Material de referencia para la ICPC.

Índice

1. Estructuras de datos.

1.1. Policy Based Data Structures.

La STL de GNU C++ implementa algunos contenedores adicionales basados en árboles. Para utilizarlos debemos añadir las siguientes librerías:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
```

Estos contenedores tienen la siguiente declaración:

```
tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag, node_update = null_node_update
    , Allocator = std::allocator<char> >
```

donde

- `Key` es el tipo de las llaves.
- `Mapped` es el tipo de los datos mapeados (`map`) o en su defecto `null_type` (`set`).
- `Cmp_Fn` es una función de comparación de llaves, declarada como `struct` con el operador `()` sobrecargado.
- `Tag` debe ser alguno de `rb_tree_tag`, `splay_tree_tag` o `ov_tree_tag`.
- `node_update` indica la información adicional de cada nodo, o en su defecto, `null_node_update`.

Split y join

Estos contenedores soportan las mismas funciones que `set` y `map`, junto con dos funciones nuevas:

```
A.split(T key, Tree B);
A.join(Tree B);
```

La función `split` mueve todos los nodos con llaves mayores que `key` del árbol `A` al árbol `B`. La función `join`, por el contrario, mueve todos los nodos del árbol `B` al árbol `A`, siempre y cuando los rangos no se traslapen. En el caso de árboles rojo-negro, ambas funciones tienen complejidad poli-logarítmica.

Iteradores de nodo

Además del clásico `Tree::iterator`, los contenedores basados en árboles implementan un tipo de iterador adicional, `node_iterator`, el cual nos permite recorrer el árbol. Así por ejemplo, las funciones

```
Tree::node_iterator root = A.node_begin();
Tree::node_iterator nil = A.node_end();
```

regresan un iterador de nodo correspondiente a la raíz y nodos nulos del árbol. Cada iterador de nodo incluye dos funciones miembro `get_l_child()` y `get_r_child()` que regresan los iteradores de nodos correspondientes a los hijos izquierdo y derecho.

Podemos hacer la conversión entre iteradores convencionales e iteradores de nodo de la siguiente manera:

```
it = *nd_it;
nd_it = it.m_p_nd;
```

La primera línea regresa el `iterator` correspondiente a un `node_iterator` y la segunda línea realiza lo contrario.

Actualización de nodos

Recordemos que `node_update` especifica la información adicional que guardará cada nodo así como la forma en que se actualiza. Este debe declararse en forma de `struct`, definiendo en su interior el tipo del dato adicional como `metadata_type`, y sobrecargando el operador `()` que indique cómo se actualizará cada nodo.

El operador `()` será llamado automáticamente, recibiendo como parámetros el nodo a actualizar y el nodo nulo. Las llamadas siempre se realizarán desde las hojas hasta la raíz. De esta manera, al actualizar la información de un nodo, la información de sus hijos ya está actualizada.

Cada iterador de nodo tiene una función miembro `get_metadata()` que regresa una referencia constante al dato adicional de ese nodo. Para modificarlo, debemos hacer antes un `const_cast<metadata_type &>`.

Por ejemplo, si queremos que cada nodo guarde el tamaño del sub-árbol correspondiente, podemos definir la etiqueta `size_node_update` de la siguiente manera:

```
template<typename node_const_iterator, typename node_iterator, typename Cmp_Fn, typename
    Allocator>
```

```
struct size_node_update {
    typedef int metadata_type;

    void operator() (node_iterator nd_it, node_const_iterator nil) {
        int lsize = 0, rsize = 0;
        if (nd_it.get_l_child() != nil)
            lsize = nd_it.get_l_child().get_metadata();
        if (nd_it.get_r_child() != nil)
            rsize = nd_it.get_r_child().get_metadata();
        const_cast<int &>(nd_it.get_metadata()) = lsize + rsize + 1;
    }
};
```

Árbol de Estadísticos de Orden

La STL incluye una etiqueta `tree_order_statistics_node_update`, que le indica a cada nodo que guarde el tamaño del sub-árbol correspondiente. Esta etiqueta incorpora dos funciones nuevas:

```
A.find_by_order(unsigned int k);
A.order_of_key(T key);
```

La función `find_by_order` regresa un iterador convencional que corresponde al k-ésimo elemento de `A` (indexado en 0). La función `order_of_key`, por su parte, regresa un entero que representa el número de elementos menores que `key`. Ambas funciones tienen complejidad logarítmica.

```
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
    ordered_set;
typedef ordered_set::node_iterator pnode;

//Imprime el arbol rotado 90 grados hacia la izquierda.
void print_tree(pnode it, pnode nil, int indent = 0) {
    if (it != nil) {
        print_tree(it.get_l_child(), nil, indent + 2);
        for (int i = 0; i < indent; ++i)
            cout << ' ';
        cout << **it << '\n';
        print_tree(it.get_r_child(), nil, indent + 2);
    }
}

int main() {
    //Datos de ejemplo.
    int n = 10;
    int arr[] = {20, 15, 50, 30, 25, 36, 10, 35, 40, 21};
    //Crea un arbol con los datos de arr y lo imprime.
    ordered_set v, w;
    for (int i = 0; i < n; ++i)
        v.insert(arr[i]);
    print_tree(v.node_begin(), v.node_end()); cout << '\n';
    //Separa el arbol en dos y los imprime.
    v.split(30, w);
    print_tree(v.node_begin(), v.node_end()); cout << '\n';
    print_tree(w.node_begin(), w.node_end()); cout << '\n';
    //Vuelve a unir ambos arboles y lo imprime.
    v.join(w);
    print_tree(v.node_begin(), v.node_end()); cout << '\n';
    //Imprime el indice de 35.
    cout << v.order_of_key(35) << '\n';
    //Imprime el 7-esimo elemento.
    cout << *v.find_by_order(7) << '\n';
    return 0;
}
```

2. Grafos.

2.1. Caminos más cortos.

Algoritmo de Dijkstra. Complejidad: $O((E+V)\log V)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <utility>
using namespace std;
#define maxv 100000 //Maximo numero de vertices.

typedef pair<int, int> edge;
#define length first
#define to     second

int V, E;                  //Numero de vertices y aristas.
vector<edge> graph[maxv]; //Aristas.

int s, dist[maxv], pred[maxv]; //Vertice inicial, distancia mas corta y predecesor.

//Encuentra el camino mas corto desde un vertice a todos los demas.
void Dijkstra() {
    fill_n(dist, V, 1e9);
    fill_n(pred, V, -1);
    dist[s] = 0;
    priority_queue<edge> pq;
    pq.push(edge(dist[s], s));
    while (!pq.empty()) {
        int curr = pq.top().to, dcurr = -pq.top().length;
        pq.pop();
        if (dist[curr] != dcurr)
            continue;
        for (edge e : graph[curr])
            if (dist[curr] + e.length < dist[e.to]) {
                dist[e.to] = dist[curr] + e.length;
                pred[e.to] = curr;
                pq.push(edge(-dist[e.to], e.to));
            }
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E >> s;
    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int u, v, d;
        cin >> u >> v >> d;
        graph[u].push_back(edge(d, v));
        graph[v].push_back(edge(d, u));
    }
    //Imprime la configuracion.
    Dijkstra();
    for (int i = 0; i < V; ++i)
        cout << i << ": " << pred[i] << ' ' << dist[i] << '\n';
    return 0;
}
```

2.2. Árbol de expansión mínima.

Algoritmo de Kruskal. Complejidad: $O(E\log V)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>
using namespace std;
```

```cpp
#define maxv 100000 //Maximo numero de vertices y aristas.

typedef pair<int, pair<int, int>> edge;
#define weight first
#define from   second.first
#define to     second.second

int V, E;         //Numero de vertices y aristas.
edge graph[maxv]; //Aristas.

int parent[maxv], Rank[maxv]; //Union-Find por rango y compresion de camino.
vector<int> MST;              //Arbol de expansion minima.

int Find(int x) {
    if (parent[x] != x)
        parent[x] = Find(parent[x]);
    return parent[x];
}

void Union(int x, int y) {
    if (Rank[x] < Rank[y])
        parent[x] = y;
    else {
        parent[y] = x;
        if (Rank[x] == Rank[y])
            Rank[x]++;
    }
}

//Encuentra el arbol de expansion minima.
int Kruskal() {
    int cost = 0;
    MST.clear();
    for (int i = 0; i < V; ++i) {
        parent[i] = i;
        Rank[i] = 0;
    }
    sort(graph, graph + E);
    for (int i = 0; i < E; ++i)
        if (Find(graph[i].from) != Find(graph[i].to)) {
            cost += graph[i].weight;
            Union(Find(graph[i].from), Find(graph[i].to));
            MST.push_back(i);
        }
    return cost;
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;
    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i)
        cin >> graph[i].from >> graph[i].to >> graph[i].weight;
    //Imprime la configuracion del arbol de expansion minima.
    cout << "Peso total: " << Kruskal() << '\n';
    for (int i : MST)
        cout << graph[i].from << ' ' << graph[i].to << ' ' << graph[i].weight << '\n';
    return 0;
}
```

2.3. Orden topológico.

Complejidad: $O(V + E)$.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
#define maxv 100000 //Maximo numero de vertices.

```

```
7  int V, E;                  //Numero de vertices y aristas.
8  vector<int> graph[maxv]; //Aristas.
9
10 int vis[maxv];          //Visitados.
11 vector<int> toposort; //Orden topologico.
12
13 bool DFS(int u) {
14     vis[u] = 1;
15     for (int v : graph[u])
16         if (vis[v] == 1 || (!vis[v] && !DFS(v)))
17             return false;
18     vis[u] = -1;
19     toposort.push_back(u);
20     return true;
21 }
22
23 //Encuentra el orden topologico. Regresa false si no existe.
24 bool TopologicalSort() {
25     toposort.clear();
26     fill_n(vis, V, false);
27     for (int u = 0; u < V; ++u)
28         if (!vis[u] && !DFS(u))
29             return false;
30     reverse(toposort.begin(), toposort.end());
31     return true;
32 }
33
34 int main() {
35     ios_base::sync_with_stdio(0); cin.tie();
36     cin >> V >> E;
37     //Lee la informacion de las aristas.
38     for (int i = 0; i < E; ++i) {
39         int from, to;
40         cin >> from >> to;
41         graph[from].push_back(to);
42     }
43     //Imprime el orden topologico
44     if (!TopologicalSort())
45         cout << "El grafo tiene ciclos.";
46     else for (int u : toposort)
47         cout << u << ' ';
48     cout << '\n';
49     return 0;
50 }
```

2.4. Componentes fuertemente conexas.

Algoritmo de Kosaraju. Complejidad: $O(V + E)$.

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <stack>
5  using namespace std;
6  #define maxv 100000 //Maximo numero de vertices.
7
8  int V, E;                                    //Numero de vertices y aristas.
9  vector<int> graph[maxv], transpose[maxv]; //Aristas y aristas transpuestas.
10
11 int component[maxv]; //Componentes fuertemente conexas.
12 stack<int> S;         //Vertices pendientes.
13 bool vis[maxv];       //Visitado.
14
15 void DFS(int u) {
16     if (!vis[u]) {
17         vis[u] = true;
18         for (int v : graph[u])
19             DFS(v);
20         S.push(u);
21     }
```

```
}

void Assign(int u, int root) {
    if (component[u] == -1) {
        component[u] = root;
        for (int v : transpose[u])
            Assign(v, root);
    }
}

//Algoritmo de Kosaraju para encontrar las componentes fuertemente conexas.
void Kosaraju() {
    fill_n(vis, V, false);
    for (int u = 0; u < V; ++u)
        DFS(u);
    fill_n(component, V, -1);
    while (!S.empty()) {
        Assign(S.top(), S.top());
        S.pop();
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;
    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int from, to;
        cin >> from >> to;
        graph[from].push_back(to);
        transpose[to].push_back(from);
    }
    //Imprime las componentes fuertemente conexas.
    Kosaraju();
    for (int i = 0; i < V; ++i)
        cout << i << ": " << component[i] << '\n';
    return 0;
}
```

2.5. Puentes y puntos de articulación.

Algoritmo de Tarjan. Complejidad: $O(V + E)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>
using namespace std;
#define maxv 100000 //Maximo numero de vertices.

int V, E;                 //Numero de vertices y aristas.
vector<int> graph[maxv]; //Aristas.

bool artpoint[maxv];              //Puntos de articulacion.
vector<pair<int, int>> bridge;    //Puentes.
int lst_id, idx[maxv], low[maxv]; //Indice de los vertices, menor vertice alcanzable.

void DFS(int u, int pred) {
    idx[u] = low[u] = lst_id++;
    int children = 0;
    for (int v : graph[u]) {
        if(idx[v] == -1) {
            DFS(v, u);
            low[u] = min(low[u], low[v]);
            children++;
            if ((pred == -1 && children > 1) || (pred != -1 && low[v] >= idx[u]))
                artpoint[u] = true;
            if (low[v] > idx[u])
                bridge.push_back(make_pair(u, v));
        }
```

```
28          else if (v != pred)
29              low[u] = min(low[u], idx[v]);
30      }
31  }
32
33  //Algoritmo de Tarjan.
34  void Tarjan() {
35      lst_id = 0;
36      fill_n(artpoint, V, false);
37      bridge.clear();
38      fill_n(idx, V, -1);
39      for (int u = 0; u < V; ++u)
40          if (idx[u] == -1)
41              DFS(u, -1);
42  }
43
44  int main() {
45      ios_base::sync_with_stdio(0); cin.tie();
46      cin >> V >> E;
47      //Lee la informacion de las aristas.
48      for (int i = 0; i < E; ++i) {
49          int u, v;
50          cin >> u >> v;
51          graph[u].push_back(v);
52          graph[v].push_back(u);
53      }
54      //Imprime los puentes y puntos de articulacion.
55      Tarjan();
56      cout << "Puntos de articulacion:\n";
57      for (int i = 0; i < V; ++i)
58          if (artpoint[i])
59              cout << i << ' ';
60      cout << "\nPuentes:\n";
61      for (int i = 0; i < bridge.size(); ++i)
62          cout << bridge[i].first << ' ' << bridge[i].second << '\n';
63      return 0;
64  }
```

2.6. Flujo máximo.

Algoritmo de Dinic. Complejidad: $O(V^2E)$ ($O(\sqrt{V}E)$ para emparejamientos bipartitos).

```
1   #include <iostream>
2   #include <algorithm>
3   #include <vector>
4   #include <queue>
5   using namespace std;
6   #define maxv 100000 //Maximo numero de vertices.
7
8   struct edge {
9       int to, rev;        //Destino, arista invertida.
10      int flow, capacity; //Flujo, capacidad.
11  };
12
13  int V, E;                   //Numero de vertices y aristas.
14  vector<edge> graph[maxv]; //Aristas.
15
16  int s, t;                     //Fuente y sumidero.
17  int level[maxv], ptr[maxv]; //Distancia desde s y numero de aristas visitadas.
18
19  //Verifica si se puede enviar flujo de s a t.
20  bool BFS() {
21      fill_n(level, V, -1);
22      level[s] = 0;
23      queue<int> Q;
24      Q.push(s);
25      while (!Q.empty() && level[t] == -1) {
26          int curr = Q.front();
27          Q.pop();
28          for (edge e : graph[curr])
```

```
29              if (level[e.to] == -1 && e.flow < e.capacity) {
30                  level[e.to] = level[curr] + 1;
31                  Q.push(e.to);
32              }
33      }
34      return level[t] != -1;
35  }
36
37  //Envia flujo de s a t.
38  int DFS(int curr, int flow) {
39      if (curr == t || !flow)
40          return flow;
41      for (int &i = ptr[curr]; i < graph[curr].size(); ++i) {
42          edge &e = graph[curr][i], &re = graph[e.to][e.rev];
43          if (level[e.to] == level[curr] + 1)
44              if (int currflow = DFS(e.to, min(flow, e.capacity - e.flow))) {
45                  e.flow += currflow;
46                  re.flow -= currflow;
47                  return currflow;
48              }
49      }
50      return 0;
51  }
52
53  //Calcula el flujo maximo de s a t.
54  int Dinic() {
55      int flow = 0;
56      while (BFS()) {
57          fill_n(ptr, V, 0);
58          while (int currflow = DFS(s, 1e9))
59              flow += currflow;
60      }
61      return flow;
62  }
63
64  int main() {
65      ios_base::sync_with_stdio(0); cin.tie();
66      cin >> V >> E >> s >> t;
67      //Lee la informacion de las aristas.
68      for (int i = 0; i < E; ++i) {
69          int from, to, capacity;
70          cin >> from >> to >> capacity;
71          graph[from].push_back(edge{to, (int)graph[to].size(), 0, capacity});
72          graph[to].push_back(edge{from, (int)graph[from].size() - 1, 0, 0});
73      }
74      //Imprime la configuracion del flujo.
75      cout << "Flujo maximo: " << Dinic() << '\n';
76      for (int i = 0; i < V; ++i)
77          for (edge e : graph[i])
78              if (e.capacity)
79                  cout << i << ' ' << e.to << ": " << e.flow << '/' << e.capacity << '\n';
80      return 0;
81  }
```

3. Matemáticas.

3.1. Big Numbers.

```
1  #include <algorithm>
2  #include <utility>
3  using namespace std;
4
5  typedef string BigInteger;
6
7  //Regresa el i-esimo digito de derecha a izquierda de un numero.
8  unsigned int digit(const BigInteger &num, unsigned int i) {
9      if (i < num.size())
10         return num[num.size() - 1 - i] - '0';
```

```
11      return 0;
12  }
13
14  //Compara dos numeros y regresa: 1 si el primero es mayor; 0 si son iguales; -1 si el
15  //segundo es mayor.
16  int compareTo(const BigInteger &a, const BigInteger &b) {
17      for (int i = max(a.size(), b.size()) - 1; i >= 0; --i) {
18          if (digit(a, i) > digit(b, i))
19              return 1;
20          if (digit(b, i) > digit(a, i))
21              return -1;
22      }
23      return 0;
24  }
25
26  //Regresa la suma de dos numeros.
27  BigInteger sum(const BigInteger &a, const BigInteger &b) {
28      BigInteger ans;
29      int carry = 0;
30      for (int i = 0; i < max(a.size(), b.size()); ++i) {
31          carry += digit(a, i) + digit(b, i);
32          ans.push_back((carry % 10) + '0');
33          carry /= 10;
34      }
35      if (carry)
36          ans.push_back(carry + '0');
37      reverse(ans.begin(), ans.end());
38      return ans;
39  }
40
41  //Regresa la diferencia de dos numeros. El primero debe ser mayor o igual que el segundo.
42  BigInteger substract(const BigInteger &a, const BigInteger &b) {
43      BigInteger ans;
44      int carry = 0;
45      for (int i = 0; i < a.size(); ++i) {
46          carry += digit(a, i) - digit(b, i);
47          if (carry >= 0) {
48              ans.push_back(carry + '0');
49              carry = 0;
50          }
51          else {
52              ans.push_back(carry + 10 + '0');
53              carry = -1;
54          }
55      }
56      while (ans.size() > 1 && ans.back() == '0')
57          ans.pop_back();
58      reverse(ans.begin(), ans.end());
59      return ans;
60  }
61
62  //Regresa el producto de dos numeros (BigInteger x int).
63  BigInteger multiply(const BigInteger &a, unsigned int b) {
64      if (b == 0)
65          return "0";
66      BigInteger ans;
67      int carry = 0;
68      for (int i = 0; i < a.size(); ++i) {
69          carry += digit(a, i) * b;
70          ans.push_back((carry % 10) + '0');
71          carry /= 10;
72      }
73      while (carry) {
74          ans.push_back((carry % 10) + '0');
75          carry /= 10;
76      }
77      reverse(ans.begin(), ans.end());
78      return ans;
79  }
80
81  //Regresa el producto de dos numeros (BigInteger x BigInteger).
```

```
82  BigInteger multiply(const BigInteger &a, const BigInteger &b) {
83      BigInteger ans;
84      for (int i = 0; i < b.size(); ++i)
85          ans = sum(ans, multiply(a, digit(b, i)).append(i, '0'));
86      return ans;
87  }
88
89  //Regresa el cociente y el residuo de la division (BigInteger / int).
90  pair<BigInteger, unsigned int> divide(const BigInteger &a, unsigned int b) {
91      pair<BigInteger, int> ans;
92      for (int i = a.size() - 1; i >= 0; --i) {
93          ans.second = 10*ans.second + digit(a, i);
94          if (!ans.first.empty() || ans.second >= b || i == 0)
95              ans.first.push_back((ans.second / b) + '0');
96          ans.second %= b;
97      }
98      return ans;
99  }
100
101 //Regresa el cociente y el residuo de la division (BigInteger / BigInteger).
102 pair<BigInteger, BigInteger> divide(const BigInteger &a, const BigInteger &b) {
103     pair<BigInteger, BigInteger> ans;
104     BigInteger table[10];
105     for (int i = 0; i < 10; ++i)
106         table[i] = multiply(b, i);
107     for (int i = a.size() - 1; i >= 0; --i) {
108         int q = 0;
109         ans.second.push_back(digit(a, i) + '0');
110         while (q < 9 && compareTo(ans.second, table[q + 1]) >= 0)
111             ++q;
112         if (!ans.first.empty() || q > 0 || i == 0)
113             ans.first.push_back(q + '0');
114         ans.second = substract(ans.second, table[q]);
115     }
116     return ans;
117 }
```

3.2. Test de Primalidad.

Algoritmo de Miller-Rabin (determinista). Complejidad: $O(\log n)$.

```
1  #include <iostream>
2  using namespace std;
3
4  long long power(__int128 base, long long expo, long long mod) {
5      if (expo == 0)
6          return 1;
7      else if (expo % 2)
8          return (base * power(base, expo - 1, mod)) % mod;
9      else {
10         __int128 p = power(base, expo / 2, mod);
11         return (p * p) % mod;
12     }
13 }
14
15 //Regresa false si n es compuesto y true si probablemente es primo.
16 bool MillerTest(long long n, long long a, int s, long long d) {
17     __int128 x = power(a, d, n);
18     if (x == 1 || x == n - 1)
19         return true;
20     for (int r = 0; r < s - 1; ++r) {
21         x = (x * x) % n;
22         if (x == n - 1)
23             return true;
24     }
25     return false;
26 }
27
28 //Regresa true n es primo.
29 bool isPrime(long long n) {
30     if (n <= 4)
31         return n == 2 || n == 3;
```

```
32      long long s, d = n - 1;
33      for (s = 0; d % 2 == 0; ++s)
34          d /= 2;
35      for (long long a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
36          if (n == a)
37              return true;
38          if (!MillerTest(n, a, s, d))
39              return false;
40      }
41      return true;
42  }
43
44  int main() {
45      ios_base::sync_with_stdio(0); cin.tie();
46      long long n;
47      while (cin >> n) {
48          if (isPrime(n))
49              cout << "Es primo.\n";
50          else
51              cout << "No es primo.\n";
52      }
53      return 0;
54  }
```

3.3. Factorización en primos.

Complejidad: $O(\pi(\sqrt{n}))$ donde $\pi(x)$ es el número de primos menores o iguales que x.

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4   #define maxn 10000000 //Raiz cuadrada del mayor numero a factorizar.
5
6   vector<long long> primes;  //Lista de primos.
7
8   //Encuentra con la Criba de Eratostenes los primos menores que maxn.
9   void find_primes() {
10      vector<bool> sieve(maxn);
11      for (long long i = 2; i < maxn; ++i)
12          if (!sieve[i]) {
13              primes.push_back(i);
14              for (long long j = i * i; j < maxn; j += i)
15                  sieve[j] = true;
16          }
17  }
18
19  //Prueba por division.
20  vector<long long> prime_factor(long long n) {
21      vector<long long> factors;
22      for (int i = 0; primes[i] * primes[i] <= n; ++i)
23          while (n % primes[i] == 0) {
24              factors.push_back(primes[i]);
25              n /= primes[i];
26          }
27      if (n != 1)
28          factors.push_back(n);
29      return factors;
30  }
31
32  int main() {
33      ios_base::sync_with_stdio(0); cin.tie();
34      long long n;
35      find_primes();
36      while (cin >> n) {
37          for (long long p : prime_factor(n))
38              cout << p << ' ';
39          cout << '\n';
40      }
41      return 0;
42  }
```

3.4. Sistemas de Ecuaciones Lineales.

Eliminación Gauss-Jordan. Complejidad $O(n^3)$.

```
#include <iostream>
#include <cmath>
using namespace std;
#define maxn 100 //Maximo numero de renglones y columnas.

//Encuentra la forma escalonada reducida de la matriz A de m x n.
void GaussJordan(int m, int n, double matrix[][maxn]) {
    for (int r = 0, c = 0; c < n; ++c) {
        int pivot = r;
        for (int i = r + 1; i < m; ++i)
            if (fabs(matrix[pivot][c]) < fabs(matrix[i][c]))
                pivot = i;
        if (fabs(matrix[pivot][c]) < 1e-6)
            continue;
        if (pivot != r)
            for (int j = c; j < n; ++j)
                swap(matrix[r][j], matrix[pivot][j]);
        for (int j = n - 1; j >= c; --j) {
            matrix[r][j] /= matrix[r][c];
            for (int i = 0; i < m; ++i)
                if (i != r)
                    matrix[i][j] -= matrix[i][c] * matrix[r][j];
        }
        ++r;
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    int m, n;
    double matrix[maxn][maxn];
    //Lee los datos de la matriz.
    cin >> m >> n;
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
                cin >> matrix[i][j];
    //Imprime la forma escalonada reducida.
    GaussJordan(m, n, matrix);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j)
                cout << matrix[i][j] << ' ';
        cout << '\n';
    }
    return 0;
}
```

3.5. Teorema Chino del Residuo.

```
#include <iostream>
#include <utility>
using namespace std;

//Algoritmo extendido de Euclides.
long long extendedEuclid(long long a, long long b, long long &x, long long &y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    else {
        long long gcd = extendedEuclid(b, a %b, y, x);
        y -= (a / b) * x;
        return gcd;
    }
}

//Teorema Chino del Residuo. Regresa el residuo y el modulo.
```

```
pair<long long, long long> ChineseRemainder(int n, long long coef[], long long mod[]) {
    pair<long long, long long> ans(0, 1);
    for (int i = 0; i < n; ++i)
        ans.second *= mod[i];
    for (int i = 0; i < n; ++i) {
        long long a = ans.second / mod[i], x, y;
        extendedEuclid(a, mod[i], x, y);
        long long tmp = (a * (x + ans.second)) %ans.second;
        ans.first = (ans.first + coef[i] * tmp) %ans.second;
    }
    return ans;
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    int n;
    cin >> n;
    //Lee los datos de las ecuaciones.
    long long coef[n], mod[n];
    for (int i = 0; i < n; ++i)
        cin >> coef[i] >> mod[i];
    //Imprime la solucion.
    pair<long long, long long> ans = ChineseRemainder(n, coef, mod);
    cout << "x = " << ans.second << "k + " << ans.first << '\n';
    return 0;
}
```

4. Geometría

4.1. Geometría 2D.

```
#include <cmath>
#include <vector>
using namespace std;
#define epsilon 1e-6 //Precision.

struct point {
    double x, y;
};
typedef const point cpoint;

point operator + (cpoint &P, cpoint &Q) {
    return point{P.x + Q.x, P.y + Q.y};
}

point operator - (cpoint &P, cpoint &Q) {
    return point{P.x - Q.x, P.y - Q.y};
}

point operator * (cpoint &P, double c) {
    return point{P.x * c, P.y * c};
}

point operator / (cpoint &P, double c) {
    return point{P.x / c, P.y / c};
}

//Regresa el producto punto de dos vectores. Es 0 cuando son ortogonales.
double dot(cpoint &P, cpoint &Q) {
    return P.x * Q.x + P.y * Q.y;
}

//Regresa la componente z del producto cruz de dos vectores. Es 0 cuando son paralelos.
double cross(cpoint &P, cpoint &Q) {
    return P.x * Q.y - P.y * Q.x;
}

//Regresa la distancia euclidiana de un punto al origen.
double norm(cpoint &P) {
    return sqrt(dot(P, P));
```

```
40  }
41
42  //Regresa la distancia euclidiana entre dos puntos.
43  double dist(cpoint &P, cpoint &Q) {
44      return norm(P - Q);
45  }
46
47  //Regresa el vector rotado 90 grados en el sentido contrario de las manecillas del reloj.
48  point rotate90ccw(cpoint &P) {
49      return point{-P.y, P.x};
50  }
51
52  //Regresa el vector rotado theta radianes en sentido contrario de las manecillas del reloj.
53  point rotateCCW(cpoint &P, double theta) {
54      return point{P.x*cos(theta) - P.y*sin(theta), P.x*sin(theta) + P.y*cos(theta)};
55  }
56
57  //Regresa la proyeccion ortogonal del vector P sobre el vector Q.
58  point projection(cpoint &P, cpoint &Q) {
59      return Q * (dot(P, Q) / dot(Q, Q));
60  }
61
62  //Regresa la distancia del punto P a la recta que pasa por A y B.
63  double distPointLine(cpoint &P, cpoint &A, cpoint &B) {
64      return dist(P, A + projection(P - A, B - A));
65  }
66
67  //Regresa true si la recta que pasa por A y B corta al segmento con extremos C y D.
68  bool lineSegmentIntersection(cpoint &A, cpoint &B, cpoint &C, cpoint &D) {
69      return cross(B - A, C - A) * cross(B - A, D - A) < 0;
70  }
71
72  //Regresa el punto de interseccion de dos rectas no paralelas AB y CD.
73  point lineLineIntersection(cpoint &A, cpoint &B, cpoint &C, cpoint &D) {
74      point v = B - A, w = D - C;
75      return A + v * (cross(C - A, w) / cross(v, w));
76  }
77
78  //Regresa el centro de la circunferencia que pasa por A, B y C.
79  point circumcenter(cpoint &A, cpoint &B, cpoint &C) {
80      point AB = (A + B) / 2, BC = (B + C) / 2;
81      return lineLineIntersection(AB, AB + rotate90ccw(A - B), BC, BC + rotate90ccw(C - B));
82  }
83
84  //Regresa las intersecciones de la recta AB con la circunferencia con centro O y radio r.
85  vector<point> lineCircleIntersection(cpoint &A, cpoint &B, cpoint &O, double r) {
86      vector<point> ans;
87      point v = B - A, w = A - O;
88      double a = dot(v, v), b = dot(v, w), c = dot(w, w) - r*r;
89      double d = b*b - a*c;
90      if (d >= -epsilon)
91          ans.push_back(A + v * ((-b + sqrt(d + epsilon))/a));
92      if (d > epsilon)
93          ans.push_back(A + v * ((-b - sqrt(d + epsilon))/a));
94      return ans;
95  }
96
97  //Regresa las intersecciones de las circunferencias con centros O1, O2 y radios r1, r2.
98  vector<point> circleCircleIntersection(cpoint &O1, double r1, cpoint &O2, double r2) {
99      vector<point> ans;
100     double d = dist(O1, O2);
101     if (r1 + r2 >= d && d + min(r1, r2) >= max(r1, r2)) {
102         point v = (O2 - O1) / d;
103         double x = (d*d + r1*r1 - r2*r2) / (2*d), y = sqrt(r1*r1 - x*x);
104         ans.push_back(O1 + v * x + rotate90ccw(v) * y);
105         if (y > epsilon)
106             ans.push_back(O1 + v * x - rotate90ccw(v) * y);
107     }
108     return ans;
109 }
110
```

```
111 //Regresa 1, 0, -1 dependiendo si el punto Q esta dentro, sobre o fuera del poligono
112 //(posiblemente no convexo) con vertices P.
113 int pointInPolygon(cpoint &Q, int n, cpoint P[]) {
114     int numCrossings = 0;
115     for (int i = 0; i < n; ++i) {
116         int j = (i + 1) %n;
117         if (fabs(dist(P[i], Q) + dist(Q, P[j]) - dist(P[i], P[j])) < epsilon)
118             return 0;
119         if (cross(P[i] - Q, P[j] - Q) * ((Q.y <= P[j].y) - (Q.y <= P[i].y)) > 0)
120             numCrossings++;
121     }
122     return numCrossings %2 ? 1 : -1;
123 }
124
125 //Regresa el area con signo del poligono (posiblemente no convexo) con vertices P.
126 double areaPolygon(int n, cpoint P[]) {
127     double area = 0;
128     for (int i = 0; i < n; ++i)
129         area += cross(P[i], P[(i+1)%n]);
130     return area / 2;
131 }
132
133 //Regresa true si el poligono con vertices P es convexo.
134 bool isConvexPolygon(int n, cpoint P[]) {
135     double orientation = cross(P[1] - P[0], P[2] - P[1]);
136     for (int i = 1; i < n; ++i)
137         if (orientation * cross(P[(i+1)%n] - P[i], P[(i+2)%n] - P[(i+1)%n]) < 0)
138             return false;
139     return true;
140 }
```

4.2. Envolvente convexa.

Algoritmo de Graham-Scan. Complejidad: $O(n \log n)$.

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <complex>
5  using namespace std;
6  #define maxn 100000 //Maximo numero de puntos.
7
8  typedef complex<int> point;
9  #define x real()
10 #define y imag()
11
12 int n;               //Numero de puntos.
13 point P[maxn];       //Lista de puntos.
14 vector<point> hull; //Envolvente convexa.
15
16 //Compara usando el producto cruz quien posee el menor angulo con centro en P[0].
17 bool compareByAngle(const point &v, const point &w) {
18     return (conj(v - P[0]) * (w - P[0])).y > 0;
19 }
20
21 //Algoritmo de Graham-Scan para encontrar la envolvente convexa.
22 void GrahamScan() {
23     int first = 0;
24     for (int i = 1; i < n; ++i)
25         if (P[i].y < P[first].y || (P[i].y == P[first].y && P[i].x < P[first].x))
26             first = i;
27     swap(P[0], P[first]);
28     sort(P + 1, P + n, compareByAngle);
29     hull.clear();
30     for (int i = 0; i < n; ++i) {
31         while (hull.size() > 1 && (conj(hull[hull.size()-1] - hull[hull.size()-2]) *
32                (P[i] - hull[hull.size()-2])).y < 0)
33             hull.pop_back();
34         hull.push_back(P[i]);
35     }
36 }
37
```

```cpp
int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    //Lee los puntos.
    cin >> n;
    for (int i = 0; i < n; ++i) {
        int px, py;
        cin >> px >> py;
        P[i] = point(px, py);
    }
    //Imprime la envolvente convexa.
    GrahamScan();
    for (int i = 0; i < hull.size(); ++i)
        cout << '(' << hull[i].x << ", " << hull[i].y << ")\n";
    return 0;
}
```

5. Strings

5.1. Búsqueda de patrones.

Arreglo Z. Complejidad: $O(|P| + |T|)$.

```cpp
#include <iostream>
using namespace std;
#define maxn 100000 //Longitud maxima de los strings.

string text, pattern, str; //Texto, patron a buscar y string auxiliar.
int Z[maxn];               //Arreglo Z.

//Construye el arreglo Z de str.
void buildZ() {
    int l = 0, r = 0;
    for (int i = 1; i < str.size(); ++i) {
        Z[i] = 0;
        if (i <= r)
            Z[i] = min(r - i + 1, Z[i - l]);
        while (i + Z[i] < str.size() && str[Z[i]] == str[i + Z[i]])
            Z[i]++;
        if (i + Z[i] - 1 > r) {
            l = i;
            r = i + Z[i] - 1;
        }
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    //Lee el texto y los patrones.
    cin >> text >> pattern;
    str = pattern + '$' + text;
    //Imprime todas las ocurrencias.
    buildZ();
    for (int i = 0; i < text.size(); ++i)
        if (Z[i + pattern.size() + 1] == pattern.size())
            cout << "Ocurrencia en la posicion " << i << '\n';
    return 0;
}
```

Aho Corasick. Complejidad: $O(|T| + |P_1| + \ldots + |P_n| + \#Ocurrencias)$.

```cpp
#include <iostream>
#include <cstring>
#include <queue>
using namespace std;
#define maxc 26     //Longitud del alfabeto.
#define maxn 100    //Maximo numero de patrones.
#define maxs 100000 //Maximo numero de nodos.

int n;                      //Numero de patrones.
```

```
10  string text, pattern[maxn]; //Texto y lista de patrones.
11
12  int nnodes;                 //Numero de nodos.
13  struct node {
14      node *nxt[maxc], *link; //Nodos adyacentes y mayor sufijo que es prefijo en el Trie.
15      bool isEnd[maxn];       //Es nodo terminal de algun patron.
16  } Trie[maxs];               //Nodos del Trie.
17
18  node *nextNode(node *curr, char c) {
19      if (!curr)
20          return Trie;
21      if (!curr->nxt[c])
22          return nextNode(curr->link, c);
23      return curr->nxt[c];
24  }
25
26  //Construye los links de cada nodo.
27  void buildLink() {
28      queue<node*> Q;
29      Q.push(Trie);
30      while (!Q.empty()) {
31          node *curr = Q.front();
32          Q.pop();
33          for (char c = 0; c < maxc; ++c)
34              if (node *nxt = curr->nxt[c]) {
35                  nxt->link = nextNode(curr->link, c);
36                  for (int i = 0; i < n; ++i)
37                      if (nxt->link->isEnd[i])
38                          nxt->isEnd[i] = true;
39                  Q.push(nxt);
40              }
41      }
42  }
43
44  //Construye el Trie de patrones.
45  void buildTrie() {
46      nnodes = 0;
47      memset(Trie, 0, sizeof(Trie));
48      for (int i = 0; i < n; ++i) {
49          node *curr = Trie;
50          for (char c : pattern[i]) {
51              if (!curr->nxt[c - 'a'])
52                  curr->nxt[c - 'a'] = Trie + (++nnodes);
53              curr = curr->nxt[c - 'a'];
54          }
55          curr->isEnd[i] = true;
56      }
57      buildLink();
58  }
59
60  int main() {
61      ios_base::sync_with_stdio(0); cin.tie();
62      //Lee el texto y los patrones.
63      cin >> text >> n;
64      for (int i = 0; i < n; ++i)
65          cin >> pattern[i];
66      buildTrie();
67      //Imprime todas las ocurrencias.
68      node *curr = Trie;
69      for (int i = 0; i < text.size(); ++i) {
70          curr = nextNode(curr, text[i] - 'a');
71          for (int j = 0; j < n; ++j)
72              if (curr->isEnd[j])
73                  cout << pattern[j] << " aparece en la posicion " << i - pattern[j].size() +
                        1 << '\n';
74      }
75      return 0;
76  }
```

5.2. Arreglo de sufijos.

Complejidad: $O(n \log n)$.

```cpp
#include <iostream>
#include <algorithm>
#include <utility>
using namespace std;
#define maxn 100000 //Longitud maxima del string.

string word;                //String.
int n, SuffixArray[maxn]; //Arreglo de sufijos.

int bucket[maxn], tempSA[maxn];          //Cubeta (RadixSort).
pair<int, int> rnk[maxn], tempRA[maxn]; //Rango (SuffixArray).

//Ordena de acuerdo a los rangos.
void RadixSort() {
    int M = max(n + 1, 256);
    for (int k = 0; k < 2; ++k) {
        fill_n(bucket, M, 0);
        for (int i = 0; i < n; ++i)
            bucket[k ? rnk[i].first : rnk[i].second]++;
        for (int i = 1; i < M; ++i)
            bucket[i] += bucket[i - 1];
        for (int i = n - 1; i >= 0; --i) {
            int nxt_id = --bucket[k ? rnk[i].first : rnk[i].second];
            tempSA[nxt_id] = SuffixArray[i];
            tempRA[nxt_id] = rnk[i];
        }
        copy(tempSA, tempSA + n, SuffixArray);
        copy(tempRA, tempRA + n, rnk);
    }
}

//Construye el arreglo de sufijos.
void buildSA() {
    n = word.size();
    for (int i = 0; i < n; ++i) {
        SuffixArray[i] = i;
        rnk[i] = make_pair(word[i], (i + 1 < n) ? word[i + 1] : 0);
    }
    RadixSort();
    for (int k = 2; k < n; k *= 2) {
        int curr = 0, prev = -1;
        for (int i = 0; i < n; ++i) {
            if (rnk[i].first != prev || rnk[i].second != rnk[i - 1].second)
                curr++;
            prev = rnk[i].first;
            rnk[i].first = curr;
            tempSA[SuffixArray[i]] = i;
        }
        for (int i = 0; i < n; ++i) {
            int nxt_id = SuffixArray[i] + k;
            rnk[i].second = (nxt_id < n) ? rnk[tempSA[nxt_id]].first : 0;
        }
        RadixSort();
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    //Lee la palabra.
    cin >> word;
    buildSA();
    //Imprime los sufijos en orden lexicografico.
    for (int i = 0; i < n; ++i)
        cout << SuffixArray[i] << ' ';
    cout << '\n';
    return 0;
}
```