

Material de referencia para la ICPC.

Índice

1. Estructuras de datos.	2
1.1. Treap.	2
2. Grafos.	3
2.1. Caminos más cortos.	3
2.2. Árbol de expansión mínima.	4
2.3. Orden topológico.	5
2.4. Componentes fuertemente conexas.	6
2.5. Puentes y puntos de articulación.	6
2.6. Flujo máximo.	7
2.7. Emparejamiento máximo.	9
3. Matemáticas.	10
3.1. Big Numbers.	10
3.2. Test de Primalidad.	12
3.3. Factorización en primos.	12
3.4. Sistemas de Ecuaciones Lineales.	13
3.5. Teorema Chino del Residuo.	14
4. Geometría	14
4.1. Geometría 2D.	14
4.2. Envolverte convexa.	16
5. Strings	17
5.1. Algoritmo de KMP.	17
5.2. Algoritmo de Aho Corasick.	18
5.3. Arreglo de sufijos.	19

1. Estructuras de datos.

1.1. Treap.

Complejidad promedio por operación: $O(\log n)$

```
1  typedef int T;           //Tipo de las llaves.
2  struct node {
3      T key;               //Llave.
4      int size, priority;  //Tamano, prioridad.
5      node *left, *right;  //Hijos izquierdo y derecho.
6
7      //Constructor.
8      node(T key, int priority) : key(key), size(1), priority(priority), left(NULL), right(
9          NULL) {};
10
11 //Regresa el tamano de un treap.
12 int size(node *root) {
13     return root ? root->size : 0;
14 }
15
16 //Actualiza el tamano de un treap y demas informacion adicional.
17 void update(node *root) {
18     if (root)
19         root->size = size(root->left) + size(root->right) + 1;
20 }
21
22 //Divide un treap en dos partes, la mitad izquierda con los elementos menores que key.
23 void split(node *root, T key, node *&left, node *&right) {
24     if (!root)
25         left = right = NULL;
26     else if (key < root->key) {
27         split(root->left, key, left, root->left);
28         right = root;
29     }
30     else {
31         split(root->right, key, root->right, right);
32         left = root;
33     }
34     update(root);
35 }
36
37 //Combina dos treaps en uno solo en el orden dado.
38 void merge(node *&root, node *left, node *right) {
39     if (!left || !right)
40         root = left ? left : right;
41     else if (left->priority > right->priority) {
42         merge(left->right, left->right, right);
43         root = left;
44     }
45     else {
46         merge(right->left, left, right->left);
47         root = right;
48     }
49     update(root);
50 }
51
52 //Divide un treap en dos partes, la mitad izquierda con los primeros k elementos.
53 void split_by_order(node *root, int k, node *&left, node *&right) {
54     if (!root)
55         left = right = NULL;
56     else if (k < size(root->left)) {
57         split(root->left, k, left, root->left);
58         right = root;
59     }
60     else {
61         split(root->right, k - 1 - size(root->left), root->right, right);
62         left = root;
63     }
64     update(root);
65 }
```

```

65 }
66
67 //Regresa el k-esimo elemento (indexado en 0) del treap.
68 node *kth_element(node *root, int k) {
69     if (k == size(root->left))
70         return root;
71     else if (k < size(root->left))
72         return kth_element(root->left, k);
73     else
74         return kth_element(root->right, k - size(root->left) - 1);
75 }
76
77 //Regresa el numero de elementos menores que key.
78 int order_of_key(node *root, T key) {
79     if (key < root->key)
80         return order_of_key(root->left, key);
81     else if (key == root->key)
82         return size(root->left) + 1;
83     else
84         return size(root->left) + 1 + order_of_key(root->right, key);
85 }

```

2. Grafos.

2.1. Caminos más cortos.

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

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <queue>
5  #include <utility>
6  using namespace std;
7  #define maxv 100000 //Maximo numero de vertices.
8
9  typedef pair<int, int> edge;
10 #define length first
11 #define to second
12
13 int V, E; //Numero de vertices y aristas.
14 vector<edge> graph[maxv]; //Aristas.
15
16 int s, dist[maxv], pred[maxv]; //Vertice inicial, distancia mas corta y predecesor.
17
18 //Encuentra el camino mas corto desde un vertice a todos los demas.
19 void Dijkstra() {
20     fill_n(dist, V, 1e9);
21     fill_n(pred, V, -1);
22     dist[s] = 0;
23     priority_queue<edge> pq;
24     pq.push(edge(dist[s], s));
25     while (!pq.empty()) {
26         int curr = pq.top().to, dcurr = -pq.top().length;
27         pq.pop();
28         if (dist[curr] != dcurr)
29             continue;
30         for (edge e : graph[curr])
31             if (dist[curr] + e.length < dist[e.to]) {
32                 dist[e.to] = dist[curr] + e.length;
33                 pred[e.to] = curr;
34                 pq.push(edge(-dist[e.to], e.to));
35             }
36     }
37 }
38
39 int main() {
40     ios_base::sync_with_stdio(0); cin.tie();
41     cin >> V >> E >> s;

```

```

42 //Lee la informacion de las aristas.
43 for (int i = 0; i < E; ++i) {
44     int u, v, d;
45     cin >> u >> v >> d;
46     graph[u].push_back(edge(d, v));
47     graph[v].push_back(edge(d, u));
48 }
49 //Imprime la configuracion.
50 Dijkstra();
51 for (int i = 0; i < V; ++i)
52     cout << i << ": " << pred[i] << ' ' << dist[i] << '\n';
53 return 0;
54 }

```

2.2. Árbol de expansión mínima.

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

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <utility>
5  using namespace std;
6  #define maxv 100000 //Maximo numero de vertices y aristas.
7
8  typedef pair<int, pair<int, int>> edge;
9  #define weight first
10 #define from second.first
11 #define to second.second
12
13 int V, E; //Numero de vertices y aristas.
14 edge graph[maxv]; //Aristas.
15
16 int parent[maxv], Rank[maxv]; //Union-Find por rango y compresion de camino.
17 vector<int> MST; //Arbol de expansion minima.
18
19 int Find(int x) {
20     if (parent[x] != x)
21         parent[x] = Find(parent[x]);
22     return parent[x];
23 }
24
25 void Union(int x, int y) {
26     if (Rank[x] < Rank[y])
27         parent[x] = y;
28     else {
29         parent[y] = x;
30         if (Rank[x] == Rank[y])
31             Rank[x]++;
32     }
33 }
34
35 //Encuentra el arbol de expansion minima.
36 int Kruskal() {
37     int cost = 0;
38     MST.clear();
39     for (int i = 0; i < V; ++i) {
40         parent[i] = i;
41         Rank[i] = 0;
42     }
43     sort(graph, graph + E);
44     for (int i = 0; i < E; ++i)
45         if (Find(graph[i].from) != Find(graph[i].to)) {
46             cost += graph[i].weight;
47             Union(Find(graph[i].from), Find(graph[i].to));
48             MST.push_back(i);
49         }
50     return cost;
51 }
52

```

```

53 int main() {
54     ios_base::sync_with_stdio(0); cin.tie();
55     cin >> V >> E;
56     //Lee la informacion de las aristas.
57     for (int i = 0; i < E; ++i)
58         cin >> graph[i].from >> graph[i].to >> graph[i].weight;
59     //Imprime la configuracion del arbol de expansion minima.
60     cout << "Peso total: " << Kruskal() << '\n';
61     for (int i : MST)
62         cout << graph[i].from << ' ' << graph[i].to << ' ' << graph[i].weight << '\n';
63     return 0;
64 }

```

2.3. Orden topológico.

Complejidad: $O(V + E)$.

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  #define maxv 100000 //Maximo numero de vertices.
6
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     }
22 }
23
24 void Assign(int u, int root) {
25     if (component[u] == -1) {
26         component[u] = root;
27         for (int v : transpose[u])
28             Assign(v, root);
29     }
30 }
31
32 //Algoritmo de Kosaraju para encontrar las componentes fuertemente conexas.
33 void Kosaraju() {
34     fill_n(vis, V, false);
35     for (int u = 0; u < V; ++u)
36         DFS(u);
37     fill_n(component, V, -1);
38     while (!S.empty()) {
39         Assign(S.top(), S.top());
40         S.pop();
41     }
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 from, to;
50         cin >> from >> to;
51         graph[from].push_back(to);
52         transpose[to].push_back(from);
53     }
54     //Imprime las componentes fuertemente conexas.
55     Kosaraju();
56     for (int i = 0; i < V; ++i)
57         cout << i << ": " << component[i] << '\n';
58     return 0;
59 }

```

2.5. Puentes y puntos de articulación.

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

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>

```

```

4  #include <utility>
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]; //Aristas.
10
11 bool artpoint[maxv]; //Puntos de articulacion.
12 vector<pair<int, int>> bridge; //Puentes.
13 int lst_id, idx[maxv], low[maxv]; //Indice de los vertices, menor vertice alcanzable.
14
15 void DFS(int u, int pred) {
16     idx[u] = low[u] = lst_id++;
17     int children = 0;
18     for (int v : graph[u]) {
19         if (idx[v] == -1) {
20             DFS(v, u);
21             low[u] = min(low[u], low[v]);
22             children++;
23             if ((pred == -1 && children > 1) || (pred != -1 && low[v] >= idx[u]))
24                 artpoint[u] = true;
25             if (low[v] > idx[u])
26                 bridge.push_back(make_pair(u, v));
27         }
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: $O(V^2E)$.

```

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 }

```

2.7. Emparejamiento máximo.

Algoritmo de Hopcroft-Karp. Complejidad: $O(\sqrt{V}E)$.

```

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  int U, V, E; //Numero de vertices en cada lado y numero de aristas.
9  vector<int> graph[maxv]; //Aristas que van de U a V.
10
11 int pairU[maxv], pairV[maxv], dist[maxv]; //Pares de vertices en el emparejamiento.
12
13 bool BFS() {
14     queue<int> Q;
15     fill_n(dist, U + 1, 1e9);
16     for (int u = 1; u <= U; ++u)
17         if (!pairU[u]) {
18             dist[u] = 0;
19             Q.push(u);
20         }
21     while (!Q.empty()) {
22         int u = Q.front();
23         Q.pop();
24         if (dist[u] < dist[0])
25             for (int v : graph[u])
26                 if (dist[pairV[v]] == 1e9) {
27                     dist[pairV[v]] = dist[u] + 1;
28                     Q.push(pairV[v]);
29                 }
30     }
31     return dist[0] != 1e9;
32 }
33
34 bool DFS(int u) {
35     if (!u)
36         return true;
37     for (int v : graph[u])
38         if (dist[pairV[v]] == dist[u] + 1 && DFS(pairV[v])) {
39             pairV[v] = u;
40             pairU[u] = v;
41             return true;
42         }
43     dist[u] = 1e9;
44     return false;
45 }
46
47 //Busca un emparejamiento maximo.
48 int HopcroftKarp() {
49     int size = 0;
50     fill_n(pairU, U + 1, 0);
51     fill_n(pairV, V + 1, 0);
52     while (BFS())
53         for (int u = 1; u <= U; ++u)
54             if (!pairU[u] && DFS(u))
55                 size++;
56     return size;
57 }
58
59 int main() {

```

```

60     ios_base::sync_with_stdio(0); cin.tie();
61     cin >> U >> V >> E;
62     //Lee la informacion de las aristas. Los vertices estan indexados en 1.
63     for (int i = 0; i < E; ++i) {
64         int u, v;
65         cin >> u >> v;
66         graph[u].push_back(v);
67     }
68     //Imprime la configuracion del emparejamiento.
69     cout << "Emparejamiento: " << HopcroftKarp() << '\n';
70     for (int u = 1; u <= U; ++u)
71         if (pairU[u])
72             cout << u << " - " << pairU[u] << '\n';
73     return 0;
74 }

```

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 subtract(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 = subtract(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 $O(n^3)$.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 #define maxn 100 //Maximo numero de renglones y columnas.
5
6 //Encuentra la forma escalonada reducida de la matriz A de m x n.
7 void GaussJordan(int m, int n, double matrix[][maxn]) {
8     for (int r = 0, c = 0; r < m && c < n; ++c) {
9         int pivot = r;
10        for (int i = r + 1; i < m; ++i)
11            if (fabs(matrix[pivot][c]) < fabs(matrix[i][c]))
12                pivot = i;
13        if (fabs(matrix[pivot][c]) < 1e-6)
14            continue;
15        if (pivot != r)
16            for (int j = c; j < n; ++j)
17                swap(matrix[r][j], matrix[pivot][j]);
18        for (int j = n - 1; j >= c; --j) {
19            matrix[r][j] /= matrix[r][c];
20            for (int i = 0; i < m; ++i)
21                if (i != r)
22                    matrix[i][j] -= matrix[i][c] * matrix[r][j];
23        }
24        ++r;
25    }
26 }
27
28 int main() {
29     ios_base::sync_with_stdio(0); cin.tie();
30     int m, n;
31     double matrix[maxn][maxn];

```

```
32 //Lee los datos de la matriz.
33 cin >> m >> n;
34 for (int i = 0; i < m; ++i)
35     for (int j = 0; j < n; ++j)
36         cin >> matrix[i][j];
37 //Imprime la forma escalonada reducida.
38 GaussJordan(m, n, matrix);
39 for (int i = 0; i < m; ++i) {
40     for (int j = 0; j < n; ++j)
41         cout << matrix[i][j] << ' ';
42     cout << '\n';
43 }
44 return 0;
45 }
```

3.5. Teorema Chino del Residuo.

```
1 #include <iostream>
2 #include <utility>
3 using namespace std;
4
5 //Algoritmo extendido de Euclides.
6 long long extendedEuclid(long long a, long long b, long long &x, long long &y) {
7     if (b == 0) {
8         x = 1; y = 0;
9         return a;
10    }
11    else {
12        long long gcd = extendedEuclid(b, a % b, y, x);
13        y -= (a / b) * x;
14        return gcd;
15    }
16 }
17
18 //Teorema Chino del Residuo. Regresa el residuo y el modulo.
19 pair<long long, long long> ChineseRemainder(int n, long long coef[], long long mod[]) {
20     pair<long long, long long> ans(0, 1);
21     for (int i = 0; i < n; ++i)
22         ans.second *= mod[i];
23     for (int i = 0; i < n; ++i) {
24         long long a = ans.second / mod[i], x, y;
25         extendedEuclid(a, mod[i], x, y);
26         long long tmp = (a * (x + ans.second)) % ans.second;
27         ans.first = (ans.first + coef[i] * tmp) % ans.second;
28     }
29     return ans;
30 }
31
32 int main() {
33     ios_base::sync_with_stdio(0); cin.tie();
34     int n;
35     cin >> n;
36     //Lee los datos de las ecuaciones.
37     long long coef[n], mod[n];
38     for (int i = 0; i < n; ++i)
39         cin >> coef[i] >> mod[i];
40     //Imprime la solucion.
41     pair<long long, long long> ans = ChineseRemainder(n, coef, mod);
42     cout << "x = " << ans.second << "k + " << ans.first << '\n';
43     return 0;
44 }
```

4. Geometría

4.1. Geometría 2D.

```
1 #include <cmath>
2 #include <vector>
```

```

3 using namespace std;
4 #define epsilon 1e-6 //Precision.
5
6 struct point {
7     double x, y;
8 };
9 typedef const point cpoint;
10
11 point operator + (cpoint &P, cpoint &Q) {
12     return point{P.x + Q.x, P.y + Q.y};
13 }
14
15 point operator - (cpoint &P, cpoint &Q) {
16     return point{P.x - Q.x, P.y - Q.y};
17 }
18
19 point operator * (cpoint &P, double c) {
20     return point{P.x * c, P.y * c};
21 }
22
23 point operator / (cpoint &P, double c) {
24     return point{P.x / c, P.y / c};
25 }
26
27 //Regresa el producto punto de dos vectores. Es 0 cuando son ortogonales.
28 double dot(cpoint &P, cpoint &Q) {
29     return P.x * Q.x + P.y * Q.y;
30 }
31
32 //Regresa la componente z del producto cruz de dos vectores. Es 0 cuando son paralelos.
33 double cross(cpoint &P, cpoint &Q) {
34     return P.x * Q.y - P.y * Q.x;
35 }
36
37 //Regresa la distancia euclidiana de un punto al origen.
38 double norm(cpoint &P) {
39     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. Envolverte 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
38 int main() {
39     ios_base::sync_with_stdio(0); cin.tie();
40     //Lee los puntos.
41     cin >> n;
42     for (int i = 0; i < n; ++i) {
43         int px, py;
44         cin >> px >> py;
45         P[i] = point(px, py);
46     }
47     //Imprime la envolvente convexa.
48     GrahamScan();
49     for (int i = 0; i < hull.size(); ++i)
50         cout << '(' << hull[i].x << ", " << hull[i].y << ")\n";
51     return 0;
52 }

```

5. Strings

5.1. Algoritmo de KMP.

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

```

1  #include <iostream>
2  using namespace std;
3  #define maxn 100000
4
5  string text, pattern; //Texto y patron.
6  int lps[maxn]; //Mayor prefijo que es sufijo propio.
7
8  void kmp_preprocess() {
9     lps[0] = 0;
10    for (int i = 1; i < pattern.size(); ++i) {

```

```

11     int j = lps[i - 1];
12     while (j > 0 && pattern[i] != pattern[j])
13         j = lps[j - 1];
14     if (pattern[i] == pattern[j])
15         j++;
16     lps[i] = j;
17 }
18 }
19
20 //Imprime todas las ocurrencias de pattern en text.
21 void kmp_search() {
22     kmp_preprocess();
23     for (int i = 0, j = 0; i < text.size(); ++i) {
24         while (j > 0 && text[i] != pattern[j])
25             j = lps[j - 1];
26         if (text[i] == pattern[j])
27             j++;
28         if (j == pattern.size()) {
29             cout << "Ocurrencia en la posicion " << i - j + 1 << '\n';
30             j = lps[j - 1];
31         }
32     }
33 }
34
35 int main() {
36     ios_base::sync_with_stdio(false); cin.tie();
37     cin >> text >> pattern;
38     kmp_search();
39     return 0;
40 }

```

5.2. Algoritmo de Aho Corasick.

Complejidad: $O(|T| + |P_1| + \dots + |P_n| + \#Ocurrencias)$.

```

1  #include <iostream>
2  #include <cstring>
3  #include <queue>
4  using namespace std;
5  #define maxc 26 //Longitud del alfabeto.
6  #define maxn 100 //Maximo numero de patrones.
7  #define maxs 100000 //Maximo numero de nodos.
8
9  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() +
74                     1 << '\n';
75     }
76     return 0;
77 }

```

5.3. Arreglo de sufijos.

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

```

1  #include <iostream>
2  #include <algorithm>
3  #include <utility>
4  using namespace std;
5  #define maxn 100000 //Longitud maxima del string.
6
7  string word; //String.
8  int n, suffixArray[maxn], lcp[maxn]; //Arreglo de sufijos y arreglo de mayor prefijo comun.
9
10 pair<pair<int, int>, int> rnk[maxn], tmp[maxn];
11 int bucket[maxn];
12
13 //Ordena los rangos. Otra opcion es usar sort(), pero la complejidad sera O(n log^2 n).
14 void RadixSort() {
15     int M = max(n, 256);
16     for (int k = 0; k < 2; ++k) {
17         fill(bucket, bucket + M, 0);
18         for (int i = 0; i < n; ++i)
19             bucket[k ? rnk[i].first.first : rnk[i].first.second]++;
20         for (int i = 1; i < M; ++i)
21             bucket[i] += bucket[i - 1];
22         for (int i = n - 1; i >= 0; --i)
23             tmp[--bucket[k ? rnk[i].first.first : rnk[i].first.second]] = rnk[i];

```

```
24     copy(tmp, tmp + n, rnk);
25 }
26 }
27
28 //Construye el arreglo de sufijos.
29 void buildSA() {
30     n = word.size();
31     for (int i = 0; i < n; ++i)
32         rnk[i] = make_pair(make_pair(word[i], word[min(i + 1, n - 1)]), i);
33     RadixSort();
34     for (int k = 2; k < n; k *= 2) {
35         int curr = 0, prev = -1;
36         for (int i = 0; i < n; ++i) {
37             if (rnk[i].first != make_pair(prev, rnk[i - 1].first.second))
38                 curr++;
39             prev = rnk[i].first.first;
40             rnk[i].first.first = curr;
41             suffixArray[rnk[i].second] = i;
42         }
43         for (int i = 0; i < n; ++i) {
44             int j = min(rnk[i].second + k, n - 1);
45             rnk[i].first.second = rnk[suffixArray[j]].first.first;
46         }
47         RadixSort();
48     }
49     for (int i = 0; i < n; ++i)
50         suffixArray[i] = rnk[i].second;
51 }
52
53 //Construye el arreglo de mayor prefijo comun.
54 void buildLCP() {
55     for (int i = 0; i < n; ++i)
56         bucket[suffixArray[i]] = i;
57     for (int i = 0, k = 0; i < n; ++i) {
58         if (bucket[i] == n - 1)
59             k = 0;
60         else {
61             int j = suffixArray[bucket[i] + 1];
62             while (i + k < n && j + k < n && word[i + k] == word[j + k])
63                 k++;
64             lcp[bucket[i]] = k;
65             if (k > 0)
66                 k--;
67         }
68     }
69 }
70
71 int main() {
72     ios_base::sync_with_stdio(0); cin.tie();
73     //Lee la palabra.
74     cin >> word;
75     word.push_back('$');
76     //Imprime el arreglo de sufijos y lcp.
77     buildSA();
78     buildLCP();
79     for (int i = 0; i < n; ++i)
80         cout << suffixArray[i] << ' ' << lcp[i] << ": " << word.substr(suffixArray[i]) << '\n';
81     return 0;
82 }
```