/* Matrix exponentiation for solving linear recurrences

* Form f(n) = f(n-1) +f(n-2) or any other similar form. */

```c
#include<stdio.h>

unsigned long long int ans[3][3];
void multiply(unsigned long long int F[3][3],unsigned long long int M[3][3],unsigned long long int ans[3][3]);
void power(unsigned long long int F[3][3],int n,unsigned long long int M[3][3]);

int main()
{
    int test_cases,num;
    scanf("%d",&test_cases);
    unsigned long long int M[3][3]={{1,4,2},{1,0,0},{0,1,0}};
    unsigned long long int F[3][3]={{1,4,2},{1,0,0},{0,1,0}};
    int MOD=1000000007;
    int sol[test_cases];

    for(int i=0;i<test_cases;i++)
    {
        scanf("%d",&num);
        if(num==0)
            sol[i]=1;
        else if(num==1)
            sol[i]=1;
        else if(num==2)
            sol[i]=5;
        else
          {
            F[0][0]=1;
            F[0][1]=4;
            F[0][2]=2;
            F[1][0]=1;
            F[1][1]=0;
            F[1][2]=0;
            F[2][0]=0;
            F[2][1]=1;
            F[2][2]=0;
            power(F,num-2,M);
            //printf("%llu:%llu:",F[0][0],F[0][1]);
            sol[i]=((F[0][0]*5)%MOD+(F[0][1]*1)%MOD+(F[0][2]*1)%MOD)%MOD;
          }
    }
    for(int i=0;i<test_cases;i++)
    {
        printf("%d\n",sol[i]);
    }
}

void multiply(unsigned long long int F[3][3],unsigned long long int M[3][3],unsigned long long int ans[3][3])
{
    int MOD=1000000007;
```

```
    unsigned long long int sum=0;
    for(int i=0;i<3;i++)
    {
        for(int j=0;j<3;j++)
        {
            sum=0;
            for(int k=0;k<3;k++)
            {
                sum=((sum%MOD)+((F[i][k]%MOD)*(M[k][j]%MOD))%MOD)%MOD;
            }
            ans[i][j]=sum;
        }
    }

    for(int i=0;i<3;i++)
    {
        for(int j=0;j<3;j++)
        {
            //printf("%lld ",ans[i][j]);
            F[i][j]=ans[i][j];
        }
        //printf("\n");
    }
}

void power(unsigned long long int F[3][3],int n,unsigned long long int M[3][3])
{
    if(n==0 || n==1)
        return;
    power(F,n/2,M);
    multiply(F,F,ans);
    if(n%2!=0)
        multiply(F,M,ans);
}
```

/* Number Theory Basic Results (Cormen):

1. gcd is the smallest integral linear combination of the two numbers i.e. gcd(a.b) = min({ma+nb: m,n are integers})
2. d | a and d | b => d | gcd(a,b)
3. gcd.an; bn/ D n gcd.a; b/
4. n, a, b positve integers …. if n | ab and gcd(a, n) = 1, then n | b
5. if a,p and b,p are relatively prime, then so are ab,p
6. p | ab => p | a or p | b
7. basic gcd algo gcd (a,b) = (b==0) ? a : gcd(b, a%b)
8. gcd(a,b) = ma+nb to return (gcd, m , n) = extended_gcd(a,b) = (b==0) ? (a, 1, 0) : {d’,m’,n’ = extended_gcd(b, a%b); return (d’, n’, m’-(floor(a/b)*n’));}

*/

/* Segment Tree */

```
#include<stdio.h>
#include<iostream>
#include<math.h>

using namespace std;
#define MAX 1000000
typedef int type;

struct tree_node
{
  type max,min,sum;
};

void create_segment_tree(type*,int,int,int,struct tree_node *);
type query(type *, int ,int, int,int ,int ,struct tree_node *);

main()
{
  type A[MAX];
  int length;    //length of the array
  //scanf("%d",&length);

  int size = 2 * pow(2,(log(length)/log(2))+1);        //size of segment tree
  cout<<size<<endl;

  int i;
  struct tree_node segment_tree[size];
  for(i = 0;i<length;i++)
          scanf("%d",&A[i]);
 create_segment_tree(A,0,length-1,0,segment_tree);
  int no_of_query;
  scanf("%d",&no_of_query);

  for(int j = 0; j < no_of_query; j++)
  {
          int left,right;
          scanf("%d %d",&left,&right);
          type result = query(A,left,right,0,0,length-1,segment_tree);
          printf("%d\n",result);
  }
}

void create_segment_tree(type *A,int left,int right,int i,struct tree_node *segment_tree)
{
  if(left==right)
  {
          segment_tree[i].max = A[left];
          segment_tree[i].sum = A[left];
          segment_tree[i].min = A[left];
```

```
        return;
  }
  int mid = (left + right) / 2;
  create_segment_tree(A,left,mid,2*i+1,segment_tree);
  create_segment_tree(A,mid+1,right,2*i+2,segment_tree);

  /*for maximum*/
  if(segment_tree[2*i+1].max >= segment_tree[2*i+2].max)
  {
        segment_tree[i].max = segment_tree[2*i+1].max;
  }
 else
  {
        segment_tree[i].max = segment_tree[2*i+2].max;
  }

  /*for minimum*/
  if(segment_tree[2*i+1].min <= segment_tree[2*i+2].min)
  {
        segment_tree[i].min = segment_tree[2*i+1].min;
  }
  else
  {
        segment_tree[i].min = segment_tree[2*i+2].min;
  }

  /*for sum */
  segment_tree[i].sum = segment_tree[2*i+1].sum + segment_tree[2*i+2].sum;
}

type query(type *A, int left_range, int right_range,int i,int left,int right,tree_node *segment_tree)
{
  /* if the query  interval does not lie in query interval */
  if( left_range > right || right_range < left )
        //return -1;
        return 0; //in case of sum
  /* if the current interval is a subset of query interval */
  if(left_range <= left && right_range >= right)
        //return segment_tree[i].max;
        return segment_tree[i].sum;     //in case of sum
                                        //return segment_tree[i].min; in case of min

  int mid = (left+right)/2;
  //type left_max = query(A,left_range,right_range,2*i+1,left,mid,segment_tree);
  //type right_max = query(A,left_range,right_range,2*i+2,mid+1,right,segment_tree);
  type left_sum = query(A,left_range,right_range,2*i+1,left,mid,segment_tree);
  type right_sum = query(A,left_range,right_range,2*i+2,mid+1,right,segment_tree);
  return left_sum+right_sum;
  /*
  if(left_max ==-1)
        return right_max;
  if(right_max == -1)
        return left_max;
  if(left_max > right_max)
```

```
return left_max;
```

/* Multiplicative inverse using Euler Method */

```
// Use 64 bits integers to avoid overflow errors during multiplication.
// p should be a prime.
// if p is not prime then use extended euclidean algorithm to find modular multiplicative inverse.
long modPow(long a, long x, long p) {
    //calculates a^x mod p in logarithmic time.
    long res = 1;
    while(x > 0) {
        if( x % 2 != 0) {
            res = (res * a) % p;
        }
        a = (a * a) % p;
        x /= 2;
    }
    return res;
}

long modInverse(long a, long p) {
    //calculates the modular multiplicative of a mod m.
    //(assuming p is prime).
    return modPow(a, p-2, p);
}
```

/* Extended Euclid for finding Multiplicative inverse */

```
unsigned int multi_inverse(unsigned b){
        long long y = 1, lasty = 0, temp;
        unsigned a = MOD;

        while (b != 0){
                unsigned quotient = a/b;
                unsigned remainder = a%b;
                a = b;
                b = remainder;

                temp = lasty - (quotient*y);
                lasty = y;
                y = temp;
        }

        unsigned ans = (lasty + MOD)%MOD;

        return ans;
}
```

/* Graph structure Template */

```
//Included Libraries
#include <iostream>
#include <cstdio>
#include <cstring>
#include <limits>
#include <vector>
#include <stack>
#include <queue>
#include <utility>
#include <algorithm>

//General Macros
#define S(a)            scanf("%d", &a)
#define S_DOUBLE(a)     scanf("%lf", &a)
#define S_STRING(a)     scanf("%s", a)
#define FOR(i,m,n) for((i) = (m); (i) <= (n); (i)++)

//Graph Macros
#define AD_NODE         pair< int, cost_type >
#define MP(a,b)     make_pair((a), (b))
#define MAX_VERT        1001                    //Maximum number of verticies
#define MAX_EDGE        1001                    //Maximum number of edge
#define MAX_COST_TYPE   numeric_limits<cost_type>::max()

using namespace std;

//data type of the cost in pair
typedef double      cost_type;

struct edge_node
{
        int v1;
        int v2;
        cost_type cost;
};

class Graph
{
        //Data Structures (general)
        int                     V;                  //The number of vertices
        int                     E;                  //The number of edges
        vector< AD_NODE >       ad_list[MAX_VERT];  //Adjacency _list
        vector< edge_node >     edge_list;          //Edge _list

        public :

        //for algorithms
        bool            visit[MAX_VERT];   //Has the vertex been visted? (in bfs, dfs)
        cost_type       dist[MAX_VERT];    //Distance from the source (in bfs, dijkstra) / cost of lightest edge(in prim)
        int           parent[MAX_VERT];  //The parent of the vertex in traversal (in all) / parent in set hierarchy (in kruskal)
        int             discover[MAX_VERT], finish[MAX_VERT];       //The discovery and finish times (in dfs)
```

```
int                          count[MAX_VERT];//counts the number of elements in this set (in kruskal)
vector< edge_node > mst;                    //Edge _list of MST

//constructor and destructors
Graph(int V, int E);

//getters and setters
int getV(){ return V; }
void setV(int V){ this->V = V; }
int getE(){ return E; }
int getdist(int index){return dist[index];}

void setE(int E){this->E = E; }
void setVE(int V, int E){this->V = V;this->E = E;}

//Adjacency list
        //input methods
        void ad_list_from_edges();
        void ad_list_from_ad_mat();
        void ad_list_from_ad_list();
        //printing
        void print_ad_list();

//Edge _list
        //input
        void edge_list_from_edges();
        //manipulate
        void sort_edge_list();
        //print
        void print_edge_list();

//Input all List from edges
void all_lists_from_edges();
template<class T>
        void print_array(T *arr);


//Graph Traversal
void bfs(int source);
void dfs(int source);

//Single-source shortest path
void dijkstra(int source);
bool bellman_ford(int source);

//Minimum spanning tree
void kruskal();
        int find_set(int v);
        void union_set(int v, int w);
        void print_mst();
void prim(int source);

//Topological Sort ⇒ not implemented yet
void topological_sort(int source);
```

```
};

int main()
{
	int T, V, E;

	S(T);
	while(T--)
	{
		S(V);
		S(E);

		Graph graph(V, E);

		graph.all_lists_from_edges();

		graph.print_edge_list();
		graph.print_ad_list();

		graph.dijkstra(1);
		graph.print_array(graph.parent);
		graph.print_array(graph.dist);
		printf("\n%d\n", graph.getdist(2));

		graph.bfs(1);
		graph.print_array(graph.dist);
		graph.print_array(graph.parent);
		graph.print_array(graph.visit);

		graph.dfs(1);
		graph.print_array(graph.discover);
		graph.print_array(graph.finish);
		graph.print_array(graph.parent);
		graph.print_array(graph.visit);

		graph.kruskal();
		graph.print_mst();

		graph.prim(1);
		graph.print_array(graph.visit);
		graph.print_array(graph.parent);
		cout << "------------------------------------" << endl;

	}

	return 0;
}

Graph::Graph(int V, int E){
	this -> V = V;
	this -> E = E;
}

/* Input structure assumed (Edges given):
```

```
 * T
 * V E
 * vi vj cij
 *   .
 *   .
 * E times
 *   .
 *   .

where :
 * cik - edge weight between i and j
 */
void Graph::ad_list_from_edges()
{
        int i;
        int vertex1;
        int vertex2;
        cost_type cost;

        FOR(i, 1, V)
                ad_list[i].clear();

        FOR(i, 1, E)
        {
                S(vertex1);
                S(vertex2);
                S_DOUBLE(cost);

                //for directed only this
                ad_list[vertex1].push_back( MP(vertex2, cost) );

                //for undirected also include this
                ad_list[vertex2].push_back( MP(vertex1, cost) );
        }
}

void Graph::edge_list_from_edges()
{
        int i;
        int vertex1;
        int vertex2;
        cost_type cost;

        edge_list.clear();

        FOR(i, 1, E)
        {
                S(vertex1);
                S(vertex2);
                S_DOUBLE(cost);

                edge_node node;
                node.v1=vertex1;
```

```
                node.v2=vertex2;
                node.cost=cost;

                edge_list.push_back(node);
        }
}


void Graph::all_lists_from_edges()
{
        int i;
        int vertex1;
        int vertex2;
        cost_type cost;

        FOR(i, 1, V)
                ad_list[i].clear();

        edge_list.clear();

        FOR(i, 1, E)
        {
                S(vertex1);
                S(vertex2);
                S_DOUBLE(cost);

                //for directed only this
                ad_list[vertex1].push_back( MP(vertex2, cost) );

                //for undirected also include this
                ad_list[vertex2].push_back( MP(vertex1, cost) );


                edge_node node;
                node.v1=vertex1;
                node.v2=vertex2;
                node.cost=cost;

                edge_list.push_back(node);
        }
}

struct Compare_edge_node
{
        bool operator()(const edge_node& l, const edge_node& r)
        {
                return (l.cost < r.cost );              // '<' for ascending and '>' for decending
        }
}compare_edge_node;

void Graph::sort_edge_list()
{
        sort(edge_list.begin(), edge_list.end(), compare_edge_node);
}
```

```
/* Input structure assumed (Adjacency matrix structure):

 * T
 * V
 * c11 c12 .... c1V
 * c21 c22 .... c2V
 * .
 * .
 * .
 * cV1 cV2 .... cVV

where :
 * cik - edge weight between i and j
 */
void Graph::ad_list_from_ad_mat()
{
	int i, j;
	cost_type cost;

	//making of ad_listacency list -- works for both directed and undirected
	FOR(i, 1, V)
	{
		ad_list[i].clear();

		//scan for verties 1 to i;
		FOR (j, 1, i-1)
		{
			S_DOUBLE(cost);
			ad_list[i].push_back(MP(j, cost));
		}

		//leave the vertex i (assuming no self loops - not to be stored in ad_listacency list)
		S_DOUBLE(cost);

		//scan for vertices i+1 to V
		FOR (j, i+1, V)
		{
			S_DOUBLE(cost);
			ad_list[i].push_back(MP(j, cost));
		}
	}
}

/* Input structure assumed (Adjacency matrix structure):

 * T
 * V
 * k1 v11|c11 v12|c12 .... v1k1|c1k1
 * k2 v21|c21 v22|c22 .... v2k2|c2k2
 * .
 * .
 * .
```

```
 * kV vV1|cV1 vV2|cV2 .... vVk1|cVk1

where :
 * k1 - Number of vertices ad_listacent to v1
 * vij - The jth vertex in the ad_listacency list of vi
 * cij - edge weight between vi and vij

 * Note : '|' above is jsut for clarity. In actual ad_list, white spaces are assumed in place of '|'.
 */
void Graph::ad_list_from_ad_list()
{
	int i, j;

	int vertex, k;
	cost_type cost;

	//making of ad_listacency list -- works for both directed and undirected
	FOR(i, 1, V)
	{
		ad_list[i].clear();

		//scan list for i;
		S(k);
		FOR (j, 1, k)
		{
			S(vertex);
			S_DOUBLE(cost);

			ad_list[i].push_back(MP(vertex, cost));
		}
	}
}

void Graph::print_ad_list()
{
	int i;
	FOR (i, 1, V)
	{
		cout << "Vertex " << i << ":\t";

		vector< AD_NODE >::iterator it;
		FOR (it, ad_list[i].begin(), --(ad_list[i].end()))
			cout << it->first << "|"<<  it->second << "\t";

		cout << endl;
	}
}

void Graph::print_edge_list()
{
	vector< edge_node >::iterator it;
	FOR (it, edge_list.begin(), --(edge_list.end()))
		cout << it->v1 << "-"<< it -> v2 << "|"<<  it->cost << endl;
```

```
		cout << endl;
}

		template <class T>
void Graph::print_array(T *arr)
{
		int i;
		FOR (i, 1, V)
				cout << arr[i] << "\t";

		cout << endl;
}

class Compare_pq
{
		public :

				bool operator()(const AD_NODE& l, const AD_NODE& r)
				{
						return (l.second > r.second);  // '<' for max_heap and '>' for min_heap
				}
};

void Graph::bfs(int source)			//assumption : connected Graph
{
		memset(visit, false, sizeof(visit));
		memset(dist, 0, sizeof(visit));
		memset(parent, 0, sizeof(visit));

		queue< int > q;

		dist[source] = 0;
		visit[source] = true;
		q.push(source);

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

				vector< AD_NODE >::iterator it;
				FOR( it, ad_list[w].begin(), --(ad_list[w].end()) )
				{
						int v = it -> first;
						if (!visit[v])
						{
								visit[v] = true;
								dist[v] = dist[w] + 1;
								parent[v] = w;
								q.push(v);
						}
				}
		}
}
```

```
void Graph::dfs(int source)                    //assumption : connected Graph
{
        memset(visit, false, sizeof(visit));
        memset(discover, 0, sizeof(visit));
        memset(finish, 0, sizeof(visit));
        memset(parent, 0, sizeof(visit));

        stack< int > s;

        vector< AD_NODE >::iterator it[V+1];
        int i;
        FOR(i, 1, V)
                it[i] = ad_list[i].begin();

        s.push(source);
        visit[source] = true;
        discover[source] = 0;

        int time = 0;
        while(!s.empty())
        {
                int w = s.top();
                time++;

                for( ;it[w] < ad_list[w].end(); it[w]++ )
                {
                        int v = it[w] -> first;
                        if (!visit[v])
                        {
                                s.push(v);
                                visit[v] = true;
                                discover[v] = time;
                                parent[v] = w;

                                break;
                        }
                }

                if (it[w] == ad_list[w].end())
                {
                        s.pop();
                        finish[w] = time;
                }
        }
}

void Graph::dijkstra(int source)
{
        for (int i=0; i<=V; i++)
        {
                dist[i] = MAX_COST_TYPE;
                parent[i] = -1;
        }
```

```
        priority_queue< AD_NODE, vector< AD_NODE >, Compare_pq > pq;

        pq.push(MP(source, 0));
        dist[source] = 0;

        while(!pq.empty())
        {
                int w = pq.top().first;
                pq.pop();

                vector< AD_NODE >::iterator it;
                FOR( it, ad_list[w].begin(), --(ad_list[w].end()) )
                {
                        int v = it -> first;
                        cost_type wv_cost = it -> second;

                        cost_type temp = dist[w] + wv_cost;

                        if ( dist[v] > temp)
                        {
                                dist[v] = temp;
                                parent[v] = w;
                                pq.push(MP(v,dist[v]));
                        }
                }
        }
}

bool Graph::bellman_ford(int source)                    //not sure about implementation
{
        int i;
        FOR(i, 1, V)
        {
                dist[i] = MAX_COST_TYPE;
                parent[i] = i;
        }

        dist[source] = 0;

        vector< edge_node >::iterator it;
        FOR(i, 1, V-1)
        {
                FOR(it, edge_list.begin(), --(edge_list.end()))
                {
                        cost_type temp = dist[it->v1] + it->cost;
                        if ( dist[it->v2] > temp)
                        {
                                dist[it->v2] = temp;
                                parent[it->v2] = it->v1;
                        }
                }
        }

        FOR(it, edge_list.begin(), --(edge_list.end()))
```

```
	{
		if ( dist[it->v2] > dist[it->v1] + it->cost)
			return false;
	}

	return true;
}

void Graph::prim(int source)
{
	for (int i=0; i<=V; i++)
	{
		dist[i] = MAX_COST_TYPE;
		parent[i] = -1;
		visit[i] = false;
	}

	priority_queue< AD_NODE, vector< AD_NODE >, Compare_pq > pq;

	pq.push(MP(source, 0));
	dist[source] = 0;

	while(!pq.empty())
	{
		int w = pq.top().first;
		visit[w] = true;
		pq.pop();

		vector< AD_NODE >::iterator it;
		FOR( it, ad_list[w].begin(), --(ad_list[w].end()) )
		{
			int v = it -> first;
			if (visit[v])
				continue;

			cost_type wv_cost = it -> second;
			if ( dist[v] > wv_cost)
			{
				dist[v] = wv_cost;
				parent[v] = w;
				pq.push(MP(v,wv_cost));
			}
		}
	}
}

void Graph::kruskal()
{
	mst.clear();
	int i;
	FOR(i, 1, V)
	{
		parent[i] = i;
		count[i] = 1;
```

```
        }

        sort_edge_list();

        vector< edge_node >::iterator it;
        FOR(it, edge_list.begin(), --(edge_list.end()))
        {
                int set_v1 = find_set(it->v1);
                int set_v2 = find_set(it->v2);

                if (set_v1 != set_v2)
                {
                        union_set(set_v1, set_v2);
                        mst.push_back(*it);
                }
        }
}

int Graph::find_set(int v)
{
        while (v != parent[v])
        {
                v = parent[v];
        }

        return v;
}

void Graph::union_set(int v1, int v2)
{
        if (count[v1] >= count[v2])
        {
                count[v1] += count[v2];
                parent[v2] = v1;
        }
        else
        {
                count[v2] += count[v1];
                parent[v1] = v2;
        }
}

void Graph::print_mst()
{
        vector< edge_node >::iterator it;
        FOR (it, mst.begin(), --(mst.end()))
                cout << it->v1 << "-"<< it -> v2 << "|"<<  it->cost << endl;

        cout << endl;
}

void Graph::topological_sort()
{
```

```
}
```