数组、链表、二叉树 链表、二叉树、队列栈的各种操作（性能，场景）

二分查找和各种变的二分查找

各类排序算法以及复杂度分析（快排、归并、堆）

各类算法题（手写）

理解并可以分析时间和空间复杂度

动态规划、贪心

红黑树、AVL树、Hash树、Tire树、B树、B+树

图算法（较少，理解两个最短路径算法）

图的表示

邻接链表和邻接矩阵两种。稀疏图常用邻接链表表示，稠密图通常用邻接矩阵表示。

广度优先搜索 BFS

广度优先搜索是最简单的图搜索算法之一，**Prim的最小生成树算法**和**Dijkstra的单源最短路径算法**都使用了类似广度优先搜索思想。

```
BFS(G, s)                              // 图G=（V,E）使用邻接链表表示
  for each vertex u ∈ G.V - {s}
      u.color = WHITE                  // u结点颜色
      u.d = ∞                          // 从源节点s到u结点的距离
      u.π = NIL                        // u结点在广度优先搜索中的前驱
  s.color = GRAY
  s.d = 0
  s.π = NIL
  Q = Empty
  ENQUEUE(Q, s)
  while Q != Empty
      u = DEQUEUE(Q)
      for each v ∈ G.Adj[u]
          if v.color == WHITE
              v.color = GRAY
              v.d = u.d + 1
              v.π = u;
              ENQUEUE(Q, v)
      u.color = BLACK
```

算法的初始化成本为O(V)，每个结点进行一次入队出队操作，因此队列操作时间为O(V)，扫描邻接链表总时间为O(E)。算法总复杂度为O(V+E)，因此广度搜索时间是图G的邻接链表大小的一个线性函数。

> E为边数，V为节点数。

深度优先遍历 DFS

深度优先搜索算法总是对最近才发现的结点v的出发边进行探索，直到该结点的所有出发边都被发现为止。一旦结点v的所有边都被发现，搜索则“回溯”到v的前驱结点（v是经过该节点才被发现的），来搜索改前驱结点的出发边。该过程一直持续到从源结点可以达到的所有结点都被发现为止。如果还存在未发现的节点，则深度优先搜索将从这些未发现的结点中任选一个作为一个新的源结点，并重复同样的搜索过程，直到所有结点被发现为止。

深度优先搜索的前驱子图可以形成一个由多棵深度优先树构成的深度优先森林。

> 算法总复杂度为O(V+E)。

```
DFS(G)                          //图G=（V,E）使用邻接链表表示
  for each vertex u ∈ G.V
      u.color = WHITE
      u.π = NIL
  time = 0;                     //time是个全局变量，用来计算时间戳
  for each vertex u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G.u)

DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY
  for each v ∈ G.Adj[u]
      if v.color == WHITE
          v.π = u
          DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time
```

拓扑排序

拓扑排序首先需要图G为有向无环图，它是G中所有结点的一种线性次序，该次序满足如下条件：如果图G包含边(u, v)，则结点u在拓扑排序中处于结点v的前面。许多实际应用中都需要使用有向无环图来指明事件的优先次序，拓扑排序可以找出这些进行事件的合理顺序。

拓扑排序算法其实很简单，分为两步：第一步，对有向无环图进行深度优先搜索排序；第二步，将所有结点按照**其完成的时间的逆序**从左向右排序，此时所有的有向边都是从左指向右。证明神马的具体见算法导论吧。

算法第一步深度优先搜索算法按时间复杂度为O(V+E)，第二步将结点插入链表最前端所需的时间为O(V)，所以总的时间复杂度为O(V+E)。

另外还有一种拓扑排序算法，其思想为首先选择一个无前驱的顶点（即入度为0的顶点，图中至少应有一个这样的顶点，否则肯定存在回路），然后从图中移去该顶点以及由他发出的所有有向边，如果图中还存在无前驱的顶点，则重复上述操作，直到操作无法进行。如果图不为空，说明图中存在回路，无法进行拓扑排序；否则移出的顶点的顺序就是对该图的一个拓扑排序。

```
Topological_Sort_II(G);
  begin
      for 每个顶点u∈V[G] do d[u]←0;  //初始化d[u],d[u]用来记录顶点u的入度

      for 每个顶点u∈V[G] do
          for 每个顶点v∈Adj[u]
              do d[v]←d[v]+1;  //统计每个顶点的入度

      CreateStack(s);  //建立一个堆栈s

      for 每个顶点u∈V[G] do
        if d[u]=0 then push(u,s);  //将度为0的顶点压入堆栈

      count←0;

      while (not Empty(s)) do
          begin
          u←top(s);  //取出栈顶元素
          pop(s);     //弹出一个栈顶元素
          count←count+1;
          R[count]←u;   //线性表R用来记录拓扑排序的结果

          for 每个顶点v∈Adj[u] do //对于每个和u相邻的节点v
            begin
              d[v]←d[v]-1;
              if d[v]=0
                  then push(v,s);  //如果出现入度为0的顶点将其压入栈
            end;
      end;

      if count<>G.size then writeln('Error! The graph has cycle.')
                  else 按次序输出R;
  end;
```

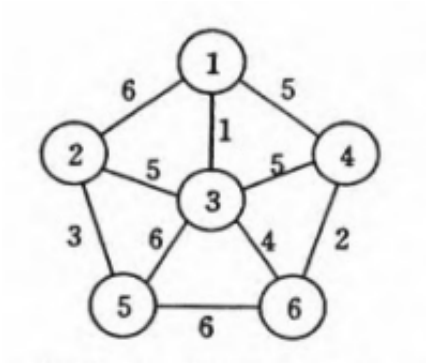
最小生成树

- Prim算法：贪心算法

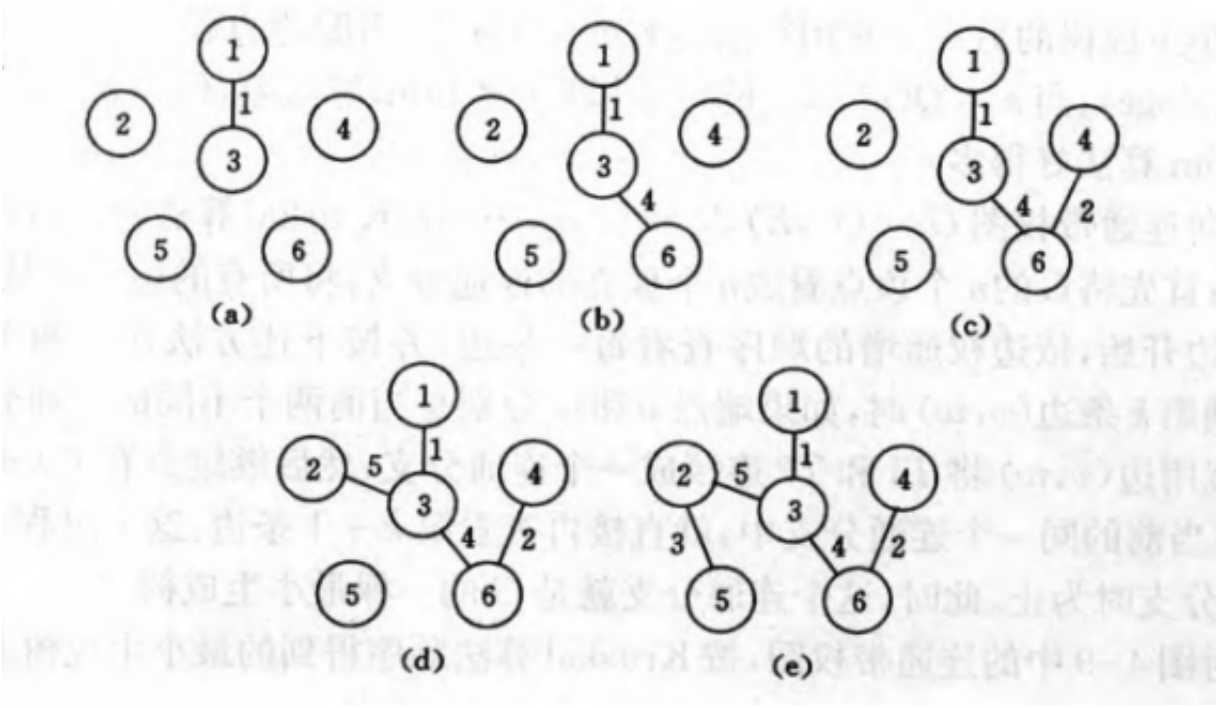
MST（Minimum Spanning Tree，最小生成树）问题有两种通用的解法，Prim算法就是其中之一，它是从点的方面考虑构建一颗MST，大致思想是：设图G顶点集合为U，首先任意选择图G中的一点作为起始点a，将该点加入集合V，再从集合U-V中找到另一点b使得点b到V中任意一点的权值最小，此时将b点也加入集合V；以此类推，现在的集合V={a，b}，再从集合U-V中找到另一点c使得点c到V中任意一点的权值最小，此时将c点加入集合V，直至所有顶点全部被加入V，此时就构建出了一颗MST。因为有N个顶点，所以该MST就有N-1条边，每一次向集合V中加入一个点，就意味着找到一条MST的边。

> 复杂度：O(n^2)

任意指定一个顶点作为起始点，放在S中。
每一步将最短的特殊边放入S中，需要n-1步，即可把所有的其他的点放入S中。算法结束。



对于这个图，Prim算法的过程为：



- Kruskal算法：克鲁斯卡尔

求解最小生成树的另一种常见算法是Kruskal算法，它比Prim算法更直观。从直观上看，Kruskal算法的做法是：每次都从剩余边中选取权值最小的，当然，这条边不能使已有的边产生回路。

算法逻辑人很容易理解，但用代码判断当前边是否会引起环的出现，则很棘手。
算法说明
为了判断环的出现，我们换个角度来理解Kruskal算法的做法：初始时，把图中的n个顶点看成是独立的n个连通分量，从树的角度看，也是n个根节点。我们选边的标准是这样的：若边上的两个顶点从属于两个不同的连通分量，则此边可取，否则考察下一条权值最小的边。

最短路径算法：迪杰斯特拉 & 弗洛伊德

Dijkstra

- **第一步**：用带权的矩阵WeiArcs来表示带权有向图，如果图中的两个顶点vi和vj是连通的，则用WeiArcs[i][j]表示这两个顶点所形成边的权值；如果vi和vj不连通，即这条边不存在，那么将WeiArcs[i][j]置为∞。
- **第二步**：设S为已求得的从某一顶点v始发的最短路径的终点的集合，且S的初始状态为空，初始化时，将始发顶点置于S集合中。那么从v出发到图中其余各个顶点vi可能达到的最短路径长度的初值为D[i]。
- **第三步**：选择一顶点vj,使得vj就是当前求得的一条从顶点v出发的最短路径的终点。此时令S = S ∪ {vj}。
- **第四步**：修改从v出发到集合V-S（V为图顶点的集合）中任一顶点vk可达的最短路径长度。如果D[j]+WeiArcs[j][k] < D[K],则D[k] = D[j] + WeiArcs[j][k]。
- **第五步**：重复操作第三步、第四步共N-1次，由此就能求得从v出发到图中其余各个顶点的最短路径。

	v0	v1	v2	v3	v4	v5
v0	∞	∞	10	∞	30	100
v1	∞	∞	5	∞	∞	∞
v2	∞	∞	∞	50	∞	∞
v3	∞	∞	∞	∞	∞	10
v4	∞	∞	∞	20	∞	60
v5	∞	∞	∞	∞	∞	∞

带权邻接矩阵

终点	起始点为 v0，D(f(x))为 v0 到各个终点路径的值				
	⇒	⇒	⇒	⇒	
v1	∞	∞	∞	∞	∞ 无
v2	D(v0,v2)=10				
v3	∞	D(v0,v2,v3)= 60	D(v0,v4,v3)= 50		
v4	D(v0,v4)=30	D(v0,v4)=30			
v5	D(v0,v5)=100	D(v0,v5)=100	D(v0,v4,v5)= 90	D(v0,v4,v3,v5) =60	
vj	**v2**	**v4**	**v3**	**v5**	
S	{v0,v2}	{v0,v2,v4}	{v0,v2,v3,v4}	{v0,v2,v3,v4,v5}	

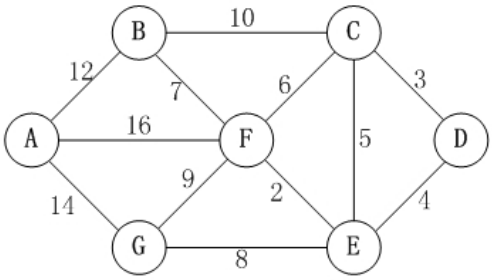
运算过程表

Floyd

通过Floyd计算图G=(V,E)中各个顶点的最短路径时，需要引入一个矩阵S，矩阵S中的元素a[i][j]表示顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

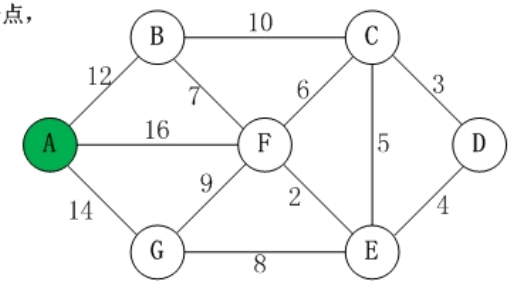
假设图G中顶点个数为N，则需要对矩阵S进行N次更新。初始时，矩阵S中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则a[i][j]=∞。 接下来开始，对矩阵S进行N次更新。第1次更新时，如果"a[i][j]的距离" > “a[i][0]+a[0][j]”(a[i][0]+a[0][j]表示"i与j之间经过第1个顶点的距离")，则更新a[i][j]为"a[i][0]+a[0][j]“。 同理，第k次更新时，如果"a[i][j]的距离” > “a[i][k]+a[k][j]"，则更新a[i][j]为"a[i][k]+a[k][j]"。更新N次之后，操作完成！

第1步：
初始化矩阵S



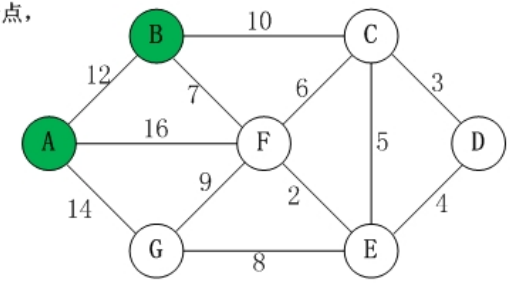
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第2步：
以顶点A为中介点，
更新矩阵S。



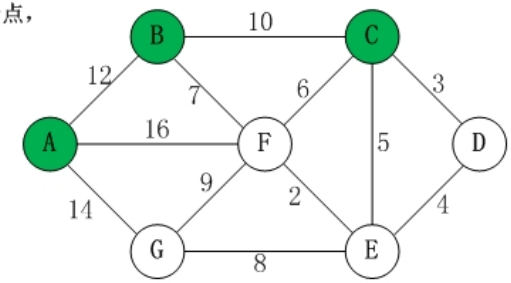
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	INF	INF	8	9	0

第3步：
以顶点B为中介点，
更新矩阵S。



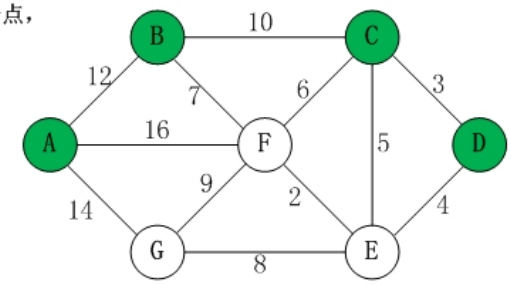
	A	B	C	D	E	F	G
A	0	12	22	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	22	10	0	3	5	6	36
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	36	INF	8	9	0

第4步：
以顶点C为中介点，
更新矩阵S。



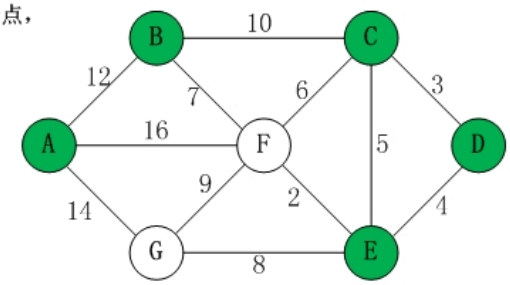
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第5步：
以顶点D为中介点，
更新矩阵S。



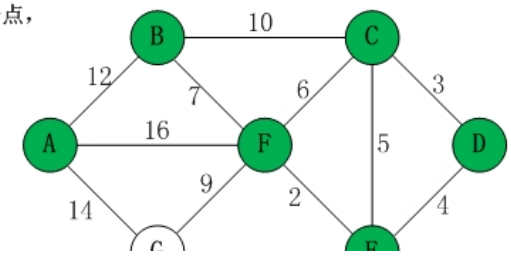
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第6步：
以顶点E为中介点，
更新矩阵S。

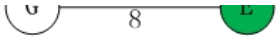


	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	23
C	22	10	0	3	5	6	13
D	25	13	3	0	4	6	12
E	27	15	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	23	13	12	8	9	0

第7步：
以顶点F为中介点，
更新矩阵S。

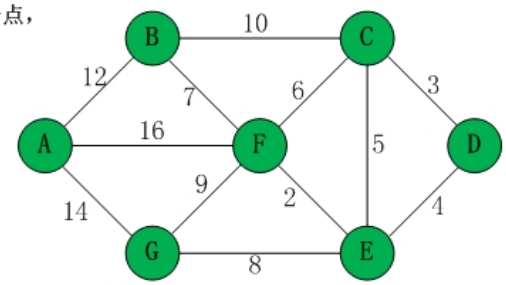


	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9



$$G\begin{bmatrix} 14 & 16 & 13 & 12 & 8 & 9 & 0 \end{bmatrix}$$

第8步：
以顶点G为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0