Repositorio

Búsqueda

Búsqueda binaria

Dado un elemento x y un vector a con n elementos ordenados de menor a mayor. Queremos encontrar la posición exacta donde se encuentra x en el vector a, y en caso de no encontrarse devolver el valor de -1. Usamos búsqueda binaria

```
int busqueda_binaria(vector<int> &a, int x)
{
    int l = 0, r = a.size()-1;
    while(l <= r)
    {
        int m = l + (r - l) / 2;

        if(a[m] == x) return m;

        if(a[m] < x) l = m + 1;
        else r = m - 1;
    }

    return -1;
}
```

Que resuelve el problema en complejidad $O(logn)$. Se puede extender el problema a alguna de sus variantes. Que pasa si queremos encontrar la posición del primer elemento estrictamente mayor que x. Para esto usamos

```
int busqueda_binaria(vector<int> &a, int x)
{
    int l = 0, r = a.size()-1;
    int res = -1;
    while(l <= r)
    {
        int m = l + (r - l) / 2;

        if(a[m] <= x) l = m + 1;
        else
        {
            res = m;
            r = m - 1;
        }
    }

    return res;
}
```

Teniendo la misma complejidad de $O(logn)$. En caso de querer la posición del primer elemento mayor o igual a x, se cambia la sentencia a[m] ≤ x por a[m] < x.

Segment tree

Dado un vector de enteros de tamaño n, queremos encontrar el mínimo elemento dentro de un rango dado. Además de esto queremos poder actualizar tal vector por rangos, es decir, queremos aumentar o disminuir los elementos del vector por un $valor$ dado en cierto rango, para así poder seguir encontrando la posición del minimo elemento de un rango dado. Esto úlitmo se conoce como lazy propagation. Para eso usamos la siguiente clase:

```
class SegmentTree
{
    public:
        vector<int> st, a, lazy;
        int n;
        int left(int p) {return (p << 1);}
        int right(int p) {return (p << 1) + 1;}

        // For update
        void push(int p)
        {
            st[left(p)] += lazy[p];
            st[right(p)] += lazy[p];
            lazy[left(p)] += lazy[p];
```

```
            lazy[right(p)] += lazy[p];
            lazy[p] = 0;
        }

        void build(int p, int l, int r)
        {
            if(l == r) st[p] = a[l];
            else
            {
                build(left(p), l, (l + r)/2);
                build(right(p), (l + r)/2 + 1, r);
                int p1 = st[left(p)], p2 = st[right(p)];

                st[p] = max(p1, p2);
            }
        }

        int query(int p, int l, int r, int i, int j)
        {
            if(i > r || j < l) return INT_MAX;
            if(i <= l && j >= r) return st[p];

            // For update
            push(p);

            int p1 = query(left(p), l, (l + r)/2, i, j);
            int p2 = query(right(p), (l + r)/2 + 1, r, i, j);

            return max(p1, p2);
        }


        // For update
        void update(int p, int valor, int l, int r, int i, int j)
        {
            if(i > r || j < l) return;
            if(i <= l && j >= r)
            {
                st[p] += valor;
                lazy[p] += valor;
                return;
            }

            push(p);
            update(left(p), valor, l, (l+r)/2, i, j);
            update(right(p), valor, (l+r)/2 + 1, r, i, j);
            st[p] = max(st[left(p)], st[right(p)]);
        }

        SegmentTree(vector<int> &A)
        {
            a = A;
            n = a.size();
            st.assign(4*n, 0);
            lazy.assign(4*n, 0);
            build(1, 0, n-1);
        }

        int query(int i, int j){return query(1, 0, n-1, i, j);}
        void update(int i, int j, int val)
        {
            update(1, val, 0, n-1, i, j);
        }
};
```

Para construir el segment tree se utiliza complejidad $O(n)$ tanto de memoria como de tiempo. Por otro lado, para actualizar y para enviar una query es necesario $O(logn)$. El segmente tree puede ir de temas sencillos como encontrar el mínimo así como el máximo número de repeticiones en un rango e incluso el vector ordenado de cierto rango.

Grafos

DFS

Dado un grafo de n nodos contando desde 0 a $n-1$ y m aristas. El grafo se realiza con lista de adyacencia global "grafo", y creamos un vector auxiliar de booleanos que identifique los nodos ya visitados, ambos con tamaño n. El DFS recorre el grafo con la idea de ultimo en llegar, primero en analizar. Luego el procedimiento para iterar sobre un grafo

```
vector<vector<int>> grafo;
vector<bool> vis;
void dfs(int nodo)
{
    stack<int> s;
    s.push(nodo);

    while(!s.empty())
    {
        nodo = s.top();
        s.pop();

        if(vis[nodo] == true) continue;
        vis[nodo] = true;

                // Iteramos sobre los hijos del nodo actual
        for(int i=0;i<int(grafo[nodo].size());++i)
        {
            if(vis[grafo[nodo][i]] == true) continue;
            s.push(grafo[nodo][i]);
        }
```

```
    }
}
```

El código corre con complejidad $O(m+n)$, donde n es la cantidad de nodos y m la cantidad de aristas. Igualmente, el código se puede realizar de forma recursiva con la misma complejidad:

```
vector<vector<int>> grafo;
vector<bool> vis;
void dfs(int nodo)
{
    if(vis[nodo] == true) return;
    vis[nodo] = true;

    for(int i=0;i<int(grafo[nodo].size());++i)
    {
        if(vis[grafo[nodo][i]] == true) continue;
        dfs(grafo[nodo][i]);
    }
}
```

BFS

Dado un grafo de n nodos contando desde 0 a $n-1$ y m aristas. El grafo se realiza con lista de adyacencia global "grafo", y creamos un vector auxiliar de booleanos que identifique los nodos ya visitados, ambos con tamaño n. El BFS funciona con el paradigma primero en llegar primero en analizar. Por lo tanto:

```
vector<vector<int>> grafo;
vector<bool> vis;
void bfs(int nodo)
{
    queue<int> q;
    q.push(nodo);

    while(!q.empty())
    {
        nodo = q.front();
        q.pop();

        if(vis[nodo] == true) continue;
        vis[nodo] = true;

        for(int i=0;i<int(grafo[nodo].size());++i)
        {
            if(vis[grafo[nodo][i]] == true) continue;
            q.push(grafo[nodo][i]);
        }
    }
}
```

El código corre con complejidad $O(m+n)$, donde n es la cantidad de nodos y m la cantidad de aristas.

Dijkstra

Dado un grafo pesado de n nodos contando desde 0 a $n-1$ y m aristas de pesos no negativos. El grafo se realiza con lista de adyacencia global "grafo", la cual para cada nodo contiene un vector de pares caracterizando con quien está conectado y el peso de la arista, y creamos un vector auxiliar de enteros inicializado en infinito (para el caso de enteros se recomiendo usar la cota INT_MAX), ambos con tamaño n. El algoritmo de Dijkstra resuelve la ruta más corta entre un nodo inicial (que llamaremos nodo) al resto de nodos en el grafo.

```
vector<vector<pair<int, int>>> grafo;
vector<int> dis;

void dijkstra(int nodo)
{
    priority_queue<array<int, 2>> pq;
    pq.push({0, nodo});
    dis[nodo] = 0;

    while(!pq.empty())
    {
        auto [d, nodo] = pq.top();
        d = -d;
        pq.pop();

        if(dis[nodo] < d) continue;
        for(auto &e:grafo[nodo])
        {
            if(dis[e.first] > d + e.second)
            {
                dis[e.first] = d + e.second;
                pq.push({-dis[e.first], e.first});
            }
        }
    }
}
```

Este algoritmo tiene complejidad de $O((n+m)logn)$, donde n es la cantidad de nodos y m cantidad de aristas.

Bellman-Ford

Dado un grafo pesado de n nodos contando desde 0 a $n-1$ y m aristas de pesos posiblemente negativo. El grafo se realiza con lista de adyacencia global "grafo", la cual para cada nodo contiene un vector de pares caracterizando con quien está conectado y el peso de la arista, y creamos un vector auxiliar de enteros inicializado en infinito (para el caso de enteros se recomiendo usar la cota INT_MAX), ambos con tamaño n. Si se quiere conocer la distancia mínima entre un nodo inicial y el resto del grafo

```
vector<vector<pair<int, int>>> grafo;
vector<int> dis;

void Bellman_Ford(int nodo)
{
```

```
    dis[nodo] = 0;
    for(int i=0;i<n-1;++i)
    {
        for(int u=0;u<n;++u)
        {
            for(auto &e:grafo[u])
            {
                dis[e.first] = min(dis[e.first], dis[u] + e.second);
            }
        }
    }
}
```

Que corre en complejidad $O(n * m)$ donde n es la cantidad de vértices y m es la cantidad de aristas. En caso de querer conocer si existen ciclos negativos, es decir, aquellos que empiecen en un nodo y terminen en si mismo con la suma de los pesos de las aristas negativo, entonces después de Bellman Ford corremos esta función

```
bool Negative_Cycle()
{
    for (int u=0;u<n;++u)
    {
        for(auto &e:grafo[u])
        {
            if(dis[e.first] > dis[u] + e.second) return true;
        }
    }

    return false;
}
```

la cual regresa positivo si existe un ciclo negativo y negativo en caso contrario. Tiene complejidad menor al Bellman Ford.

Floyd Warshall

Dado un grafo pesado con n nodos a través de una matriz de adyacencia (considerando 0 para la entrada i, i). Queremos encontrar la distancia mínima entre cualquier par de nodos. Para eso, realizamos la llamada a la siguiente función:

```
int n;
vector<vector<int>> grafo;
void Floyd_Warshall()
{
    for(int k=0;k<n;++k)
    {
        for(int i=0;i<n;++i)
        {
            for(int j=0;j<n;++j)
            {
                grafo[i][j] = min(grafo[i][j], grafo[i][k] + grafo[k][j]);
            }
        }
    }
}
```

Este código se corre con complejidad $O(n^3)$, por lo que computacionalmente solo tiene sentido para $n \leq 400$.

Kruskal

Dado un grafo de n vértices y m aristas. El grafo se realiza con lista de adyacencia global "grafo", la cual para cada nodo contiene un vector de pares caracterizando con quien está conectado y el peso de la arista. Se desea encontrar el árbol mínimo de expansión (MST), este es una estructura tipo árbol para todo par de nodos u y v exista un camino que los conecte, minimizando el peso total del árbol. Cabe recalcar que si el grafo no es conexo, el resultado será un bosque (conjunto de árboles). Para esto:

```
int n;
vector<vector<pair<int, int>>> grafo;

ld Kruskal()
{
    priority_queue<array<int, 3>> pq;

    for(int i=0;i<n;++i)
    {
        for(auto &e:grafo[i])
        {
            pq.push({-e.second, i, e.first});
        }
    }

    int res = 0;

    Union_Find UF(n);
    while(!pq.empty())
    {
        auto [d, x, y] = pq.top();
        d = -d;
        pq.pop();

        cout << x << " " << y << " " << d << nl;

        if(UF.unionn(x, y))
        {
            res += d;
        }
    }

    return res;
}
```

Donde retornamos el peso total del árbol construido. La complejidad de este algoritmo es de $O(mlogm)$. El requerimento básico para este problema es incluir el Union Find, que se encuentra a continuación. Sin este, la complejidad del algoritmo sería considerablemente mayor.

Union Find

Creamos la clase de Union Find con la finalidad de saber si se forma un ciclo cuando añadimos una arista más a un conjunto ya establecido de aristas. Cuando se crea la clase se necesita pasar la cantidad de nodos como parámetro, de tal forma que:

```
class Union_Find
{
    public:
        int n;
        vector<int> p, h;

        Union_Find(int n)
        {
            this -> n = n;
            p.assign(n, 0);
            h.assign(n, 0);

            for(int i=0;i<n;++i)
            {
                p[i] = i;
            }
        }

        int findd(int i)
        {
            if(p[i] != i) p[i] = findd(p[i]);
            return p[i];
        }

        bool same(int i, int j)
        {
            return findd(i) == findd(j);
        }

        bool unionn(int i, int j)
        {
            if(!same(i, j))
            {
                int x = findd(i), y = findd(j);

                if(h[x] > h[y]) p[y] = x;
                else
                {
                    p[x] = y;
                    if(h[x] == h[y]) ++h[y];
                }

                return true;
            }

            return false;
        }
};
```

El código corre en tiempo prácticamente constante, es decir, $O(n)$. Una forma de usarlo es exhibida en el Kruskal. Cuando se corre la función unionn se realiza la unión de los dos nodos dados a través de su arista en común en dado caso de que no compartan mismo padre (es decir, que no exista ya un camino entre estos dos nodos).

Matemática

Criba de Eratóstenes

Dado un entero n, este algoritmo devuelve los números primos en el intervalo $[2, n]$.

```
vector<int> Criba(int n)
{
    vector<int> primos;
    vector<int> menor(n + 1, 0);

    for (int i=2; i<=n; ++i)
    {
        if (menor[i] == 0)
        {
            primos.push_back(i);
            menor[i] = i;
        }

        for (int j=0; j<primos.size() && primos[j]<=menor[i] && i*primos[j]<=n; ++j)
        {
            menor[i * primos[j]] = primos[j];
        }
    }

    return primos;
}
```

Este código tiene complejidad $O(n)$.

GCD extendido

Dados dos enteros a, b, la siguiente función regresa $gcd(a, b)$ y además calcula los valores x, y tal que $ax + by = gcd(a, b)$

```
int extended_gcd(int a, int b, int &x, int &y)
{
    if (a == 0)
    {
        x = 0;
        y = 1;
        return b;
    }

    int x1, y1;
    int gcd = extended_gcd(b%a, a, x1, y1);

    x = y1 - (b/a) * x1;
    y = x1;

    return gcd;
}
```

Complejidad: $O(log(min(a,b)))$

Congruencia lineal

Dada la expresión $ax \equiv b(mod n)$. Si se conocen a, b y n queremos encontrar todos los valores de $x \in [0, n-1]$ tal que cumplen tal sentencia. De esta forma:

```
vector<int> linear_congruence(int a, int b, int n)
{
    a = a%n;
    b = b%n;

    int u = 0, v = 0;
    int d = extended_gcd(a, n, u, v);

    vector<int> res;
    if(b % d != 0) return res;

    int x0 = (u * (b/d)) % n;
    if(x0 < 0) x0 += n;

    for(int i=0;i<d;++i) res.push_back((x0 + i * (n/d)) % n);

    return res;
}
```

La complejidad del algoritmo es considerada como $O(log(min(a,n)))$.

Exponenciación binaria

Dado un número a y su exponente b, la exponenciación binaria devuelve el valor a^b en $O(logb)$

```
ll bin_pow(ll a, ll b)
{
    if(b == 0) return 1;

    ll res = bin_pow(a, b/2);

    if(b%2 == 0) return res * res;
    return res * res * a;
}
```

Exponenciación binaria de matrices

Dada una matriz cuadrada A (en forma de vector de vectores) y un exponente *b*, la exponenciación binaria de matrices resuelve A^b

```
vector<vector<ll>> multiplicacion_matrices(vector<vector<ll>> &a, vector<vector<ll>> &b)
{
    vector<vector<ll>> res(a.size(), vector<ll>(b[0].size(), 0));

    for(int i=0;i<res.size();++i)
    {
        for(int j=0;j<res[i].size();++j)
        {
            for(int k=0;k<a[i].size();++k) res[i][j] += a[i][k] * b[k][j];
        }
    }
    return res;
}

vector<vector<ll>> exponenciacion_matriz(vector<vector<ll>> a, int n)
{
    if(n == 0)
    {
        for(int i=0;i<a.size();++i)
        {
            for(int j=0;j<a.size();++j)
            {
                if(i == j) a[i][j] = 1;
                else a[i][j] = 0;
            }
        }
        return a;
    }
    if(n == 1) return a;

    vector<vector<ll>> res = exponenciacion_matriz(a, n/2);
    res = multiplicacion_matrices(res, res);
```

```
    if(n%2 == 0) return res;
    return multiplicacion_matrices(a, res);
}
```

El código corre en $O(Alogb)$, donde A representa la cantidad de operaciones necesarias para la multiplicación de dos matrices. Estas se utilizan para resolver relaciones de recurrencia.

Cadena

KMP

El algoritmo de Knuth-Morris-Pratt (KMP) es una técnica eficiente para buscar subcadenas en un texto, usando información sobre coincidencias previas para evitar comparaciones redundantes. Dada dos cadenas, p y t, donde la primera es un patrón y la segunda es un texto, KMP encuentra el número de veces que aparece el patrón p en t y en que índices se encuentran.

```
vector<int> b;
void kmpPreprocess(string &p)
{
    b.resize(p.size() + 1);
    int i = 0, j = -1;
    b[0] = -1;
    while(i < int(p.size()))
    {
        while(j >= 0 && p[i] != p[j]) j = b[j];
        ++i, ++j;
        b[i] = j;
    }
}

void kmpSearch(string &p, string &t)
{
    int i = 0, j = 0;
    while(i < int(t.size()))
    {
        while(j >= 0 && t[i] != p[j]) j = b[j];
        ++i, ++j;
        if(j == int(p.size()))
        {
            cout << i-j << endl;
            j = b[j];
        }
    }
}
```

Complejidad $O(m+n)$, donde m es la longitud de la cadena t y n la longitud de la cadena p.

Función z

Otra manera más sencilla de implementar para buscar la cantidad de veces que aparece un patrón en una cadena es a través de la función z. Esta función devuelve un vector de enteros z con tamaño igual a una cadena dada s. Tal vector almacena la cantidad máxima de caracteres empezando de la posición i que coincide con los primeros caracteres de s. Esto se ve así:

```
vector<int> funcion_z(string &s)
{
    int n = s.size();
    vector<int> z(n);
    int l = 0, r = 0;

    for(int i=1;i<n;++i)
    {
        if(i < r) z[i] = min(r - i, z[i - l]);

        while(i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];

        if(i + z[i] > r)
        {
            l = i;
            r = i + z[i];
        }
    }
    z[0] = n;

    return z;
}
```

Este algoritmo tiene complejidad $O(n+m)$. Si se desea encontrar un patrón p de tamaño m en un texto t de tamaño n, se deben concatenar el patrón y el texto de forma que $s = p + \$ + t$, donde el signo de pesos es un símbolo cualquiera que no esté en p o t y sirve como separador entre patrón y texto. Luego, lanza la función z. Con el vector retornado, se verifica el número de veces que aparece el tamaño del patrón (m) en el vector y ese será el resultado buscado.

Trie

Un trie es una estructura de datos tipo árbol utilizada principalmente para almacenar y buscar palabras o cadenas de caracteres, donde cada nodo representa un carácter y las rutas desde la raíz hasta los nodos hoja forman las palabras completas. Dado una colección de cadenas

```
// Cantidad de posibles elementos de las cadenas
int K = 26;

struct Nodo
{
    vector<Nodo*> hijos;
    bool fin;

    Nodo()
    {
        fin = false;
        hijos.resize(26, nullptr);
    }
};
```

```
struct Trie
{
    Nodo *root;

    Trie()
    {
        root = new Nodo;
    }

    void insertar(string &s)
    {
        Nodo *actual = root;
        for(char &c:s)
        {
            if(actual -> hijos[c-'a'] == nullptr)
            {
                actual -> hijos[c-'a'] = new Nodo;
            }
            actual = actual -> hijos[c-'a'];
        }
        actual -> fin = true;
    }
};
```

donde cada nodo almacena si es el fin de una palabra. Al usar vector de punteros, la complejidad para añadir una cadena es de $O(n)$ en tiempo y $O(n*K)$ de memoria, donde n es la longitud de la cadena. Por otro lado, para buscar en el trie tiene complejidad $O(n)$ en tiempo y $O(1)$ de memoria. Sin embargo existe otra posibilidad, que es usar mapas. Cambiando la estructura del nodo a:

```
struct Nodo
{
    map<char, Nodo*> hijos;
    bool fin;

    Nodo()
    {
        fin = false;
    }
};
```

el cual tiene complejidad para añadir una cadena de $O(nlogK)$ en tiempo y $O(n)$ de memoria. Por otro lado, para consulta tiene la misma complejidad de tiempo y $O(1)$ de memoria.

Arrays

Longest increasing subsequence (LIS)

Dado un vector a de n enteros. Queremos encontrar los valores de la subcadena monótona creciente (o su tamaño). Recordemos que una subcadena es un subconjunto de elementos del vector original que hereda el orden del vector original. Por lo tanto:

```
int longest_increasing_subsequence(vector<int> &a)
{
    vector<int> res;
    for(int i=0;i<a.size();++i)
    {
        auto p = lower_bound(res.begin(), res.end(), a[i]);

        if(p == res.end()) res.push_back(a[i]);
        else *p = a[i];
    }

    return res.size();
}
```

Este programa hace uso de la búsqueda binaria incluida en c++. De tal forma que la complejidad total del código es de $O(nlogn)$. En caso de querer retornar el vector con los valores, se debe cambiar la función a vector<int> y retornar solamente res. Para hacerla no decreciente es suficiente con cambiar lower_bound por upper_bound.

Algoritmos específicos

Sweep line para intervalos

Dada una sucesión de intervalos de la forma $[a_i, b_i]$. Se quiere calcular la cantidad máxima de intervalos que se intersectan en al menos un punto. Para esto:

```
int n;
vector<pair<int, int>> intervalos;

int sweep_line()
{
    int res = 0;

    vector<pair<int, int>> a;
    for(int i=0;i<n;++i)
    {
        a.push_back({intervalos[i].first, 1});
        a.push_back({intervalos[i].second + 1, -1});
    }

    sort(a.begin(), a.end());

    int actual = 0;
    for(auto [i, j] : a)
    {
        actual += j;
        res = max(res, actual);
    }
```

```
    return res;
}
```

Este código es tiene complejidad $O(nlogn)$, debido a que se debe de ordenar los nuevos intervalos. Se puede optimizar la parte de push_back si se crea el vector a con una dimensión definida desde un principio.

Trucos

- ++i más rápido que i++
- Para estructuras no dinámica, array es más rápido que vector
- __builtin_popcount(num): Retorna la cantidad de bits encendidos de num.