

Data Structure Final Review

RandomStar in 2020.01

Chapter 1 Introduction and Algorithm Analysis

- 算法的几个要素：输入，输出，明确性(Definiteness)，有穷性(Finiteness)，高效性(Effectiveness)

1.1 时间复杂度和空间复杂度

- 几个基本的运算规则
 - 顺序结构：直接相加
 - 循环中：复杂度=一次循环的复杂度x循环次数
 - 嵌套循环中：循环规模的乘积x一次循环的复杂度
 - if/else语句：选其中复杂度最高的

1.2 最大子列和问题的分析

- 算法1：使用3层嵌套循环直接计算，复杂度为 $O(N^3)$
- 算法2：使用两层循环并设置标记位，复杂度为 $O(N^2)$
- 算法3：使用归并的方法(这一部分PPT上的代码比较复杂，可以好好看看)，复杂度为 $O(N \lg N)$
- 算法4：一种在线算法，复杂度时线性的，代码如下

```
1  int MaxSubsequenceSum(int A[],int N) {
2      int ThisSum = 0,MaxSum = 0,j;
3      for(j = 0; j < N; j++) {
4          ThisSum += A[j];
5          if(ThisSum > MaxSum)
6              MaxSum = ThisSum;
7          else if(ThisSum < 0)
8              ThisSum = 0;
9      }
10     return MaxSum;
11 }
```

Chapter 2 Linked List, Stacks and Queues

2.1 List的抽象数据结构(ADT)

- 包含如下操作：
 - 获得长度
 - 打印列表
 - 清空
 - 查询一个元素
 - 插入删除
 - 找到下一个
 - 找最值

2.1.1 Array List

- 需要估计好数组的最大长度，找第K个元素的时间复杂度是常数级别的，而插入和删除的时间复杂度是 $O(N)$

2.1.2 Linked List

- 插入和删除消耗常数时间，而查询第K个的时间复杂度是 $O(N)$
- ☆链表的冒泡排序

```

1 List BubbleSort(List L)
2 {
3     if(L->Next == NULL || L->Next->Next == NULL)
4         return L;
5     List p = L;
6     while(p->Next->Next != NULL) {
7         if(p->Next->key > p->Next->Next->key) {
8             List q = p->Next;
9             p->Next = q->Next;
10            q->Next = p->Next->Next;
11            p->Next->Next = q;
12        }
13    }
14 }

```

2.2 栈Stack

- 是一个LIFO的列表(Last-in-First-out)
- 支持这样一些操作
 - Push 将一个元素添加到栈的末尾
 - Pop 弹出栈的末尾元素

2.3 队列 Queue

- 是一种FIFO的列表(First-in-First-out)
- 支持如下操作
 - Enqueue 将元素添加到队列的末尾
 - Dequeue 将位于队列最前面的元素弹出

chapter 3 Trees

3.1 定义

- 树是一系列结点构成的(也可以为空) 并且包含
 - 一个根节点r
 - 若干和r相连的子树(subtree)
 - 一个N个节点的树一定有N-1条边
- 几个树中的基本概念
 - 父节点，子节点，同辈节点，叶节点
 - 节点的度数：节点子树的个数，空节点的度数为0
 - 树的度数：节点度数的最大值
 - 祖先ancestors：所有拥有通往这个节点路径的节点
 - 后代decendants：这个节点子树中的所有节点，不包含自己
 - 深度depth 从根节点到当前节点的路径长度，根节点的深度为0
 - 高度height 从叶节点到当前节点路径的最大长度，叶节点的高度为0，空节点的高度为-1



3.2 二叉树 Binary Tree

- 每一个节点最多有2个子节点的树叫做二叉树
- 二叉树的遍历：
 - 前序遍历：按照上左右的顺序递归地遍历
 - 中序遍历：按照左上右的顺序递归地遍历
 - 后序遍历：按照左右上的顺序递归地遍历

```
1 void PreOrder(Tree T)
2 {
3     if(T == NULL) return;
4     printf("%d ", T->value);
5     PreOrder(T->Left);
6     PreOrder(T->Right);
7 }
8
9 void InOrder(Tree T)
10 {
11     if(T == NULL) return;
12     InOrder(T->Left);
13     printf("%d ", T->value);
14     InOrder(T->Right);
15 }
16
17 void PostOrder(Tree T)
18 {
19     if(T == NULL) return;
20     PostOrder(T->Left);
21     PostOrder(T->Right);
22     printf("%d ", T->value);
23 }
```

- 二叉树的性质：
 - 第 i 层上最多可以用 2^{i-1} 个节点
 - 深度为 K 的二叉树最多拥有 $2^k - 1$ 个节点

3.3 二叉搜索树BST

- 定义：
 - 左子树的键值都不超过根节点
 - 右子树的键值都大于根节点
 - 左右子树都是二叉搜索树
 - 只有一个节点的树和空树是二叉搜索树
- 二叉搜索树的几个基本操作 
 - 查找一个键值：递归地进行查询，比要查的小查右边，比要查的大查左边
 - 找到最小/最大的键值：直接找最左边的或者最右边的节点
 -  插入：先查找键值，找到合适的位置在进行插入
 - 以上三个操作的时间复杂度都是 $O(h)$ 而 h 是树的高度，最好情况下 $h = O(\log N)$

Chapter 4: Heaps(Priority Queues)

- 二叉堆
 - 实现方式：一棵用数组表示的完全二叉树

- 完全二叉树的特点：1-H-1层的节点是满的，第H层的节点从左边开始依次放置没有空的，其中H是整棵树的高度
- 高度为H的完全二叉树有 $[2^H, 2^{H+1} - 1]$ 个节点
- 二叉堆的性质：
 - 对于下标为K的节点，其父节点的下标是K/2，其子节点的下标是2K和2K+1
 - 分为最小堆和最大堆两种
 - 最小堆，所有的父节点中的值都要小于子节点
 - 最大堆，所有的父节点中的值要大于子节点
- 堆的几种操作：
 - 插入：向下调整到合适的位置

```

1 void Insert(PriorityHeap H, int X) {
2     int i;
3     if(IsFull(H) == 1) return;
4     else {
5         H->size++;
6         for(i = H->size; H->Elements[i/2] > X; i=i/2)
7             H->Elements[i] = H->Elements[i/2];
8         H->Element[i]=X;
9     }
10 }

```

- 删除最小值(最小堆)：直接删除根节点，向上调整

```

1 int DeleteMin(PriorityHeap H)
2 {
3     int i, child, Min, Last;
4     if (IsEmpty(H) == 1)
5         return H->Element[0];
6     else
7     {
8         Min = H->Element[1];
9         Last = H->Element[H->size--];
10        for (i = 1; 2 * i <= H->size; i = child)
11        {
12            child = i * 2;
13            if (child != H->size && H->Element[child + 1] < H-
14                >Element[child])
15                child++;
16            if (Last > H->Element[child])
17                H->Elements[i] = H->Elements[child];
18            else
19                break;
20        }
21        H->Elements[i] = Last;
22        return Min;
23    }
24 }

```

- 一个常见的应用：找数组中第K小的元素
 - 将数组插入一个最小堆中，不断deletemin，K次之后的删除的值就是数组中第K小的元素



Chapter 5: Disjoint Set

- 等价类的定义：一个定义在集合S上的关系是一个等价关系当且仅当它具有完备性，自反性和传递性
- 并查集的操作
 - Union 操作：普通的union，根据size/height进行union
 - 负数表示这个节点是根节点，并且负数的绝对值表示其元素个数
 - 正数表示当前下标的数据的根节点的编号

- Find 操作



```
1 void Union(DisjSet S,int root1,int root 2)
2 {
3     S[root1]=root2;
4 }
5
6 int Find(DisjSet S,int x)
7 {
8     while(S[x] > 0)
9         x = S[x];
10    return x;
11 }
```

- 路径压缩：每次合并直接连接到根上面，避免路径过长

```
1 int Find(DisjSet S, int x)
2 {
3     if(S[x] <= 0) return x;
4     else return S[x] = Find(S,S[x]);
5 }
6 //Another Method
7 int Find(DisjSet S, int x)
8 {
9     int root,train,lead;
10    for(root = x, S[root] > 0;x = S[root]);
11    for(trail = x; trail != root;trail = lead)
12    {
13        lead = S[trail];
14        S[trail] = root;
15    }
16    return root;
17 }
```

- 根据大小来合并：将小的合并到大的上面去

```

1 void Union(DisjSet S,int root1,int root 2)
2 {
3     if(S[root1] <= S[root2]) //root1 is larger
4     {
5         S[root1] += S[root2];
6         S[root2] = root1; //insert root2 to root 1
7     } else {
8         S[root2] += S[root1];
9         S[root1] = root2;
10    }
11 }

```

Chapter 6: Graph

6.1 图的基本概念

- 有向图/无向图：区别在于边是否有方向
- 完全图：图中的所有节点两两相连，对于N个点有 $N(N+1)/2$ 条边
- 子图G'：顶点和边都是图G的子集
- 路径、路径的长度
 - 路径分为简单路径和环两种
- 连通分量：图G的一个最大连通子图
- 强连通有向图：任意两个顶点之间存在有向的路径可以到达
- 顶点V的度数
 - 有向图中分为出度和入度
 - 总而言之表示这个顶点所在的边的数量，其中有向图还区分出去的边和进入的边

6.2 拓扑排序

- 定义：
 - 拓扑逻辑顺序是顶点的一种线性排列，如果存在顶点i指向顶点j的边，那么拓扑排序中i一定出现在j的前面
 - 只有有向无环图才有拓扑排序，并且可能不唯一
- 实现拓扑排序的算法

```

1 #define INF 123456789
2 int TopNum[Max];
3 void TopSort(Graph G)
4 {
5     int Q[Max], rear, front, counter;
6     rear = 0;
7     front = 0;
8     counter = 0;
9     int v, w, i, j;
10    //Find the head vertices
11    for (v = 0; v < G->NV; v++)
12    {
13        if (Indegree[v] == 0)
14            Q[rear++] = v;
15    }
16    while (rear - front != 0)
17    {

```

```

18     v = Q[front++];
19     TopNum[v] = ++counter;
20     for (w = 0; w < G->nv; w++)
21     {
22         if (G[v][w] != INF)
23         {
24             Indegree[w]--;
25             if (Indegree[w] == 0)
26                 Q[rear++] = w;
27         }
28     }
29 }
30 if (counter != G->Nv)
31     return; // The graph has a circle
32 }

```

6.3 最短路径算法 Dijkstra Algorithm

- 基本的思路：
 - 在未访问的顶点中，寻找一个和目标距离最短的顶点V
 - 如果没有找到，就停止，如果找到了，将V标记位已访问
 - 对所有和V相邻的节点W，更新最多路径距离的值
- 代码实现

```

1 void Dijkstra(MGraph Graph, int dist[], vertex S)
2 {
3     //count[MAX] means the number of shortest paths, count[S]=1;
4     int visit[MAX] = {0}, i, j;
5     int n = Graph->Nv;
6     for (i = 0; i < n; i++)
7         dist[i] = INF;
8     dist[S] = 0;
9     for (;;)
10    {
11        int u = -1, v, min = INF;
12        for (i = 0; i < n; i++)
13        {
14            if (dist[i] < min && visit[i] == 0)
15            {
16                min = dist[i];
17                u = i;
18            }
19        }
20        if (u == -1)
21            break;
22        visit[u] = 1;
23        for (v = 0; v < n; v++)
24        {
25            if (Graph->G[u][v] < INF && visit[v] == 0)
26            {
27                if (dist[v] > dist[u] + Graph->G[u][v])
28                    dist[v] = dist[u] + Graph->G[u][v];
29                //count[v]=count[u];
30                //path[v]=u;

```

```

31         //if(dist[v]==dist[u]+Graph->G[u][v])
32         //count[v]+=count[u];
33     }
34 }
35 }
36 for (i = 0; i < n; i++)
37     if (dist[i] == INF)
38         dist[i] = -1;
39 }

```

- 算法的时间复杂度是 $O(|V|^2 + |E|)$

6.4 网络流 Network Flow

- 目标：在图G中找到从s出发到t的最大流，步骤如下
 - 在G中找到一条从s到t的路径
 - 将这条路径的最短边长从每一条边中减去，并将这个数值加入结果中
 - 更新图G并删除长度为0的边
 - 重复上述步骤直到不存在s到t的路径

```

1  int minlen=INF;
2  int maxflow(int s,int e,int n)
3  {
4      int i,result=0;
5      while(1)
6      {
7          if(search(s,e,n)==0)
8              return result;
9          for(i=e;i!=s;i=pre[i])
10             if(G[pre[i]][i]<minlen)
11                 minlen=G[pre[i]][i];
12          for(i=e;i!=s;i=pre[i])
13          {
14              G[pre[i]][i]-=minlen;
15              G[i][pre[i]]+=minlen;
16          }
17          result+=minlen;
18      }
19      return result;
20 }
21
22 int search(int s,int e,int n)
23 {
24     int v,i;
25     rear=0;
26     front=0;
27     memset(visit,0,sizeof(visit));
28     q[rear]=s;
29     rear++;
30     visit[s]=1;
31     while(rear-front!=0)
32     {
33         v=q[front];
34         front++;

```



```

35     for(i=0;i<n;i++)
36     {
37         if(visit[i]==0&&G[v][i]!=0)
38         {
39             q[rear]=i;
40             rear++;
41             visit[i]=1;
42             pre[i]=v;
43             if(i==e){
44                 return 1;
45             }
46         }
47     }
48 }
49 return 0;
50 }

```

- 算法的时间复杂度是 $O(|E|^2 \log |V|)$

6.5 最小生成树和DFS

- DFS的基本模式：从一个顶点V开始，遍历所有和V相邻并且未访问的顶点，需要递归地进行

```

1 void DFS(vertex v)
2 {
3     visit[v]=1;
4     int w;
5     for(w=0;w<n;w++) {
6         if(visit[w] == 0 && G[v][u] < INF)
7             DFS(w);
8     }
9 }

```

- 一个基本的应用：找连通分量

```

1 void ListComponents(Graph G)
2 {
3     for each v in G
4         if(visit[v]==0) {
5             DFS(v);
6             printf("\n");
7         }
8 }

```

- Prim算法和Kruskal算法都是贪心算法
 - 具体的算法看PPT就可以了，一种是DFS的算法，一种是BFS的算法

6.6 一些历年卷里难以判断的题目

- 图论中一些难以判断的结论（都是对的）
 - If e is the only shortest edge in the weighted graph G , then e must be in the minimum spanning tree of G .
 - If the BFS sequence of a graph is $1\ 2\ 3\ 4\ \dots$, and if there is an edge between vertices 1 and 4, then there must be an edge between the vertices 1 and 3.

- In a directed graph G with at least two vertices, if DFS from any vertex can visit every other vertices, then the topological order must NOT exist.
- Suppose that a graph is represented by an adjacency matrix. If there exist non-zero entries in the matrix, yet all the entries below the diagonal are zeros, then this graph must be a directed graph.
- 欧拉回路/欧拉路径：遍历图 G 中的每一条路径
 - 无向图存在欧拉回路，当且仅当该图的所有顶点度数都为偶数且连通
 - 有向图存在欧拉回路，当且仅当所有的出度等于入度且图要连通
- 哈密顿路径/哈密顿回路：恰好通过图 G 的每个节点一次
- Kruskal's algorithm is to grow the minimum spanning tree by adding one edge, and thus an associated vertex, to the tree in each stage. (**FALSE**)
- 关于拓扑逻辑排序
 - If a graph has a topological sequence, then its adjacency matrix must be triangular.
 - 错的，在无向图中不一定
 - If V_i precedes V_j in a topological sequence, then there must be a path from V_i to V_j .
 - 错的，不一定有
 - If the adjacency matrix is triangular, then the corresponding directed graph must have a unique topological sequence.
 - 错的，可以举出反例
 - In a DAG, if for any pair of distinct vertices V_i and V_j , there is a path either from V_i to V_j or from V_j to V_i , then the DAG must have a unique topological sequence.
 - 对的

Chapter 7: Sort

7.1 插入排序

- 最好的情况是 $O(N)$ 最坏的情况是 $O(N^2)$
 - N 个元素中的平均inversion个数为 $I = \frac{N(N+1)}{4}$ 并且时间复杂度为 $O(I + N)$

```

1 void InsertionSort(int a[], int n)
2 {
3     int j, p, tmp;
4     for (p = 1; p < n; p++)
5     {
6         tmp = a[p];
7         for (j = p; j > 0 && a[j - 1] > tmp; j--)
8             a[j] = a[j - 1];
9         a[j] = tmp;
10    }
11 }

```

7.2 希尔排序

- 定义一系列间隔，每次按照间隔进行排序，并且每一轮的间隔不断减小，直到变成1

```

1 void ShellSort(int a[], int n)
2 {
3     int i, j, increment, tmp;

```

```

4   for (increment = n / 2; increment > 0; increment /= 2)
5   {
6       for (i = increment; i < n; i++)
7       {
8           tmp = a[i];
9           for (j = i; j >= increment; j -= increment)
10          {
11              if (tmp < a[j - increment])
12              {
13                  a[j] = a[j - increment];
14              }
15              else
16                  break;
17          }
18          a[j] = tmp;
19      }
20  }
21 }

```

- 最差的时间复杂度依然是平方级别的

7.3 堆排序

- 用建堆+delete max操作来进行排序，时间复杂度为 $O(N \lg N)$

```

1  #define leftchild(i) (2 * (i) + 1)
2  //different from traditonal heap,a[] start from index 0;
3  void PercDown(int a[], int i, int n)
4  {
5      int child, tmp;
6      for (tmp = a[i]; leftchild(i) < n; i = child)
7      {
8          child = leftchild(i);
9          if (child != n - 1 && a[child + 1] > a[child])
10             child++;
11             if (a[child] > tmp)
12                 a[i] = a[child];
13             else
14                 break;
15         }
16         a[i] = tmp;
17     }
18
19 void HeapSort(int a[], int n)
20 {
21     int i;
22     for (i = n / 2; i >= 0; i--) //build heap
23         PercDown(a, i, n);
24     for (i = n - 1; i > 0; i--)
25     {
26         int t = a[0];
27         a[0] = a[i];
28         a[i] = t;
29         PercDown(a, 0, i);
30     }

```

7.4 归并排序

- 将数组分成两路进行排序然后将两个数组合并成一个，时间复杂度是 $O(N \lg N)$

```

1 void MSort(int a[], int tmp[], int left, int right)
2 {
3     int center = (left + right) / 2;
4     if (left < right)
5     {
6         MSort(a, tmp, left, center);
7         MSort(a, tmp, center + 1, right);
8         Merge(a, tmp, left, center + 1, right);
9     }
10 }
11
12 void MergeSort(int a, int n)
13 {
14     int tmp[Max];
15     MSort(a, tmp, 0, n - 1);
16     //need O(n) extra space
17 }
18
19 void Merge(int a[], int tmp[], int lpos, int rpos, int rightend)
20 {
21     int i, leftend, num, tmppos;
22     leftend = rpos - 1;
23     tmppos = lpos;
24     num = rightend - lpos + 1;
25     while (lpos <= leftend && rpos <= rightend)
26     {
27         if (a[lpos] <= a[rpos])
28             tmp[tmppos++] = a[lpos++];
29         else
30             tmp[tmppos++] = a[rpos++];
31     }
32     while (lpos <= leftend)
33         tmp[tmppos++] = a[lpos++];
34     while (rpos <= rightend)
35         tmp[tmppos++] = a[rpos++];
36     for (i = 0; i < num; i++, rightend--)
37         a[rightend] = tmp[rightend];
38 }

```

- 一种迭代式的实现方式

```

1 void merge_pass(ElementType list[], ElementType sorted[], int N, int
   length);
2
3 void merge_sort(ElementType list[], int N)
4 {
5     ElementType extra[MAXN]; /* the extra space required */
6     int length = 1;          /* current length of sublist being merged */

```

```

7     while (length < N)
8     {
9         merge_pass(list, extra, N, length); /* merge list into extra */
10        output(extra, N);
11        length *= 2;
12        merge_pass(extra, list, N, length); /* merge extra back to list */
13        output(list, N);
14        length *= 2;
15    }
16 }
17
18 void merge_pass(ElementType list[], ElementType sorted[], int N, int
length)
19 {
20     int i, j;
21     for (i = 0; i < N; i += 2 * length)
22     {
23         int x, y, z;
24         x = i;
25         y = i + length;
26         z = x;
27         while (x < i + length && y < i + 2 * length && x < n && y < n)
28         {
29             if (list[x] < list[y])
30                 sorted[z++] = list[x++];
31             else
32                 sorted[z++] = list[y++];
33         }
34         if (x < i + length)
35             while (x < i + length)
36                 sorted[z++] = list[x++];
37         if (y < i + 2 * length)
38             while (y < i + 2 * length)
39                 sorted[z++] = list[y++];
40     }
41 }

```

7.5 快速排序：已知的最快排序方式

- 需要选择一个pivot，每次将比pivot小的放到左边，大的放到右边
 - 最坏的复杂度依然是平方复杂度，但是最优的复杂度是 $O(N \lg N)$ ，并且平均复杂度是 $O(N \lg N)$
 - 一个结论：基于比较的排序方法的时间复杂度至少是 $O(N \lg N)$ 级别的

```

1 int median3(int a[], int left, int right)
2 {
3     int center = (left + right) / 2;
4     if (a[left] > a[center])
5         swap(a[left], a[center]);
6     if (a[left] > a[right])
7         swap(a[left], a[right]);
8     if (a[center] > a[right])
9         swap(a[right], a[center]);
10    //these steps makes a[left]<=a[center]<=a[right]

```

```

11     swap(a[center], a[right - 1]) //hide pivot
12     //only need to sort a[left+1]~ a[right-2]
13     return a[right - 1];
14 }
15
16 void Qsort(int a[], int left, int right)
17 {
18     int i, j, pivot;
19     pivot = median3(a, left, right);
20     i = left, j = right - 1;
21     while (1)
22     {
23         while (a[++i] < pivot)
24         {
25         }
26         while (a[--j] > pivot)
27         {
28         }
29         if (i < j)
30             swap(a[i], a[j]);
31         else
32             break;
33     }
34     swap(a[i], a[right - 1]);
35     Qsort(a, left, i - 1);
36     Qsort(a, i + 1, right);
37 }
38
39 void QuickSort(int a[], int n)
40 {
41     Qsort(a, 0, n - 1);
42 }

```

7.6 桶排序，基数排序

- 桶排序：时间换空间的经典方法
- 基数排序：用Least Significant Digit first(LSD)的方法进行排序
 - 具体的可以看PPT怎么进行操作

7.7 总结：

排序算法的稳定性

- 不稳定的排序：堆排序，快速排序，希尔排序，直接选择排序
- 稳定的排序：基数排序，冒泡排序，插入排序，归并排序

数组排序呈现的特征

- 堆排序：一般没什么特征
- 归并排序：排序中会出现连续若干个数字已经排好了顺序
- 快速排序：有一些数，前面的都比这个数小，后面的都比他大，具体要看run了多少次
- 选择排序：每一次都会选出最大或者最小的数排在最后应该出现的位置

Chapter 8: Hash

8.1 定义和基本概念

- 哈希函数 $f(x)$ 的结果表示 x 在哈希表中的位置
- T 表示不同的 x 的个数
- n 表示哈希表中的位置个数
- b 表示哈希表中bucket的个数
- s 表示槽的个数
- identifier density $= n / T$
- loading density $\lambda = n / sb$
- collision 冲突：两个值被放到了同一个bucket中
- overflow 溢出：bucket超过了承载的上限

8.2 解决collision

- 需要找到另一个空的地方来放置待放入的元素
- 常用的方法：
 - 线性探查，平方探查，看PPT和做题就可以理解两种方法是怎么操作的