1. C语言

1.1 指针 & 引用

```
int n = 1;
int *p = &n;  // 指针
int &a = n;   // 引用
```

```
typedef int *ptrInt;
```

1.2 运算符优先级

1. 数组访问、结构体访问、后缀自增
2. 前缀自增、非、取地址、解引用（从右到左）
3. 乘除摸加减、左移右移
4. 关系、位、逻辑
5. 三元条件、赋值（从右到左）
6. 逗号

例题

```
int a = 3, b = 2, c = 5, d = 4, x = 0, y = 0;
(x = a > b) || (y = c > d); // x,y = 1,0
```

```
int a[] = {2, 4, 6, 8};
int *p = a;
int b = *p++, c = *p; // b,c = 2,4
```

```
int a = 1, b = 2, c;
c = ++b = a++;  // a,b,c = 2,1,1
```

```
int a = 5, b = 10;
a *= 1 & 2 > 1 && (b = 0) ? true : false; // a,b = 0,0
```

```
c = (a = 6, b = a--); // a,b,c = 5,6,6
```

```
int a = 5, b = 4, c;
c = b++ && a == b;  // a,b,c = 5,5,1
```

2. 绪论

2.1 数据结构

- 逻辑结构
  - 线性结构
    - 线性表
    - 栈和队列
    - 字符串
  - 非线性结构
    - 集合结构
    - 树结构
    - 图结构（网状结构）
- 物理结构（存储结构）
  - 顺序存储结构
  - 链式存储结构

2.2 算法

- 特性：有穷性、确定性、可行性、输入、输出
- 评价标准：正确性、可读性、健壮性、高效性
- 效率度量：时间复杂度、空间复杂度

时间复杂度的计算

```
x = 90, y = 100;
while (y > 0)
  if (x > 100) x -= 10, y--;
  else x++;
```

```
x = 0;
for (i = 1; i < n; i++)
  for (j = 1; j <= n - i; j++)
    x++;
```

```
i = 1;
while (i <= n)
  i *= 3;
```

```
x = n, y = 0;
while (x >= (y + 1) * (y + 1))
  y++;
```

以上时间复杂度 $O(1)$，$O(n^2)$，$O(\log_3 n)$，$O(\sqrt{n})$

语句频度 & 时间复杂度

3. 线性表

3.1 顺序表

常用于随机存取，存储密度 100%

3.2 链表

常用于增删改，若单链表指针域、数据域大小一致，则存储密度 50%

单链表

易混淆概念

- 头指针
- 头结点
- 首元结点

头结点作用

- 数据元素操作相同
- 头指针都非空

循环链表

设尾指针有时可简化操作，如合并线性表

3.3 有序表合并

- 合并后按原序排列
  - 最好时间复杂度 $O(\min(m,n))$
  - 最坏时间复杂度 $O(m+n)$
- 合并后按逆序排列
  - 最好时间复杂度 $O(m+n)$
  - 最坏时间复杂度 $O(m+n)$

当 m,n 非常接近或相差很大，$O(m+n)$ 都接近于 $O(\max(m,n))$

4. 栈和队列

4.1 栈

后进先出，用于数制转换、括号匹配、表达式求值

4.2 队列

先进先出，用于舞伴问题

循环队列

解决“假溢出”问题

队列容量 SIZE

```
Q.front == Q.rear;  // 队空
(Q.rear + 1) % SIZE == Q.front; // 队满
n = (Q.rear + SIZE - Q.front) % SIZE; // 当前元素个数
```

5. 串和数组

5.1 串

数据元素受限的线性表

模式匹配

- *BF* $O(m*n)$
- *KMP* $O(m+n)$

5.2 KMP

next 数组

- $\texttt{next}[1]=0$
- $\texttt{next}[j]$ 为 $P_1\cdots P_{j-1}$ 最长公共前后缀长度 $+1$

nextval 数组

- $\texttt{nextval}[1]=0$
- $\texttt{nextval}[j]=\begin{cases}\texttt{next}[j], & P_j\neq P_{\texttt{next}[j]}\\ \texttt{nextval}[\texttt{next}[j]], & P_j=P_{\texttt{next}[j]}\end{cases}$

例题

j	1	2	3	4	5	6	7
P	a	b	a	b	a	a	a
$\texttt{next}[j]$	0	1	1	2	3	4	2
$\texttt{nextval}[j]$	0	1	0	1	0	4	2

5.3 数组

存储地址计算

- 优先行存储
- 优先列存储

6. 树

术语

- 结点的度：其子树个数
- 树的度：Max 结点的度
- 树的深度（高度）：从 1 开始算

树的存储结构

- 双亲表示法

- 孩子表示法
- 孩子兄弟法（左孩子，右兄弟）

6.1 二叉树

二叉树是逻辑结构，二叉链表是物理结构

$\mathbf{n}_0 = \mathbf{n}_2 + \mathbf{1}$

完全二叉树

具有 $\mathbf{n}$ 个结点的完全二叉树的深度为 $\lfloor \log_2 \mathbf{n} \rfloor + \mathbf{1}$

二叉树的遍历

- 先序遍历
- 中序遍历
- 后序遍历

根据（先序&中序）或（中序&后序）序列构建唯一二叉树。*P125*

波兰式（前缀） & 逆波兰式 *P122*

6.2 Huffman 树

术语

- 树的路径长度：$\sum$ 结点路径长度
- 结点的带权路径长度：结点路径长度 $\times$ 权
- 树的带权路径长度 *WPL*：$\sum$ 叶子的带权路径长度
- Huffman 树：*WPL* 最小的二叉树

Huffman 树的构造

1. 选两棵权值最小的树合成，权值记为 a, b
2. 合成的二叉树根结点权值为 $a + b$
3. 重复直到仅剩一棵树

6.3 Huffman 编码

最优前缀码

Huffman 编码的构造

1. 标记 Huffman 树的左分支为 0，右分支为 1
2. 根结点到叶子节点路径上的 01 序列，就是该结点的最优前缀码

7. 图

术语

- 度、入度、出度：有向图的度 $=$ 出度 $+$ 入度
- 简单路径、简单回路
- 网：带权的图
- 连通图、连通分量：无向图
- 强连通图、强连通分量：有向图

7.1 图的存储

邻接矩阵表示唯一，邻接表表示不唯一

邻接矩阵

- 无权图由 $0, 1$ 构成
- 网由 w, ∞ 构成

邻接表

由顶点顺序表、邻接点链表构成

优缺点的对比

	邻接矩阵	邻接表
判断边是否存在	✓	
计算顶点的度	✓	
增删顶点	✓	
统计边数		✓
空间复杂度		✓

7.2 最小生成树

Prim

1. 选一个点 v 加入 U
2. 多选一个点 $v \notin U$ 加入，使增加的权值最小
3. 重复

Kruskal

1. 将边按权值排序
2. 选取最短的边
   - 形成回路，丢弃
   - 不形成回路，加入
3. 重复

7.3 拓扑排序

用于有向无环图，是关键路径算法的基础

AOV-网构造拓扑序列

1. 选入度为 0 的顶点，并输出
2. 删除该顶点及其出边
3. 重复，若输出顶点少于总顶点，则存在环

7.4 图的算法比较

DFS & BFS

- 邻接矩阵 $O(n^2)$
- 邻接表 $O(n+e)$

最小生成树 MST

- *Prim* $O(n^2)$
- *Kruskal* $O(e\log_2 e)$

最短路

- *Dijkstra* $O(n^2)$
- *Floyd* $O(n^3)$

8. 查找

8.1 线性表

查找算法比较

- 顺序查找 $O(n)$
- 折半查找 $O(\log_2 n)$
- 分块查找 $O(\sqrt{n})$

判定树

查找成功至少 $\lfloor \log_2 \mathbf{n} \rfloor$ 次

8.2 树表

二叉排序树

- 左子树都小于根，右子树都大于根
- 中序遍历可得到单增序列
- 无序序列生成二叉查找树

8.3 散列表

术语

- 散列表长 m
- 散列函数 $H(key)$
- 冲突 $a \neq b, H(a) = H(b)$，其中 a 与 b 为同义词

冲突的处理

线性探测会产生 “二次聚集”

- 开放地址法 $H_i = (H(key) + d_i)\%m$
  - 线性探测法 $d_i = 1, 2, 3, \cdots, m-1$
  - 二次探测法 $d_i = 1^2, -1^2, 2^2, -2^2, \cdots, +k^2, -k^2 (k \leq m/2)$
  - 伪随机探测法
- 链地址法 *P224*

散列表的平均查找长度 **ASL**

- 查找成功 $\mathbf{ASL_{succ}}$：$\sum \mathbf{p_{succ}} \times \mathbf{cnt_i}$
- 查找失败 $\mathbf{ASL_{unsucc}}$：$\sum \mathbf{p_{unsucc}} \times \mathbf{cnt_i}$

9. 排序

- 插入排序
  - 直接插入排序
  - 折半插入排序
  - 希尔排序
- 交换排序
  - 冒泡排序
  - 快速排序
- 选择排序
  - 简单选择排序
  - 堆排序
- 归并排序
- 基数排序
- 外部排序

9.1 插入排序

当“记录个数较少”、“序列基本有序”时，`直接插入排序`效率较高

- 直接插入排序：从后向前查找，插入到合适位置
- 折半插入排序：折半查找，插入到合适位置
- 希尔排序：分组插入，从“减少记录个数”、“序列基本有序”改进直接插入

9.2 交换排序

冒泡排序

1. 两两比较相邻的记录，逆序则交换位置
2. 若一趟排序没有交换过记录，则完成排序

快速排序

1. 每趟排序从右边开始比较
2. 逆序或相等则交换位置，之后换另一边继续
3. 若 `low` $<$ `high` 则继续，否则进行下一趟排序
4. 若待排序列中只有一个记录，则完成排序

> 对序列 49 38 65 97 76 13 27 $\overline{49}$ 排序，结果如下：
>
> 第 1 趟：{27 38 13} 49 {76 97 65 $\overline{49}$}
>
> 第 2 趟：{13} 27 {38} 49 {76 97 65 $\overline{49}$}
>
> 第 3 趟：13 27 38 49 {$\overline{49}$ 65} 76 {97}
>
> 第 4 趟：13 27 38 49 $\overline{49}$ {65} 76 97

9.3 选择排序

堆排序

1. 建初堆：从最后一个非终端结点开始，堆调整使其是大根堆
2. 交换：堆顶元素和最后一个结点交换，形成新堆，调整成大根堆
3. 堆调整：该结点之后的结点必满足大根堆

9.4 归并排序

2-路归并

> 对序列 [49] [38] [65] [97] [76] [13] [27] 排序，结果如下：
>
> 第 1 趟：[38 49] [65 97] [13 76] [27]
>
> 第 2 趟：[38 49 65 97] [13 27 76]
>
> 第 3 趟：[13 27 38 49 65 76 97]

9.5 分配类排序

不是 关键字比较 ，而是通过 “分配”与“收集” 实现排序

9.6 排序性能比较

排序选取

- 记录较少
  - 基本有序：直接插入、冒泡
- 记录较多

- 随机：快排
- 基本有序但不稳定：堆排序
- 基本有序且稳定：归并

不稳定排序

希尔排序、快速排序、堆排序

时间复杂度较小

- 快排 $O(n\log_2 n)$
- 堆排序 $O(n\log_2 n)$
- 归并 $O(n\log_2 n)$
- 希尔 $O(n^{1.3})$
- 其他 $O(n^2)$

空间复杂度较大

- 归并 $O(n)$
- 快排 $O(\log_2 n)$
- 其他 $O(1)$

10. 题目

10.1 选填

C语言

函数内定义的是`局部变量`，只在`定义它的函数范围内`有效

结构化程序设计中，三种基本结构是`顺序结构、选择结构、循环结构`

C语言以`函数`为程序的基本单位，变量必须`先`定义`后`使用

C语言源程序要经过`预处理、编译、汇编、链接`才能执行

C语言基本数据类型`int, char, float, double, short, long`

按照变量的作用域可将变量划分为`全局变量、局部变量`

C语言程序总是从`主函数`开始执行

数据结构

衡量算法的两个主要指标：`时间复杂度、空间复杂度`

树的存储结构有三种表示方法：`双亲表示法、孩子表示法、孩子兄弟表示法`

图进行广搜常用`队列`实现算法，进行深搜常用`栈`实现算法

一个递归算法必须包括：`终止条件、递归部分`

栈和队列都是`限制存储点的线性结构`

为便于确定顶点的入度，可以建立`逆邻接表`

10.2 编程题

1. 有序表的合并

将两个递增的有序链表 La 和 Lb 合并为一个递增的有序链表 Lc。不开辟新的存储空间，不允许有重复的数据。

```
void MergeList(LinkList &La, LinkList &Lb, LinkList &Lc)
{
  pa = La->next;
  pb = Lb->next;
  Lc = pc = La;
  while (pa && pb)
  {
    if (pa->data < pb->data)
    {
      pc->next = pa;
      pc = pa;
      pa = pa->next;
    }
    else if (pa->data > pb->data)
    {
      pc->next = pb;
      pc = pb;
      pb = pb->next;
    }
    else
    {
      pc->next - pa;
      pc = pa;
      pa = pa->next;
      q = pb->next;
      delete pb;
      pb = q;
    }
  }
  pc->next = pa ? pa : pb;
  delete Lb;
}
```

将两个非递减的有序链表 La 和 Lb 合并为一个非递增的有序链表 Lc。不开辟新的存储空间，允许有重复的数据。

```
void MergeList(LinkList &La, LinkList &Lb, LinkList &Lc)
{
  pa = La->next;
  pb = Lb->next;
  Lc = pc = La;
  Lc->next = NULL;
  while (pa || pb)
  {
    if (!pa)
      q = pb, pb = pb->next;
    else if (!pb)
      q = pa, pa = pa->next;
    else if (pa->data <= pb->data)
      q = pa, pa - pa->next;
    else
      q = pb, pb = pb->next;
    q->next = Lc->next;
    Lc->next = q;
  }
  delete Lb;
}
```

2. 杂

将链表逆转，不开辟新的存储空间。

```
void Inverse(LinkList &L)
{
  p = L->next;
  L->next = NULL;
  while (p != NULL)
  {
    q = p->next;
    p->next = L->next;
    L->next = p;
    p = q;
  }
}
```

3. 递归

已知 *f* 为单链表的表头指针，链表中仅存储整型数据，用递归算法实现

（1）求链表中的最大整数

（2）求链表的结点个数

（3）求所有整数的平均值

```
int GetMax(LinkList p)
{
  if (!p->next)
    return p->data;
  else
  {
    int max = GetMax(p->next);
    return p->data >= max ? p->data : max;
  }
}

int GetLength(LinkList p)
{
  if (!p->next)
    return 1;
  else
    return GetLength(p->next) + 1;
}

double GetAverage(LinkList p, int n)
{
  if (!p->next)
    return p->data;
  else
  {
    double ave = GetAverage(p->next, n - 1);
    return (ave * (n - 1) + p->data) / n;
  }
}
```

4. 二叉树

以二叉链表作为二叉树的存储结构，编写算法

(1) 统计二叉树的叶子结点个数
(2) 判别两棵树是否相等
(3) 交换二叉树每个结点的左孩子和右孩子
(4) 设计二叉树的双序遍历算法
(5) 计算二叉树的最大宽度
(6) 用按层次顺序遍历二叉树的方法，统计树中具有度为1的结点数目
(7) 求任意二叉树中第一条最长的路径长度，并输出此路径上各结点的值
(8) 输出二叉树中从每个叶子结点到根结点的路径