```
class Template
{
public:
        /* ---------------------------------------Eratosthenes筛法----------------
----------------------- */
        vector<bool> Eratosthenes_Sieve(int n);

        /* ------------------------------------------欧几里得算法------------------
----------------------- */
        int gcd(int a, int b);

        /* ----------------------------------------扩展欧几里得算法----------------
------------------------ */
        void ex_gcd(int a, int b, int &x, int &y);

        /* -------------------------------------------分解质因数-------------------
----------------------- */
        vector<int> Prime_Factor(int n);

        /* --------------------------------------------归并排序-------------------
------------------------ */
        void merge_sort(vector<int> &target);

        /* --------------------------------------------快速排序-------------------
------------------------ */
        void quick_sort(vector<int> &target);

        /* -------------------------------------------链式前向星-------------------
----------------------- */
        struct edge
        {
                int v, next;//edges数组存储所有的边，v表示该边的终点，next表示与该边同
起点的下一条边所在edges中的位置
        } edges[1000];
        int head[500] = {-1};//head数组存储以下标值为起点的最后一条边所在edges数组中的
位置
        int counter = 0;    //counter用来遍历edges数组
        void add_edge(int u, int v)
        {
                edges[counter].v = v;
                edges[counter].next = head[u];
                head[u] = counter++;
        }

        void traverse(int u)
        { for (int i = head[u]; i != -1; i = edges[i].next); }

        /* ------------------------------------------广度优先搜索------------------
------------------------ */
        void BFS(vector<vector<int>> adjacency_matrix, vector<bool> &known, int
source);
```

```
        /* ------------------------------------------深度优先搜索-------------------------------------------- */
        void DFS(vector<vector<int>> adjacency_matrix, vector<bool> &known, int source);

        /* -------------------------------------------拓扑排序---------------------------------------------- */
        vector<int> topological_sort(vector<list<int>> adjacency_list);

        /* ----------------------------------------Dijkstra(堆优化)--------------------------------------- */
        void dijkstra(vector<vector<int>> adjacency_matrix, vector<bool> &known, vector<int> &Distance, int source);

        /* -----------------------------------------SPFA(负环判断)---------------------------------------- */
        void SPFA(vector<vector<int>> adjacency_matrix, vector<bool> &contain, vector<int> &Distance, int source);

        /* ----------------------------------------Floyd(路径记录)----------------------------------------*/
        void Floyd(vector<vector<int>> adjacency_matrix, vector<vector<int>> &Distance, vector<vector<int>> &next_vertex);

        /* --------------------------------------------Prim---------------------------------------------- */
        void prim(vector<vector<int>> adjacency_matrix, vector<int> &weight, vector<int> &previous);

        /* -------------------------------------------Kruskal------------------------------------------- */
        int kruskal(vector<pair<int, pair<int, int>>> &edges);

        /* -----------------------------------------Dinic(最大流)----------------------------------------- */
        struct Dinic
        {
                static const int SIZE = 2000;
                struct edge
                {
                        int v, next, capacity, flow;
                } edges[SIZE];
                int head[SIZE], level[SIZE], current[SIZE];
                int counter, v_num;

                Dinic(int v_num)
                {
                        counter = 0;
                        memset(head, 0xff, sizeof(head));
                        this->v_num = v_num;
                }

                void add_edge(int u, int v, int capacity);
```

```
            bool BFS(int start, int end);

            int DFS(int cur, int end, int flow);

            int max_flow(int start, int end);
    };

    /* --------------------------------------MCMF(最小费用最大流)--------------
------------------------ */
    struct MCMF
    {
            static const int SIZE = 2000;
            struct edge
            {
                    int u, v, next, capacity, flow, cost;
            } edges[SIZE];
            int head[SIZE], Distance[SIZE], contain[SIZE], current_flow[SIZE],
path[SIZE];
            int counter, v_num;

            MCMF(int v_num)
            {
                    counter = 0;
                    memset(head, 0xff, sizeof(head));
                    this->v_num = v_num;
            }

            void add_edge(int u, int v, int cost, int capacity);

            bool SPFA(int start, int end, int &flow, int &cost);

            pair<int, int> mincost_maxflow(int start, int end);
    };

    /* -----------------------------------------最长无重复子串-----------------
------------------------ */
    int Longest_substring(string s);

    /* --------------------------------------Manacher算法(最长回文)------------
------------------------*/
    pair<int, int> manacher(string &s);

    /* -----------------------------------------KMP(字符串匹配)----------------
------------------------*/
    int KMP(string a, string b);

    /* -------------------------------------并查集(按秩合并+路径压缩)-----------
-------------------------*/
    struct union_find
    {
            union_find(int n);                 //初始化(共含有n个点)
            int find(int x);                 //获取点x所属的连通分量的id
            void Union(int x1, int x2);         //连接点x1与x2
            vector<int> id;                     //每个点所属的连通分量的id
```

```
		vector<int> weight;		//每个连通分量所含的点数(权重)
	};

	/* -----------------------------------------BIT(二叉索引树)----------------
-------------------------*/
	struct BIT
	{
		static const int SIZE = 1000;
		int c[SIZE];

		int lowbit(int x)
		{ return x & (-x); }

		int sum(int x);

		void add(int x, int d)

	};

private:
	void merge_sort_recursive(vector<int> &target, std::vector<int> &copy,
size_t start, size_t end);

	void quick_sort_recursive(vector<int> &target, int start, int end);

	static const int INF = numeric_limits<int>::max();
};
```

```
/*Eratosthenes筛法
**筛选出n以内的所有质数
**返回参数res中如果res[i] == false则i为质数
***解释:对于p<=n && p>1的所有整数p,标记所有1p,2p,3p,4p......则未标记的数即为质数
***    !res[i]:只需判断p为素数的情况，若p非素数则p与p的倍数在之前的循环已经标记过
***    i * i:内层循环只需从i*i开始因为之前的循环已经标记过i * x(x<i)的情况
*/
vector<bool> Template::Eratosthenes_Sieve(int n)
{
	vector<bool> res(n + 1, false);
	res[0] = true;
	res[1] = true;
	for (int i = 2; i <= n; i++)
	{
		if (!res[i])
		{
			for (int j = i * i; j <= n; j += i)
			{
				res[j] = true;
			}
		}
	}
	return res;
```

```
}

/*欧几里得算法（辗转相除法）
**找出a与b的最大公约数
**返回参数为a与b的最大公约数
*/
int Template::gcd(int a, int b)
{
        return b == 0 ? a : gcd(b, a % b);
}


/*扩展欧几里得算法
**找出ax + by = gcd(a,b)的一个x,y整数解
**参数x,y即为上述整数的一对整数解
***解释：
***             推理1：当b = 0时ax + by = gcd(a,b) = a,此时x = 1,取y = 0;
***             推理2：设ax1 + by1 = gcd(a,b), bx2 + a%by2 = gcd(b,a%b) 由欧几里得
算法递归可知gcd(a,b) = gcd(b,a%b)
***                     则可得等式a(x1) + b(y1) = a(y2) + b(x2 - (a/b)*y2)视a,b为
未知数由等式恒等定理可得
***                     递推关系 x1 = y2 , y1 = x2 - (a/b)*y2;
*/
void Template::ex_gcd(int a, int b, int &x, int &y)
{
        if (b == 0)
        {
                x = 1;
                y = 0;
                return;
        }
        int x1, y1;
        ex_gcd(b, a % b, x1, y1);
        x = y1;
        y = x1 - (a / b) * y1;
}


/*分解质因数（唯一分解定理）
**将整数n用多个质数相乘的形式表示
**返回参数res中的元素即为n的质数因子
***算术基本定理可表述为：任何一个大于1的自然数 N,如果N不为质数，那么N可以唯一分解成有限个
质数的乘积;
*/
vector<int> Template::Prime_Factor(int n)
{
        vector<int> res;
        for (int i = 2; i <= n; i++)
        {
                while (n % i == 0)
                {
                        res.push_back(i);
                        n /= i;
                }
```

```
        }
        return res;
}


/*归并排序
**以归并排序的方法排序容器target
***
*/
void Template::merge_sort(vector<int> &target)
{
        vector<int> copy = target;
        merge_sort_recursive(target, copy, 0, target.size() - 1);
}

void Template::merge_sort_recursive(vector<int> &target, std::vector<int> &copy,
size_t start, size_t end)
{
        if (start >= end) return;
        int mid = (end - start + 1) / 2 + start;
        merge_sort_recursive(target, copy, start, mid - 1);
        merge_sort_recursive(target, copy, mid, end);
        int start1 = start, start2 = mid, counter = start;
        while (start1 <= mid - 1 && start2 <= end)
                target[counter++] = copy[start1] < copy[start2] ? copy[start1++] :
copy[start2++];
        while (start2 <= end)
                target[counter++] = copy[start2++];
        while (start1 <= mid - 1)
                target[counter++] = copy[start1++];
        for (int i = start; i <= end; i++)
                copy[i] = target[i];
}


/*快速排序
**以快速排序的方法排序容器vector
*/
void Template::quick_sort(vector<int> &target)
{
        quick_sort_recursive(target, 0, target.size() - 1);
}

void Template::quick_sort_recursive(vector<int> &target, int start, int end)
{
        if (start >= end)
                return;
        int pivot_element = target[end];
        int flag = start;
        for (int j = start; j <= end - 1; j++)
        {
                if (target[j] < pivot_element)
                        std::swap(target[flag++], target[j]);
        }
```

```
        std::swap(target[flag], target[end]);
        quick_sort_recursive(target, start, flag - 1);
        quick_sort_recursive(target, flag + 1, end);
}


/*广度优先搜索
**参数列表中:adjacency_matrix[a][b]的值若为0则代表a不与b相连
**                  source代表起点
**          known若为true则代表此点曾经访问过，默认为false
***解释：以广度优先的方式从起点source开始遍历整个图
*/
void Template::BFS(vector<vector<int>> adjacency_matrix, vector<bool> &known, int
source)
{
        queue<int> que;
        que.push(source);
        known[source] = true;
        while (!que.empty())
        {
                int tmp = que.front();
                que.pop();
                for (int i = 0; i < adjacency_matrix[tmp].size(); i++)
                {
                        if (!adjacency_matrix[tmp][i] || known[i])
                                continue;
                        que.push(i);
                        known[i] = true;
                }
        }
}


/*深度优先搜索
**参数列表中:adjacency_matrix[a][b]的值若为0则代表a不与b相连
**                  source代表起点
**          known若为true则代表此点曾经访问过，默认为false
***解释：以深度优先的方式从起点source开始遍历整个图
*/
void Template::DFS(vector<vector<int>> adjacency_matrix, vector<bool> &known, int
source)
{
        known[source] = true;
        for (int i = 0; i < adjacency_matrix[source].size(); i++)
        {
                if (!adjacency_matrix[source][i] || known[i])
                        continue;
                DFS(adjacency_matrix, known, i);
        }
}


/*拓扑排序
**以带队列的方式对图进行拓扑排序
```

```
**返回参数为排序后顶点的顺序
***解释：0：将所有入度为0的顶点存入队列
***             1：不断的弹出队列中的顶点元素，每弹出一个顶点元素，标记此顶点并将计数器
加1，然后通过邻接列表访问此顶点指向的所有顶点
***             3：将每个顶点的入度减1，若减1后入度为0则将此顶点存入队列 返回第0步
*/
vector<int> Template::topological_sort(vector<list<int>> adjacency_list)
{
        map<int, int> vertices_indgree;                                          //
全部顶点的入度表 first为顶点名称 second为此顶点的入度
        vector<int> res;                                                        //拓扑
排序后所有顶点的下标表 first为顶点名称 second为此顶点所处的位置
        bool cycle_found = false;                                                //若
检测到图中有环则cycle_found = true;

        //构建入度表
        for (int i = 0; i < adjacency_list.size(); i++)
        {
                vertices_indgree.insert({i, 0});
        }
        for (int i = 0; i < adjacency_list.size(); i++)
        {
                for (auto itr = adjacency_list[i].begin(); itr !=
adjacency_list[i].end(); itr++)
                {
                        vertices_indgree[*itr]++;
                }
        }
        //构建完毕

        queue<pair<int, int>> que;
        int counter = 0;
        for (int i = 0; i < vertices_indgree.size(); i++)
        {
                if (vertices_indgree[i] == 0)
                {
                        que.push({i, vertices_indgree[i]});
                }
        }
        while (!que.empty())
        {
                pair<int, int> vertice = que.front();
                que.pop();
                res.push_back(vertice.first);
                counter++;
                vertices_indgree[vertice.first] = -1;                            //标
记此顶点入度为-1以确保不会在访问此顶点
                for (auto itr = adjacency_list[vertice.first].begin(); itr !=
adjacency_list[vertice.first].end(); itr++)
                {
                        if (--vertices_indgree[*itr] == 0)
                        {
                                que.push({*itr, vertices_indgree[*itr]});
                        }
```

```
                }
        }
        if (counter != vertices_indgree.size())
        {
                cycle_found = true;
        }

        return res;
}


/*Dijkstra(堆优化)
**参数列表中:adjacency_matrix[a][b]的值若为INF则代表a不与b相连，若值大于0则为a到b的边的
权重
**                  source代表原点，以该点进行路径计算
**          known若为true则代表此点曾经访问过，默认为false
**          Distance代表此点与原点的最短路径，默认为INF
**          (Distance与adjacency_matrix的默认值INF应视题意做出调整,INF默认为
numeric_limits<int>::max())
***解释:1.获取优先队列(小顶堆)que的顶元素
***    2.访问所有该点所指向的点，并对其进行松弛
***    3.若被松弛的点从未被访问过，则将其压入优先队列中
*/
void Template::dijkstra(vector<vector<int>> adjacency_matrix, vector<bool> &known,
vector<int> &Distance, int source)
{
        Distance[source] = 0;
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> que;
        que.push({Distance[source], source});
        while (!que.empty())
        {
                int tmp = que.top().second;
                que.pop();
                if(known[tmp])
                        continue;
                known[tmp] = true;
                for (int i = 0; i < adjacency_matrix[tmp].size(); i++)
                {
                        if (adjacency_matrix[tmp][i] == INF)
                                continue;
                        if (Distance[i] > Distance[tmp] + adjacency_matrix[tmp]
[i])
                        {
                                Distance[i] = Distance[tmp] +
adjacency_matrix[tmp][i];
                                que.push({Distance[i], i});
                        }
                }
        }
}


/*SPFA(负环判断)
```

```
**参数列表中:adjacency_matrix[a][b]的值若为INF则代表a不与b相连，若值非0则为a到b的边的权
重
**                  source代表原点，以该点进行路径计算
**          contain若为true则代表此点在队列当中，默认为false
**          Distance代表此点与原点的最短路径，默认为INF
**          (Distance与adjacency_matrix的默认值INF应视题意做出调整,INF默认为
numeric_limits<int>::max())
***解释:1.通过容器in_times记录每个节点进入队列的次数，若次数大于总节点数则该图内包含负环
***     2.获取队列的队头元素
***     3.访问该节点所指向的节点，对被指向节点进行松弛
***     4.若松弛成功，将被指向的节点压入队列，并更新该点进入队列的次数
*/
void Template::SPFA(vector<vector<int>> adjacency_matrix, vector<bool> &contain,
vector<int> &Distance, int source)
{
        vector<int> in_times(adjacency_matrix.size(), 0);
        Distance[source] = 0;
        queue<int> que;
        que.push(source);
        in_times[source]++;
        contain[source] = true;
        while (!que.empty())
        {
                int tmp = que.front();
                que.pop();
                contain[tmp] = false;
                for (int i = 0; i < adjacency_matrix[tmp].size(); i++)
                {
                        if (adjacency_matrix[tmp][i] == INF)
                                continue;
                        if (Distance[i] > Distance[tmp] + adjacency_matrix[tmp]
[i])
                        {
                                Distance[i] = Distance[tmp] +
adjacency_matrix[tmp][i];
                                if (!contain[i])
                                {
                                        que.push(i);
                                        in_times[i]++;
                                        contain[i] = true;
                                        if (in_times[i] > adjacency_matrix.size())
                                        {
                                                cout << "!!!Negative Circle
Founded!!!";
                                                return;
                                        }
                                }
                        }
                }
        }
}


/*Floyd(路径记录)
```

```
**
*/
void
Template::Floyd(vector<vector<int>> adjacency_matrix, vector<vector<int>>
&Distance, vector<vector<int>> &next_vertex)
{
	//初始化Distance 与 next_vertex
	int vertex_num = adjacency_matrix.size();
	for (int i = 0; i < vertex_num; i++)
	{
		for (int j = 0; j < vertex_num; j++)
		{
			Distance[i][j] = adjacency_matrix[i][j];
			next_vertex[i][j] = j;
		}
	}
	//Floyd
	for (int mid = 0; mid < vertex_num; mid++)
	{
		for (int start = 0; start < vertex_num; start++)
		{
			for (int end = 0; end < vertex_num; end++)
			{
				if (Distance[start][end] > Distance[start][mid] +
Distance[mid][end])
				{
					Distance[start][end] = Distance[start]
[mid] + Distance[mid][end];
					next_vertex[start][end] =
next_vertex[start][mid];
				}
			}
		}
	}
}


/*Prim(最小生成树)
**参数列表中:adjacency_matrix[a][b]的值若为INF则代表a不与b相连，若值非INF则为a到b的边
的权重
**		weight代表在最小生成树中以此点为终点的边的权重，默认为INF
**		previous代表在最小生成树中此点的父节点，默认为-1
**		(weight与adjacency_matrix的默认值INF应视题意做出调整,INF默认为
numeric_limits<int>::max())
***解释:1.获取优先队列(小顶堆)que的顶元素
***	2.访问所有该点所指向的未被确认点，并对其进行松弛
***	3.若松弛成功，则将其压入优先队列中，并更新其父节点
***	(思路与Dijkstra相同)
*/
void Template::prim(vector<vector<int>> adjacency_matrix, vector<int> &weight,
vector<int> &previous)
{
	int source = 0;
	vector<bool> known(n, false);
```

```
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> que;
        weight[source] = 0;
        que.push({weight[source], source});
        while (!que.empty())
        {
                pair<int, int> tmp = que.top();
                int vertex = tmp.second;
                que.pop();
                known[vertex] = true;
                for (int i = 0; i < adjacency_matrix[vertex].size(); i++)
                {
                        if (!known[i] && weight[i] > adjacency_matrix[vertex][i])
                        {
                                weight[i] = adjacency_matrix[vertex][i];
                                previous[i] = vertex;
                                que.push({weight[i], i});
                        }
                }
        }
}


/*Kruskal(最小生成树(可用于带负权值的边的图))
**参数列表中:edges为图中所有的边first为边的权重second为边的起点与终点
**返回参数:最小生成树的总权值
***解释:1.先将边的集合以权重升序排序
***     2.每次选取权值最小的边，尝试将其添加进最小生成树中
***     3.若添加此边后最小生成树中不存在环，则添加成功(借助并查集判断)
*/
int Template::kruskal(vector<pair<int, pair<int, int>>> &edges)
{
        int weight = 0;
        union_find d_set(edges.size());
        sort(edges.begin(), edges.end());
        for (int i = 0; i < edges.size(); i++)
        {
                int u = edges[i].second.first;
                int v = edges[i].second.second;
                int u_id = d_set.find(u);
                int v_id = d_set.find(v);
                if (u_id != v_id)
                {
                        d_set.Union(u_id, v_id);
                        weight += edges[i].first;
                }
        }
        return weight;
}


/*Dinic (最大流)
**参数start为起点end为终点,返回参数为start到end的最大流
***解释:1.通过BFS构建各个点的层级图(level)
```

```
***    2.若成功访问至终点,则多次通过DFS由层级图进行增广,用current记录曾经访问过的边(优
化)
***    3.当BFS无法再访问至终点时,返回结果为最大流
***    (level记录一次BFS后该点的层级,current记录以该点为起点的下一条边在edges中的下标)
*/
void Template::Dinic::add_edge(int u, int v, int capacity)
{
        edges[counter] = {v, head[u], capacity, 0};
        head[u] = counter++;
        edges[counter] = {u, head[v], 0, 0};
        head[v] = counter++;
}

bool Template::Dinic::BFS(int start, int end)
{
        memset(level, 0xff, sizeof(level));
        level[start] = 0;
        queue<int> que;
        que.push(start);
        while (!que.empty())
        {
                int tmp = que.front();
                que.pop();
                for (int i = head[tmp]; i != -1; i = edges[i].next)
                {
                        if (level[edges[i].v] == -1 && edges[i].flow <
edges[i].capacity)
                        {
                                level[edges[i].v] = level[tmp] + 1;
                                que.push(edges[i].v);
                        }
                }
        }
        return level[end] != -1;
}

int Template::Dinic::DFS(int cur, int end, int flow)
{
        if (cur == end)
                return flow;
        for (int &i = current[cur]; i != -1; i = edges[i].next)
        {
                if (level[edges[i].v] == level[cur] + 1 && edges[i].flow <
edges[i].capacity)
                {
                        int next_flow = DFS(edges[i].v, end, min(flow,
edges[i].capacity - edges[i].flow));
                        if (next_flow > 0)
                        {
                                edges[i].flow += next_flow;
                                edges[i ^ 1].flow -= next_flow;
                                return next_flow;
                        }
                }
```

```
        }
        return 0;
}

int Template::Dinic::max_flow(int start, int end)
{
        if (start == end)
                return -1;
        int res = 0;
        while (BFS(start, end))
        {
                for (int i = 0; i < v_num; i++)
                        current[i] = head[i];
                while (long long flow = DFS(start, end, INF))
                        res += flow;
        }
        return res;
}


/*MCMF(最小费用最大流)
**参数:start为起点,end为终点,返回参数.first为最大流.second为最小费用
***解释:1.通过SPFA算法找出总费用最小的增广路径
***    2. 松弛成功后需记录下在该点的流量,与所经路径
***    3.一轮SPFA结束后更新总流量与总费用,并更新增广路径中所有边的流量与反向边的流量
***    (current_flow记录流过该节点的流量,path记录在最小费用增广路径中以该点为终点的边所
在edges中的下标)
*/
void Template::MCMF::add_edge(int u, int v, int cost, int capacity)
{
        edges[counter] = {u, v, head[u], capacity, 0, cost};
        head[u] = counter++;
        edges[counter] = {v, u, head[v], 0, 0, -cost};
        head[v] = counter++;
}

bool Template::MCMF::SPFA(int start, int end, int &flow, int &cost)
{
        for (int i = 0; i < v_num; i++)
        {
                Distance[i] = INF;
                contain[i] = 0;
        }
        queue<int> que;
        Distance[start] = 0;
        contain[start] = 1;
        path[start] = 0;
        current_flow[start] = INF;
        que.push(start);
        while (!que.empty())
        {
                int tmp = que.front();
                que.pop();
                contain[tmp] = 0;
```

```
                for (int i = head[tmp]; i != -1; i = edges[i].next)
                {
                        edge &e = edges[i];
                        if (Distance[e.v] > Distance[tmp] + e.cost && e.capacity >
e.flow)
                        {
                                Distance[e.v] = Distance[tmp] + e.cost;
                                path[e.v] = i;
                                current_flow[e.v] = min(current_flow[tmp],
e.capacity - e.flow);
                                if (!contain[e.v])
                                {
                                        que.push(e.v);
                                        contain[e.v] = 1;
                                }
                        }
                }
        }
        if (Distance[end] == INF)
                return false;
        flow += current_flow[end];
        cost += Distance[end] * current_flow[end];
        for (int i = end; i != start; i = edges[path[i]].u)
        {
                edges[path[i]].flow += current_flow[end];
                edges[path[i] ^ 1].flow -= current_flow[end];
        }
        return true;
}

pair<int, int> Template::MCMF::mincost_maxflow(int start, int end)
{
        int flow = 0, cost = 0;
        while (SPFA(start, end, flow, cost));
        return {flow, cost};
}


/*最长无重复子串
**找出串s的最长无重复子串 例如"abcabcbb"的最长无重复子串为"abc"长度为3
**返回参数即为最长无重复子串的长度
***
*/
int Template::Longest_substring(string s)
{
        int length = s.length(), res = 0;
        unordered_map<char, int> hash_map;
        int low = 0;
        for (int high = 0; high < length; high++)
        {
                auto itr = hash_map.find(s[high]);
                if (itr != hash_map.end())
                {
                        low = max(itr->second, low);
```

```
                }
                bool flag;
                flag = hash_map.insert({s[high], high + 1}).second;
                if (!flag)
                {
                        hash_map[s[high]] = high + 1;
                }
                res = max(res, high + 1 - low);
        }
        return res;
}


/*Mannacher算法(最长回文)
**返回参数中first代表最长回文长度, second代表最长回文的对称点位置
***解释：0：通过在字符串中插入间隔符消除回文长度奇偶性的问题(此时回文的长度必定为奇数)
***     1：通过radius[i]表示以第i个字符为对称轴时回文的半径长度如 #a#a#的半径为3,显而易见半径的长度-1即为出去间隔符的回文的长度, aa长为2
***     2：通过max_right表示所有曾访问过的回文字符串所能接触到的最右端的位置，max_right_pos表示此回文对称轴所在的位置
***     3：若i>max_right则表明此位置从未被探测过，此时记radius[i]为1
***        若i<max_right则此时观察i关于max_right_pos的对称点j（2*max_right_pos-i)
***             若 以j为轴的串的最左端 在 以max_right_pos为轴的串的最左端 的右边此时记radius[i]为radius[j]
***             若 以j为轴的串的最左端 在 以max_right_pos为轴的串的最左端 的左边此时记radius[i]为max_right-i
***             此时可得语句radius[i] = min(radius[2*max_right_pos-i],max_right-i);
***     4：标记radius[i]后继续以i为轴进行探测，当左右两端字符不相等时终止，每次探测成功便对radius[i]++
***     5：探测完毕后尝试更新max_right,max_right_pos与res
*/
pair<int, int> Template::manacher(string &s)
{
        //对字符串插入标记
        char spliter = 1;
        string s_new;
        for (int i = 0; i < s.length(); i++)
        {
                s_new.push_back(spliter);
                s_new.push_back(s[i]);
        }
        s_new.push_back(spliter);
        s = s_new;
        vector<int> radius(s.length(), 0);
        //插入完毕
        pair<int, int> res = {0, 0};
        int max_right = 0;
        int max_right_pos = 0;
        for (int i = 0; i < s.length(); i++)
        {
                i < max_right ? radius[i] = min(radius[2 * max_right_pos - i], max_right - i) : radius[i] = 1;
                while (i - radius[i] >= 0 && i + radius[i] < s.length() && s[i - radius[i]] == s[i + radius[i]])
```

```
                        radius[i]++;
                if (radius[i] + i - 1 > max_right)
                {
                        max_right = radius[i] + i - 1;
                        max_right_pos = i;
                }
                if (res.first < radius[i] - 1)
                {
                        res.first = radius[i] - 1;
                        res.second = i;
                }
        }
        return res;
}


/*KMP算法(字符串匹配)
**
*/
int Template::KMP(string a, string b)
{
        //构建部分匹配表
        int b_length = b.length();
        vector<int> partial_match_table(b_length, 0);
        partial_match_table[0] = -1;
        int j = -1;
        for (int i = 1; i < b_length; i++)
        {
                while (j > -1 && b[j + 1] != b[i])
                        j = partial_match_table[j];
                if (b[j + 1] == b[i])
                        j = j + 1;
                partial_match_table[i] = j;
        }
        //构建完毕
        int a_length = a.length();
        j = -1;
        for (int i = 0; i < a_length; i++)
        {
                while (j > -1 && b[j + 1] != a[i])
                        j = partial_match_table[j];
                if (b[j + 1] == a[i])
                        j = j + 1;
                if (j == b_length - 1)
                        return i - b_length + 1;
        }
        return -1;
}


/*并查集(加权优化+路径压缩)
**通过加权树的方法对其优化，使时间复杂度降至最低
***解释：1.引入树的结构用来表示连通分量，初始时有n个数(n个连通分量)
***     2.每次进行Union操作时将小树的根节点合并到大树的根节点上，同时也要增加大树的权值
```

```
***       非根节点的id并非所属连通分量的id而是其父节点的名称
***       只有根节点的id等于根节点的名称，同时也代表着所属连通分量的id
***     3.每次进行find操作时若此节点非根节点，则不断迭代直至找出根节点，找出根节点后即可
获取所属连通分量的id
*/
Template::union_find::union_find(int n)
{
        id.resize(n);
        weight.resize(n);
        for (int i = 0; i < n; i++)
        {
                id[i] = i;
                weight[i] = 1;
        }
}

int Template::union_find::find(int x)
{
        if (x != id[x])
        {
                id[x] = find(id[x]);
        }
        return x;
}

void Template::union_find::Union(int x1, int x2)
{
        int x1_id = find(x1);
        int x2_id = find(x2);
        if (x1_id == x2_id)
                return;
        if (weight[x1_id] > weight[x2_id])
        {
                id[x2_id] = x1_id;
                weight[x1_id] += weight[x2_id];
        } else
        {
                id[x1_id] = x2_id;
                weight[x2_id] += weight[x1_id];
        }
}

/*BIT(二叉索引树)
**lowbit(x)返回x二进制中从右至左第一个1为止的数的值
**数组c[x]存储区间(x-lowbit(x),x]的合
**sum(x)计算第1~x的元素的合
**add(x,d)将第x元素增加d
*/
int Template::BIT::sum(int x)
{

        int res = 0;
        for (int i = x; i > 0; i -= lowbit(i))
                res += c[i];
```

```
        return res;

}

void Template::BIT::add(int x, int d)
{

        for (int i = x; i <= SIZE; i += lowbit(i))
                c[i] += d;

}
```