C程序设计专题考试内容整理

程序结构*Program Structure*
- 存储类关键字*storage class specifier*
  - 基本概念
  - 典型例题
- 函数指针
  - 定义函数指针*function pointer*
  - 返回类型为函数指针的函数原型*function prototype*
  - 函数参数为函数指针
  - `[] () *`等运算符优先级
- typedef
- #define
- 标准头文件结构

————————————

算法设计

递归
- 递归的概念
- 递归要素
- 递归分类
  - 分类方法一
  - 分类方法二
- 递归举例
  - （1）阶乘
  - （2）fibonacci数列
  - （3）最大公约数/欧几里得算法/gcd
  - （4）汉诺塔 Hanoi Towers
  - （5）牛顿迭代法计算平方根
  - （6）快速幂计算
  - （7）整数顺序/逆序输出

搜索与排序 *Searching&Sorting*

Search
- Linear Search *线性搜索*
- Binary Search *二分搜索*
  - 使用前提：**已排序的数据**
  - 数学模型
  - 递归实现
  - 迭代实现

Sort
- Select *选择排序*
  - 排序思路（递归）
  - 代码实现
  - 时间复杂度
  - 改进思路
- Bubble Sort *冒泡排序*
  - 排序思路
  - 代码实现
  - 时间复杂度
- Insert Sort *插入排序*
  - 排序思路

数据结构 *Data Structure*

结构*Struct*

结构声明和变量定义

- 结构体类型不占内存，定义变量占内存

在c语言中，不允许有常量的数据类型是（结构 ）

若程序有以下的说明和定义：

```
struct abc
{ int x;char y; } //没加;
 struct abc s1,s2;
```

则会发生的情况是（）

- 嵌套结构

如果结构变量s中的生日是“1984年11月11日”，下列对其生日的正确赋值是（）。

```
struct student
{
  int no;
  char name[20];
  char sex;
  struct{
    int year;
    int month;
    int day;
  }birth;
};
struct student s;
```

`->` `.`和其他运算符优先级

- 单目运算符 `[] () . ->`优先级最高，这四个结合律左到右
- 其他单目右到左

For the following declarations of structure and variables, the correct description of the expression `*p->str++;` is __.

```
struct {
    int no;
    char *str;
} a={1,"abc"}, *p=&a;
```

```
++ acts on the pointer str
```

向函数传递参数

- 可以传递整个结构
- 可以传递结构指针
- 可以传递结构成员

以下`scanf`函数调用语句中不正确的是__。

```
struct pupil {
    char name[20];
    int age;
    int sex;
} pup[5], *p=pup;
```

A. `scanf("%s", pup[0].name);` *数组名本身是一指针*

B. `scanf("%d", &pup[0].age);`

C. `scanf("%d", p->age);` `p->age` *是一个int*

D. `scanf("%d", &(p->sex));`

> `scanf(format,指针)`

结构赋值

- 可以两个结构赋值
- 可以结构内成员赋值
- 注意数组和指针的区别

For the following declarations, assignment expression __ is not correct.

```
struct Student {
    long num;
    char name[20];
} st1, st2={101, "Tom"}, *p=&st1;
```

```
A.  st1 = st2
```

```
B.  p->name = st2.name √(数组不等于指针，不能直接复制)
```

```
C.  p->num = st2.num
```

```
D.  *p=st2
```

结构数组指针

```
The value of expression *((int *)(p+1)+2) is __.

static struct {
    int x, y[3];
} a[3] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}}, *p;
p = a+1;
```

```
After executing the following code fragment, the value of variable z is ______.

static struct{ int x, y[3];
} a[3]={{0},{5,6,7},{10,12}}, *p=a+3; int z;
z=*((int *)(p-1)-3);
```

链表*Linked List*

基本链表类型

单向单头链表

- 数据结构

```
typedef struct _Node {
  int value;
  struct _Node *next;
} Node;

typedef struct {
  Node *head;//仅有头指针
} List;

List list;
List *plist = &list;
```

- 头插法

```
void insert_head (List *plist,int x) {
    Node *p = (Node *) malloc(sizeof(Node));
    p->value = x;
    p->next = plist->head;
    plist->head = p;
}
```

- 尾插法

```
//appendtail:Boundary-空表
```

```
void append_tail (List *plist,int x) {
    Node *tail = (Node *)malloc(sizeof(struct _Node));
    tail->value = x;
    tail->next = NULL;
    if (plist->head) {
        Node *p = plist->head;
        for (;p->next;p=p->next) ;
        p->next = tail;
    }
    else {
        plist->head = tail;
    }
}
```

- 按值删除所有结点

```
void list_remove(List *list, int value) {
    Node *p=list->head,*q=list->head;
    while(p) {
        if (p->value == value) {
            if (list->head == p) {//删除头结点
                list->head = q = p->next;
                free(p);
                p = q;
            } else {//删除中间结点
                q->next = p->next;
                free(p);
                p = q->next;
            }
        } else {//不删除结点
            q = p;
            p = p->next;
        }
    }
}
```

- 遍历*iterate*

查找实质是遍历

```
void list_iterate(List *list, void (*func)(int v)) {
    for (Node*p = list->head;p;p=p->next) {
        func(p->data);
    }
}
```

- 销毁

```
void clear (List *plist) {
    for (Node *p = plist->head,*q = NULL;p;p = q) {
        q = p->next;
        free(p);
    }
}
```

单向双头链表

- 数据结构

```
typedef struct _node Node;
typedef struct {
    Node *head;
    Node *tail;//比单头链表多尾指针
} List;
```

- 创建链表（多尾指针）

```
List list_create() {
    List list;
    list.head = list.tail = NULL; //头尾指针置为NULL
    return list;
}
```

- 尾插法（有尾指针，尾插方便许多）

```
void list_append(List *list, int v) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->data = v,p->next = NULL;//创建并初始化新结点
    if (list->tail) {//情况1:链表非空
        list->tail->next = p;//变更尾指针位置
        list->tail = p;
    } else {//情况2:空链表
        list->head = list->tail = p;//变更头尾指针位置
    }
}
```

- 头插法（空链表时多维护尾指针）

```
void list_insert(List *list, int v) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->data = v,p->next = NULL; //创建并初始化新结点
    if (list->head) {//情况1:链表非空
        p->next = list->head;//变更头指针位置
        list->head = p;
    } else {//情况2:空链表
        list->head = list->tail = p; //变更头尾指针位置
    }
}
```

- 按值删除某结点（多维护尾指针。分两大类，四小种）

```
void list_remove(List *list, int v) {
if (list->head && list->head != list->tail) {//假定链表非空且至少有两个结点
/*以下这段代码实际上也可以放在for循环中，没必要单独讨论该情况*/
        if (list->head->data == v) {//情况1:如果要删除的结点是头结点
            Node *p = list->head;
            list->head = p->next;//改变头指针位置
            free(p);
            return;
        }
        for (Node *p = list->head->next,*q = list->head;p;q = p,p = p->next) {
            if (p->data == v) {
                if (p == list->tail) {//情况2:如果要删除尾结点
                    list->tail = q;
                    q->next = NULL;//这里很重要，使尾结点后继为NULL
                    free(p);
                } else {
                    q->next = p->next;//情况3:中间结点
                    free(p);
                    p = q->next;
                }
                break;
            }
        }
    }
}
```

单向有哨兵

- 创建哨兵链表（头结点）

```
void create_head(List *plist) {
    Node *p = (Node*)malloc(sizeof(Node));
    p->value=0,p->next=NULL;
    plist->head = p;
}
```

- 头插法（实际的头指针是`head->next`）

```
void insert_head(List *plist,int x) {
    Node *p = (Node*) malloc(sizeof(Node));
    p->value = x,p->next = NULL;
    p->next = plist->head->next;
    plist->head->next = p;
    /*比较一下无头结点的写法
    p->next = plist->head;
    plist->head = p;
    */
}
```

- 尾插法（不用考虑空表情况）

```
void append_tail(List *plist,int x) {
    Node *tail = (Node*)malloc(sizeof(Node));
    tail->value=x,tail->next=NULL;
    Node*p=plist->head;
    for (;p->next;p=p->next) ;
    p->next = tail;
    /*比较无头结点，空表头指针为空，需要单独考虑（而设置了哨兵后，即使是空表，头指针也不为空）
    if (plist->head) {
        Node *p = plist->head;
        for (;p->next;p=p->next) ;
        p->next = tail;
    }
    else {
        plist->head = tail;
    }
```

- 删除（不用单独考虑删除头指针情况）

```
void remove(List *plist,int x) {
    for (Node*p = plist->head,*q = plist->head->next;p;q = p,p = p->next) {
        if (p->value == x) {
            q->next = p->next;
            free(p);
        }
    }
}
```

基本操作及其复杂度

创建链表 - O(1)

```
List head;
head = NULL;
```

头插法 - O(1)

```
Node *p = (Node *)malloc(sizeof(struct Node));
p->data = val,p->next = NULL;//create a node

p->next = head;
head = p;
```

尾插法 - 单头O(n),双头O(1)

```
Node *p = (Node *)malloc(sizeof(struct Node));
p->data = val,p->next = NULL;

if (tail) {
  tail->next = p;
  tail = p;
}
else {
  head = tail = p;
}
```

按值删除所有结点 - O(n)

```
void list_remove(List *list, int value) {
    Node *p=list->head,*q=list->head;
    while(p) {
        if (p->value == value) {
            if (list->head == p) {//删除头结点
                list->head = q = p->next;
                free(p);
                p = q;
            } else {//删除中间结点
                q->next = p->next;
                free(p);
                p = q->next;
            }
        } else {//不删除结点
            q = p;
```

```
            p = p->next;
        }
    }
}
```

按值/按位置搜索某一结点 (Linear) - O(n)

```
int loc = 0;
for (Node *p=head;p;p = p->next) {
  if (p->data == x) {
    return loc;
  }
  loc++;
}
```

销毁 - O(n)

```
for (Node *p = head,*q;p;p = q){
  q = p->next;
  free(p);
}
```

- 注意`->`左边不能是`NULL`

简单程序及其复杂度

奇偶结点重组-*19A*

- 要求重排后`1-3-5-2-4`
- 要求空间复杂度为`O(1)`，即利用原有结点，至多创建了一个哨兵结点
- 已知`CreateNode(int data)`

```
Linklist Rearrange(Linklist head) {
  ListNode* current = head;
  Linklist even = CreateNode(0);
  ListNode* even_tail = even;
  ListNode* odd_tail = NULL;
  int even = 0;
  while (odd_tail) {
    if (!even) //current指向奇数
    {
      odd_tail = current;
    }
    else {
      even_tail->next = current;
      even_tail = current;//尾插法
      odd_tail->next = current->next;
    }
```

```
    current = current->next;
    even = 1 - even;
  }
  even_tail->next = NULL;
  current->next = even->next;//不是even--相当于一个哨兵结点
  return head;
}
```

分离奇偶值结点

- 空间复杂度O(1)，利用原结点
- 十分类似于上题

```
struct ListNode *getodd( struct ListNode **L ) {
    Node *odd = (Node*)malloc(sizeof(struct ListNode));
    odd->data = 0,odd->next = NULL;
    Node *odd_tail = odd;
    Node *cur = *L;
    Node *even_tail = NULL;
    while (cur) {
        if (cur->data%2) {
            odd_tail->next = cur;
            odd_tail = cur;
            if (cur == *L) {//判断第一个是否为奇数
                *L = (*L)->next;
            }
            else {
                even_tail->next = cur->next;
            }
        }
        else {
            even_tail = cur;
        }
        cur = cur->next;
    }
    return odd->next;//odd本身是哨兵结点
}
```

链表实现Merge - 时间O(n)，空间O(1)

- 数组merge，空间复杂度O(n),必须要新开辟`b[n]`
- 与数组显著不同的是，链表实现利用原结点，只新建了哨兵结点

```
typedef struct Node Node;
List Merge( List L1, List L2 ) {
    List merge = (List)malloc(sizeof(Node));
    merge->Data = 0,merge->Next = NULL;
    Node *merge_tail = merge;
    Node *tail1 = L1->Next,*tail2 = L2->Next;
```

```
    while (tail1 && tail2) {
        if (tail1->Data < tail2->Data) {
            merge_tail->Next = tail1;
            merge_tail = tail1;//尾插法
            tail1 = tail1->Next;
        } else {
            merge_tail->Next = tail2;
            merge_tail = tail2;//尾插法
            tail2 = tail2->Next;
        }
    }
    merge_tail->Next = tail1 ? tail1 : tail2;
    L1->Next = NULL,L2->Next = NULL;
    return merge;
}
```

链表逆置 - O(n)

- 利用头插法

```
typedef struct ListNode Node;
void insert_head(Node **head,int x) {
    Node *p = (Node*)malloc(sizeof(Node));
    p->data=x,p->next=NULL;
    p->next=*head;
    *head = p;
}
struct ListNode *reverse( struct ListNode *head ){
    Node *head2 = NULL;
    for (Node*p = head;p;p=p->next) {
        insert_head(&head2,p->data);
    }
    return head2;
}
```

在递增链表中插入新结点 - O(n)

- 插入排序的一趟，链表实现

```
List Insert( List L, ElementType X ) {
//思路：先定位最后一个比X小的结点q（while循环）即q->Data<X && q->Next->Data>X，然后把X插在该结点后面
    List p = (List) malloc(sizeof(struct Node));
    p->Data = X,p->Next = NULL;//创建新结点
    List q = L;
    if (L) {
        while (q->Next && q->Next->Data < X) q = q->Next;
        p->Next = q->Next;//在链表中间插入一结点
        q->Next = p;
```

```
    } else {
        L = p;//特殊情况
    }
    return L;
}
```

用单向链表完成多项式计算

- 因式分解

```
struct node {
  int coe;
  int exp;
  struct node *next;
} ;
typedef struct node node;
int polynomial(node *h,int x) {
  if (h == NULL) return 0;
  int result = 0;
  int last = h->exp,cur;
  for (node *p = h;p;last = cur,p = p->next)
    cur = p->exp;
    for (int i=last;i>cur;i--) result *= x;
    result += p->coe;
  }
  for (int i=last;i>0;i--) result *= x;
  return result;
}
```

循环链表之猴子选大王

- 与单向链表差别在`tail->next = head`

```
linklist *CreateCircle( int n ) {
    linklist *head = NULL,*last = NULL;
    for (int i=1;i<=n;i++) {
        linklist * p = (linklist*) malloc(sizeof(linklist));
        p->number = i,p->next = NULL;
        scanf("%d",&(p->mydata));

        if (head) {
            last->next = p;
            last = p;
        } else {
            head = last = p;
        }
    }
    last->next = head;
    return head;
```

```
}

int KingOfMonkey(int n,linklist *head) {
        linklist *p = head,*q = head;
        int cnt = 0;
        for (int i=0;i<n-1;i++) {
            q = q->next;//找到尾结点
        }
        int d = q->mydata;

        while (p->next != p) //循环退出条件,链表中只剩一个元素
        {
            cnt++;
            if (cnt == d) {
                d = p->mydata;
                cnt = 0;
                printf("Delete No:%d\n",p->number);
                q->next = p->next;
                free(p);
                p = q->next;
            }
            else {
                q = p;
                p = p->next;
            }
        }
        return p->number;
}
```

队列

队列的基本概念

- 队列：“先进先出”（FIFO）线性表
- 插入操作只能在队尾(rear)进行，删除操作只能在队首(front)进行
- 储存结构：顺序存储结构/链表结构实现

数据结构

- 单端队列

```
struct _queue {
  int *pBase;
  int front;
  int rear;
  int maxsize;
} QUEUE,*PQUEUE;
```

或者单向双头链表

- 双端队列deque　*英标 [dek]*

de -- double ended 双端队列（两边都可以插入和删除），双向双头链表

循环队列

引入循环队列的原因

- 线性队列浪费front以前的空间

CreateQueue

```
void CreateQueue (PQUEUE Q,int maxsize) {
    Q->pBase = (int*) malloc(sizeof(int)*maxsize);
    front = rear = 0;
    Q->maxsize = maxsize;
}
```

EmptyQueue

```
int EmptyQueue(PQUEUE Q) {
    return Q->front == Q->rear; //队列空的唯一情况
}
```

- `front == rear`

FullQueue

```
int FullQueue(PQUEUE Q) {
    return (Q->rear+1)%(Q->maxsize) == Q->front; //括号是不必要的。实质是(rear+1)%size ==
front
}
```

- `(rear+1)%size == front`

EnQueue

```
int EnQueue(PQUEUE Q,int val) {
    if (FullQueue(Q)) return 0;
    Q->pBase[Q->rear] = val;
    Q->rear = (Q->rear+1)%Q->maxsize;// rear = (rear+1)%size
    return 1;
}
```

- `q[rear] = val;`
- `rear = (rear+1)%size`

DeQueue

```
int DeQueue(PQUEUE Q,int *val) {
    if (EmptyQueue(Q)) return 0;
    *val = Q->pBase[Q->front];
    Q->front = (Q->front+1)%Q->maxsize; // front = (front+1)%size
    return 1;
}
```

- `*val = q[front];`
- `front = (front+1)%size;`

栈*Stack*

栈的基本概念

FILO 先进后出线性表

数据结构

常用数组

Pop

Push

程序结构*Program Structure*

存储类关键字*storage class specifier*

基本概念

- 包括：**auto，extern，static，register ，typedef等**
- 声明变量时，最多使用一个存储类关键字

  > `typedef register int x;`编译不能通过

- 分类：
  - 从变量的作用域角度（空间）来分，可以分为**全局变量** 和**局部变量**
  - 从变量的存在时间角度（生存期）来分，可以分为**静态存储方式**和**动态存储方式**
- 修饰关系：直接修饰变量，而不是数据类型。

  > static int *p　*a static pointer to integer，换言之static先修饰p*

典型例题

C语言中静态变量和外部变量的初始化是在___阶段完成的。

> 编译

`sizeof(int)`可计算整型所占的内存字节数，但是`sizeof( )`并不是一个函数，而是一个运算符（操作符，operator）。

> √

C语言中定义的全局变量存放在堆区，局部变量存放在栈区。

> x

函数指针

- 注意区分 ***function prototype*** 和 ***definition***

定义函数指针*function pointer*

```
int *(*p)(char *,double);
```

> 理解：()优先级最高，(*p)表明是指针

写出函数指针，函数返回类型为***void***，参数有两个：一个是***interger variable***,一个是 ***a pointer to an array of 10 integers***

```
void (*p)(int,int(*q)[10]);
```

> 这道题坑主要在第二个参数

返回类型为函数指针的函数原型*function prototype*

写出func函数原型，参数表是***int***，返回一个pointer to the function ***void f(int n)***

```
void(*func(int))(int)
```

> 理解：最内层()优先级最高，func()表明是函数。*func表明返回类型是指针
>
> 注意区别`void (*func) (int)`,这是一个指针

函数参数为函数指针

```
int f(int a) {return a;}
printf("%d",g(1,f));

int g(int c,int f(int)) {
  return f(c);
}

int g(int c,int (*f)(int)) {
  return (*f)(c);
}
//以上两种写法都是对的,即函数名称作为参量时
//既可以直接引用，也可以当作指针
```

`[] () *`等运算符优先级

以下哪个选项中的`p`是指针：

```
int* *p();
```

```
int *p();
```

```
int (*p)[5];  √
```

```
int *p[6];
```

typedef

- 一般数据类型起别名

```
typedef int myint;
typedef struct node node;
```

- 函数指针的typedef

```
typedef int *(*T)(char *,double);//语法完全类似于定义函数指针
T p;//定义函数指针
```

#define

- 只是复制粘贴

```
#define char* type1
typedef char* type2
type1 s1,s2;
type2 s3,s4;
```

等价于

```
char* s1,s2;//s2不是指针
char *s3,*s4;
```

- 还有乘法运算没括号，此处略

标准头文件结构

```
#ifndef _LINKLIST_
#define _LINKLIST_
#endif
```

算法设计

递归

递归的概念

1. 本质：重复 repetition
2. 实现：calling itself
3. 数学模型：分段函数
4. 分而治之,divide and conquer

递归要素

1. 基准条件(base case)：递归的终点
2. 有递进(make progress)：从最大的状态开始，每一次重复都向base case收敛
3. 自我实现(always believe)
4. 简洁性 *举例如fibo数列递归，树状递归若无"记忆"会有大量重复计算*

递归分类

分类方法一

1. 线性递归linear：每次进入函数只会call自己一次 *例子如阶乘*,f(n) = n*f(n-1)
2. 树形递归tree：每次进入函数会call自己两次及以上 *如fibo, fib[n-1] + fib[n-2]*

分类方法二

1. 真递归：递进时分解，回归时计算。 *阶乘和fib都是真递归*
2. 尾递归/伪递归：递进时做计算，回归时不做计算。*这种递归都可以暴力改成循环。*

递归举例

（1）阶乘

线性 真递归

```
    long factorial(int n) {
        if (n < 1) return 1;
        else return n*factorial(n-1);
    }
```

（2）fibonacci数列

树状 *(重复计算次数很多很多)* **真递归**

```
int fib(int n) {
    if (n <= 1) return n;
    else return fib(n-1) +fib(n-2);
}
```

- 调试小技巧：利用缩进indent显示递归递进

```
int fib(int n,int indent) {
    for (int i=0;i<indent;i++) {
        printf(" ");
    }
    printf("%d\n",n);
    if (n<=1) return n;
    else {
        return fib(n-1,indent+1) + fib(n-2,indent+1);
    }
}
```

- 优化——“记忆”消除重复计算

```
int a[100] = {0,1}; *此处为了演示数组大小随意填写*
int fib(int n) {
    if (n == 0 || a[n] > 0) return a[n];
    else {
        a[n] = fib(n-1) + fib(n-2);
        return a[n];
    }
}
```

（3）最大公约数/欧几里得算法/gcd

线性 伪递归 *可改成循环*

- 数学模型

  gcd(x,y) = x, y=0
  gcd(y,x%y) , y>0

- 递归实现

```
int gcd(int x,int y ) {
    if (y==0) return x;
    else return gcd(y,x%y);
}
```

伪递归：分解时计算，返回时不计算

- 迭代实现（循环）

修改方法：整个函数用while(1)包起来，base case变成if break;递进改成对相应变量赋值

```
int gcd(int x,int y) {
    while(1) {
        if (y==0) break;
        else {
            int t = x%y;
            x = y;
            y = t;
        }
    }
    return x;
}
```

（4）汉诺塔 Hanoi Towers

树状递归 真递归（计算是printf操作）

```
//我觉得这个代码真的很神奇！！
void Move(int n,char src,char temp,char des) {
    if (n>0) {
        Move(n-1,src,des,temp);
        printf("move %d from %c to %c\n",n,src,des);
        Move(n-1,temp,src,des);
    }
}
```

（5）牛顿迭代法计算平方根

- 数学模型

f(n) = x/2, n==1
    (f(n-1) + x/f(n-1))/2 ,n>1
收敛条件为 f(n)*f(n) == x 或 fabs(f(n) - f(n-1)) < e

改善版本
f(x,guess) = guess , fabs(guess*guess - x) < eps(精度)
f(x,(guess + x/guess)/2) ,otherwise

- 递归实现

线性 伪递归

```
double newsqrt(double x, double g) {
    if (fabs(g*g - x) <0.0001) {
        return g;
    }
    else {
        return newsqrt(x,(g+x/g)/2);
    }
}
```

- 迭代实现

```
double newsqrt(double x, double g) {
    while(1) {
        if (fabs(g*g - x) <0.0001) break;
        double t = (g+x/g)/2;
        g = t;
    }
    return g;
}
```

- 比较迭代和递归

迭代：循环，最小状态展开至最大状态
递归：最大状态分解至最小状态

（6）快速幂计算

- 数学模型

  x^n = 1 , n=0
  = (x^(n/2))^2 , n>0 && n%2==0
  = x*x^(n-1) , n>0 && n%2!=0

- 递归实现

```
long quickpow(int x,int n) {
    if (n==0) return 1;
    else if (n%2 == 0) {
        long result = quickpow(x,n/2);
        return result*result;
    }
    else {
        return x*quickpow(x,n-1);
    }
}
```

（7）整数顺序/逆序输出

- 核心问题是`printf`和递归语句的顺序
- 顺序输出

```
void printdigits( int n ) {
    if (n < 10) printf("%d\n",n);
    else {
        printdigits(n/10);
        printf("%d",n%10);
    }
}
```

- 逆序输出

```
void printdigits( int n ) {
    if (n < 10) printf("%d\n",n);
    else {
        printf("%d",n%10);
        printdigits(n/10);
    }
}
```

搜索与排序 *Searching&Sorting*

Search

Linear Search *线性搜索*

- 链表实现/数组实现
- 复杂度O(n)

Binary Search *二分搜索*

使用前提：已排序的数据

数学模型

x为数据集，s为搜索目标

f(x,s)　　= can not find, begin > end

= x[mid], s == x[mid]

= f(lower half of x,s), s < x[mid]

= f(higher half of x,s), s > x[mid]

递归实现

```
int bsearch(int a[], int begin, int end, int x) {
  if (begin > end) return -1;
  int mid = (begin+end)/2;
  if (a[mid] == x) return mid;
  else if (a[mid] < x) return bsearch(a,mid+1,end,x);
  else return bsearch(a,begin,mid-1,x);
}
```

- **伪递归**（递进时计算，回归时不计算）
- **线性递归**

迭代实现

```
int bsearch(int a[], int begin, int end, int x) {
  int ret = -1;
  while (begin <= end) {
    int mid = (begin+end)/2;
    if (a[mid] == x) {
      ret = mid;
      break;//不要漏了这句
    }
    else if (a[mid] < x) begin = mid+1;
    else end = mid-1;
  }
  return ret;
}
```

- 时间复杂度 O(logN) *log 表示 log2*
- 迭代比递归更容易看出时间复杂度

Sort

Select *选择排序*

排序思路（递归）

- 找到最大值，放在最后
- `sort(a,len-1)`

代码实现

递归

- Version1

```
void select(int a[], int len) {
  if (len > 0) {
    int loc = findmax(a,len);
    //swap(a[loc],a[len-1]);
    int t = a[loc];
    a[loc] = a[len-1];
    a[len-1] = t;
    select(a,len-1);
  }
}

int findmax(int a[],int len) {
  int max = 0;
  for (int i=1;i<len;i++) {
    if (a[i] > a[max]) max = i;
  }
  return max;
}
```

- Version2

```
void select(int a[], int len) {
  if (len > 0) {
    int max = 0;
    for (int i=1;i<len;i++) {
      if (a[i]>a[max]) max = i;
    }
    int t = a[max];
    a[max] = a[len-1];
    a[len-1] = t;
    select(a,len-1);
  }
}
```

> 线性递归
>
> 伪递归

迭代

- Version1 *while循环*

```
void select(int a[], int len) {
  while (len>0) {
    int max = 0;
    for (int i=1;i<len;i++) {
      if (a[i]>a[max]) max = i;
    }
    int t = a[max];
    a[max] = a[len-1];
    a[len-1] = t;
    len--;
  }
}
```

- Version2 *for循环*

```
void select(int a[], int len) {
  for (;len>0;len--)
    int max = 0;
    for (int i=1;i<len;i++) {
      if (a[i]>a[max]) max = i;
    }
    int t = a[max];
    a[max] = a[len-1];
    a[len-1] = t;
  }
}
```

- Version3 *较为常见的双重循环写法*

```
void select(int a[],int len) {
  for (int i=0;i<len-1;i++) {
    int min = i;
    for (int j=i+1;j<len;j++) {
      if (a[j]<a[min]) min = j;
    }
    int t = a[i];
    a[i] = a[min];
    a[min] = t;
  }
}
```

时间复杂度

O(n^2),双重循环

改进思路

- 每次同时找最大值和最小值；
- 找最大值和次大值；
- 找最大、次大、最小、次小
    - 但是仍然是O(n^2)

Bubble Sort *冒泡排序*

排序思路

- 每一次遍历中，两两比较相邻的两项，将较大的后移
- 冒泡：最大项像气泡一样浮到数组尾部

代码实现

- 基本版本

```
void bubble(int a[],int len) {
  for (int i=0;i<len;i++) {
    for (int j=i;j<len-1;j++) {
      if (a[j] > a[j+1]) {
        int t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    }
  }
}
```

- 改进版本

Bubble sort一般在第n次遍历之前已经结束

```
void bubble(int a[],int len) {
  for (int i=0;i<len;i++) {
    int flag;
    for (int j=0;j<len-i-1;j++) {//注意内层循环始终
      flag = 0;
      if (a[j] > a[j+1]) {
        int t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
        flag++;
      }
    }
    if (!flag) break;
  }
}
```

时间复杂度

- **时间复杂度** O(N^2)，和选择排序相似
- 考虑复杂度时，仅考虑循环次数，不考虑循环内做了多少事情

Insert Sort *插入排序*

排序思路

有n个元素的数列，先使前n-1个元素有序，再将第n个元素插入其中

代码实现

- 递归算法

```
//1 3 5 7 9 4
void InsertSort(int a[],int n) {
  if (n>0) InsertSort(a,n-1);//将前n-1个元素排好
  int x = a[n-1],j = n-2;
  while (j>=0 && a[j]>x) {
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = x;//插入第n个元素到有序位置
}
```

线性递归,真递归

注意到递进过程只是将n个元素分成单个元素

实际上可以不管递进，直接写回归——迭代

- 迭代算法

```
void InsertSort(int a[],int n) {
  for (int i=1;i<n;i++) {
    int x = a[i], j = i-1;//x是要排序元素的值，j标记有序数列的末位置
    while (j>=0 && a[j]>x) {
      a[j+1] = a[j];
      j--;
    }
    a[j+1] = x;
  }
}
```

时间复杂度

迭代算法可见，O(N^2)

适用场景

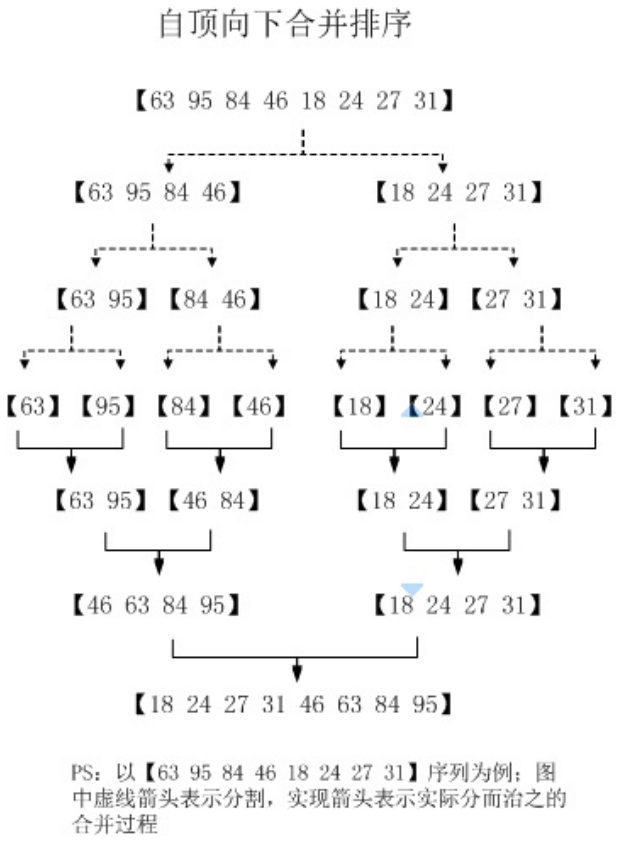
- 将一个新的数据放到原有有序的数列中——O(N)
- 动态缓慢地加入新的数据

Merge Sort *归并排序*

排序思路

1. 先将长度为N的无序序列分割平均分割为两段
2. 然后分别对前半段进行归并排序、后半段进行归并排序
3. 最后再将排序好的前半段和后半段归并

过程（2）中进行递归求解



PS：以【63 95 84 46 18 24 27 31】序列为例；图中虚线箭头表示分割，实现箭头表示实际分而治之的合并过程

- 牺牲空间换取时间
- 分而治之-*Divide&Conquer*，核心思想就是**分解、求解、合并**

TopDown *自顶向下的递归*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 8

/*a是目标数组，b是临时数组*/
/*Version1*/
void merge(int a[], int begin, int mid, int end, int b[])
{
    /*mid是前一半的末尾，后一半的开始*/
    int i = begin, j = mid, k = begin;
    while (i < mid && j < end)
    {
        if (a[i] < a[j])
            b[k++] = a[i++];
```

```
        else
            b[k++] = a[j++];
    }
    while (i < mid)
        b[k++] = a[i++];
    while (j < end)
        b[k++] = a[j++];
    //merge完以后b要重新放回a里，否则白排了!!!
    for (int i = begin;i<end;i++) {
        a[i] = b[i];
    }

    /*可视化*/
    for (int i = 0; i < begin; i++)
        printf("    ");
    for (int i = begin; i < end; i++)
    {
        printf("%4d",b[i]);
    }
    for (int i = end; i < SIZE; i++)
        printf("    ");
    printf("\n");
}

/*begin is inclusive, end is exclusive*/
void mergeSort(int a[], int begin, int end, int b[])
{
    //base case
    if (end - begin < 2)
        return;
    int mid = (begin + end) / 2;
    mergeSort(a, begin, mid, b);
    mergeSort(a, mid, end, b);
    merge(a, begin, mid, end, b); //合二为一
}

int main()
{
    int a[SIZE], b[SIZE];
    srand(0); //seed == 0
    for (int i = 0; i < SIZE; i++)
    {
        a[i] = rand() % 150;
    }
    for (int i = 0; i < SIZE; i++)
    {
        printf("%4d", a[i]);
    }
    printf("\n");
```

```
    mergeSort(a, 0, SIZE, b);
    for (int i = 0; i < SIZE; i++)
    {
        printf("%4d", a[i]);
    }
}
```

BottomUp *自底向上的迭代*

- Topdown递进过程实际上是切割，没有做任何其他事情；因此可以考虑舍弃递进，直接合并
- 双重循环，外层循环控制合并次数(logN)，内层循环控制排序小单元的步长

```
int min(int a,int b) {
    if (a>b) return b;
    else return a;
}
/*merge，将两个有序数列合并成一个，TopDown和BottomUp完全一致*/
void merge(int a[], int begin, int end, int b[])
{
    /*mid是前一半的末尾，后一半的开始*/
    int mid = (begin+end)/2;
    int i = begin, j = mid, k = begin;
    while (i < mid && j < end)
    {
        if (a[i] < a[j])
            b[k++] = a[i++];
        else
            b[k++] = a[j++];
    }
    while (i < mid)
        b[k++] = a[i++];
    while (j < end)
        b[k++] = a[j++];
    //merge完以后b要重新放回a里，否则白排了!!!
    for (int i = begin;i<end;i++) {
        a[i] = b[i];
    }

/*BottomUp*/
void mergeSort(int a[], int n, int b[])
{
    for (int width = 1;width < n;width *= 2) {//合并次数log2N
        for (int i=0;i<n;i = i + width*2) {//控制小数组的长度
            merge(a,min(i,n),min(i+width*2,n),b);//begin is inclusive, end is exclusive
        }
    }
}
```

上面是错的,mid不是中间

```
void Merge(int a[],int begin,int mid,int end) {
    int i = begin,j = mid,k = i;
    int *b = (int *)malloc(sizeof(int)*(end-begin));
    while (i<mid && j<end) {
        if (a[j] < a[i]) b[k++] = a[j++];
        else b[k++] = a[i++];
    }
    while (i<mid) b[k++] = a[i++];
    while (j<end) b[k++] = a[j++];

    for (int i=begin;i<end;i++) {
        a[i] = b[i];
    }
    for (int i = 0; i < begin; i++)
        printf("    ");
    for (int i = begin; i < end; i++)
    {
        printf("%4d",b[i]);
    }
    for (int i = end; i < MAXSIZE; i++)
        printf("    ");
    printf("\n");
}

void* MergeSort(parray *a)
{
    for (int width = 1;width < MAXSIZE;width *= 2) {
        for (int i=0;i<MAXSIZE;i = i + width*2) {
            Merge(a->pBase,i,i+width,min(i+width*2,MAXSIZE));//begin is inclusive, end
is exclusive
        }
    }
}
```

时间复杂度

- O(NlogN)
- 递进（分解）无循环，不消耗时间
- 合并的时候 共logN层,每一层循环遍历N次，共计N*logN

Quick Sort *快速排序*

排序思路

- 取出基准数*pivot*，使pivot左边的数比它小，右边的数比它大
- 对左边和右边分别快速排序(递归过程)

代码实现

- **快排递归**

```
void QuickSort(int a[],int left,int right) {
    if (left < right) {
        int pivot = GetPivot(a,left,right);
        QuickSort(a,left,pivot-1);
        QuickSort(a,pivot+1,right);
    }
}
```

```
//qsort : increment v[left] ... v[right]
void qsort (int v[],int left,int right) {
    int i,last;
    void swap (int v[],int i,int j);
    if (left >= right) return;//如果分组中只有一个元素，则不用排序
    swap(v,left,(left+right)/2);//把最中间的元素换到最左边
    last = left; //定位 比划分元素小 的最后一个元素的位置，便于结束本轮快排时将 划分元素 插入last的位置
    for (i = left+1;i <= right;i++) {
        if (v[i] < v[left]) swap(v,++last,i); //将小于划分元素的数移到左边，并标记最后一个小的数的位置
    }
    swap(v,left,last);//把划分元素放回去
    qsort(v,left,last-1);
    qsort(v,last+1,right);//分两组继续快排，递归进行
}
```

时间复杂度

考虑到最好情况，每次都是均匀划分，则运算成本为：

$$T(n) = 2 * T(\frac{n-1}{2}) + O(n)$$

为方便运算，将式子看做：

$$T(n) = 2 * T(\frac{n}{2}) + n$$

$$T(n) = 2 * 2[T(\frac{n}{4}) + \frac{n}{2}] + n = 2^2 T(\frac{n}{4}) + 2n = \cdots\cdots$$

假设$2^k = n$

$$T(n) = 2^k * T(1) + k * n$$

不难看出复杂度为O(nlogn)。

但如果是最坏情况，比如[1,2,3,4,5]，若一数组末尾元素作为划分标准，那么计算的成本就变为了：

$$T(n) = T(n-1) + O(n)$$

很明显，复杂度变成了O(n^2)。

- Worst case:O(N^2)
- Best case:O(NlogN)
- Average case:O(NlogN)
- 不稳定

算法复杂度O()

算法时间复杂度

- 通常指worst case，有时也指average case(指明)
- O标记法：省略系数和小项

典型例题

- Inserting a node into a descending-order(降序) linked list with N nodes needs ***O(n)*** comparisons at average.
- Given a data set of N(N=10^6) integers which is within the range of the whole integers, and unsorted, most of the data are duplicated except one. Given a good sort function which has an complexity of O(NlogN), to find out the single one, the complexity of the best algorithm is ***O(nlogn)***
- when sorting n objects, if input array is **already sorted**, the **Bubble Sort** algorithm has **O(n)** time complexity.

- Which one of the following algorithms is NOT an O(n) algorithm?

  A. Finding someone in your telephone book; 二分搜索O(nlogn)

  B. Linear Search;

  C. Deletion of a specific element in a double-linked List (unsorted);

  D. Comparing two strings.

- Which one of the following algorithms is NOT an O(1) time complexity algorithm?

  A. Calculating the average value of the **first three** elements of a double-linked list;
  B. Searching in a stack;
  C. Accessing to the **third** element of a single-linked list;
  D. Accessing to the **third** element of an array.

- Binary search uses at worst ***O(logN)***, at average ***O(logN)*** , and at best ***O(1)*** comparisons.

图形库

基本概念

- 坐标单位是inch
- 坐标系和正常使用的坐标系一样，原点在左下角

基本函数

main及初始化

```
void Main()
{
  InitGraphics();
  OpenConsole();//如果要输入字符
  CloseConsole();
}
```

绘图函数

- 获取屏幕中央坐标

`cx = GetWindowWidth()/2;`

`cy = GetWindowHeight()/2;`

- 获取当前坐标

```
x = GetCurrentX();
y = GetCurrentY();
```

- 移动画笔
  - 把画笔移到(x,y)处
  - 不画线

```
MovePen(x, y);
```

- 画线
  - 从当前位置开始画
  - 参数是偏移量，单位是inch
  - 新的画笔位置是线末端

```
DrawLine(dx, dy);
```

- 画圆
  - 始终从当前位置开始画
  - 第二个参数
    - 如果是0，则从🕒开始画
    - 如果是90，则从🕑开始画
    - 如果是180，则从🕘开始画
    - 如果是-90，则从🕛开始画
  - 第三个参数：圆的角度，单位是度

```
DrawArc(r, 0, 360);
```

- 中心圆

```
//自己写一个画中心圆的函数!
void DrawCenterCircle(double x,double y,double r) {
  MovePen(x+r,y);
  DrawArc(r,0,360);
}
```

计时器

- 计时器函数原型

```
void TimerEvent(int timerID);//timerID是计时器编号
```

- 主程序中注册回调函数

`registerTimerEvent(TimerEvent);//注册回调函数`

- 打开和关闭计时器

`startTimer(timerID,interval);//每隔interval启动计时器，单位ms`

`cancelTimer(timerID);`

键盘

- 键盘函数原型

`void KeyboardEventProcess(int key,int event);`

> `key`是按键的虚拟码，如`VS_ESCAPE`
>
> `event`只有两种，`KEY_DOWN`和`KEY_UP`

- 注册键盘回调函数

`registerKeyboardEvent(KeyboardEventProcess);`

鼠标

- 鼠标函数原型

- 注册鼠标回调函数

应用举例

闪烁 - SetEraseMode 和 TimerEvent

```
static bool isDisplayCircle = true;
if (timerID == TIMER_BLINK100) {
  bool erasemode = GetEraseMode();//初始化擦除函数
  SetEraseMode(isDisplayCircle);//设置是否擦除
  DrawCenterCircle(cx,cy,radius);
  SetEraseMode(erasemode);
  isDisplayCircle=!isDisplayCircle;
}
```

- 函数功能：*Flash a circle drawn in the center of a window once every 500 milliseconds. The ESCAPE key is used as a switch to toggle the blink.*

```
#include <windows.h>
#include "genlib.h"
#include "graphics.h"
#define TIMERB 1
```

```
const int mseconds = 500;
static double ccx = 1.0, ccy = 1.0; static double radius = 1.0;
static bool isB = FALSE;
static bool isD = TRUE;
void DrawCenteredCircle(double x, double y, double r); void KeyboardEventProcess(int
key,int event);
void TimerEventProcess(int timerID);
void Main()
{
  InitGraphics();
  registerKeyboardEvent(KeyboardEventProcess);
  registerTimerEvent(TimerEventProcess);
  ccx = GetWindowWidth()/2;
  ccy = GetWindowHeight()/2;
  DrawCenteredCircle(ccx, ccy, radius);
  if(isB) startTimer(TIMERB, mseconds);
}

void DrawCenteredCircle(double x, double y, double r) {
  MovePen(x+r, y);
  DrawArc(r, 0.0, 360.0); }
void KeyboardEventProcess(int key,int event) { if(event == KEY_DOWN && key ==
VK_ESCAPE) {
  isB = !isB;
  if (isB) {
    startTimer(TIMERB, mseconds);
  }
  else {
    cancelTimer(TIMERB);
    DrawCenteredCircle(ccx, ccy, radius);
  }

void TimerEventProcess(int timerID)
{
  if(timerID == TIMERB)
  {
    bool erasemode = GetEraseMode();
    SetEraseMode(isD); //true，则画笔设置为背景色；false，则画笔为正常颜色
    DrawCenteredCircle(ccx, ccy, radius);
    SetEraseMode(erasemode);
    isD=!isD;
  }
}
```

分形树

```
void FractalTree(int n, double x, double y, double length, double theta) {
  if (n > 0) {//分形次数，也即递归树深度
  double radians = theta / 180.0 * PI;//角度转弧度
  int dx = length * cos(radians);//计算x方向偏移量
  int dy = length * sin(radians);//计算y方向偏移量
  MovePen(x, y);
  DrawLine(dx, dy);//画第一条竖线
  FractalTree(n-1, x+dx, y+dy, length*0.75, theta + 15); //右树
  FractalTree(n-1, x+dx, y+dy, length*0.75, theta - 15);//左树
  }
}
void Main() {
  int n;
  double length;
  InitGraphics();
  OpenConsole();
  n = GetInteger();
  length = GetReal();
  CloseConsole();
  FractalTree(n, GetWindowWidth()/2.0, 0, length, 90);
  return;
}
```

等边三角形分形

```
#include "graphics.h"
#include <math.h>
#define LEN 6.0
#define PI 3.14159
#define EPS 0.05
typedef struct {
double x, y;
} VERTEX;
VERTEX MidPoint(VERTEX A, VERTEX B);
void DrawTriangle(VERTEX A, VERTEX B, VERTEX C);
void FraTriangle(VERTEX A, VERTEX B, VERTEX C);
void Main()
{
  VERTEX A, B, C;
  double cx, cy;
  InitGraphics();
  cx = GetWindowWidth()/2;
  cy = GetWindowHeight()/2;
  A.x = cx;
  A.y = cy + LEN/2*sin(PI/3);
  B.x = cx - LEN/2;
```

```
  B.y = cy - LEN/2*sin(PI/3);
  C.x = cx + LEN/2;
  C.y = B.y;
  FraTriangle(A, B, C);
}
void DrawTriangle(VERTEX A, VERTEX B, VERTEX C)/*Draw △ABC*/
{
  MovePen(A.x, A.y);
  DrawLine(B.x-A.x, B.y-A.y);
  DrawLine(C.x-B.x, C.y-B.y);
  DrawLine(A.x-C.x, A.y-C.y);
}
VERTEX MidPoint(VERTEX A, VERTEX B) {
  VERTEX mAB;
  mAB.x = (A.x + B.x) / 2;
  mAB.y = (A.y + B.y) / 2;
  return ___(1)___ ;//mAB
}
void FraTriangle(VERTEX A, VERTEX B, VERTEX C) {
  VERTEX mAB, mBC, mCA;
  if (fabs(A.x-B.x) < EPS) return;
  ___(2)___;//DrawTriangle(A,B,C);
  mAB = MidPoint(A, B); mBC = MidPoint(B, C); mCA = MidPoint(C, A);
  ___(3)___;//FracTriangle(A,mAB,mCA);
  ___(4)___;//FracTriangle(mAB,B,mBC);
  ___(5)___;//FracTriangle(mCA,mBC,C);

}
```

- 总结以上两个程序，都是在Main里直接调用递归函数
- 将n次分形，拆解成1 + (n-1)次分形。即先画出第一步，后面由递归实现
- 分形树的 **第一步** 是画出树干
- 三角形的 **第一步** 是画出大三角形