#以下为常用图论算法，收集起来便于查看

注：一下图论算法均使用vector存图

1.Dijstra()//用于计算单源最短路，适合于正权图，无法处理负权图，基于贪心思想和优先队列实现

注意：在本算法中一个节点可能会被多次加入队列，但由于小根堆的性质，他只会进行一次更新

可能会多次更新，但出队列时一定是最短距离

注：Dijstra算法的优先队列要重定义<,

**friend bool operator <(const NODE& a,const NODE&b)**

**{**

**return a.d>b.d;**

**}**

且这次更新时该节点到原点已经是最短距离（PS：这也时Dijstra算法不能处理负权图的原因，由于负权边的存在

使得贪心思想不再正确）

void Dijkstra(int s)

{

memset(d,INF,sizeof(d));//d[]是距离数组，存储某个点到指定源点的当前最短距离

memset(vis,0,sizeof(vis));//vis[]是标记数组，记录某个节点是否更新过

d[s]=0;

pq.push({s,d[s]});//pq为优先队列，小根堆，按照距离大小依次弹出

while(!pq.empty())

{

DIST u=pq.top();

pq.pop();

if(vis[u.id])

continue;

vis[u.id]=1;

int id=u.id;

for(int i=0;i<G[id].size();i++){

EDGE e=G[id][i];

if(d[e.to]>d[id]+e.cost)

{

d[e.to]=d[id]+e.cost;

pq.push({e.to,d[e.to]});

}

}

}

}

2.spfa()//用于计算单源最短路，正负权图均可处理，基于队列和边更新思想实现

注：spfa可进一步分为bfs实现和dfs实现，一般来讲，dfs更倾向于判负环，可以提高算法速度，bfs倾向于处理带

负权的最短路或最长路

bool spfa\_bfs(int s)

{

memset(d,INF,sizeof(d));//d[]是距离数组，存储某个点到指定源点的当前最短距离

memset(vis,0,sizeof(vis));//vis是标记数组，标记某个节点是否在队列中

memset(cnt,0,sizeof(cnt));//cnt是计数数组，记录每个点进入队列的次数，判负环

d[s]=0;

queue<int>q;

q.push(s);

cnt[s]++;

vis[s]=1;

while(!q.empty()){

int u=q.front();

q.pop();

vis[u]=0;

for(int i=0;i<G[u].size();i++){

EDGE e=G[u][i];

if(d[e.to]>d[u]+e.cost){

d[e.to]=d[u]+e.cost;

if(!vis[e.to]){

vis[e.to]=1;

q.push(e.to);

cnt[e.to]=cnt[u]+1;

if(cnt[e.to]>n)

return true;//表示存在负环

}

}

}

}

return false;

}

bool spfa\_dfs(int pos)

{

vis[pos]=1;

for(int i=0;i<G[pos].size();i++){

EDGE e=G[pos][i];

if(d[e.to]>d[pos]+e.to){

if(vis[e.to])

return false;//找到负环

vis[e.to]=true;

d[e.to]=d[pos]+e.to;

if(!spfa(ans,e.to))

return false;

}

}

vis[pos]=0;

return true;

}

3.Folyd();//适用于多源最短路以及涉及到最短路中间的节点问题时，扩展算法可以记录最短路的个数

注：Folyd算法需要用邻接矩阵存图

void Floyd()

{

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

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

for(int j=1;j<=n;j++){

if(f[i][j]>f[i][k]+f[k][j])

f[i][j]=f[j][i]=f[i][k]+f[k][j];

}

return ;

}

4.Kusral()//最小生成树问题，结合并查集实现

int Kusral()

{

int k=0;

int ans=0;

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

EDGE e=edge[i];

int fa\_x=find(e.from);

int fa\_y=find(e.to);

if(fa\_x!=fa\_y){

k++;

ans+=edge[i].cost;

my\_union(fa\_x,fa\_y);

if(k>=p)

break;

}

}

return ans;

}

5.并查集,可以处理元素合并的问题，扩展算法有带权并查集和种类并查集

int find(int x)

{

if(x==fa[x])

return x;

return fa[x]=find(fa[x]);

}

void my\_union(int x,int y)

{

int fa\_x=find(x);

int fa\_y=find(y);

fa[fa\_y]=fa\_x;

}

6.拓扑排序,从顶点的入度出发不断遍历

void tuopu\_sort()

{

queue<int>p;

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

if(in[i]==1)

p.push(i);

}

while(k>0){

ans++;

while(!p.empty()){

int u=p.front();

p.pop();

cout<<u;

in[u]--;

k--;

if(k<1)

break;

for(int i=0;i<G[u].size();i++)

{

EDGE e=G[u][i];

in[e.to]--;

if(in[e.to]==1){

p.push(e.to);

}

}

}

}

}

7.Tarjon()//强连通图，缩点

void Tarjon(int x)//dfs枚举强联通分量

{

dfn[x]=low[x]=++tot;

st.push(x);

vis[x]=1;

for(int i=0;i<G[x].size();i++){

EDGE e=G[x][i];

if(!dfn[e.to]){

//如果还没有访问过这个子节点，那么就递归访问

Tarjon(e.to);

//访问完之后记得回溯更新low值

low[x]=min(low[x],low[e.to]);

}

else if(vis[e.to]){

//如果访问过该节点，且该节点仍然在栈中，那么就利用该节点更新当前节点的low值

//很大概率是出现了一个环，也就是强联通分量

low[x]=min(low[x],dfn[e.to]);

}

}

//如果当前节点外联的边都遍历完毕，那么首先确定该节点是否为强联通子图的根节点

if(low[x]==dfn[x]){

//是根节点，那么完成出栈操作

Tot++;

while(1){

int top=st.top();

vis[top]=0;

st.pop();

in[top]=tot;//记录新点编号

if(top==x)

break;

}

}

}

**8.Krusal算法延伸（Krusal重构树）**

Krusal重构树算法是Krusal算法的延伸，在利用并查集建立最小生成树时，每次合并两个集合都创建一个虚拟节点作为刚合并的两个祖先节点的父节点，该虚拟节点的权值表示两个子树中所有节点之间路径中最长边的最小值（具体最值取法取决于对于边的排序顺序），该算法适合于解决求图中两个点直接所有路径中最长边的最小值或最短边的最大值，Krsual重构树具有优良的性质，其中任意两个叶子结点代表原图中的节点，任意两个叶子结点之间路径中最大值的最小值即为两个节点最近公共祖先的权值（可以结合LCA问题求解，利用树上倍增思想）。

基础模板如下：

（以2021年ICPC上海站H题Life is a game 为例）

1. #include<bits/stdc++.h>
2. **using** **namespace** std;
4. **typedef** **long** **long** ll;
5. **const** **int** maxn=2e5+10;
6. **int** n,m,q;
7. ll a[maxn];
8. **int** fa[maxn];//维护并查集
9. ll d[maxn];//维护每条边的权值，也即是重构树中非叶子结点的权值
10. vector<**int**>e[maxn];//重构Krusal树
11. ll down[maxn];//维护该节点下所有叶子节点（也即是代表城市）的权值和
12. **int** f[maxn][20];//倍增数组，用来回溯树
13. **int** find(**int** x)
14. {
15. **if**(x==fa[x])**return** x;
16. **return** fa[x]=find(fa[x]);//路径压缩
17. }
18. **struct** EDGE{
19. **int** u;
20. **int** v;
21. **int** cost;
22. }edge[maxn];
23. **bool** cmp(EDGE x,EDGE y)
24. {
25. **return** x.cost<y.cost;
26. }
28. **int** Krusal()
29. {
30. //初始化并查集
31. **for**(**int** i=1;i<=2\*n;i++)fa[i]=i;
32. **int** node=n;
33. **for**(**int** i=1;i<=m;i++){
34. **int** fa\_u=find(edge[i].u);
35. **int** fa\_v=find(edge[i].v);
36. **if**(fa\_u!=fa\_v){
37. d[++node]=edge[i].cost;
38. fa[fa\_u]=fa[fa\_v]=node;
39. e[node].push\_back(fa\_u);
40. e[node].push\_back(fa\_v);
41. }
42. }
43. **return** node;
44. }
45. **int** build(**int** x)
46. {
47. ll sum=a[x];
48. **for**(**int** i=0;i<e[x].size();i++){
49. **int** j=e[x][i];//x节点的孩子
50. f[j][0]=x;//从j节点向上走一步到达的节点是i
51. **for**(**int** s=1;s<20;s++){
52. f[j][s]=f[f[j][s-1]][s-1];
53. }
54. sum+=build(j);
55. }
56. down[x]=sum;
57. **return** sum;
58. }
59. **void** Solution(**int** root)
60. {
61. **int** x,k;
62. scanf("%d %d",&x,&k);
63. ll ans=a[x]+k;
64. d[0]=1e18;//0代表跳出了树的范围，要给一个极大值，表示跳不到
65. **while**(x!=root){
66. **int** tmp=x;
67. **for**(**int** i=19;i>=0;i--){
68. //倍增向上找
69. **if**(d[f[x][i]]<=ans){
70. //如果当前的权值足以走到x的第2的i次方个父亲处，那么直接跳到该处
71. //注：因为Krusal重构树从上到下，非叶子结点的权值不增，所以只需要判断深度最小的权值即可，也即是
72. //这一路上最大的权值
73. x=f[x][i];
74. ans=k+down[x];
75. }
76. }
77. **if**(x==tmp){
78. //如果一步都走不了了,直接跳出
79. **break**;
80. }
81. }
82. printf("%lld\n",ans);
83. }
84. **int** main()
85. {
86. scanf("%d %d %d",&n,&m,&q);
87. **for**(**int** i=1;i<=n;i++)scanf("%lld",&a[i]);
88. **for**(**int** i=1;i<=m;i++){
89. scanf("%d %d %d",&edge[i].u,&edge[i].v,&edge[i].cost);
90. }
91. sort(edge+1,edge+m+1,cmp);
92. **int** root=Krusal();//建立重构树
93. build(root);//初始化ST查询表以及down数组
94. **while**(q--){
95. Solution(root);
96. }
97. }