Resumo STL

September 14, 2018

1 Containers Importantes

Os mais usados são **vector**, **pair**, **map**, **set**.

1.1 Vector

Definido em **<vector>**.

Funções definidas:

- empty, size, clear push_back, pop_back, insert, erase, begin, end

1.2 Balanced Search Tree (Map/Set)

Tabela de símbolos genérica, com chave e valor.
Definido em **<map>** e **<set>**.
A diferença é que map tem chave e valor, e set tem só valor.
Funções definidas:

- clear, insert, insert_or_assign, emplace, emplace_hint, try_emplace, erase, swap, extract, merge
- count, find, contains, equal_range, lower_bound, upper_bound
- key_comp, value_comp

```
// Construtores
map<string, vector<int> > mapa;
set<string> conjunto;

const map<std::string, int> init {
 {"this", 100},
};

bool cmp()(int lhs, int rhs) const {
   return lhs > rhs;
}
map<int, int, cmp> mapa_compare;
```

1.3 Bitset

Definido em **bitset**

- test, all, any, none, count
- set, reset, flip
- to_string, to_ulong, to_ullong
- &= |= ≙ = -> and, or, xor, not
- «, » shift left and right

1.4 Linked List

Dois tipos de lista: list e forward_list.

- clear, insert, emplace, erase, push_back, emplace_back, pop_back, push_front, emplace_front, pop_front, resize, swap, (emplace_after, erase_after
- merge, splice, remove, remove_if, reverse, unique, sort

1.5 Stack

1.6 Queue and Double Queue

1.7 Pair

Pair é uma struct, e não uma classe.
Pra fazer um pair:

```
#typedef pair<int,int> p;
p.make_pair(first, second);
```

1.8 Heap (priority_queue)

1.9 Hash Table (Unordered_map)

1.10 Graph

2 Algoritmos Importantes

- find(first, last, value):
- copy(first, last, result):
- swap(a, b):
- remove(first, last, value):
- unique(first, last [,pred]):

- reverse(first, last):
- sort(first, last [, comp]):
- stable_sort(first, last [,comp]):
- merge(first1, last1, first2, last2, result [,comp]):
- inplace_merge():
- includes(first1, last1, first2, last [,compare]):
- is_sorted(first, last [,compare]):
- min(a, b), max(a, b):
- iota(first, last, value):
- binary_search(first, last, value [,compare]);
- next_permutation(first, last [,compare]);
- min_element(first, last):
- max_element(first, lat):

3 Misc

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstdio>
#include <climits>

using namespace std;

#define _inline(f...) f() __attribute__((always_inline)); f
#define foreach(i, c) for( __typeof( (c).begin() ) i = (c).begin(); i != (c).end(); ++
#define all(x) (x).begin,(x).end
#define fori(i, n) for(int i = 0; i < (n); i++)
#define INF 1000000000

typedef long long    ll
typedef pair<int,int> ii;
typedef pair<int,ii>  iii
typedef vector<int> vi;
typedef vector<ii> vii;

//memset(arr, 0, sizeof arr); // Clear array

// Imprimir N casas de Pi
```

```
double pi = 2 * acos(0,0);
cin << n << pi;
printf("%.␣*lf\n", n, pi);

// Imprimir numeros distintos ordenados
sort(ALL(v)); UNIQUE(v);

// Gen todos os subsets
int p[20], N = 2
for(i = 0; i < 20; i++) p[i] = i
for(i = 0; i < (i << N); i++)
    for(j = 0; j < N; j++)
        if (i & (i << j))
            printf("%d␣", p[j])
    printf("\n")

/*—————————— GRAPH —————————— */

int V, E, weight, a, b;
int adj_mat[100][100];
vector<vii> adj_list;
priority_queue<pair<int, ii> > edge_list;

vector<bool> visited;

void printGraph() {
    foreach(it, adj_list) {
        cout << "No:␣" << (it - adj_list.begin()) << endl;
        foreach(it2, *it) {
            printf("␣␣␣Vizinho:␣%d,␣Peso:␣%d\n", it2->first, it2->second);
        }
    }
}

void addEdge(int v, int u, int w) {
    adj_list[v].push_back(make_pair(u, w));
    E++;
}

#define DFS_WHITE -1
#define DFS_BLACK 1 // Visited
vi dfs_num;
int numCC;
/* numCC = 0;
 * dfs_num.assign(V, DFS_WHITE)
 * fori(i, V)
 *     if(dfs_num[i] == WHITE)
 *         printf("Component %d:" ++numCC), dfs(i), printf("\n");
```

```
 */
void dfs(int node) {
   dfs_num[node] = DFS_BLACK;
   fori(j, adj_list.size()) {
      ii v = adj_list[node][j];
      if(dfs_num[v.first] == DFS_WHITE)
         dfs(v.first);
   }
}

/* Colorir as aretas do grafo, dfs_num vai ter as cores */
/* numCC = 0;
 * dfs_num.assign(V, DFS_WHITE);
 * fori(i, v)
 *    if(dfs_num[i] == DFS_WHITE)
 *       flood_fill(i, ++numCC);
 */
void flood_fill(int u, int color) {
   dfs_num[u] = color;
   fori(j, adj_list.size()) {
      ii v = adj_list[u][j];
      if(dfs_num[v.first] == DFS_WHITE)
         flood_fill(v.first, color);
   }
}

vi topoSort;
/* topoSort.clear()
 * dfs_num.assign(V, WHITE)
 * fori(i, V)
 *    if (dfs_num == WHITE)
 *       top_sort(i)
 * reverse(topoSort.begin(), topoSort.end());
 */
void top_sort(int u) {
   dfs_num[u] = DFS_BLACK;
   fori(j, adj_list.size()) {
      ii v = adj_list[u][j];
      if(dfs_num[v.first] == DFS_WHITE)
         top_sort(v.first);
   }
   topoSort.push_back(u);
}

/* Organizar as arestas de um grafo em: forward, back, bidirecional */
#define DFS_GRAY 2
vi dfs_parent;
/* numCC = 0
```

```
 * dfs_num.assign(V, WHITE), dfs_parent.assign(V, -1)
 * fori(i, V)
 *     if(dfs_num == WHITE)
 *         printf("Component %d:\n", ++numCC), graph_check(i);
 */
void graph_check(int u) {
   dfs_num[u] = DFS_GRAY;
   fori(j, adj_list.size()){
      ii v = AdjList[u][j];
      if (dfs_num[v.first] == DFS_WHITE) {
         dfs_parent[v.first] = u;
         graph_check(v.first);
      }
      else if (dfs_num[v.first] == DFS_GRAY) {
         if (v.first == dfs_parent[u]
               printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first, v.first, u);
               else
               printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
               }
               else if (dfs_num[v.first] == DFS_BLACK)
               printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
               }
               dfs_num[u] = DFS_BLACK;
               }

               /* Checar se um grafo pode ser colorido de 2 cores diferentes */
               void bipartite_check(int s) {
               queue<int> q;
               q.push(s);
               vi color(V, INF);
               color[s] = 0;
               bool isB = true;
               while(!q.empty() && isB) {
               int u = q.front(); q.pop();
               fori(j, adj_list.size()) {
                  ii v = adj_list[u][j];
                  if(color[v.first] == INF) {
                     color[v.first] = 1-color[u];
                     q.push(v.first);
                  } else if (color[v.first] == color[u]) {
                     isB = false;
                     break;
                  }
               }
               }
               }

vi dfs_low;
```

```
vi arti_vtx;
int dfsNCont = 0,dfsRoot, rootChild;
/* dfs_low.assign(V, 0);
 * dfs_parent.assign(V, -1), arti_vtx.assign(V, 0)
 * fori(i, V)
 *     if(dfs_num == WHITE)
 *         dfsRoot = i; rootChild = 0;
 *         articulation(i);
 *         arti_vtx[dfsRoot] = (rootChild >1);
 */
void articulation(int u) {
   dfs_low[u] = dfs_num[u] = dfsNCont++;
   fori(j, adj_list.size()) {
      ii v = adj_list[u][j];
      if(dfs_num[v.first] == DFS_WHITE) {
         dfs_parent[v.first] = u;
         if(u == dfsRoot) rootChild++;
         articulation(v.first);
         if (dfs_low[v.first] >= dfs_num[u])
            arti_vtx[u] = true;
         if (dfs_low[v.first] > dfs_num[u])
            printf("␣Edge␣[%d,%d]␣is␣a␣bridge\n", u, v.first);
         dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
      } else if(v.first != dfs_parent[u])
         dfs_low[u] = min(dfs_low[u], dfs_num[v.first])
   }
}

void strongly_connected(int u) {
}

/* Kruskal and Prim */
vi taken;
priority_queue<ii> pq;
void process(int vtx) {
   taken[vtx] = 1;
   fori(j, adj_list.size()) {
      ii v = adj_list[vtx][j];
      if (!taken[v.first])
         pq.push(ii(-v.second, -v.first));
   }
}

void MST() {
   //edge_list.push_back(make_pair(w, ii(u, v)));
   sort(edge_list.begin(), edge_list.end());
   int mst_cost = 0;
   UnionFind UF(v);
```

```
    fori(i, E) {
        pair<int, ii> front = edge_list[i];
        if (!UF.isSameSet(front.second.first, front.second.second)) {
            mst_cost += front.first;
            UF.unionSet(front.second.first, front.second.second);
        }
    }
}

/* Dijkstra: SSSP em grafo com pesos positivos */
void dijkstra(int s) {
    vi dist(V, INF); dist[s] = 0;
    priority_queue<ii, vector<ii>, greater<ii> > pq;
    pq.push(ii(0, s));
    while(!pq.empty()) {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if(d > dist[u]) continue;
        fori(i, adj_list[u].size()) {
            ii v = adj_list[u][j];
            if(dist[u]+v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
    fori(i, V) {
        printf("SSSP(%d,␣%d)␣=␣%d\n", s, i dist[i]);
    }
}


/* Bellman Ford: SSSP em grafo com pesos negativos */
void bellmanFord(int s) {
    vi dist(V, INF);
    dist[s] = 0;
    for (int i = 0; i < V - 1; i++)
        for (int u = 0; u < V; u++)
            for (int j = 0; j < (int)AdjList[u].size(); j++) {
                ii v = AdjList[u][j];
                dist[v.first] = min(dist[v.first], dist[u] + v.second);
            }

    bool hasNegativeCycle = false;
    fori(u, V) {
        fori(j, adj_list.size()) {
            ii v = adj_list[u][j];
            if(dist[v.first] > dist[u] + v.second)
```

```
            hasNegativeCycle = true;
        }
    }
}

/* Floyd-Warshall: All pairs shortest path */
void floydWarshall() {
    fori(k, V) {
        fori(i, V) {
            fori(j, V) {
                adj_mat[i][j] = min(adj_mat[i][j], adj_mat[i][k]+adj_mat[k][j]);
            }
        }
    }
    fori(i, V) {
        fori(j, V) {
            printf("APSP(%d,␣%d)␣=␣%d\n", i, j, adj_mat[i][j]);
/* Imprimir o caminho
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        p[i][j] = i;
for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
                AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
                p[i][j] = p[k][j];
            }
*/
}

void transitiveClosure() {
    fori(k, V)
        fori(i, V)
            fori(j, V)
                adjMat[i][j] |= (adjMat[i][k] & adjMat[k][j])
}

/* Network Flow: Edmond Karp */
int res[V][V], mf, f, s, t;
vi p;
void aug(int v, int minEdge) {
    if(v == s) {
        f = minEdge;
        return;
    } else if(p[v] != -1) {
        aug(p[v] -= f);
        res[v][p[v]] += f
```

```
    }
}
void karp() {
    mf = 0;
    while(true) {
        f = 0;
        vi dist(V, INF);
        dist[s] = 0;
        queue<int> q;
        q.push(s);
        p.assign(V, -1);
        while(!q.empty()) {
            int u = q.front(); q.pop();
            if(u == t) break;
            fori(v, V)
                if(res[u][v] > 0 && dist[v] == INF)
                    dist[v] = dist[u]+1, q.push(v), p[v] = u;
        }
        aug(t, INF);
        if(f == 0) break;
        mf += f;
    }
    printf("%d\n", mf);
}

/*----------- Union Find Disjoint Set -----------*/

class UnionFind {
    private:
        vi p, rank, setSize;
        int numSets;
    public:
        UnionFind(int N) {
            setSize.assign(N, 1);
            rank.assign(N, 0);
            p.assign(N, 0);
            fori(i, N)
                p[i] = i;
            numSets = N;
        }
        int findSet(int i) {
            return (p[i] == i) ? i : (p[i] = findSet(p[i]));
        }
        bool isSameSet(int i, int j) {
            return findSet(i) == findSet(j);
        }
        void unionSet(int i, int j) {
            if(!isSameSet(i, j)) {
```

```
            numSets--;
            int x = findSet(i), y = findSet(j);
            if(rank[x] > rank[y]) {
               p[y] = x;
               setSize[x] += setSize[y];
            } else {
               p[x] = y;
               setSize[y] += setSize[x];
               if(rank[x] == rank[y])
                  rank[y]++;
            }
         }
      }
      int numDisjointSets(){ return numSets; }
      int sizeOfSet(int i) { reuturn setSize[findSet(i)]; }
}

/*----------- Segment Tree -----------*/
/* Responder consultas com ranges dinamicas
 * Ex: Achar o index do menor elemento na range[i,j]
 */

class SegmentTree{
   private:
      vi st, A;
      int n;
      int left(int p) { return p << 1; }
      int right(int p) { return (p << 1) -1; }
      void build(int p, int L, int R) {
         if(L == R)
            st[p] = 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] = (A[p1] <= A[p2]) ? p1 : p2;
         }
      }
      int rmq(int p, int L, int R, int i, int j) {
         if(i > R || j < L) return -1;
         if(L >= i && R <= j) return st[p];

         int p1 = rmq(left(p), L, (L+R)/2, i, j);
         int p2 = rmq(right(p), (L+R)/2+1, R, i, j);

         if(p1 == -1) return p2;
         if(p2 == -1) return p1;
         return (A[p1] <= A[p2]) ? p1 : p2;
```

```
        }
        int update_point(int p, int L, int R, int idx, int new_value) {
            int i = idx, int j = idx;
            if (i > R || j < L) return st[p];
            if (L == i && R == j) {
                A[i] = new_value;
                return st[p] = L;
            }
            int p1, p2;
            p1 = update_point(left(p), L, (L+R)/2, idx, new_value);
            p2 = update_point(right(p), (L+R)/2+1, R, idx, new_value);
            return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
        }

    public:
        SegmentTree(const vi &_A) {
            A = _A;
            n = (int) A.size();
            st.assign(4*n, 0);
            build(1, 0, n-1);
        }
        int rmq(int i, j) { return rmq(1, 0, n-1, i, j);}
        int update_point(int idx, int new_value) {
            return update_point(1, 0, n-1, idx, new_value);
        }
};

/*----------- Fenwick Tree -----------*/
/* AKA Binary Indexed Tree - Dynamic Cumulative Frequency Table
 * Tambem pode ser usada pra resolver problemas de Range Sum Query
 */
class FenwickTree {
    private:
        vi ft;

    public:
        FenwickTree(){}
        FenwickTree(int n) { ft.assign(n+1, 0); }
        int rsq(int b) {
            int sum = 0;
            for(; b; b -= LSOne(b)) sum += ft[b];
            return sum;
        }
        int rsq(int a, int b) {
            return rsq(b) - (a == 1 ? 0 : rsq(a-1));
        }
        void adjust(int k, int v) {
            for(; k < (int)ft.size(); k += LSOne(k))
```

```
            ft[k] += v;
        }
};

/*------------ Max 1D Range Sum ------------*/
int range_sum(vi *a) {
    int run_sum = 0, ans = 0;
    fori(i, a->size()) {
        if(run_sum + a->at(i) >= 0) {
            run_sum += a->at(i);
            ans = max(ans, run_sum);
        } else
            run_sum = 0;
    }
    return anx;
}

/*------------ Longest Increasing Sequence ------------*/
void reconstruct_print(int end, vi *a, vi *p) {
    int x = end;
    stack<int> s;
    for(; p[x] >= 0; x = p[x]) s.push(a[x]);
    printf("[%d", a[x]);
    for(; !s.empty(); s.pop()) printf(", %d", s.top());
    printf("]\n");
}

int lis(vi *a) {
    int L[MAX_N], L_id[MAX_N], P[MAX_N];
    int lis = 0, lis_end = 0;
    fori(i, a->size()) {
        int pos = lower_bound(L, L+lis, a->at(i)) - L;
        L[pos] = A[i];
        L_id[pos] = i;
        P[i] = pos ? L_id[pos-1] : -1;
        if(pos+1 > lis) {
            lis = pos+1;
            lis_end = 1;
        }
        printf("Considering element A[%d] = %d\n", i, A[i]);
        printf("LIS ending at A[%d] is of length %d: ", i, pos + 1);
        reconstruct_print(i, A, P);
        print_array("L is now", L, lis);
        printf("\n");
    }

    printf("Final LIS is of length %d: ", lis);
    reconstruct_print(lis_end, A, P);
```

```
    return 0;

}

/*========== Math ==========*/
class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int caseNo = 1;
        while(1) {
            int N = sc.nextInt(), F = sc.nextInt();
            if(N == 0 && F == 0) break;
            for(int i = 0; i < N; i) {
                BigInteger V = sc.nextBigInteger();
                sum = sum.add(V);
            }
        }

        // Ler um inteiro e converter pra base b;
        BigInteger p = new BigInteger(sc.next(), b);

        // Greatest Commmon Divisor
        BigInterger p, q;
        BigInteger gcd = p.gcd(q);
        sout(p.divise(gcd) + "/" + q.divide(gcd));

        // Modulos
        // x elevado a y mod n
        sout(x.modPow(y, n));
    }
}

/* Binomial Coefficient
 * C(n, 0) = C(n,n) = 1
 * C(n, k) = C(n-1, k-1)+C(n-1, k)
 */

/* Catalan Numbers
 * Cat(n) = (2n!)/(n!n!(n+1))
 * Cat(n+1) = [(2n+2)(2n+1)]/[(n+2)(n+1)] * cat(n)
 */

// Sieve of Eratosthenes
void sieve(ll upperbound) {
    _sieve_size = upperbound+1;
    bs.set();
    bs[0] = bs[1] = 0;
    for(ll i = 2; i <= _sieve_size; i++)
```

```
      if(bs[i]) {
         for(ll j = i*i; j <= _sieve_size; j+= i)
            bs[j] = 0;
         primes.push_back((int)i);
      }
}
bool isPrime(ll N) {
   if(N <= _sieve_size) return bs[N];
   fori(i, primes.size())
      if(N % primes[i] == 0) return false;
   return true;
}

int main() {
   int aux[] = {18, 17, 13, 19, 15, 11, 20};
   vi arr = vector<int>(begin(aux), end(aux));
   vi seg_tree = vi(arr.size()*4, 0);
   segTree(&arr, &seg_tree,1, 0, arr.size()-1);
   cout << rmq(&arr, &seg_tree, 1, 3) << endl;
   cout << rmq(&arr, &seg_tree, 4, 5) << endl;
   cout << rmq(&arr, &seg_tree, 3, 4) << endl;

   return 0;
}
```