
 * Output: <maximum flow value - minimum cost value>
```

```
 * Complexity: O(|V|^2) per augmentation
 *             max flow: O(|V|^3) augmentations
 *             min cost max flow: O(|V|^4 * MAX_EDGE_COST) augmentations
 */
typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;
    VPII dad;

    MinCostMaxFlow(int N) :
        N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
        found(N), dist(N), pi(N), width(N), dad(N) {}

    void AddEdge(int from, int to, L cap, L cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }

    void Relax(int s, int k, L cap, L cost, int dir) {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k]) {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }

    L Dijkstra(int s, int t) {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;

        while (s != -1) {
            int best = -1;
            found[s] = true;
            for (int k = 0; k < N; k++) {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }

        for (int k = 0; k < N; k++)
            pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }

    pair<L, L> GetMaxFlow(int s, int t) {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t)) {
            totflow += amt;
            for (int x = t; x != s; x = dad[x].first) {
                if (dad[x].second == 1) {
                    flow[dad[x].first][x] += amt;
                    totcost += amt * cost[dad[x].first][x];
                }
                else {
                    flow[x][dad[x].first] -= amt;
                    totcost -= amt * cost[x][dad[x].first];
                }
            }
        }
        return make_pair(totflow, totcost);
    }
};
```

5.8 Tarjan

```
int index, ct;
vector<bool> I;
// L indica el indice del conjunto fuertemente conexo al que pertenece
    cada nodo
```

```cpp
vector<int> D, L, S;
vector<vector<int> > V; // listas de adyacencia

void tarjan (unsigned n) {
    D[n] = L[n] = index++;
    S.push_back(n);
    I[n] = true;
    for (unsigned i = 0; i < V[n].size(); ++i) {
        if (D[V[n][i]] < 0) {
            tarjan(V[n][i]);
            L[n] = MIN(L[n], L[V[n][i]]);
        }
        else if (I[V[n][i]])
            L[n] = MIN(L[n], D[V[n][i]]);
    }
    if (D[n] == L[n]) {
        ++ct;
        // todos los nodos eliminados de S pertenecen al mismo scc
        while (S[S.size() - 1] != n) {
            I[S.back()] = false;
            S.pop_back();
        }
        I[n] = false;
        S.pop_back();
    }
}

void scc() {
    index = ct = 0;
    I = vector<bool>(V.size(), false);
    D = vector<int>(V.size(), -1);
    L = vector<int>(V.size());
    S.clear();
    for (unsigned n = 0; n < V.size(); ++n)
        if (D[n] < 0)
          tarjan(n);
    // ct = numero total de scc
}
```

5.9 Topological Sort

```cpp
vector<int> A[101]; // adjacency list (directed graph without cycles)
int inbound[101]; // number of nodes that point to each node
vector<int> fo; // final order

// M = number of nodes (there might be 'lonely' nodes)
void toposort(int M) {
    stack<int> order;
    int current;

    // Search for roots (identifiers might change between
    // problems (e.g. 1 to M))
    for(int m = 0; m < M; m++){
        if(inbound[m] == 0)
            order.push(m);
    }

    // Start topsort from roots
    while(!order.empty()){
        // Pop from stack
        current = order.top();
        order.pop();
        // Save order in fo
        fo.push_back(current);
        // Add childs only if inbound is 0
        for (int i = 0; i < A[current].size(); ++i)
        {
            inbound[A[current][i]]--;
            if (inbound[A[current][i]] == 0)
                order.push(A[current][i]);
        }
    }
}

int main() {
    A[0].push_back(1); A[0].push_back(2); A[2].push_back(1);
    inbound[0] = 0; inbound[1] = 2; inbound[2] = 1;
    toposort(3);
    for (int i = 0; i < fo.size(); ++i) cout << fo[i] << " ";
    // 0 2 1
}
```

6 Math

6.1 Catalan Numbers

```
unsigned long long v[34]; // 1, 1, 2, 5, 14, 42, 132, 429, 1430, ...
void catalan(){
    v[0] = 1;
    for (int i = 1; i < 34; ++i){
        unsigned long long sum = 0;
        for (int j = 0; j < i; ++j){
            sum += v[j] * v[i-j-1];
        }
        v[i] = sum;
    }
}
```

6.2 Complex Numbers

```
// Complex number class, from Stanford's Notebook. Required for FFT
struct cpx {
    cpx(){}
    cpx(double aa):a(aa){}
    cpx(double aa, double bb):a(aa),b(bb){}
    double a, b;
    double modsq(void) const { return a * a + b * b; }
    cpx bar(void) const { return cpx(a, -b); }
};
cpx operator +(cpx a, cpx b) { return cpx(a.a + b.a, a.b + b.b); }
cpx operator *(cpx a, cpx b) {
    return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}
cpx operator /(cpx a, cpx b) {
    cpx r = a * b.bar();
    return cpx(r.a / b.modsq(), r.b / b.modsq());
}
cpx EXP(double theta) { return cpx(cos(theta),sin(theta)); }
```

6.3 Exponent

```
template <typename T,typename U> T expo(T &t, U n) {
    if (n == U(0)) return T(1);
    else {
        T u = expo(t, n/2);
        if (n%2 > 0) return u*u*t;
        else return u*u;
    }
}
```

6.4 FFT

```
// from Stanford's notebook:
    https://web.stanford.edu/~liszt90/acm/notebook.html
// in:    input array
// out:   output array
// step:  {SET TO 1} (used internally)
// size:  length of the input/output {MUST BE A POWER OF 2}
// dir:   either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j *
    k / size)
const double two_pi = 4 * acos(0);
void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
    if(size < 1) return;
    if(size == 1)
    {
        out[0] = in[0];
        return;
    }
    FFT(in, out, step * 2, size / 2, dir);
    FFT(in + step, out + size / 2, step * 2, size / 2, dir);
    for(int i = 0 ; i < size / 2 ; i++)
    {
        cpx even = out[i];
        cpx odd = out[i + size / 2];
        out[i] = even + EXP(dir * two_pi * i / size) * odd;
        out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) /
            size) * odd;
    }
}
```

6.5 Greatest Common Divisor

```
// in algorithm library: __gcd(a, b)
int gcd(int a, int b) {
    if (a < b) return gcd(b, a);
```

```
    else if (a%b == 0) return b;
    else return gcd(b, a%b);
}

gcd(a,b)*lcm(a,b) = a*b
```

6.6 Matrix Multiplication

```
#define SIZE 15  // tamano de matriz cuadrado
#define MOD 10007 // modulo de la multiplicacion
struct matriz {
    int v[SIZE][SIZE];
    matriz() { init(); } // matriz de 0's
    matriz(int x) {    // matriz con x's en la diagonal
        init();
        for (int i = 0; i < SIZE; i++) v[i][i] = x;
    }
    void init() {
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++) v[i][j] = 0;
    }
    // multiplicacion de matrices modulo MOD
    matriz operator*(matriz &m) {
        matriz n;
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++)
                for (int k = 0; k < SIZE; k++)
                    n.v[i][j] = (n.v[i][j] + v[i][k]*m.v[k][j])%MOD;
        return n;
    }
};
```

6.7 Newton Method

```
long double tolerance = 1E-6;
long double c0 = 1.0;
long double c1 = 1.0;
bool solutionFound = false;

// find the value of 'c' that makes the function equal to = 0
// might also be used in optimization problems setting y as
// the first derivative and yprime as the second
while (true)
{
        long double y = /* formula of the original function */;
        long double yprime = /* formula of the first derivative respect to
            c */;
        c1 = c0 - y / yprime;
        if ((fabs(c1 - c0) / fabs(c1)) < tolerance)
        {
                solutionFound = true;
                break;
        }
        c0 = c1;
}
```

6.8 Primes

```
int v[10000]; // primes

void savePrimes()
{
    int k = 0;
    v[k++] = 2;
    for (int i = 3; i <= 10010; i += 2) {
        bool b = true;
        for (int j = 0; b && v[j] * v[j] <= i; j++)
            b = i%v[j] > 0;
        if (b)
            v[k++] = i;
    }
}

bool isPrime(int x){
        bool prime = true;
        for (int j = 0; prime && v[j] * v[j] <= x; j++)
                prime = x%v[j] > 0;
        return prime;
}

// probar si un numero x <= 100000000 es primo
int main()
{
    savePrimes();
```

```
    cout << isPrime(4);
}
```

7 Sequences

7.1 Binary Search

```
// binary_search function can be found at algorithm library
// devuelve el i mas pequeno tal que t <= v[i]
// si no existe tal i, devuelve v.SZ
template<typename T> int bb(T t, vector<T> &v) {
    int a = 0, b = v.SZ;
    while (a < b) {
        int m = (a + b)/2;
        if (v[m] < t) a = m+1; else b = m;
    }
    return a;
}
```

7.2 Ternary Search

```
double E = 0.0000001; // tolerance
double L = 200000; // R and L are extreme possible values...
double R = -200000; // ... for the optimized parameter
while (1) {
    double dist = R - L;
    if (fabs(dist) < E) break;
    double leftThird = L + dist / 3;
    double rightThird = R - dist / 3;
    // f is the function which we are optimizing
    if (f(leftThird) < f(rightThird))
        R = rightThird;
    else
        L = leftThird;
}
```

7.3 Vector Partition

```
bidirectional_iterator partition(bidirectional_iterator start,
                                 bidirectional_iterator end,
                                 Predicate p);

bool IsOdd(int i) {return (i%2==1);}

int main () {
    vector<int> myvector;
    vector<int>::iterator it, bound;

    // set some values:
    for (int i=1; i<10; ++i)
        myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

    bound = partition(myvector.begin(), myvector.end(), IsOdd);

    // print out content:
    cout << "odd members:";
    for (it=myvector.begin(); it!=bound; ++it)
        cout << " " << *it;
    cout << "\neven members:";
    for (it=bound; it!=myvector.end(); ++it)
        cout << " " << *it;
    cout << endl;
}
```

8 Strings

8.1 KMP

```
/*Search of substring in O(n+k)*/
void TablaKMP(string T,vector<int> &F)
{
    int pos = 2; // posicion actual en F
    int cnd = 0; // ndice en T del siguiente carcter del actual candidato
         en la subcadena

    F[0] = -1;
    while(pos <= T.size())
    {
        if(T[pos - 1] == T[cnd] )
        {//siguiente candidato coincidente en la cadena
```

```
            cnd++;
            F[pos] = cnd;
            pos++;
        }else if(cnd > 0)
        {//si fallan coincidencias consecutivas entonces asignamos valor
             conocido la primera vez
            cnd = F[cnd];
        }else{
            F[pos] = 0 ;
            pos++;
        }
    }
}
vector<int> KMPSearch(string T, string P)//T: texto donde se busca ,P:
     palabra a buscar ,salida: vector de posiciones match
{
    int k = 0 ; //puntero de T
    int i = 0 ; //avance en P

    vector<int> F(T.size(),0),sol;

    if(T.size() >= P.size())
    {
        TablaKMP(T,F);//optimizacin para no repetir busquedas de
             subcadenas que no hacen match
        while(k+i < T.size())
        {
            if(P[i] == T[k+i])
            {
                if(i == P.size()-1)
                {
                    sol.push_back(k); //modificando el return podemos
                         devolver todos los matches
                }
                i++;
            }else{
                k += i-F[i];
                if(i > 0)
                {
                    i = F[i];
                }
            }
        }
    }
    return sol;
}

int main(){
    string T = "PARTICIPARIA CON MI PARACAIDAS PARTICULAR";
    string P = "A";
    vector<int> founds = KMPSearch(T,P);
    for(int i = 0 ; i < founds.size();++i)
    {
        cout<<founds[i]<<endl;
    }
}
```

8.2 Suffix Arrays

```
// Suffix array construction in O(L log^2 L) time. Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:  string s
//
// OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)
//         of substring s[i...L-1] in the list of sorted suffixes.
//         That is, if we take the inverse of the permutation suffix[],
//         we get the actual suffix array.

#include <vector>
#include <iostream>
#include <string>

using namespace std;

struct SuffixArray {
    const int L;
    string s;
    vector<vector<int> > P;
    vector<pair<pair<int,int>,int> > M;

    SuffixArray(const string &s) : L(s.length()), s(s), P(1,
         vector<int>(L, 0)), M(L) {
        for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
        for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
            P.push_back(vector<int>(L, 0));
            for (int i = 0; i < L; i++)
```

```
                M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ?
                    P[level-1][i + skip] : -1000), i);
            sort(M.begin(), M.end());
            for (int i = 0; i < L; i++)
                P[level][M[i].second] = (i > 0 && M[i].first ==
                    M[i-1].first) ? P[level][M[i-1].second] : i;
        }
    }

    vector<int> GetSuffixArray() { return P.back(); }

    // returns the length of the longest common prefix of s[i...L-1] and
        s[j...L-1]
    int LongestCommonPrefix(int i, int j) {
        int len = 0;
        if (i == j) return L - i;
        for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
            if (P[k][i] == P[k][j]) {
                i += 1 << k;
                j += 1 << k;
                len += 1 << k;
            }
        }
        return len;
    }
};

int main() {
    // bobocel is the 0'th suffix
    //  obocel is the 5'th suffix
    //   bocel is the 1'st suffix
    //    ocel is the 6'th suffix
    //     cel is the 2'nd suffix
    //      el is the 3'rd suffix
    //       l is the 4'th suffix
    SuffixArray suffix("bobocel");
    vector<int> v = suffix.GetSuffixArray();

    // Expected output: 0 5 1 6 2 3 4
    //                  2
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```