目录

[第三章 栈 3](#_Toc60248736)

[数组栈 3](#_Toc60248737)

[STL栈 3](#_Toc60248738)

[整型数组栈 4](#_Toc60248739)

[单调栈 4](#_Toc60248740)

[第四章 队列 4](#_Toc60248741)

[数组队列 4](#_Toc60248742)

[STL队列 5](#_Toc60248743)

[STL双端队列 5](#_Toc60248744)

[整型数组队列 5](#_Toc60248745)

[单调队列 6](#_Toc60248746)

[第五章 排序 6](#_Toc60248747)

[STL快速排序 6](#_Toc60248748)

[自定义排序-lambda表达式 7](#_Toc60248749)

[vector/string的全排序 7](#_Toc60248750)

[归并排序（附带统计逆序对数） 7](#_Toc60248751)

[第六章 树 8](#_Toc60248752)

[树的储存与遍历 8](#_Toc60248753)

[孩子表示法 8](#_Toc60248754)

[儿子兄弟表示法，主要用于前后序遍历 9](#_Toc60248755)

[图表示法，主要用于从特定点遍历 9](#_Toc60248756)

[近似满二叉树 10](#_Toc60248757)

[二叉搜索树：见专题 10](#_Toc60248758)

[哈夫曼树：见堆 10](#_Toc60248759)

[第六章 二叉搜索树 10](#_Toc60248760)

[STL 11](#_Toc60248761)

[非 STL 11](#_Toc60248762)

[第八章 优先队列 15](#_Toc60248763)

[STL 15](#_Toc60248764)

[整数大根堆 15](#_Toc60248765)

[整数小根堆 15](#_Toc60248766)

[数组 15](#_Toc60248767)

[可并堆/左偏树 17](#_Toc60248768)

[哈夫曼树 18](#_Toc60248769)

[第九章 并查集 18](#_Toc60248770)

[朴素并查集 18](#_Toc60248771)

[朴素并查集（压行） 19](#_Toc60248772)

[按秩合并并查集O(log n) 19](#_Toc60248773)

[第十章 图 20](#_Toc60248774)

[图的表示 20](#_Toc60248775)

[邻接矩阵表示 20](#_Toc60248776)

[邻接表表示（非STL） 20](#_Toc60248777)

[邻接表表示（STL） 20](#_Toc60248778)

[图的遍历 21](#_Toc60248779)

[广搜 21](#_Toc60248780)

[深搜 21](#_Toc60248781)

[最短路：见专题 22](#_Toc60248782)

[最小生成树 22](#_Toc60248783)

[prim 22](#_Toc60248784)

[kruskal 22](#_Toc60248785)

[第十章 最短路 23](#_Toc60248786)

[朴素 dijkstra，O(n^2)，不能处理负边 23](#_Toc60248787)

[堆优化的 dijkstra，O(e log e)，不能处理负边 24](#_Toc60248788)

[Bellman-Ford，O(VE)，可以处理负边、判负环 24](#_Toc60248789)

[floyd ，O(n^3)，可以动态加点 25](#_Toc60248790)

[其他 26](#_Toc60248791)

[位运算操作 26](#_Toc60248792)

[筛素数 26](#_Toc60248793)

[埃氏筛O(n log log n) 26](#_Toc60248794)

# 第三章 栈

## 数组栈

struct StackElem{

//...

};

StackElem stk[MAXN];

int cnt=0,siz=MAXN;

//StackElem \*cur=stk,\*maxcur=stk+MAXN;

bool push(StackElem &e){

if(cnt==siz) return false; //cur==maxcur

stk[cnt++]=e; //\*(cur++)=e;

return true;

}

bool pop(){

if(cnt==0) return false; //cur==stk

--cnt;//--cur;

return true;

}

StackElem top(){

if(cnt==0) return NULLE;//cur==stk

return stk[cnt-1];//\*(cur-1)

}

bool empty(){

return cnt==0;//cur==stk

}

int size(){

return cnt;//cur-stk;

}

## STL栈

stack<StackElem> stk;

stk.push(e);

stk.pop();

stk.top();

stk.empty();

stk.size();

## 整型数组栈

int stk[MAXN],cnt=0;

stk[cnt++]=e;

--cnt;

stk[cnt-1];

(cnt==0);

cnt;

## 单调栈

int f(StackElem &a,StackElem &b) {}//单调性的函数

stack<StackElem> stk;

void push(StackElem &e){

while( stk.size() && !f(stk.top(),e) ) stk.pop();

stk.push(e);

}

# 第四章 队列

## 数组队列

struct QueueElem{

//...

};

QueueElem que[MAXN];

int head=0,tail=0,siz=MAXN;

//QueueElem \*head=que,\*tail=que;

bool push(QueueElem &e){

if(tail==siz) return false; //tail==que+MAXN

que[tail++]=e; //\*(tail++)=e;

return true;

}

bool pop(){

if(head==tail) return false;

++head;

return true;

}

QueueElem front(){

if(head==tail) return NULLE;

return que[head];//\*head

}

bool empty(){

return head==tail;

}

int size(){

return tail-head;

}

## STL队列

queue<QueueElem> que;

que.push(e);

que.pop();

que.front();

que.empty();

que.size();

## STL双端队列

deque<QueueElem> deq;

deq.push\_front(e);

deq.push\_back(e);

deq.pop\_front();

deq.pop\_back();

deq.front();

deq.back();

deq.empty();

deq.size();

## 整型数组队列

int que[MAXN],head=0,tail=0;

que[tail++]=e;

++head;

que[head];

(head==tail);

(tail-head);

## 单调队列

QueueElem que[MAXN];

int head=0,tail=0;

int f(QueueElem &a,QueueElem &b) {}//单调性函数

void push(QueueElem &e){

if( f(que[tail-1],e) ){

que[tail++]=e;

return ;

}

int l=head,r=tail-1,mid,pos=head;

while(l<=r){

mid=(l+r)/2;

if( f(que[mid],e) ) pos=r,r=mid-1;

else l=mid+1;

}

que[pos]=e;

}

# 第五章 排序

## STL快速排序

sort(a,a+n);//默认从小到大，不稳定

stable\_sort(a,a+n);//默认从小到大，稳定，常数更大

//自定义排序-自定义函数

bool cmp(const Elem &a,const Elem &b){//自定义排序函数，返回值为 true 时，a 排前

//...

}

sort(a,a+n,cmp);

stable\_sort(a,a+n,cmp);

### 自定义排序-lambda表达式

sort(a,a+n,[](const Elem& a,const Elem& b){

//...

});

stable\_sort(a,a+n,[](const Elem& a,const Elem& b){

//...

});

sort(a,a+n,[](const Elem& a,const Elem& b){

return a>b;//从大到小排序

});

### vector/string的全排序

vector<Elem> v;//string s

sort(v.begin(),v.begin()+v.size()); //sort(v.begin(),v.end());

stable\_sort(v.begin(),v.begin()+v.size()); //stable\_sort(v.begin(),v.end());

## 归并排序（附带统计逆序对数）

Elem a[MAXN],b[MAXN];

int merge\_sort(int head,int tail){

if(head==tail) return 0;

int l1=head,r1=(head+tail)/2,l2=r1+1,r2=tail,ans=0,pos=head;

ans+=merge\_sort(l1,r1);

ans+=merge\_sort(l2,r2);

while(l1<=r1&&l2<=r2){

if(a[l1]<=a[l2]) b[pos++]=a[l1++];

else b[pos++]=a[l2++],ans+=r1-l1+1;

}

while(l1<=r1) b[pos++]=a[l1++];

while(l2<=r2) b[pos++]=a[l2++];

for(int i=head;i<=tail;i++) a[i]=b[i];

return ans;

}

# 第六章 树

## 树的储存与遍历

### 孩子表示法

struct node{

//data

int child[maxn];//各个孩子的地址

int cntchild;

}n[maxn];

void preorder(int root){

work(root);//work() 代指操作，例如输出、计算或什么都不做，下同

for(int i=0;i<n[root].cntchild;i++)

preorder(n[root].child[i]);

}

void postorder(int root){

for(int i=0;i<n[root].cntchild;i++)

preorder(n[root].child[i]);

work(root);

}

void inorder(int root){

if(n[root].cntchild)

inorder(n[root].child[0]);

work(root);

for(int i=1;i<n[root].cntchild;i++)

inorder(n[root].child[i]);

}

void bfs(int root){//层序遍历

queue<int> q;

q.push(root);

while( q.size() ){

int now=q.front();

q.pop();

for(int i=0;i<n[now].cntchild;i++)

q.push(n[now].child[i]);

work(now);

}

}

### 儿子兄弟表示法，主要用于前后序遍历

struct node{

//data

int child,bro;

}n[maxn];

void preorder(int root){

work(root);

preorder(n[root].child);

preorder(n[root].bro);

}

void postorder(int root){

postorder(n[root].child);

work(root);

postorder(n[root].bro);

}

### 图表示法，主要用于从特定点遍历

vector<int> Edg[maxn];

int pa[maxn];

void addEdge(int u,int v){

Edg[u].push(v);

Edg[v].push(u);

}

void dfs(int root,int fa){

pa[root]=fa;

for(auto to : Edg[root])

if(to!=fa)

dfs(to,root);

work(root);

}

//dfs(u,-1)

void bfs(int root){

pa[root]=-1;

queue<int> q;

q.push(root);

while( q.size() ){

int now=q.front();

q.pop();

for(auto to : Edg[now])

if(to!=pa[now]){

q.push(to);

pa[to]=now;

}

work(now);

}

}

### 近似满二叉树

Elem data[maxn];

int lc(int pos) { return 2\*pos; }//pos<<1

int rc(int pos) { return 2\*pos+1; }//pos<<1|1

void preorder(int root){

work(root);

preorder(lc(root));//preorder(root<<1)

preorder(rc(root));//preorder(root<<1|1)

}

void postorder(int root){

preorder(lc(root));//preorder(root<<1)

preorder(rc(root));//preorder(root<<1|1)

work(root);

}

void inorder(int root){

preorder(lc(root));//preorder(root<<1)

work(root);

preorder(rc(root));//preorder(root<<1|1)

}

## 二叉搜索树：见专题

## 哈夫曼树：见堆

# 第六章 二叉搜索树

## STL

set<Elem> s;//不可重集

multiset<Elem> s;//可重集

bool exist(Elem& data){

return s.find(data)!=s.end();

}

s.insert(data);

s.erase(data);

/\*

可重集删除一个元素

s.erase( s.find(data) );

\*/

s.lower\_bound(data);

//非严格后继，找到了返回迭代器（类似指针），未找到返回 s.end()

s.upper\_bound(data);//严格后继

//找前驱需要用相反数建 set

Elem minElem(){

return \*s.begin();

}

Elem maxElem(){

return \*s.rbegin();

}

int Range(Elem &x,Elem &y){

int ans=0;

for(auto p=s.lower\_bound(x);p!=s.end()&&\*p<y;p++) ans++;

//set<Elem>::iterator p=s.lower\_bound(x)

return ans;

}

## 非 STL

struct node{

Elem data;

node \*lc,\*rc,\*pa;

}\*root;

node\* find(node \*p,Elem& data){

if(!p||p->data==data)

return p;//找到或到达空节点

else if(p->data>data)

return find(p->lc,data);

else

return find(p->rc,data);

}

void insert(node \*p,Elem& data){

if(p->data==data)

work(p);

else if(p->data>data){

if(p->lc) insert(p->lc,data);

else p->lc=newNode(data,p);//新建节点

}

else{

if(p->rc) insert(p->rc,data);

else p->rc=newNode(data,p);

}

}

node \*preElem(node \*p,Elem& data){//求前驱

if(p->data>=data){

if(p->lc) return preElem(p->lc,data);

else if(p->data>data) return NULL;

else return p;//非严格前驱

//严格前驱：NULL

}

else{

if(p->rc) return preElem(p->rc,data)

else return p;

}

}

node \*postElem(node \*p,Elem& data){//求后继

if(p->data<=data){

if(p->rc) return preElem(p->rc,data);

else if(p->data<data) return NULL;

else return p;//非严格后继

//严格后继：NULL

}

else{

if(p->lc) return preElem(p->lc,data)

else return p;

}

}

node \*minElem(node \*p){//最小元素

if(p->lc) return minElem(p->lc);

else return p;

}

node \*maxElem(node \*p){//最大元素

if(p->rc) return maxElem(p->lc);

else return p;

}

int size(node \*p){

if(!p) return 0;

else return size(p->lc)+size(p->rc)+1;

}

int Range(node \*p,Elem &x,Elem &y){//范围 [x,y] 内元素个数（递归）

if(!p) return 0;

else if(p->data<x) return Range(p->rc,x,y);

else if(p->data>y) return Range(p->lc,x,y);

else return Range(p->lc,x,y)+Range(p->rc,x,y)+1;

}

int Range(Elem &x,Elem &y){//范围 [x,y] 内元素个数（递推）

node \*q=postElem(root,x);//非严格后继

if(!q) return 0;

int ans=0;

while(q&&q->data<y){

ans++;

q=postElem(root,\*q);

}

return ans;

}

node \*erase(Elem &data){//返回新的根

node \*p;

if(!root) return NULL;

if(root->data==data){//删除树根

if( root->lc==0 && root->rc==0 ){//平凡树

delete root;

return NULL;

}

else if( root->lc==0 || root->rc==0 ){

if(root->lc) p=root->lc;

else p=root->rc;

delete root;

return p;

}

else{

p=preElem(root,data);

swapData(root,p);//交换数据

if(p->lc){

if(p==p->pa->lc) p->pa->lc=p->lc;

else p->pa->rc=p->lc;

}

else{

if(p==p->pa->lc) p->pa->lc=p->rc;

else p->pa->rc=p->rc;

}

delete p;

return root;

}

}

p=find(root,data);

if( p->lc==0 && p->rc==0 ){

if(p==p->pa->lc) p->pa->lc=0;

else p->pa->rc=0;

}

else if( p->lc==0 || p->rc==0 ){

if(p->lc){

if(p==p->pa->lc) p->pa->lc=p->lc;

else p->pa->rc=p->lc;

}

else{

if(p==p->pa->lc) p->pa->lc=p->rc;

else p->pa->rc=p->rc;

}

}

else{

node \*q=preElem(root,data);

swapData(p,q);

p=q;

if(p->lc){

if(p==p->pa->lc) p->pa->lc=p->lc;

else p->pa->rc=p->lc;

}

else{

if(p==p->pa->lc) p->pa->lc=p->rc;

else p->pa->rc=p->rc;

}

}

delete p;

return root;

}

# 第八章 优先队列

## STL

priority\_queue<Elem> pq;

pq.push(data);

pq.pop();

pq.top();

### 整数大根堆

pq.push(val);

int maxV=pq.top();

### 整数小根堆

pq.push(-val) ;

int minV=-pq.top();

## 数组

Elem pq[maxn];//堆必须从 1 开始存

int size;

bool f(Elem& a,Elem& b){//比较函数，a 是否应该是 b 的根

//...

}

void push(Elem& data){

pq[size++]=data;

for(int pos=size;pos>=1;pos/=2)

if( f(pq[pos],pq[pos/2]) ) swap(pq[pos],pq[pos/2]);

else break;

}

void pop(){

pq[1]=pq[size];

for(int pos=1;pos<=size;)

if(pos\*2>size) break;

else if(pos\*2+1>size){

if( f(pq[pos],pq[pos\*2]) ) break;

else{

swap(pq[pos],pq[pos\*2]);

pos=pos\*2;

}

}

else{

if( f(pq[pos],pq[pos\*2]) && f(pq[pos],pq[pos\*2+1]) )

break;

else if( f(pq[pos\*2],pq[pos\*2+1]) ){

swap(pq[pos],pq[pos\*2]);

pos=pos\*2;

}

else{

swap(pq[pos],pq[pos\*2+1]);

pos=pos\*2+1;

}

}

}

Elem& top(){

return pq[1];

}

void built(int n){

size=n;

for(int i=n/2;i>=1;i--)

for(int pos=1;pos<=size;)

if(pos\*2>size) break;

else if(pos\*2+1>size){

if( f(pq[pos],pq[pos\*2]) ) break;

else{

swap(pq[pos],pq[pos\*2]);

pos=pos\*2;

}

}

else{

if( f(pq[pos],pq[pos\*2]) && f(pq[pos],pq[pos\*2+1]) )

break;

else if( f(pq[pos\*2],pq[pos\*2+1]) ){

swap(pq[pos],pq[pos\*2]);

pos=pos\*2;

}

else{

swap(pq[pos],pq[pos\*2+1]);

pos=pos\*2+1;

}

}

}

## 可并堆/左偏树

struct node{

int s;

Elem data;

node \*lc,\*rc,\*pa,\*top;

bool poped;

}n[maxn];

bool f(node \*x,node \*y){//判断函数， x 是否在 y 上面

//...

}

node \*top(node \*p){

if( !p->top->poped&&p->top!=p )

return p->top=top(p->top);//根未删除，路径压缩

else

if(p->pa) return p->top=top(p->pa);//根已删除，路径压缩

else

return p;//自己是根

}

node \*merge(node \*x,node \*y){

if(!x) return y;

if(!y) return x;

if( !f(x,y) ) swap(x,y);//令堆顶为 x

x->rc=merge(x->rc,y);

x->rc->pa=x;

if( !x->lc || x->rc->s>x->lc->s)

swap(x->rc,x->lc);//不满足左偏树，调整

if(x->rc) x->s=x->rc->s+1;

else x->s=1;

return x;

}

void pop(node \*p){

p->poped=1;

if(p->lc) p->lc->top=p->lc;

if(p->rc) p->rc->top=p->rc;

p=merge(p->lc,p->rc);

if(p) p->pa=0;

}

## 哈夫曼树

struct node{

Elem data;

node \*lc,\*rc;

};

bool operator < (node \*a,node \*b){//a<b 排序方式

//...

}

priority\_queue<\*node> pq;

void merge(){

while( pq.size()>1 ){

node \*now=newNode();

now->lc=pq.top();

pq.pop();

now->rc=pq.top();

pq.pop();

now->data=calc( now->lc->data , now->rc->data );

//calc(a,b) 表示将这两棵树合并后，新的值；哈夫曼树一般是 a+b

pq.push(now);

}

}

# 第九章 并查集

## 朴素并查集

int pa[maxn],cnt;

void init(int n){

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

pa[i]=i;

cnt=n;

}

int find(int u){

return (u==pa[u])?u:(pa[u]==find(pa[u]));

}

bool isunion(int u,int v){

return find(u)==find(v);

}

void merge(int u,int v){

if(isunion(u,v)) return ;

pa[find(u)]=find(v);

cnt--;

}

## 朴素并查集（压行）

int pa[maxn],cnt;

void init(int n) { for(int i=1;i<=n;i++) pa[i]=i; cnt=n; }

int find(int u) { return (u==pa[u])?u:(pa[u]=find(pa[u])); }

bool isunion(int u,int v) { return find(u)==find(v); }

void merge(int u,int v) { cnt-=!isunion(u,v); pa[find(u)]=find(v); }

## 按秩合并并查集O(log n)

int pa[maxn],cnt;

void init(int n){

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

pa[i]=-1;

cnt=n;

}

int find(int u){

return (pa[u]<0)?u:(pa[u]==find(pa[u]));

}

bool isunion(int u,int v){

return find(u)==find(v);

}

void merge(int u,int v){

u=find(u);

v=find(v);

if(u==v) return ;

if(pa[u]<pa[v]) swap(u,v);

pa[v]-=pa[u];

pa[u]=v;

cnt--;

}

# 第十章 图

## 图的表示

### 邻接矩阵表示

int g[maxn][maxn];

void addEdge(int u,int v,int w=1){//u->v ，边权为 w 的边（无权图则 w=1）

g[u][v]=1;

//无向图加上 g[v][u]=1 ，下同

}

### 邻接表表示（非STL）

struct link{

int to,weight;

link\* nxt;

link(link\* nxt\_,int to\_,int weight\_=0):nxt(nxt\_),to(to\_),weight(weight\_) {}

}\*fir[maxn];

void addEdge(int u,int v,int w=1){

fir[u]=new link(fir[u],v,w);

//fir[v]=new link(fir[v],u,w);

}

### 邻接表表示（STL）

#### 无权图

vector<int> to[maxn];

void addEdge(int u,int v){

to[u].push\_back(v);

to[v].push\_back(u);

}

#### 有权图

typedef pair<int,int> pii;

vector<pii> to[maxn];

void addEdge(int u,int v,int w){

to[u].push\_back( pii(v,w) );

//to[v].push\_back( pii(u,w) );

}

//以下以有权有向图的邻接链表（STL） 表示法为例

## 图的遍历

bool vis[maxn];

### 广搜

queue<int> q;

void bfs(int s){

q.push(s);

vis[s]=1;

while( q.size() ){

int now=q.front();

q.pop();

work(now);

for(auto p : to[now])

if(!vis[p]){

vis[p]=1;

q.push(p);

}

}

}

### 深搜

void dfs(int now){

vis[now]=1;

work(now);

for(auto p : to[now])

if(!vis[p])

dfs(p);

}

## 最短路：见专题

## 最小生成树

### prim

int lowcost[maxn];

bool vis[maxn];

int prim(int s=0){//O(n^2)

lowcost[s]=0;

vis[s]=1;

for(int i=0;i<n;i++) lowcost[i]=INF;

for(auto p : to[s]) lowcost[p.first]=p.second;

int ans=0;

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

int minpos=0;

while(vis[pos]) minpos++;

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

if(!vis[j]&&lowcost[j]<lowcost[minpos])

minpos=j;

if(lowcost[minpos]==INF) return INF;

ans+=lowcost[minpos];

vis[minpos]=1;

for(auto p : to[minpos])

if(!vis[p.first])

lowcost[p.first]=min(lowcost[p.first],p.second);

}

return ans;

}

### kruskal

int pa[maxn],cnt;

void init(int n) { for(int i=1;i<=n;i++) pa[i]=i; cnt=n; }

int find(int u) { return (u==pa[u])?u:(pa[u]=find(pa[u])); }

bool isunion(int u,int v) { return find(u)==find(v); }

void merge(int u,int v) { cnt-=!isunion(u,v); pa[find(u)]=find(v); }

//并查集

struct Edge{

int u,v,w;

};

bool operator < (const Edge &a,const Edge &b){

return a.w>b.w;

}

priority\_queue<Edge> pq;

int kruskal(){//O(elog e)

int ans=0;

while( pq.size() ){

Edge now=pq.top();

pq.pop();

if( isunion(now.u,now.v) ) continue;

merge(now.u,now.v);

ans+=now.w;

}

if(cnt==1) return ans;

else return INF;

}

# 第十章 最短路

## 朴素 dijkstra，O(n^2)，不能处理负边

int dis[maxn];

bool vis[maxn];

void dijkstra(int s) {

for(int i=0;i<n;i++) dis[i]=INF;

dis[s]=0;

vis[s]=1;

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

int minpos=0;

while(vis[minpos]) minpos++;

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

if(dis[j]<dis[minpos])

minpos=j;

vis[minpos]=1;

for(auto p : to[minpos])

dis[p.first]=min(dis[p.first],dis[minpos]+p.second);

}

}

## 堆优化的 dijkstra，O(e log e)，不能处理负边

typedef pair<int,int> pii;

int dis[maxn];

priority\_queue<pii> pq;

void dijkstra(int s){

for(int i=0;i<n;i++) dis[i]=INF;

dis[s]=0;

pq.push( pii(0,s) );

//第一维为距离的相反数（小根堆），第二维为点

while( pq.size() ){

pii now=pq.top();

pq.pop();

if(dis[now.second]<-now.first) continue;

//答案距离更小，说明该点已经被更新过

for(auto p : to[now.second])

if(dis[p.first]>dis[now.second]+p.second){

dis[p.first]=dis[now.second]+p.second;

pq.push( pii(p.first,-dis[p.first]) );

}

}

}

## Bellman-Ford，O(VE)，可以处理负边、判负环

queue<int> q;

int dis[maxn],count[maxn];

bool inq[maxn];//In Queue or Not

bool bellman(int s){

while(q.size()) q.pop();

q.push(s);

inq[s]=1;

count[s]=1;

while( q.size() ){

int now=q.front();

q.pop();

inq[now]=0;

for(auto p : to[now])

if(dis[p.first]<dis[now]+p.second){

dis[p.first]=dis[now]+p.second;

if(inq[p.first]) continue;

q.push(p.first);

inq[p.first]=1;

count[p.first]++;

if(count[p.first]==n)

return 0;//出现负环

}

}

return 1;

}

## floyd ，O(n^3)，可以动态加点

int dis[maxn][maxn];

void update(int k) {

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

for(int j=0;j<n;j++)

dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);

}

void floyd(){

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

for(int j=0;j<n;j++)

dis[i][j]=INF;

for(int k=0;k<n;k++){

for(auto p : to[k])

dis[k][p.first]=min(dis[k][p.first],p.second);

update(k);

}

}

//多源最短路，在无负边时，可以从各个顶点开始，跑 dijkstra

//复杂度为 O(n^3) 或 O(ne log e)

# 其他

## 位运算操作

全集： u=(1ll<<n)-1

空集： 0

集合并： a|b

集合交： a&b

相对补集： a^(a&b)

绝对补集： (~a)&u

对称差集： a^b

子集： a&b==b// b 是a 的子集

真子集： (a&b==b)&(a!=b)// b 是 a 的真子集

属于： (a>>i)&1

不属于： !((a>>i)&1)

## 筛素数

### 埃氏筛O(n log log n)

bool isnotprime[maxn];

int prime[maxn],cntprime;

void sieve(int n){

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

if(isnotprime[i]) continue;

prime[cntprime++]=i;

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

isnotprime[j]=1;

}

}