堆栈 :
字符串 : KMP算法
二叉树 :
树 :
图 : dijkstra算法, floyd算法, Kruskal算法
排序 : 归并
检索 :

- 所有的算法只包含尽可能简化的模版, 不记录数据, 但会做预处理

1. KMP算法

```
//s[1] 对齐第一个字符, s[n] 是最后一个字符, next[i]是第i个字符的前缀后缀最长匹配长度
next[1] = 0;
for (int i=2,j=0;i<=n;i++){
    while (j && s[i]!=s[j+1]) j = next[j]; // 一直往前找, 直到找到一个匹配的或者j=0
    if (s[i]==s[j+1])j++; // 找到了一个匹配的
    next[i] = j; // 保存匹配长度
}
```

2. dijkstra算法 (用堆实现)

```
int w[maxn][maxn]; //已memset(w, 0x3f, sizeof(w)),且连边的数据已加载
int dis[maxn],vis[maxn]; // dis最短距离, vis是否访问
int n; // 顶点数, 从1开始

void dijkstra(int s){
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> q; // first 
        memset(dis,0x3f,sizeof(dis));
        dis [s] = 0;
        q.push({0,s});// 起点
        while(!q.empty()){
                int u = q.top().second;// 取出最小距离的点
                q.pop();
                if (vis[u])continue;
                vis[u] = 1;// 标记已访问
                for (int i=1;i<=n;i++){
                        int w0 = min(w[i][u],w[u][i]);
                        dis[i] = min(dis[i],dis[u]+w0);// 更新距离
                        q.push({dis[i],i});// 加入队列
                }
        }
}
```

3. floyd算法

```
void floyd(){// g是邻接矩阵, 已经memset(g, 0x3f, sizeof(g))过且已经加载了边的数据
        for(int i=1;i<=n;i++) g[i][i]=0;
        for(int k=1;k<=n;k++){
                for(int i=1;i<=n;i++){
                        for(int j=1;j<=n;j++){
                                g[i][j] = min(g[i][j],g[i][k]+g[k][j]);
                                g[j][i] = g[i][j]; //无向图同步更新
                        }
                }
        }
}
```

4. Kruskal算法

```cpp
struct Edge {
    int u, v, w;  // u, v: 两个节点，w: 边的权重
};
vector<Edge> E, Etree;  // 存储所有的边
int fa[114514];  // 用于并查集的父节点数组
int find(int x) { // 查找函数，路径压缩优化
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}
void Kruskal(){
    for (int i = 1; i <= n; i++) {  // 初始化并查集
        fa[i] = i;
    }
    sort(E.begin(), E.end(), [](Edge a, Edge b) {
        return a.w < b.w;
    });// 按照边的权重升序排序
    for (Edge e : E) {
        int u = find(e.u), v = find(e.v);  // 查找 u, v 的
        if (u != v) {
                        fa[u] = v;  // 合并 u, v
                        Etree.push_back(e);  // 将边加入最小生成树
                }
    }
}
```

5. 归并排序

```cpp
void merge_sort(int l, int r){
    if (l == r) return; // 递归边界
    int mid = (l + r) >> 1; // 取中点
    merge_sort(l, mid); // 递归排序左半部分
    merge_sort(mid + 1, r); // 递归排序右半部分
    int i = l, j = mid + 1, k = 0; // i, j 分别指向左右两部分的起点，k 用于合并
    while (i <= mid && j <= r) { // 合并两部分
        if (a[i] <= a[j]) b[k++] = a[i++];
        else b[k++] = a[j++];
    }
    while (i <= mid) b[k++] = a[i++]; // 处理剩余部分
    while (j <= r) b[k++] = a[j++];
    for (int i = l; i <= r; i++) a[i] = b[i - l]; // 将合并后的数组复制回原数组
}
```

6. 二分法

```
bool check(int x) {}// 判断 x 是否满足条件
int binarySearch(int left, int right) {
        if (check(right)) return right; // 如果 right 满足条件，直接返回 (因为处理不了)
    while (left < right) {
        int mid = left + (right - left) / 2; // 防溢出
        if (check(mid)) left = mid + 1;
        else right = mid;
    }
    return left - 1; // 返回符合条件的最大值
}
```

Trick1:

```
ios::sync_with_stdio(false); // 关闭输入输出流的同步，加快cincout速度
```