Contenido

[2-SAT Problem 1](#_Toc497699630)

[Suffix Array NlogN with LCP 2](#_Toc497699631)

[Convex Hull Trick (minimo) 5](#_Toc497699632)

[Convex Hull Trick (maximo) 6](#_Toc497699633)

[Dynamic Convex Hull Trick (minimo) 7](#_Toc497699634)

[Python Tricks 8](#_Toc497699635)

[Mínima cantidad de operaciones para hacer un arreglo estrictamente creciente 9](#_Toc497699636)

[Menor diferencia en un interval 9](#_Toc497699637)

[Dominator Tree 11](#_Toc497699638)

[Cycle-Finding. 13](#_Toc497699639)

[Hanoi Towers 13](#_Toc497699640)

[Necklase and Bracelet 13](#_Toc497699641)

[Inverso Modular 14](#_Toc497699642)

[Graph concepts and theorems 14](#_Toc497699643)

# 2-SAT Problem

const int MAXN=4e5+5;

int n,k;

vector<int>g[MAXN],gi[MAXN];

vector<bool> band;

vector<int> comp;

int TS[MAXN];

int NOT(int x)

{

if(x>n)return x-n;

return n+x;

}

void dfs1 (int nod)

{

band[nod] = true;

int t=g[nod].size();

for (int i=0;i<t;++i)

{

int nn = g[nod][i];

if(!band[nn])

dfs1(nn);

}

TS[k--]=nod;

}

void dfs2 (int nod, int c)

{

comp[nod] = c;

int t=gi[nod].size();

for (int i=0; i<t; ++i)

{

int nn = gi[nod][i];

if (comp[nn] == -1)

dfs2 (nn, c);

}

}

void add\_edge(int a,int b)

{

g[NOT(a)].push\_back(b);

g[NOT(b)].push\_back(a);

gi[b].push\_back(NOT(a));

gi[a].push\_back(NOT(b));

}

int main()

{

int m;

scanf("%d%d",&n,&m);

int a,b;char ca,cb;

for(int i=1;i<=m;i++)

{

scanf("%d %c %d %c",&a,&ca,&b,&cb);

if(ca=='N')a=NOT(a);

if(cb=='N')b=NOT(b);

add\_edge(a,b);

}

k=2\*n;

band.assign (2\*n+1, false);

for (int i=1; i<=2\*n; ++i)

if (!band[i])

dfs1 (i);

comp.assign (2\*n+1, -1);

for (int i=1, j=1; i<=2\*n; ++i)

{

int nod = TS[i];

if (comp[nod] == -1)

dfs2 (nod, j++);

}

for (int i=1; i<=n; ++i)

if(comp[i]==comp[NOT(i)])

{

printf("IMPOSSIBLE");

return 0;

}

for (int i=1; i<=n; ++i)

{

if(comp[i]>comp[NOT(i)])

printf("Y");

else

printf("N");

}

}

# Suffix Array NlogN with LCP

Example problem: N cadenas, hallar para cada cadena el menor substring que no aparece en ninguna otra cadena.

#include <bits/stdc++.h>

using namespace std;

#define ll long long

#define MAX 1000005

char s[MAX];

int SA[MAX],wa[MAX], wb[MAX], we[MAX], wv[MAX],S[MAX],A[MAX];

int LLCP[MAX],ID[MAX];

void Sufix\_Array(char \*cad,int \*SA,int N){

N++;

int i, j, p, \*x = wa, \*y = wb, range = 256;

memset(we, 0, range \* sizeof(int));

for (i = 0; i < N; i++)

we[ x[i] = cad[i] ]++;

for (i = 1; i < range; i++) we[i] += we[ i-1 ];

for (i = N - 1; i >= 0; i--)

SA[ --we[ x[i] ] ] = i;

for (j = p = 1; p < N; j <<= 1, range = p){

for (p = 0, i = N - j; i < N; y[p++] = i , i++) ;

for (i = 0; i < N; i++)

if (SA[i] >= j) y[p++] = SA[i] - j;

for (i = 0; i < N; i++)

wv[i] = x[ y[i] ];

memset(we, 0, range \* sizeof(int));

for (i = 0; i< N; i++)

we[ wv[i] ]++;

for (i = 1; i < range; i++) we[i] += we[i-1];

for (i = N-1; i >= 0; i--) SA[--we[wv[i]]] = y[i];

swap(x, y); x[SA[0]] = 0;

for (p = i = 1; i < N; i++)

if(y[SA[i]] == y[SA[i-1]] && y[SA[i]+j] == y[SA[i-1]+j])

x[SA[i]] = p - 1; else x[SA[i]] = p++;

}

N--;

}

int rank[MAX], LCP [MAX];

void FindLCP(char \*cad, int \*SA, int N){

int i, j, k;

for (i = 1; i <= N; i++)

rank[ SA[i] ] = i;

for (k = i = 0; i < N; LCP [rank[i++]] = k)

for (k ? k-- : 0,j = SA[rank[i]-1]; cad[i + k] == cad[j + k];

k++);

}

typedef pair<int,int>par;

stack<par>st;

char cad[MAX];

char aux[MAX];

int IZ[MAX],DE[MAX];

int sol[MAX];

int CadSol[MAX];

int n;

int M[MAX][20];

int lcp(int a,int b)

{

int ret=1<<30;

a++;

if(a>b)swap(a,b);

int lg=(int)log2(b-a+1);

ret=min(LCP[M[a][lg]],LCP[M[b-(1<<lg)+1][lg]]);

return ret;

}

int main() {

cin.tie(0);

ios\_base::sync\_with\_stdio(0);

int T;

cin >> T;

int laux=0;

for(int i=1;i<=T;i++)

{

sol[i]=1<<30;

n=strlen(cad);

cin >> aux; strcat(aux,"#");

laux=strlen(aux);

for(int j=n;j<n+laux;j++)

cad[j]=aux[j-n],

ID[j]=i,LLCP[j]=n+laux-j-1; // Guardo la cadena a la que pertenece cada caracter

// y calculo el limite del LCP para cada sufijo

}

n = strlen(cad);

Sufix\_Array(cad, SA, n);

FindLCP(cad, SA, n);

// RMQ para calcular LCP en intervalos

for(int i=1;i<=n;i++)

M[i][0]=i;

for(int i=1;(1<<i)-1<=n;i++)

for(int j=1;j+(1<<i)-1<=n;j++)

if(LCP[M[j][i-1]]<LCP[M[j+(1<<(i-1))][i-1]])

M[j][i]=M[j][i-1];

else

M[j][i]=M[j+(1<<(i-1))][i-1];

for(int i=1;i<=n;i++)

{

LCP[i]=min(LCP[i],LLCP[SA[i]]);

}

//Para cada sufijo busco el primero a la izquierda que no pertenece

//a la misma cadena

st.push(par(ID[SA[1]],1));

IZ[1]=-1;

for(int i=2;i<=n;i++)

{

while(!st.empty() && st.top().first==ID[SA[i]])

st.pop();

if(!st.empty())

IZ[i]=st.top().second;

else

IZ[i]=-1;

st.push(par(ID[SA[i]],i));

}

while(!st.empty())st.pop();

//Para cada sufijo busco el primero a la derecha que no pertenece

//a la misma cadena

st.push(par(ID[SA[n]],n));

DE[n]=-1;

for(int i=n-1;i>=1;i--)

{

while(!st.empty() && st.top().first==ID[SA[i]])

st.pop();

if(!st.empty())

DE[i]=st.top().second;

else

DE[i]=-1;

st.push(par(ID[SA[i]],i));

}

//Calculo para cada cadena el menor substring que no aparece

//en ninguna otra cadena

for(int i=1;i<=n;i++)

{

if(cad[SA[i]]=='#')continue;

if(DE[i]!=-1 && IZ[i]!=-1)

{

if(max(lcp(IZ[i],i),lcp(i,DE[i]))<LLCP[SA[i]] && sol[ID[SA[i]]]>max(lcp(IZ[i],i),lcp(i,DE[i]))+1)

sol[ID[SA[i]]]=max(lcp(IZ[i],i),lcp(i,DE[i]))+1,CadSol[ID[SA[i]]]=SA[i];

}

else

if(DE[i]!=-1){

if(lcp(i,DE[i])<LLCP[SA[i]] && sol[ID[SA[i]]]>lcp(i,DE[i])+1)

sol[ID[SA[i]]]=lcp(i,DE[i])+1,CadSol[ID[SA[i]]]=SA[i];

}

else

if(lcp(IZ[i],i)<LLCP[SA[i]] && sol[ID[SA[i]]]>lcp(IZ[i],i)+1)

sol[ID[SA[i]]]=lcp(IZ[i],i)+1,CadSol[ID[SA[i]]]=SA[i];

}

for(int i=1;i<=T;i++)

if(sol[i]==1<<30)

cout << "IMPOSSIBLE" << '\n';

else

{

for(int j=CadSol[i];j<CadSol[i]+sol[i];j++)

cout << cad[j];

cout << '\n';

}

return 0;

}

# Convex Hull Trick (minimo)

#include<bits/stdc++.h>

using namespace std;

int pointer;

vector<long long> M;

vector<long long> B;

bool bad(int l1,int l2,int l3)

{

return (B[l3]-B[l1])\*(M[l1]-M[l2])<(B[l2]-B[l1])\*(M[l1]-M[l3]);

}

void add(long long m,long long b)

{

M.push\_back(m);

B.push\_back(b);

while (M.size()>=3&&bad(M.size()-3,M.size()-2,M.size()-1))

{

M.erase(M.end()-2);

B.erase(B.end()-2);

}

}

long long query(long long x)

{

if (pointer>M.size())

pointer=M.size()-1;

while (pointer<M.size()-1&&

M[pointer+1]\*x+B[pointer+1]<M[pointer]\*x+B[pointer])

pointer++;

return M[pointer]\*x+B[pointer];

}

long long intersect(int i) {

return (B[i+1]-B[i])/(M[i]-M[i+1]);

}

long long query\_BS(long long x)

{

int lo = 0, hi = M.size() - 2, res = -1;

while (lo <= hi) {

int mid = lo + (hi - lo) / 2;

if (intersect(mid) <= x) res = mid, lo = mid + 1;

else hi = mid - 1;

}

return M[res+1]\*x + B[res+1];

}

int main()

{

int M,N,i;

pair<int,int> a[50000];

pair<int,int> rect[50000];

scanf("%d",&M);

for (i=0; i<M; i++)

scanf("%d %d",&a[i].first,&a[i].second);

sort(a,a+M);

for (i=0,N=0; i<M; i++)

{

while (N>0&&rect[N-1].second<=a[i].second)

N--;

rect[N++]=a[i];

}

long long cost;

add(rect[0].second,0);

pointer=0;

for (i=0; i<N; i++)

{

cost=query\_BS(rect[i].first);

//cost=query(rect[i].first);

if (i<N)

add(rect[i+1].second,cost);

}

printf("%lld\n",cost);

return 0;

}

# Convex Hull Trick (maximo)

#include<bits/stdc++.h>

using namespace std;

long long A[200005];

long long sum[200005];

// Convex Hull Trick

//Convex Hull Optimization1 dp[i]=min j<i {dp[j] + b[j] \* a[i] } b[j]>=b[j+1], a[i]<=a[i+1](opcional) O(n^2)->O(n)

//Convex Hull Optimization2 dp[i][j]=min k<j {dp[i-1][k] + b[k] \* a[j] } b[k]>=b[k+1], a[j]<=a[j+1](opcional) O(kn^2)->O(kn)

struct convex\_hull\_trick {

vector< pair<double,double> > h;

double intersect(int i) {

return (h[i+1].second-h[i].second)/(h[i].first-h[i+1].first);

}

void add(double m, double b) {

h.push\_back(make\_pair(m,b));

while (h.size() >= 3) {

int n = h.size();

if (intersect(n-3) < intersect(n-2)) break;

swap(h[n-2], h[n-1]);

h.pop\_back();

}

}

double get\_max(double x) {

int lo = 0, hi = h.size() - 2, res = -1;

while (lo <= hi) {

int mid = lo + (hi - lo) / 2;

if (intersect(mid) <= x) res = mid, lo = mid + 1;

else hi = mid - 1;

}

return h[res+1].first\*x + h[res+1].second;

}

void clear()

{

h.clear();

}

}CHT;

int main()

{

int N;

cin >> N;

long long s=0;

for(int i=1;i<=N;i++)

cin >> A[i],sum[i]=A[i]+sum[i-1],s+=1ll\*i\*A[i];

CHT.add(1,0);

long long sol=0;

for(int i=2;i<=N;i++)

{

sol=max(sol,sum[i-1]-1ll\*A[i]\*i+(long long)CHT.get\_max(A[i]));

CHT.add(i,-sum[i-1]);

}

CHT.clear();

CHT.add(-N,-sum[N]);

for(int i=N-1;i>=1;i--)

{

sol=max(sol,-1ll\*A[i]\*i+sum[i]+(long long)CHT.get\_max(-A[i]));

CHT.add(-i,-sum[i]);

}

cout << s+sol << '\n';

return 0;

}

# Dynamic Convex Hull Trick (minimo)

#include<bits/stdc++.h>

using namespace std;

#define MAXH 600006

typedef long long ll;

const int MAXN=3e5+5;

ll P[MAXN],H[MAXN],A[MAXN],dp[MAXN];

const ll INF=1ll<<60;

struct Line{

long long a, b;

Line(long long a = 0, long long b = INF): a(a), b(b) {}//-INF para maximo

long long getval(long long x) {

return a\*x + b;

}

};

struct Node{

int l, r;

Line line;

Node \*left, \*right;

Node(int l, int r): l(l), r(r), line(), left(NULL), right(NULL) {}

void Push() {

if (l < r && !left) {

left = new Node(l, (l + r)/2);

right = new Node((l + r)/2 + 1, r);

}

}

void update(Line L) {

Push();

Line A = line, B = L;

if (A.getval(l) < B.getval(l)) swap(A,B); //> para maximo

if (A.getval(r) >= B.getval(r)) line = B; //<= para maximo

else if (l != r) {

int mid = (l + r)/2;

if (A.getval(mid) < B.getval(mid)) {// > para maximo

line = A;

left -> update(B);

} else {

line = B;

right -> update(A);

}

}

}

long long get(int x) {

long long ans = line.getval(x);

int mid = (l + r)/2;

if (x <= mid && left) ans = min(ans, left -> get(x));

if (x > mid && right) ans = min(ans, right -> get(x));

return ans;

}

}\*ST[4\*MAXN];

void update(int nod,int I,int F,ll m,ll n,int pos)

{

if(!ST[nod]) ST[nod] = new Node(1, MAXH);//1-MAXH rango de las querys

if(F<pos || I>pos)return;

if(I==F)

{

ST[nod]->update(Line(m,n));

return;

}

int piv=(I+F)/2;

update(2\*nod,I,piv,m,n,pos);

update(2\*nod+1,piv+1,F,m,n,pos);

ST[nod]->update(Line(m,n));

}

ll query(int nod,int I,int F,int a,int b, ll x)

{

if(!ST[nod]) ST[nod] = new Node(1, MAXH);

if(I>b || F<a)return 1ll<<60;

if(F<=b && I>=a)return ST[nod]->get(x);

int piv=(I+F)/2;

ll v1=query(2\*nod,I,piv,a,b,x);

ll v2=query(2\*nod+1,piv+1,F,a,b,x);

return min(v1,v2);

}

int main()

{

int N;

cin >> N;

for(int i=1;i<=N;i++)

cin >> P[i];

for(int i=1;i<=N;i++)

cin >> A[i];

for(int i=1;i<=N;i++)

cin >> H[i];

dp[1]=A[1];

update(1,1,N,-2ll\*H[1],dp[1]+H[1]\*H[1],P[1]);

for(int i=2;i<=N;i++)

{

dp[i]=query(1,1,N,1,P[i],H[i])+H[i]\*H[i]+A[i];

update(1,1,N,-2ll\*H[i],dp[i]+H[i]\*H[i],P[i]);

}

cout << dp[N] << '\n';

return 0;

}

# Python Tricks

0- implemetar el codigo asi

def main():

#implementar la solucion

main()

1-//para leer

import sys

a = sys.stdin.readline().slpit(' ')

sys.stdout.write("Hello World\n")

2-arrays

v = [0] \* 10 //[0, 0, 0, 0, 0, 0, 0, 0, 0, 0] para crear un arreglo con 10 elementos inicializados en 0

v = [[0]\*N]\*N matriz de N\*N -> (0,N-1)

3- v.append(n) -> equivalente de push\_back(n)

4- for i in xrange(1,n) -> for( int i = 1; i < n; i++ )

5- from itertools import permutations ->importa la fincion permutations

//para usarla

for p in permutations([1,2,3,4])

print(p)//imprime todas las permutaciones de la lista pasada como param como listas

6- from itertools import combinations ->importa la funcion combinations

v = [1,2,3,4,5]

for c in combinations(v, 2):

print(" ".join(c))//imprime las combinaciones de v en 2

7- a,b = b,a -> equivale swap(a,b)

8- remover duplicados de una lista

a = [1,2,3,4,1,2,3,4]

a = list(set(a))

9- concaternar cadenas

a = ["a", "b", "c"]

concat = ''.join(a);

10- cola

import collections

help(collections.deque)

# Mínima cantidad de operaciones para hacer un arreglo estrictamente creciente

#include<stdio.h>

#include<queue>

int main()

{

int n, t;

long long ans = 0;

std::priority\_queue<int> Q;

scanf("%d%d", &n, &t);

Q.push(t);

for(int i=1; i<n; i++)

{

scanf("%d", &t); t-=i;

Q.push(t);

if(Q.top() > t)

{

ans += Q.top() - t;

Q.pop();

Q.push(t);

}

}

printf("%lld", ans);

return 0;

}

# Menor diferencia en un interval

#include<bits/stdc++.h>

using namespace std;

typedef pair<int,int>par;

typedef pair<par,int>trio;

bool com(const trio &s,const trio &p)

{

return s.first.second<p.first.second;

}

const int MAXN=1e5+5;

const int MAXQ=3e5+5;

struct Stree

{

multiset<int>MS;

int md;

Stree(int d=0)

{

MS.clear();

md=d;

}

}ST[4\*MAXN];

int A[MAXN];

int minval;

int sol[MAXQ];

trio q[MAXQ];

void build(int nod,int I,int F)

{

if(I==F)

ST[nod].MS.insert(A[I]),

ST[nod].md=1<<30;

else

{

int piv=(I+F)/2;

build(2\*nod,I,piv);

build(2\*nod+1,piv+1,F);

for(int i=I;i<=F;i++)

ST[nod].MS.insert(A[i]);

ST[nod].md=1<<30;

}

}

int mindif(int nod,int val)

{

multiset<int>::iterator it=ST[nod].MS.lower\_bound(val);

int d1=1<<30,d2=1<<30;

if(it!=ST[nod].MS.end())d1=abs(\*it-val);

if(it!=ST[nod].MS.begin())

{

it--;

d2=abs(\*it-val);

}

return min(d1,d2);

}

void update(int nod,int I,int F,int a,int b,int val)

{

if(I>b || F<a)return;

if(mindif(nod,val)>=minval)

return;

if(I==F)

{

ST[nod].md=min(ST[nod].md,abs(\*(ST[nod].MS.begin())-val));

minval=min(minval,ST[nod].md);

return;

}

int piv=(I+F)/2;

update(2\*nod+1,piv+1,F,a,b,val);

update(2\*nod,I,piv,a,b,val);

ST[nod].md=min(ST[2\*nod].md,ST[2\*nod+1].md);

minval=min(minval,ST[nod].md);

}

int query(int nod,int I,int F,int a,int b)

{

if(I>b || F<a)return 1<<30;

if(I>=a && F<=b)

return ST[nod].md;

int piv=(I+F)/2;

int d1=query(2\*nod,I,piv,a,b);

int d2=query(2\*nod+1,piv+1,F,a,b);

return min(d1,d2);

}

int main()

{

int N;

cin >> N;

for(int i=1;i<=N;i++)

cin >> A[i];

int Q;

cin >> Q;

for(int i=1;i<=Q;i++)

{

int a,b;

cin >> a >> b;if(a>b)swap(a,b);

q[i]=trio(par(a,b),i);

}

sort(q+1,q+Q+1,com);

build(1,1,N);

int ultimo\_actualizado=0;

for(int i=1;i<=Q;i++)

{

for(int j=ultimo\_actualizado+1;j<=q[i].first.second;j++)

minval=1<<30,update(1,1,N,1,j-1,A[j]);

ultimo\_actualizado=q[i].first.second;

sol[q[i].second]=query(1,1,N,q[i].first.first,q[i].first.second);

}

for(int i=1;i<=Q;i++)

cout << sol[i] << '\n';

return 0;

}

# Dominator Tree

Dominator Tree (Lengauer-Tarjan)

Complexity: O(m log n)

struct graph

{

int n;

vector<vector<int>> adj, radj;

graph(int n) : n(n), adj(n), radj(n) {}

void add\_edge(int src, int dst)

{

adj[src].push\_back(dst);

radj[dst].push\_back(src);

}

vector<int> rank, semi, low, anc;

int eval(int v)

{

if (anc[v] < n && anc[anc[v]] < n)

{

int x = eval(anc[v]);

if (rank[semi[low[v]]] > rank[semi[x]])

low[v] = x;

anc[v] = anc[anc[v]];

}

return low[v];

}

vector<int> prev, ord;

void dfs(int u)

{

rank[u] = ord.size();

ord.push\_back(u);

for (auto v : adj[u])

{

if (rank[v] < n)

continue;

dfs(v);

prev[v] = u;

}

}

vector<int> idom; // idom[u] is an immediate dominator of u

void dominator\_tree(int r)

{

idom.assign(n, n);

prev = rank = anc = idom;

semi.resize(n);

iota(semi.begin(), semi.end(), 0);

low = semi;

ord.clear();

dfs(r);

vector<vector<int>> dom(n);

for (int i = (int) ord.size() - 1; i >= 1; --i)

{

int w = ord[i];

for (auto v : radj[w])

{

int u = eval(v);

if (rank[semi[w]] > rank[semi[u]])

semi[w] = semi[u];

}

dom[semi[w]].push\_back(w);

anc[w] = prev[w];

for (int v : dom[prev[w]])

{

int u = eval(v);

idom[v] = (rank[prev[w]] > rank[semi[u]]

? u : prev[w]);

}

dom[prev[w]].clear();

}

for (int i = 1; i < (int) ord.size(); ++i)

{

int w = ord[i];

if (idom[w] != semi[w])

idom[w] = idom[idom[w]];

}

}

vector<int> dominators(int u)

{

vector<int> S;

for (; u < n; u = idom[u])

S.push\_back(u);

return S;

}

};

# Cycle-Finding.

An implementation of Floyd’s Cycle-Finding algorithm.

par find\_cycle() {

int t = f(x0), h = f(t), mu = 0, lam = 1;

while (t != h) t = f(t), h = f(f(h));

h = x0;

while (t != h) t = f(t), h = f(h), mu++;

h = f(t);

while (t != h) h = f(h), lam++;

return par(mu, lam);

}

# Hanoi Towers

//Towers of Hanoi

void move( int n, char from, char to, char aux ) {

if ( n == 1 )

printf( "Move disk from %c to %c\n", from, to );

else {

move( n - 1, from, aux, to );

printf( "Move disk from %c to %c\n", from, to );

move( n - 1, aux, to, from );

}

}

int main() {

scanf( "%d", &n );

move( n, 'A', 'C', 'B' );

getch();

return 0;

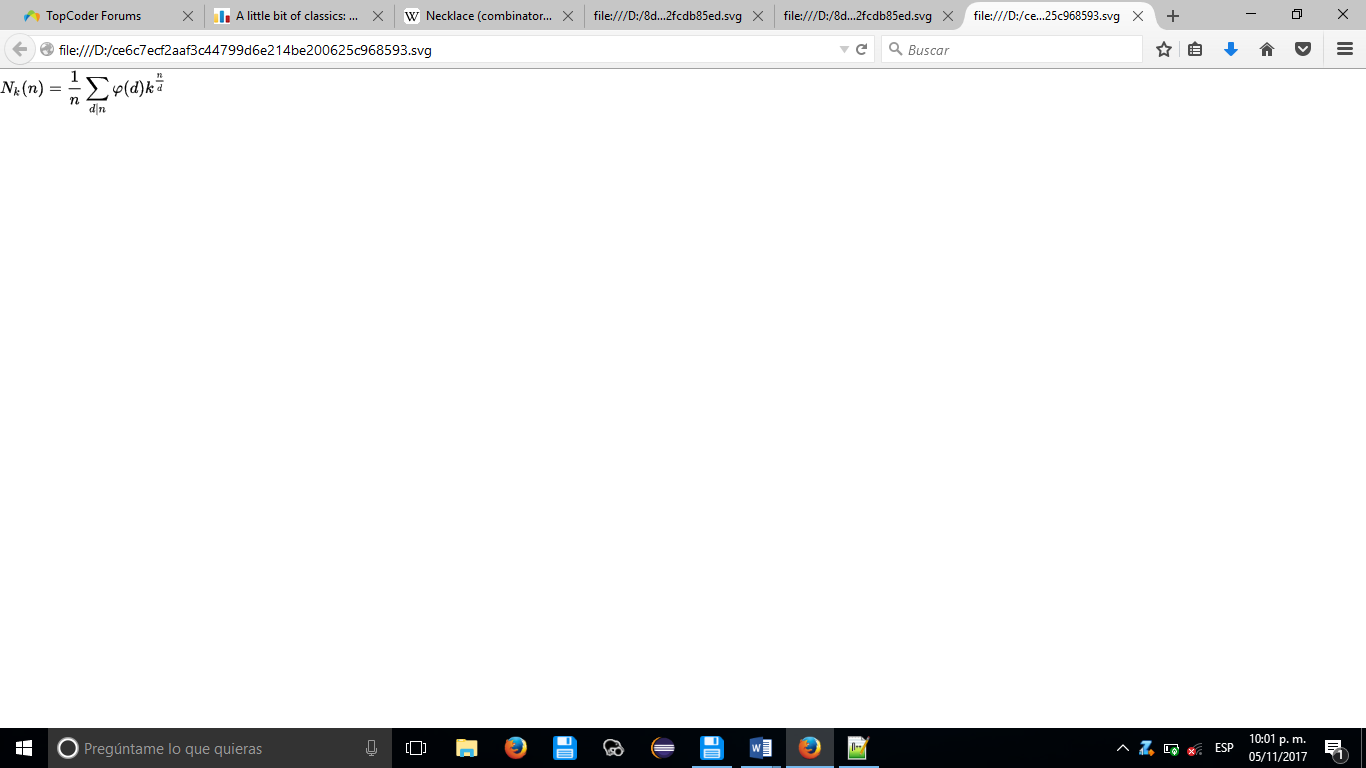
}

# Necklase and Bracelet

In [combinatorics](https://en.wikipedia.org/wiki/Combinatorics), a *k*-ary **necklace** of length *n* is an [equivalence class](https://en.wikipedia.org/wiki/Equivalence_class) of *n*-character [strings](https://en.wikipedia.org/wiki/String_%28computer_science%29#Formal_theory) over an [alphabet](https://en.wikipedia.org/wiki/Alphabet_%28computer_science%29) of size *k*, taking all [rotations](https://en.wikipedia.org/wiki/Circular_shift) as equivalent. It represents a structure with *n* circularly connected beads of up to *k* different colors.

A *k*-ary **bracelet**, also referred to as a **turnover** (or **free**) **necklace**, is a necklace such that strings may also be equivalent under reflection. That is, given two strings, if each is the reverse of the other then they belong to the same equivalence class. For this reason, a necklace might also be called a **fixed necklace** to distinguish it from a turnover necklace.

**Number of necklaces**

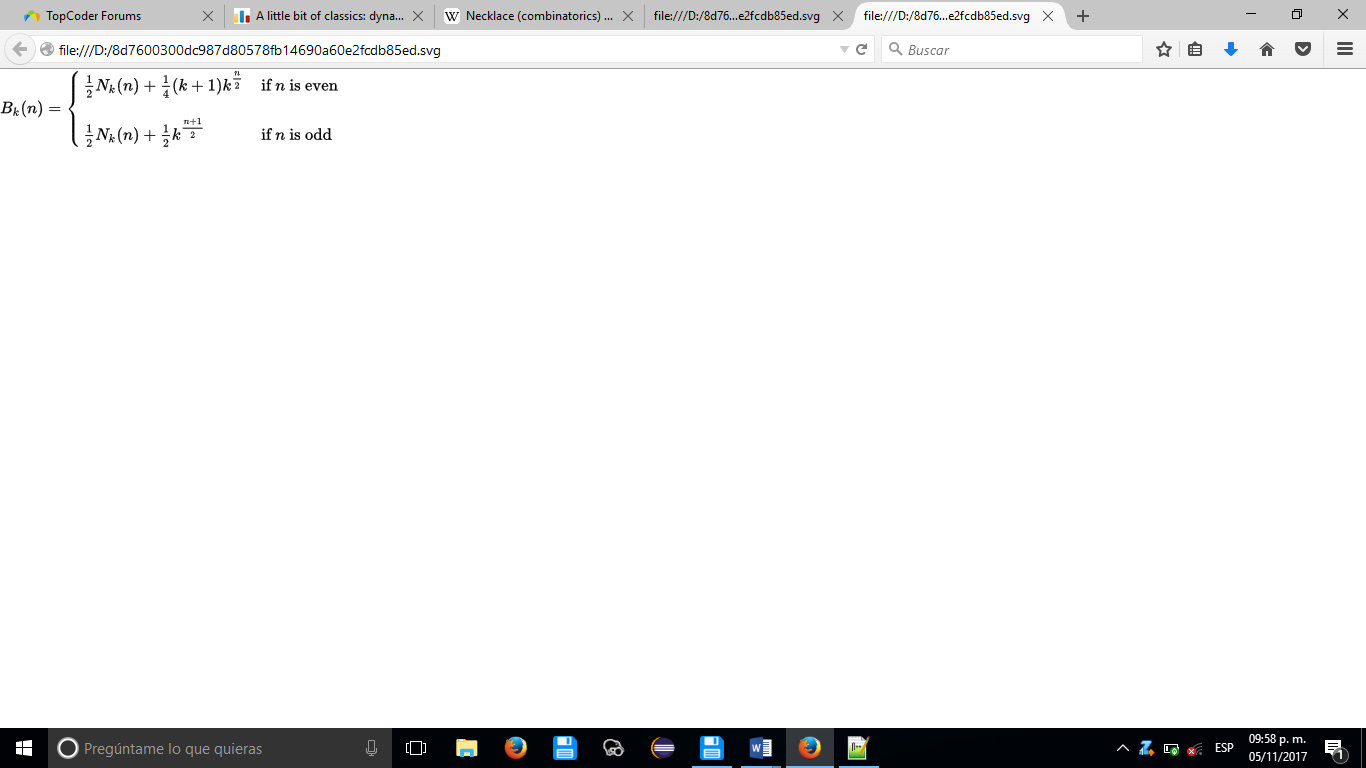
There are

N k ( n ) = 1 n ∑ d ∣ n φ ( d ) k n d {\displaystyle N\_{k}(n)={\frac {1}{n}}\sum \_{d\mid n}\varphi (d)k^{\frac {n}{d}}}

different *k*-ary necklaces of length *n*, where *φ* is [Euler's totient function](https://en.wikipedia.org/wiki/Euler%27s_totient_function).

**Number of bracelets**

There are

B k ( n ) = { 1 2 N k ( n ) + 1 4 ( k + 1 ) k n 2 if  n  is even 1 2 N k ( n ) + 1 2 k n + 1 2 if  n  is odd {\displaystyle B\_{k}(n)={\begin{cases}{\tfrac {1}{2}}N\_{k}(n)+{\tfrac {1}{4}}(k+1)k^{\frac {n}{2}}&{\text{if }}n{\text{ is even}}\\[10px]{\tfrac {1}{2}}N\_{k}(n)+{\tfrac {1}{2}}k^{\frac {n+1}{2}}&{\text{if }}n{\text{ is odd}}\end{cases}}}

different *k*-ary bracelets of length *n*, where *Nk*(*n*) is the number of *k*-ary necklaces of length *n*.

# Inverso Modular

inline int inv(int a) {

a %= mod;

if (a < 0) a += mod;

int b = mod, u = 0, v = 1;

while (a) {

int t = b / a;

b -= t \* a; swap(a, b);

u -= t \* v; swap(u, v);

}

assert(b == 1);

if (u < 0) u += mod;

return u;

}

# Graph concepts and theorems

**Chromatic Number:** Minimum number of colors required to color Graph *G* such that no two adjacent vertex gets same color.

*Brooks' Therorem:* For any connected undirected graph *G* with maximum degree Δ, the chromatic number of *G* is at most Δ unless *G* is a complete graph or an odd cycle, in which case the chromatic number is Δ + 1.

**Independent Vertex Set:** Set of vertices in a graph *G* ,no two of which are adjacent that means no two vertices in this set is connected by an edge. (no two of which are adjacent).

α(*G*) *Vertex Independence Number* is the cardinality of Largest Independence set.

**Vertex Cover:** Set of Vertices in a Graph *G*, such that each Edge in *G* incident on atleast one Vertex in the Set.

*Vertex C*



*overing Number* is the cardinality of minimum vertex cover.

* A set of vertices is a vertex cover if and only if its complement is an independent set.
* The number of vertices of Graph *G* is equal to sum of minimum vertex covering number and maximum vertex independence number.

C:\Users\Yuri\AppData\Local\Temp\maftemp-76ee4bd8\1509726050724_892\1509622526101_129\index_files\05fd05a2738130589a6d25dae6858b2fecad4420.png

**Edge Cover:** Set of Edges in a Graph *G*, such that every vertex of *G* is incident on atleast one of the edges in the set.

ρ(*G*) *Edge Covering number* is the cardinality of minimum edge cover.

**Dominating Set:** Set of Vertices ,such that every vertex not in Set is adjacent to at least one member of Set OR Every vertex of *G* is adjacent to some Vertex of the Set.

γ(*G*) *Dominance number* is the number of Vertices in the smallest Dominating Set.

* Independent Set is Maximal Independent iff it is a Dominating Set.

OR

* Independent Set is Dominating Set iff it is Maximal Independent.

Thus,

* Any maximal independent set in a graph is necessarily also a minimal dominating set.
* Dominance Number is always less than or equal to Independence Number.

**Matching:** Set of edges of *G* ,such that no two of which are adjacent (pairwise non-adjacent edges) or no two edges share a common vertex.

ν(*G*) *Matching number* (edge independence number) is the size of maximum matching.

Matching -> non-adjacent edges

Vertex independent set -> non-adjacent vertex.

* In any graph without isolated vertices, the sum of the matching number and the edge covering number equals the number of vertices.

ν(*G*) + ρ(*G*) = *n*

If Graph *G* = (*V*, *E*) is Bipartite then ,

1. *Vertex Independence number* is equal to *Edge covering number*.
2. *Matching number* is equal to *vertex covering number*. **(König's theorem)**

**Clique :**

A clique of a graph *G* is a complete subgraph of *G*, and the clique of largest possible size is referred to as a *maximum clique*.

ω(*G*) *Clique Number* is the number of vertices in maximum clique in *G*.

* χ(*G*) ≥ ω(*G*) *Chromatic Number* of *G* is always greater than or equal to *Clique number*.

A Maximal Independent vertex set of a graph *G* is equivalent to a Maximal Clique on the graph complement C:\Users\Yuri\AppData\Local\Temp\maftemp-76ee4bd8\1509726050724_892\1509622526101_129\index_files\c3ddb7fc9c27037f44efd679442efb5fc092c1d9.png.