```
ETF_CUMULATIVE:
        for(int i = 2;i < MAXN;i++) p[i] = i ;
        for(int i = 2;i < MAXN;i++) if(p[i] == i)
        for(int j = i;j < MAXN;j += i)
              p[j] = p[j] / i * (i - 1) ;
        for(int i = 2;i < MAXN;i++) p[i]=p[i]*p[i];
        for(int i = 2;i < MAXN;i++) p[i] += p[i - 1] ;
ETF_BRUTE:
       int fi(int n){

              int result = n;
              for(ull i=2;i*i<= n;i++){
                if (n % i == 0) result -= result / i;
                while (n % i == 0) n /= i;
              }
              if (n > 1) result -= result / n;
              return result;
       }
SEIVE:
/      ------------------ seive
const LL MAXN=1e6;
bool primes[MAXN+10];
int prime[MAXN+10];
int pindex=-1;
void seive_init(){
       #define FORLL(i,a,b,inc) for(LL i=a;i<=b;i+=inc)
       FORLL(i,0,MAXN,1)primes[i]=1;
    primes[1]=0,primes[0]=0;
    FORLL(i,2,MAXN,1)if(primes[i]==1)FORLL(j,i*i,MAXN,i)primes[j]=0;
    FORLL(i,2,MAXN,1)if(primes[i]!=0)prime[++pindex]=i;
}
Pair<Factors,power> :
vector<pair<LL,LL> > get_fact(LL x){
       vector<pair<LL,LL> >ret;
       int index=0;
       while(x>1){
              LL c=0;
              while(x%prime[index]==0){
                     c++;x/=prime[index];
              }if(c)
              ret.pb(mp((LL)prime[index],c));
              index++;
              if(index>pindex){
                     ret.pb(mp(x,1));
                     return ret;
              }
       }
       return ret;
}
INV_Factorials:
LL fact[1000000+10];
LL invfact[1000000+10];
LL MOD=1000003LL;
#define MAXFACT 1000000
LL  modpow(LL a,LL b,LL c){
    LL  x=1,y=a;
    while(b>0){
         if(b&1) x=(x*y)%c;
         y=(y*y)%c;
         b>>=1;
```

```
    }
    return (x);
}
void pre(){
       fact[0]=1;
       for(LL i=1;i<=MAXFACT;i++){
              fact[i]=(fact[i-1]*i)%MOD;
       }
       invfact[MAXFACT]=modpow(fact[MAXFACT],MOD-2,MOD);
       for(int i=MAXFACT-1;i>=0;i--){
              invfact[i]=(invfact[i+1]*(i+1))%MOD;
       }
}
LL C(LL n,LL k){
       if(n<0 || k<0 || k>n)return 0;
       LL r=fact[n];
       r=(r*invfact[k])%MOD;
       r=(r*invfact[n-k])%MOD;
       return r;
}
```

Sum of divisors:
S=(p^(a+1)-1)/(p-1)
Derangements: D(0)=1,D(1)=0; D[n]=(n-1)*(D[n-1]+D[n-2]);

DP BITMASK BASE TEMPLATE FOR TSP

```
int dp[1<<9];
            REP(j,(1<<9))dp[j]=1e8;
            int m=8;
            LL maxval=0;
            dp[0]=0;
            for(int mask=1;mask<(1<<m);mask++){
                maxval=1e8;
                for(int j=0;j<m;j++){
                    if((mask&(1<<j))){
                        int haa=(mask& ~(1<<j));
                        int co=__builtin_popcount(mask)-1;
                        ll val=T[j][co];
                        ll temp=dp[haa];
                        maxval=min(maxval,temp+val);
                    }
                }
                dp[mask]=maxval;
            }
            ans=min(ans,dp[(1<<m)-1]);
```

SEGMENT TREE

```
int tree[SIZE*2];
int ar[SIZE];
// query returns index of min_value
void initialize(int node,int left,int right){
     if(left==right){
            tree[node]=left;
     }
     else{
            initialize(2*node,left,(left+right)/2);
            initialize(2*node+1,((left+right)/2)+1,right);
            if(ar[tree[2*node]]<=ar[tree[2*node+1]])
                   tree[node]=tree[2*node];
            else
```

```
                    tree[node]=tree[2*node+1];

     }
}
int query(int node,int left,int right,int x,int y){
    int p1,p2;
    if(x>right || y <left) return -1;
    else if(left>=x&&right<=y) return tree[node];
    p1=query(2*node,left,(left+right)/2,x,y);
    p2=query(2*node+1,(left+right)/2+1,right,x,y);
    if(p1==-1)
           return tree[node]=p2;
    if(p2==-1)
           return tree[node]=p1;
    if(ar[p1]<=ar[p2])
          return tree[node]=p1;
    return tree[node]=p2;
}
void update(int node,int left,int right,int index,int val){
     if(left==right){
            tree[node]=left;
            ar[index]=val;
            }
     else{
         int mid=(left+right)/2;
         if(index<=mid)
            update(2*node,left,(left+right)/2,index,val);
         else
            update(2*node+1,(left+right)/2+1,right,index,val);
         if(ar[tree[2*node]]<=ar[tree[2*node+1]])
                tree[node]=tree[2*node];
         else tree[node]=tree[2*node+1];
     }
}
initialize(1,0,SIZE-1);
update(1,0,SIZE-1,x,y);
query(1,0,SIZE-1,x,y)

BIT:
void update(int idx,int val){
     while(idx<=maxval){
          tree[idx]+=val;
          idx+=(idx & -idx);
     }
}
int query(int idx){
    int sum=0;
    while(idx>0){
        sum+=tree[idx];
        idx-=(idx & -idx);
    }
    return sum;
}
int findG(int cumFre){// find kth smallest element
        int hi = maxval,lo = 1;
        while(lo < hi) {
                int mid = lo + (hi - lo) / 2;
                if(query(mid) < cumFre)
                        lo = mid + 1;
                else
                        hi = mid;
```

```
        }
        return lo;
}
int binaryse(int n,int maxindex){
      int low=0;
      int high=maxindex;
      int mid;
      while(low<=high){
            mid=low+(high-low)/2;
            if(allindexes[mid]<n)
                  low=mid+1;
            else if(allindexes[mid]>n)
                  high=mid-1;
            else{
                 return mid;
                 break;
            }
     }
     return mid;
}
```

Stirling Numbers:
2nd: is the number of ways to partition a set of n objects into k non-empty subsets
S[n+1,k]=k*S[n,k]+S[n,k-1]
S[0,0]=1,
s[n,0]=s[0,n]=0 for n>0

1st:The unsigned Stirling numbers of the first kind count the number of permutations of n elements with k disjoint cycles.
S[n+1,k]=n*S[n,k]+S[n,k-1]
S[0,0]=1,
s[n,0]=s[0,n]=0 for n>0
EXPO:

```
#define fill(ar,val) memset(ar,val,sizeof ar)
#define N 3
#define MOD 100000
int ni;
struct matr
{
       LL _[N][N];
};
int k;
matr one;
matr operator+(const matr &a,const matr &b)
{
       matr c;
       for(int i=0;i<ni;i++)
              for(int j=0;j<ni;j++)
                     c._[i][j]=(a._[i][j]+b._[i][j])%MOD;
       return c;
}

matr operator*(const matr &a,const matr &b)
{
       matr c;
       fill(c._,0);
       for(int i=0;i<ni;i++)
              for(int j=0;j<ni;j++)
                     for(int k=0;k<ni;k++)
                            c._[i][j]=(c._[i][j]+LL(a._[i][k])*b._[k][j])%MOD;
       return c;
```

```
}
matr power(matr a,int n)
{
	matr p=one;
	for(;n;)
	{
		if(n%2) p=p*a;
		if(n/=2) a=a*a;
	}
	return p;
}
matr summ(matr a,int n){
	if(n==1)
		return a;
	if(n&1)
		return a+(a*summ(a,n-1));
	else
		return (one+power(a,n/2))*summ(a,n/2);
}
```

TRIE

```
struct node{
	int number_of_words;
	int number_of_prefixes;
	struct node *child[12];
};
struct node *root=NULL;
struct node* initialise(struct node* vertex){
     vertex=(struct node*)malloc(sizeof(struct node));
     vertex->number_of_words=0;
     vertex->number_of_prefixes=0;
     for(int i=0;i<12;i++){
             vertex->child[i]=NULL;
     }
     return vertex;
}
int toint(char x){
    return x-'0';
}
char input_string[12];
int flag=0;
void addWords(struct node *vertex,int index){
     if(input_string[index]=='\0'&& vertex->number_of_prefixes==0){
                     // Check whether this s new word
          vertex->number_of_words++;
          vertex->number_of_prefixes++;
          return;
     }
     else if(input_string[index]=='\0'&& vertex->number_of_prefixes!=0){
                     // Check whether this word occured before
          flag=1;
          return;
     }
     else{
                     // Else Create the Child[new character];
               char x=input_string[index];
          if(vertex->child[toint(x)]==NULL){
              vertex->child[toint(x)]=initialise(vertex->child[toint(x)]);
          }
          vertex->number_of_prefixes+=1;
```

```
          addWords(vertex->child[toint(x)],index+1);
     }
}
```

Extended-gcd:
 if ax + by = gcd(a,b) = 1, i.e. a and be are co-primes, then x is the modular multiplicative inverse of a modulo b, and similarly, y is the modular multiplicative inverse of b modulo a.

```
/*
Takes a, b as input.
Returns gcd(a, b).
Updates x, y via pointer reference.
*/
int Extended_Euclid(int A, int B, int *X, int *Y)
{
    int x, y, u, v, m, n, a, b, q, r;

    /* B = A(0) + B(1) */
    x = 0; y = 1;

    /* A = A(1) + B(0) */
    u = 1; v = 0;

    for (a = A, b = B; 0 != a; b = a, a = r, x = u, y = v, u = m, v = n)
    {
        /* b = aq + r and 0 <= r < a */
        q = b / a;
        r = b % a;

        /* r = Ax + By - aq = Ax + By - (Au + Bv)q = A(x - uq) + B(y - vq) */
        m = x - (u * q);
        n = y - (v * q);
    }

    /* Ax + By = gcd(A, B) */
    *X = x; *Y = y;

    return b;
}
//    g = Extended_Euclid(a, b, &x, &y);
```

// SCC

```
vector<int>graph[30000+10];
vector<int>reverse_graph[30000+10];
int visited[30000+10];
int mute_node[30000+10];
stack<int>st;
int no_of_vertices=0;
vector<int>components[30000+10];
int no_of_components=0;
int mygroup_number[300000+10];
void dfs1(int vertex){
        visited[vertex]=1;
        for(int i=0;i<graph[vertex].size();i++){
                int u=graph[vertex][i];
                if(visited[u]==-1)
                        dfs1(u);
        }
```

```
        st.push(vertex);
}
void create_reverse_graph(){
        for(int i=0;i<no_of_vertices;i++){
                for(int j=0;j<graph[i].size();j++){
                        int u=i;
                        int v=graph[i][j];
                        reverse_graph[v].push_back(u);
                }
        }
}
void dfs2(int vertex){
        mute_node[vertex]=1;
        for(int i=0;i<reverse_graph[vertex].size();i++){
                int u=reverse_graph[vertex][i];
                if(mute_node[u]==-1){
                        dfs2(u);
                }
        }
        components[no_of_components].push_back(vertex);
}
set<pair<int,int> > s;
// DFS for the answer
// no of leaves in DAG= answer
bool incoming [30000+20];
int main(){
                s.clear();
                REP(i,30000+10){
                        graph[i].clear();
                        reverse_graph[i].clear();
                        mute_node[i]=0;
                        visited[i]=-1;
                        components[i].clear();
                        mygroup_number[i]=-1;
                        no_of_vertices=0;
                        no_of_components=0;
                        DAGS[i].clear();
                        graph2[i].clear();
                        IN[i]=-1;OUT[i]=-1;
                }
                int n,m;
                SS(n,m);
                no_of_vertices=n;
                int x,y;
                REP(i,m){
                        SS(x,y);x--;y--;
                        graph[x].pb(y);
                }
                for(int i=0;i<n;i++){
                        if(visited[i]==-1)
                                dfs1(i);
                }
                create_reverse_graph();
                memset(mute_node,-1,sizeof mute_node);
                while(!st.empty()){
                        int tp=st.top();
                        st.pop();
                        if(mute_node[tp]==-1){
                                dfs2(tp);
                                no_of_components++;
                        }
                }
                for(int i=0;i<no_of_components;i++){
                        for(int j=0;j<components[i].size();j++){
                                mygroup_number[components[i][j]]=i;
                        }
                }
                // Remove SCCs to simple DAG
                for(int i=0;i<n;i++){
```

```
                        for(int j=0;j<graph[i].size();j++){
                                int u=i;
                                int v=graph[i][j];
                                int u_num=mygroup_number[u];
                                int v_num=mygroup_number[v];
                                if(u_num!=v_num)
                                        s.insert(mp(u_num,v_num));
                        }
                }
                REP(i,30000+6)graph[i].clear();
                set<pair<int,int> >::iterator it;
                for(it=s.begin();it!=s.end();++it){
                        pair<int,int>p=*it;
                        graph[it->first].pb(it->second);
                }
}
Top-sort:
vector<int>top;
void dfs(int i){
        visited[i]=1;
        REP(g,graph[i].size()){
                if(visited[graph[i][g]]==-1){
                        dfs(graph[i][g]);
                }
        }
        top.pb(i);
}
vector<int> get_top(){
                set<int>s1;
                REP(i,no_of_components)if(incoming[i]==0)s1.insert(i);
                set<int>::iterator it15;
                memset(visited,-1,sizeof visited);
                for(it15=s1.begin();it15!=s1.end();it15++)
                        dfs(*it15);
                reverse(top.begin(),top.end());
                return top;
}
```

Dijkstra:

```
struct edge{
        int u,v,w;
        edge(int a,int b,int c){
                u=a,v=b,w=c;
        }
};
bool operator < ( edge e1, edge e2){
        return e1.w<e2.w;
}
vector<int>input;
int parent[200];
int cost[200];
vector<edge>graph[210];
int djikstra(int s,int t){
        priority_queue<edge>q;
        q.push(edge(0,0,0));
        REP(i,200)parent[i]=-1,cost[i]=1e8;
        cost[0]=0;
        while(q.size()>0){
                edge top=q.top();
                q.pop();
                int node=top.u;
                int C=top.w;
                if(node==t)continue;
                if(C<=cost[node]){
                        for(int i=0;i<graph[node].size();i++){
                                int to=graph[node][i].v;
                                int c=graph[node][i].w;
                                if(C+c<=cost[to]){
```

```
                                        cost[to]=C+c;
                                        q.push(edge(to,0,c+C));
                                }
                        }
                }
        }
        return cost[t];
}
```

Disjoint:

```
int fa[MAXN], rank[MAXN];
void init(){
        for(int i=0;i<MAXN;i++)fa[i]=i,rank[i]=0;
}
inline int Find(int u){
    if (fa[u] == u)
        return u;
    return fa[u] = Find(fa[u]);
}
inline void Union(int u, int v){
    if (rank[u] > rank[v])
        fa[v] = u;
    else
    {
        fa[u] = v;
        if (rank[u] == rank[v]) ++rank[v];
    }
}
```

MAX-FLOW:

```
int G[300][300];
int mark[300];
// int dfs(int total vertives,int from,int to,int curflow
int dfs(int n, int s, int t, int by){
        if (s == t)
            return by;
        mark[s] =1;
        for (int i = 0; i < n; ++i)
            if (G[s][i] > 0 &&!mark[i])
            {
                int b = dfs(n, i, t, min(by, G[s][i]));
                if (b > 0)
                {
                    G[s][i] -= b;
                    G[i][s] += b;
                    return b;
                }
            }
        return 0;
    }
//int flow (int total vertices, int from,int to);
int maxflow(int n, int s, int t)
    {
        int res = 0;
        while (true)
        {
            memset(mark,0,sizeof mark);
            int by = dfs(n, s, t,(int)100000000);
            res += by;
            if (by == 0)
                break;
        }
        return res;
    }
```

MATCHING:

```
vector<int>graph[MAXN];
```

```
int lefts[MAXN];
int rights[MAXN];
int visited[MAXN];
bool find_match(int where) {
     // the previous column was not matched
     if (where == -1)
       return true;
     for (int i = 0; i < graph[where].size(); ++ i) {
       int match = graph[where][i];
       if (visited[match] == false) {
         visited[match] = true;
         if (find_match(rights[match])) {
           rights[match] = where;
           return true;
         }
       }
     }
     return false;
   }
int solve_getMatch(int n){
        int ret=0;
        memset(rights,-1,sizeof rights);
        memset(lefts,-1, sizeof lefts);
        for(int i=0;i<n;i++){
                memset(visited,0,sizeof visited);
                ret+=find_match(i);
        }
        return ret;
}
```

Euler Conditions:

1. An undirected graph has an Eulerian cycle if and only if every vertex has even degree, and all of its vertices with nonzero degree belong to a single connected component.

2.An undirected graph can be decomposed into edge-disjoint cycles if and only if all of its vertices have even degree. So, a graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint cycles and its nonzero-degree vertices belong to a single connected component.

3.An undirected graph has an Eulerian trail if and only if at most two vertices have odd degree, and if all of its vertices with nonzero degree belong to a single connected component.

4.A directed graph has an Eulerian cycle if and only if every vertex has equal in degree and out degree, and all of its vertices with nonzero degree belong to a single strongly connected component. Equivalently, a directed graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint directed cycles and all of its vertices with nonzero degree belong to a single strongly connected component.

5.A directed graph has an Eulerian trail if and only if at most one vertex has (out-degree) − (in-degree) = 1, at most one vertex has (in-degree) − (out-degree) = 1, every other vertex has equal in-degree and out-degree, and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph

Construction of Eulerian Tour:

Given an undirected tree presented as a set of edges, the Euler tour representation (ETR) can be constructed in parallel as follows:
We construct a symmetric list of directed edges:
For each undirected edge {u,v} in the tree, insert (u,v) and (v,u) in the edge list.
Sort the edge list lexicographically. (Here we assume that the nodes of the tree are ordered, and that the root is the first element in this order.)
Construct adjacency lists for each node (called next) and a map from nodes to the first entries of the adjacency lists (called first):
For each edge (u,v) in the list, do in parallel:
If the previous edge (x,y) has $x \neq u$, i.e. starts from a different node, set first(u) = (u,v)
Else if x = u, i.e. starts from the same node, set next(x,y) = (u,v)
Construct an edge list (called succ) in Euler tour order by setting pointers succ(u,v) for all edges (u,v) in parallel according to the following rule:

```
succ(u,v)={next(v,u)      next(v,u)!=nil
          first(v)        otherwise    }
```

The resulting list succ will be circular.
The overall construction takes work W(n) = O(sort(n)) (the time it takes to sort n items in parallel) if the tree has n nodes, as in trees the number of edges is one less than the number of nodes.If the tree has a root,

we can split the circular list succ at that root. In that case, we can speak of advance and retreat edges: given a pair of nodes u,v, the first occurrence of either (u,v) or (v,u) in the ETR is called the advance edge, and the second occurrence is called the retreat edge. This appeals to the intuition that the first time such an edge is traversed the distance to the root is increased, while the second time the distance decreases.

Another algo:

Hierholzer's algorithm

Hierholzer's 1873 paper provides a different method for finding Euler cycles that is more efficient than Fleury's algorithm:

Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.

As long as there exists a vertex v that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from v, following unused edges until returning to v, and join the tour formed in this way to the previous tour.

By using a data structure such as a doubly linked list to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each, so the overall algorithm takes linear time.

Bellman-Ford:

```
procedure BellmanFord(list vertices, list edges, vertex source)
   // This implementation takes in a graph, represented as lists of vertices
   // and edges, and modifies the vertices so that their distance and
   // predecessor attributes store the shortest paths.

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then v.distance := 0
       else v.distance := infinity
       v.predecessor := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge uv in edges: // uv is the edge from u to v
           u := uv.source
           v := uv.destination
           if u.distance + uv.weight < v.distance:
               v.distance := u.distance + uv.weight
               v.predecessor := u

   // Step 3: check for negative-weight cycles
   for each edge uv in edges:
       u := uv.source
       v := uv.destination
       if u.distance + uv.weight < v.distance:
           error "Graph contains a negative-weight cycle"
```

Hungarian:

```
#define INF              (int)1e9
int maxNode=60;
int cost[60][60],X[60],Y[60];
void hungarian(int nx, int ny)
{
  int Lx[maxNode], Ly[maxNode], Q[maxNode], prev[maxNode];

  memset(X, -1, sizeof X); memset(Y, -1, sizeof Y);
  memset(Lx, 0, sizeof Lx); memset(Ly, 0, sizeof Ly);

  for(int i = 0; i < nx; i++)
    for(int j = 0; j < ny; j++)
      Lx[i] = max( Lx[i], cost[i][j]);

  for(int i = 0;i < nx; )
  {
    //cout<<i<<endl;
```

```
    memset( prev, -1, sizeof prev);
    int head = 0, tail = 0;
    for(Q[tail++] = i; head < tail && X[i] < 0; head ++)
    {
      int u = Q[head];
      for(int v = 0; v < ny && X[i] < 0; v++)
      {
        if( prev[v] >= 0 || Lx[u] + Ly[v] > cost[u][v] ) continue;
        if( Y[v] >=  0)
          prev[v] = u,  Q[tail++] = Y[v];
        else
        {
          prev[v] = u;
          for(int at = v; at >=0; )
          {
            u = Y[at] = prev[at];
            //swap X[u] , at
            int a = X[u];
            X[u] = at;
            at = a;
          }
        }
      }

    }

    if( X[i] >=0) i ++;
    else
    {
      int alpha = INF;
      for(int head = 0; head < tail; head++)
      {
        int u = Q[head];
        for(int v = 0; v < ny; v++)
          if( prev[v] == -1)
            alpha = min ( alpha, Lx[u] + Ly[v] - cost[u][v] );
      }

      for(int head = 0; head < tail; head ++)
        Lx[Q[head]] -= alpha;

      for(int v = 0; v < ny; v++)
        if( prev[v] >= 0 )
          Ly[v] += alpha;
    }
  }

}
```

```
getting sol:
         for(int i=0;i<A.size();i++){
            for(int j=0;j<B.size();j++){
              int val=get(A[i],B[j]);
                 cost[i][j]=val;
            }
         }
         hungarian(A.size(),A.size());
         int total=0;
         for(int i=0;i<A.size();i++){
            total+=cost[i][X[i]];
         }
```

Articulation:

```
struct Node {
 int level;int low;int noChildren;
 vector<int> adj;
}nd[MAXN];
int res;
```

```
bool isArtic[MAXN];
void dfs(int x) {
         nd[x].low = nd[x].level;
         int i, y;
         rep(i, nd[x].adj.size()) {
                  y = nd[x].adj[i];
                  if( nd[y].level == -1) { //unvisited
                        nd[y].level = nd[x].level + 1;
                        nd[x].noChildren++;
                        dfs(y);
                        nd[x].low = min(nd[x].low, nd[y].low);
                        if(nd[x].level > 0 && nd[y].low >= nd[x].level) { // x is a non-root node and there's
no way from y to any upper level of x
                              isArtic[x] = 1;
                        }
                  }
                  else if (nd[y].level < nd[x].level - 1) { //y's depth is lower than x's parent....so its a
back edge
                           nd[x].low = min(nd[x].low, nd[y].level);
                  }
         }
         if(nd[x].level == 0) { //root node
                  if(nd[x].noChildren >= 2) isArtic[x] = 1;
         }
}
ARTIC(){
         rep(i,n) nd[i].adj.clear(), nd[i].noChildren = 0,  nd[i].low = nd[i].level = -1;
         memset(isArtic, 0, sizeof(isArtic));
         nd[0].level = 0;
         dfs(0);
         res = 0;
         rep(i,n) if(isArtic[i]) res++;
}
```

2D BIT

```
int query(int x,int y){
       int result=0;
       int tempy=0;
       while(x>0){
            tempy=y;
            while(tempy>0){
                   result+=arr[x][tempy];
                   tempy-=(tempy & -tempy);
                   }
           x-=(x & -x);
     }
     return result;
}
void update(int x,int y,int num){
     int tempy=0;
     while(x<=maxval){
            tempy=y;
            while(tempy<=maxval){
                   arr[x][tempy]+=num;
                   tempy+=(tempy & -tempy);
                  }
           x+=(x &-x);
    }
}
```