数据结构与算法分析

程序是由算法，数据结构，程序设计设计方法，语言环境和工具四个部分组成，算法是思想和灵魂，数据结构是骨架，两者都很重要。学一门编程语言不难，但是要精通一些算法和数据结构方面的东西，需要悟性。刚好在 C 语言的学习中看到排序算法那一章，就阻塞一下，准备看完《数据结构与算法分析》这本书，加油^< ^

一、 前言

1.1 程序语言简述和特点

数据结构主要研究组织大量数据的方法；
而算法分析则是对算法运行时间的评估；
C 代替 Pascal 的使用使得动态分配数组称为可能；
本书讨论的内容：

- 选择问题：设一组 N 个数而要确定其中第 k 个最大者；
- 流行字谜问题：
- 等等

算法：任何良定义的计算过程，把输入转换成输出的计算步骤的一个序列

1.2 数学知识复习

1) 指数；
2) 对数；

$$\log_A B = \frac{\log_c B}{\log_c A}$$

$$\log AB = \log A + \log B$$

$$\log X < X\left(X > 0 \text{ 都成立}\right)$$

$$a^{\log_b n} = a^{\frac{\log n}{\log b}} = a^{\frac{\log a}{\log b}\times\frac{\log n}{\log a}} = a^{\log_b a\times\log_a n} = a^{\log_a n^{\log_b a}} = n^{\log_b a}$$

3) 级数：

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$，当 $A < 0$，且 $N \to \infty$时，结果 $\to \frac{1}{1-A}$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$$ (用数学归纳法可以证明)

$$调和数\ H_N = \sum_{i=1}^{N} \frac{1}{i} = \log_e N$$

4) 横运算（同余）：意味着 A 和 B 去除以 N 所得的余数相同

$$A \equiv B(mod\ N)$$
$$A + C \equiv B + C(mod\ N)$$
$$AD \equiv BD(mod\ N)$$

$$若A \equiv B(mod\ C)，则A^N \equiv B^N(mod\ C)$$

5) 证明方法：数学归纳法和反证法

1.3 递归(recursive)简论

定义：当一个函数可以用它自己去定义的时候称为递归；
C 允许递归，但是 C 仅仅是提供递归思想的一种企图，不是所有的数学递归都可以有效的，或者说是正确的由 C 额递归模拟来实现。（我估计是考虑会有数值越界等情况的出现）
基本的写法都需要有一个条件作为退出的触发

```
int recursive(){
 if(){
      return 0 //结束递归，并提供初值
     }else{
     //调用递归函数
     }
}
```

往往有这样的观点：能不用递归就不用递归，递归都可以用迭代来代替。
递归的四条法则：

- 基准情形
- 不断推进
- 设计法则
  假设所有的递归调用都能运行
  递归的主要问题是隐含的簿记开销，虽然这些开销几乎都是合理的，不仅简化了算法设计，而且简化了算法
- 合成效益法则
  在求解同一个问题的同一个实例的时候，切勿在不同的递归调用中做重

复性的事情。

每层递归中，包含三个部分：

- 分解（divide）：将问题划分为一些子问题，子问题与原问题一样，只是规模比较小；
- 解决（Conquer）：步骤递归地求解出子问题，如果子问题规模足够小，则停止递归，直接求解；
- 合并（Combine）：将子问题的解组合成原问题；

1.4 我对递归的理解

首先把算法变成函数的递推关系式，从结果推导到初始。

一般写法：

```
int recrusive(int a){

    while(持续条件){
        if(初始条件 1){
            //初始结果 1；
        }else if(初始条件 2){
            //初始结果 2；
        ……
        }else{
            //递归操作
             if(递归条件 1){
                 return recrusive(int a)
             } else if(递归条件 2){
                 return recrusive(int a)
             }
             ……
             return recrusive(int a)      //其余条件
        }
    }
    return  持续条件不满足时的结果
}
```

循环则是从初始推导到结果

```
While(){
    If(结束条件){
        Return;
    }else(){

        Break;
```

}

}

Return

二、 算法分析

算法：为求解一个问题需要遵循的被清楚地制定的简单指令的集合

2.1 数学基础

渐近符号

- 定义 1：如果存在正常数 c 和 n0 使得当 N>n0 时，$T(N) < c\mathcal{F}(N)$，则记为$T(N) = O(\mathcal{F}(N))$，给出了渐进上界；
- 定义 2：如果存在正常数 c 和 n0 使得当 N>n0 时，T(N)>cg(N)，则记为 $T(N) = \Omega(g(N))$，给出了渐进下界；
- 定义 3： $T(N) = \Theta(\mathcal{F}(N))$当且仅当$T(N) = O(\mathcal{F}(N))$且$T(N) = \Omega(\mathcal{F}(N))$；此时我们称$\mathcal{F}(N)$是T(N)的渐进紧确界（asymptotically tight bound），并同时给出了渐进上界和渐进下界；
- 定义 4：如果$T(N) = O(p(N))$且$T(N) \neq \Theta(p(N))$，则$T(N) = o(p(N))$

注意：$O(\mathcal{F}(N))$和$\Omega(g(N))$和$\Theta(\mathcal{F}(N))$都是集合；使用=号表示是该集合中的元素$\mathcal{F}(N)$本身必是渐进非负的，否则集合为空；
$\Theta(1)$表示常量或者某个变量的常量函数集合；
若渐记符号出现在函数或者不等式之中，代表在该渐记符号集合下的我们不关心的匿名函数。
以上概念目的是建立一种数量比较的相对的级别，比较两个数的大小其实并没有什么太大的意义，我们一般比较他们的相对增长率（relative rate of growth）。

比如$1000N = O(N^2)\big(N\ 平方级\big)$，这种记法称为大 O 记法，人们常常不说“……级的”，而是说“大 O……”，此外还有小 o 记法，那是严格的渐近上界，ω表示严格的渐近下界。

- 法则 1：如果$T(N) = O(\mathcal{F}(N))$而且$T(N) = O(g(N))$，那么

$$T1(N) + T2(N) = \max\big(O\big(\mathcal{F}(N)\big),\ O(g(N))\big)$$

$$T1(N) \times T2(N) = O\big(\mathcal{F}(N) \times g(N)\big)$$

- 法则 2：如果T(N)是一个 k 次多项式，则$T(N) = \Theta(N^k)$
- 法则 3：对于任意常数 k，$(\log N)^k = O(N)$

典型的增长率比较：常数<对数级<对数平方级<线性级<线性级*对数级<平方级

<立方级<指数级
比较的时候可以使用工具——洛必达法则

- 结果为 0，意味着$\mathrm{T(N)}=\mathrm{o}(\mathcal{F}(\mathrm{N}))$
- 结果为 c≠0，意味着$\mathrm{T(N)}=\Theta(\mathcal{F}(\mathrm{N}))$
- 结果为∞，意味着$\mathrm{T(N)}=\Omega(\mathcal{F}(\mathrm{N}))$
- 结果为极限摇摆，二者无关

2.2 循环不变式

主要用来帮助理解算法的正确性：

- 初始化：从初始化循环计数变量开始，到第一次判断之前的过程；
- 保持：如果某次的循环迭代为真，那么下次的跌打仍为真；
- 终止：亦即循环结束；

其实就是数学归纳法的思想。

2.3 求解递归式的方法

代入法
猜测解的形式；
用数学归纳法求出解中的常数，并证明解是正确的；
如经典的分治排序法：$\mathrm{T}(N)=2\mathrm{T}(N/2)+\mathrm{O}(N)$
猜测$\mathrm{T}(N)=O(N\log N)$
假定对任意 N>m，当 c>0 时，有$\mathrm{T}(N)<cN\log N$，$\mathrm{T}(N/2)<cN/2\log N/2$
则

$$\mathrm{T}(N)=2\mathrm{T}(N/2)+\Theta(N)<2\times c\,N/2\log N/2+N=cN\log N/2+N$$
$$=cN\log N-(\mathrm{cN}\log 2-N)$$

其中，当 c>=1/ log2 的时候就成立。
递归树法
猜测，给根部和叶部分配，然后求和，但是也非常的麻烦，还不如用代入法呢
主方法
对于递归式：

$$\mathrm{T}(N)=a\mathrm{T}(N/b)+\mathrm{f}(N)$$

若对于某个常数 ε >0 有$\mathrm{f}(N)=O(n^{\log_b a-\epsilon})$，则$\mathrm{T}(N)=\Theta\left(n^{\log_b a}\right)$；
若$\mathrm{f}(N)=\Theta\left(n^{\log_b a}\right)$，则有$\mathrm{T}(N)=\Theta\left(n^{\log_b a}\log n\right)$；
若对于某个常数$\epsilon>0$，有$\mathrm{f}(N)=\Omega\left(n^{\log_b a+\varepsilon}\right)$，且对某个常熟 c<1 和所有足够大的 n 有$a\mathrm{f}(N/b)\le c\mathrm{f}(N)$，则$\mathrm{T}(N)=\Theta\left(\mathrm{f}(N)\right)$；

2.4 模型和要分析的问题

RAM(Random Access Machine)模型假设

- 计算模型是一台标准的计算机
- 执行过程中指令是一条一条执行的，严格的单线程
- 简单的指令系统：加法，乘法，比较和赋值
- 任何指令的执行都恰好花费一个时间单元
- 整数有固定的 4 个字节，或者 32 个比特
- 无限内存
- 不对当代计算机的内存层次进行建模，也就是说，我们没有对高速缓存和虚拟内存进行建模；

假设的局限性：不是所有运算都恰好花费相同的时间，没有考虑数据读写的时间

要分析的问题

- 输入规模
  其最佳概念依赖于所研究的问题；
- 运行时间
  这个运行时间一般取最恶劣情况下的时间，一方面对所有的输入提供了一个界限；另一方面平均情况下的时间比较难去计算

2.5 运行时间的计算

为了简化分析，约定：

- 不存在特定的时间单位，因此，我们抛弃一些常数系数和低阶项，计算大 O 运算时间。
- 变量声明不计算时间；
- 比较，自增和结果返回各占一个时间单位；

一般法则：

- For 循环：一次 for 循环的运行时间至多是该 for 循环内语句运行时间乘以迭代的次数，$O(N^1)$；
- 嵌套的 For 循环：需要从外之内分析，嵌套在内部的语句的总的运行时间为该语句的运行时间乘以该组所有 for 循环的大小的乘积 $O\left(N^{嵌套的次数n}\right)$。
- 顺序语句 $O(N^0)$
- IF/ELSE 语句

```
If(Condition){
  S1
}ELSE{
  S2
}
```

一个 if/else 判断语句的运行时间从不超过判断再加上 S1 和 S2 中运行时

间长者的总运行时间

- 函数迭代的运行时间也是迭代运算出来的;

函数迭代计算的时间计算：

如：

```
int func(int n){
    if(n<2){                          //  1
        return 1;                     //  2
    }else{
        return func(n-1) + func(n-2)  //  3
    }
}
```

第 N 次计算的时间近似于斐波那契数列每一项小于$\left(\frac{5}{3}\right)^N$，可见他是一个指数级的时间的，通过转换为 for 循环，能大大减少运行的时间。程序之所以缓慢，是由于第三行中的 func(n-2)已经被 func(n-1)计算过一遍，这不符合递归的第四条法则（合成效益法则）

基本策略是从内至外展开

验证一个程序是否是$O(\mathcal{F}(N))$，一个常用的技巧是对 N 的某个范围（通常是 2 大的倍数隔开）计算比值$T(N)/\mathcal{F}(N)$，其中$T(N)$是凭经验观察到的时间，如果$\mathcal{F}(N)$是运行是按的理想近似，那么所算出的值收敛于一个正常数，如果算出来的值收敛于 0，则说明$\mathcal{F}(N)$估计过大，否则过小。

三、 表，栈和队列

本章讨论三种数据结构

3.1 抽象数据类型

模块化的优点

1) 调试简单
2) 分工容易
3) 在例程中修改简单（所谓历程，我猜测是将模块集中在一起的主程序）

抽象数据类型（abstract data type）是一些操作的集合，或者说对诸如表，集合，图和他们的操作一起看成是抽象数据类型

如对于集合 ADT，可以有诸如并（union）和交（intersection），测量大小（size）以及取余（complement）等操作

3.2 表 ADT

1) 表的大小
   形如 A1，A2，A3，A4，An 的表，大小为 N，N=0 则称为空表；
   对于除空表以外的所有表，Ai+1 后继 Ai，或者说 Ai 前驱 Ai+1；
2) 链表
   为了避免插入和删除的线性开销，需要允许表可以不连续存储，否则表的部分或者全部需要整体移动，这是链表出现的原因。
   链表是一系列不必在内存中相连的结构体，每个结构体均包含有表元素和指向后继元结构的指针（Next 指针，最后一个结构的 Next 指针指向 NULL）。
   其具体定义和结构如下：

```
#ifndef _list_h
    struct node;
    typedef struct node *Ptrtonode;
    typedef Ptrtonode List;
    typedef Ptrtonode Position;

    //操作函数的定义
     IsEmpty(Position p，List L)
     isLast(Position p，List L)
     find(ElementType X，List L)
     findPrevious(ElementType X，List L)
     delete (ElementType X，List L)
     insert(ElementType X，List L, Position P)
     deleteList(List L)

#endif

struct node{
    ElementType Element;
    Position Next;
}
```

3) 说明：List 和 Position 都是链表指针的数据类型，而 L 和 P 分别是其具体的指针变量，代表每个元素的首地址，L 一般指的是带表头的表的表头 header，而 P 指的是任意位置的链表元素
4) 空表判断 IsEmpty(Position p，List L)

```
int isEmpty(Position p，List L){

    return L->Next==NULL;
}
```

5) 链表结束判断 isLast(Position p，List L)

```
int isLast(Position p，List L){
```

```
        return p->Next==NULL;
    }
```

6) 查找 find(ElementType X，List L)

```
    int find(Element X，List L){

        Position P=L->Next;
        While(P->Next!=NULL && P->Element!=X){
            P=P->Next;
        }
        return p;
    }
```

7) 前驱查找 findPrevious(ElementType X，List L)

```
    int find(Element X，List L){

        Position P=L->Next;
        While(P->Next!=NULL && P->Next->Element!=X){
            P=P->Next;
        }
        return p;
    }
```

8) 删除 delete(ElementType X，List L)

```
    void delete (Element X，List L){

        Position P=findPrevious(X, L);
        If(p->Next!=NULL){
            P->= P->Next ->Next;
            free(P->Next);
        }
    }
```

9) 插入 insert(ElementType X，List L, Position P)

```
    void insert (Element X，List L){

        Position insertP=(Position)malloc(sizeof(struct Node));
        If(insert==NULL){
            Printf("Out of Space !!");
        }else{
            insertP->Element=X;
            insertP->Next= P->Next;
            P->Next= insertP;
        }
    }
```

10) 删除表 deleteList(List L)

```
void deleteList(List L){

    Position Tem,;
    Position P=L->Next;
    while(P!=NULL){
        Tem= P->Next;
        free(P);
        P=Tem;
    }
}
```

11) 双链表

应对倒序扫描链表，增加一个指向前一个地址的 Last 指针；

当然了，这样做的话插入和删除的开销都会增加一倍，另一方面，他简化删除的操作（这个怎么理解？）

12) 循环链表

让最后一个单元的 Next 指针指向第一个单元，如果有表头 header，就让他只想表头

13) 双向循环链表

14) 常见错误：

Memory access violation/segmentation violation，原因一般是出现初始化变量失败或者出现野指针的情况；

调用 malloc 的次数应该应该等于表的大小，若有表头则加 1，少一点不行，多一点浪费空间；

对链表的某一个元素进行删除操作的时候，应该先用一个临时变量存储被删除变量的 Next 指针，然后释放被删除元素堆空间，最后把被删除变量前驱元的 Next 指针指向临时变量

3.3 C primer plus 书上一个成熟的程序

头文件 list.h

```
/* ****************************************************************
 *       Filename: list.h
 *       Description:
 *       Version: 1.0
 *       Created: 2017/01/19
 *       Revision: none
 *       Compiler: gcc
 *       Author: Lv Hongbin
 *       Company: Shanghai JiaoTong Univerity
 * ****************************************************************/
```

```
#ifndef LIST_H_
#define LIST_H_
#include<stdbool.h>
/* ****************************************
 *  Type Name:       List
 *  Type Attributes:  Item item;
 *                              struct node *next;
 *
 *  Type Operation:       void InitializeList(List *plist);
 *                  boolean ListIsEmpty(const List *plist);
 *                  boolean ListIsFull(const List *plist);
 *                  unsigned int ListItemCount(const List *plist);
 *                  boolean AddItem(Item item, List *plist);
 *                  void Traverse(const List *plist, void(*pfun)(Item item));
 *                  void EmptyTheList(List *plist);
 * ****************************************/

#define TSIZE 45

    struct film{

        char title[TSIZE];
        int rating;
    };
    typedef struct film Item;

    typedef struct node{

        Item item;
        struct node *next;
    }Node;

    typedef Node * List;

/*  function prototype    */


/*  Function:    Initialize a list              */
/*  Precondition:    plist point at a list          */
/*  Postcondition:    make a list NULL              */
void InitializeList(List *plist);
```

```
/*    Function:     Makesure a list empty                */
/*    Precondition:     plist point at an initialized list */
/*    Postcondition:    Empty?true:false             */
bool ListIsEmpty(const List *plist);

/*    Function:     Makesure a list    full              */
/*    Precondition:     plist point at an initialized list */
/*    Postcondition:    Full?true:false                   */
bool ListIsFull(const List *plist);

/*    Function:     Initialize a list                */
/*    Precondition:     plist point at an initialized list */
/*    Postcondition:    make a list NULL                  */
unsigned int ListItemCount(const List *plist);

/*    Function:     Initialize a list                */
/*    Precondition:     plist point at an initialized list */
/*    Postcondition:    make a list NULL                  */
bool AddItem(Item item, List *plist);

/*    Function:     Apply func on every item         */
/*    Precondition:     plist point at an initialized list */
/*                  pfun point at a func without return */
/*    Postcondition:    Apply func on every item         */
void Traverse(const List *plist, void(*pfun)(Item item));

/*    Function:     Delete    a list                 */
/*    Precondition:     plist point at a list             */
/*    Postcondition:    Make a list empty                 */
void EmptyTheList(List *plist);

#endif
```

具体实现文件 list.c

```
/* ************************************************************
 *         Filename: list.c
 *         Description:
 *         Version: 1.0
 *         Created: 2017/01/21
 *         Revision: none
 *         Compiler: gcc
```

```
 *        Author: Lv Hongbin
 *        Company: Shanghai JiaoTong Univerity
 * ********************************************************/

#include<stdio.h>
#include<stdlib.h>
#include "list.h"

void InitializeList(List *plist){

    *plist=NULL;
}

bool    ListIsEmpty(const List *plist){

    return *plist==NULL;
}

bool ListIsFull(const List *plist){

    Node *pt;
    pt=(Node *)malloc(sizeof(Node));
    if (pt==NULL){
        return true;
    }else{
        free(pt);
        return false;
    }
}

unsigned int ListItemCount(const List *plist){

    unsigned int count=0;
    Node *temp=*plist;
    while(temp!=NULL){
        count++;
        temp=temp->next;
    }
    return count;
}

bool AddItem(Item item, List *plist){
```

```
    Node *pnew;
    Node *temp=*plist;
    pnew=(Node *)malloc(sizeof(Node));
    if(pnew==NULL){
        return false;
    }
    pnew->item=item;
    pnew->next=NULL;
    if(temp==NULL){
        *plist=pnew;
    }else{
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next=pnew;
    }
    return true;

}

void Traverse(const List *plist, void(*pfun)(Item item)){

    Node *temp=*plist;
    while(temp!=NULL){
        pfun(temp->item);
        temp=temp->next;
    }
}

void EmptyTheList(List *plist){

    Node *tem;
    while(*plist!=NULL){
                    tem=(*plist)->next;
                    free(*plist);
        *plist=tem;
         }
}
```

主程序：film.c

```
/* ****************************************************************
 *       Filename: film.c
 *       Description:
 *       Version: 1.0
 *       Created: 2017/01/21
 *       Revision: none
 *       Compiler: gcc
 *       Author: Lv Hongbin
 *       Company: Shanghai JiaoTong Univerity
 * ****************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include "list.h"

char * s_gets(char *ctitle, int N);
void showmovies(Item item);
int main(){

    List movies;
    Item temp;

    InitializeList(&movies);
    if(ListIsFull(&movies)){
        fprintf(stderr,"No memory avaliable! Bye!\n");
        exit(1);
    }

    puts("Enter the first movie title:");
    while(s_gets(temp.title,TSIZE)!=NULL &&    temp.title[0]!='\0' ){
        puts("Enter your rating<0-10>: ");
        scanf("%d",&(temp.rating));
        while(getchar()!='\n'){
            continue;
        }
        if(AddItem(temp, &movies)==false){
            fprintf(stderr,"Problem allocating memory!\n");
            break;
        }
        if(ListIsFull(&movies)){
                    puts("The lists is full.");
                    break;
```

```
            }
        puts("Enter the next movie title(empty line to stop):");
    }

    if(ListIsEmpty(&movies)){
        printf("No data entered.");
    }else{
        printf("Here is the movie list:\n");
        Traverse(&movies,showmovies);
    }
    printf("You enter %d movies.\n", ListItemCount(&movies));

    EmptyTheList(&movies);
    printf("Bye!\n");
    return 0;
}

void showmovies(Item item){

    printf("Movie:%s Rating:%d\n",item.title,item.rating);
}

char *s_gets(char *ctitle, int N){

    char *newcTitle;
    int i=0;
    newcTitle=fgets(ctitle,N,stdin);
    if(newcTitle){
        while(ctitle[i]!='\n' && ctitle[i]!='\0'){
            i++;
        }
        if(ctitle[i]=='\n'){
            ctitle[i]='\0';
        }else{
            while(getchar()!='\n'){
                continue;
            }
        }
    }
    return newcTitle;
}
```

结果：

```
[root@MiWiFi-R3G-srv /home/lvhongbin/Desktop/cStudy]#gcc list.c film.c -o film.out
[root@MiWiFi-R3G-srv /home/lvhongbin/Desktop/cStudy]#./film.out
Enter the first movie title:
Hello,Lv Hongbin
Enter your rating<0-10>:
10
Enter the next movie title(empty line to stop):
Goodbye, Lv Hongbin
Enter your rating<0-10>:
9
Enter the next movie title(empty line to stop):

Here is the movie list:
Movie:Hello,Lv Hongbin Rating:10
Movie:Goodbye, Lv Hongbin Rating:9
You enter 2 movies.
Bye!
```

提示：考虑函数的形参限定修饰符为 const 的作用与如何在常指针的条件下进行传值。

3.4 数 C 描述与 C primer plus 的不同及疑问

我对比看了一下两本书关于链表的设计和观点，数 C 描述的链表是可以从中间插入或者删除链表元素，但是 C primer plus 主张链表只能从末尾添加，而且无法删除单个元素，只能一次性把链表删除。这就有点奇怪了。但是 C primer plus 的那个程序看上去像是真实项目中的程序，有说服力一点，而数 C 描述上的就比较像是业余选手一样。我比较倾向于相信 C primer plus 书上的观点，毕竟是国外各所名牌大学所用的教材，而且用了那么多年，存在即合理。

3.5 队列_C primer plus 观点

队列（queue）是拥有两个特殊属性的链表（linked list）：
新项只能添加到链表的结尾；
只能从链表的开头移除项；
队列是一种“先进先出”（first in, first out FIFO）的数据形式

3.6 栈 ADT 的指针实现

定义：栈（Stack）是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，称为栈顶（top）。
基本操作：Push（进栈）和 Pop (出栈)，对空栈进行 Pop 一般认为是栈 ADT 错误，另一方面，运行 Push 时空间用尽这是一个实现错误，但不是 ADT 错误。
栈的实现：由于栈是一个表，任何实现表的方法都可以实现栈，可以使用数组，也可以使用指针。

头文件：stack.h

```
/* ****************************************************************
 *         Filename: stack.h
 *     Description:
 *          Version: 1.0
 *          Created: 2017/01/28
 *         Revision: none
 *         Compiler: gcc
 *           Author: Lv Hongbin
 *          Company: Shanghai JiaoTong Univerity
 * ****************************************************************/

#ifndef STACK_H_
#define STACK_H_
#include<stdbool.h>
#define TSIZE 20

typedef struct item{
        char name[TSIZE];
        int age;
}Item;

typedef struct node{
        Item item;
        struct node *next;
}Node;

typedef Node *PtrToNode;
typedef PtrToNode Stack;


bool IsEmpty(const Stack *stack);
```

```
bool IsFull(void);
Stack CreatStack(void);
void InitializeStack(Stack *stack);
bool Push(const Item *item,Stack *stack);
Item Top(const Stack *stack);
bool Pop(Stack *stack);
void MakeEmpty(Stack *stack);
void DisposeStack(Stack *stack);

#endif
```

接口实现文件：stack.c

```
/* ************************************************************
 *           Filename: stack.c
 *    Description:
 *            Version: 1.0
 *            Created: 2017/01/28
 *           Revision: none
 *           Compiler: gcc
 *             Author: Lv Hongbin
 *            Company: Shanghai JiaoTong Univerity
 * ************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"stack.h"

bool IsEmpty(const Stack *stack){

        return (*stack)->next==NULL;
}

bool IsFull(void){

        Node *node=(Node *)malloc(sizeof(Node));
        if(node==NULL){
                return true;
        }else{
                free(node);
                return false;
        }
```

```
}

Stack CreatStack(void){

        if(IsFull()){
                fprintf(stderr,"Out of space!\n");
        }else{
                Stack stack=(Stack)malloc(sizeof(Node));
                stack->next=NULL;
                MakeEmpty(&stack);
                return stack;
        }
}

void InitializeStack(Stack *stack){

        if(IsFull()){
                fprintf(stderr,"The stack is empty!\n");
                exit(0);
        }else{
                memset((*stack)->item.name,'\0',TSIZE);

                (*stack)->item.age=0;
                (*stack)->next=NULL;
        }
}

bool Push(const Item *item,Stack *stack){

        if(IsFull()){
                fprintf(stderr,"Out of space!\n");
                return false;
        }else{
                Node *node=(Node *)malloc(sizeof(Node));
                node->item=*item;
                node->next=(*stack)->next;
                (*stack)->next=node;
                return true;
        }
}

Item Top(const Stack *stack){
```

```
        if(IsEmpty(stack)){
                fprintf(stderr,"The stack is empty!\n");
                exit(0);
        }else{
                return (*stack)->next->item;
        }
}

bool Pop(Stack *stack){

        if(IsEmpty(stack)){
                fprintf(stderr,"The stack is empty!\n");
                return false;
        }else{
                Node *tem=(*stack)->next;
                (*stack)->next=tem->next;
                free(tem);
                tem=NULL;
                return true;
        }

}

void MakeEmpty(Stack *stack){

        if(!IsEmpty(stack)){
                Pop(stack);
                MakeEmpty(stack);
        }
}

void DisposeStack(Stack *stack){


}
```

测试文件：13_stack.c

```
/* ****************************************************************
        *           Filename: 13_stack.c
        *    Description:
        *            Version: 1.0
```

```
 *           Created: 2017/01/28
 *          Revision: none
 *          Compiler: gcc
 *            Author: Lv Hongbin
 *           Company: Shanghai JiaoTong Univerity
 *
********************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"stack.h"

int main(){

        puts("\n/* ***************************************");
        puts("\t\t---Stack---");
        puts("[q] quit\t [a] push an Item\t [s] Top an Item\t [d] Pop an Item\t
[e] empty Stack");
        char cSelect, name[TSIZE];
        int age;
        Item item;
        Stack stack=CreatStack();
        while(1){
                printf("\nPlease input a command: ");
                cSelect=getchar();
                while(getchar()!='\n'){
                        continue;
                }
                if(cSelect=='q'){
                        break;
                }
                if(cSelect=='a'){
                        memset(item.name,'\0',TSIZE);
                        item.age=0;
                        printf("Please input a name: ");
                        scanf("%s",item.name);
                        while(getchar()!='\n'){
                                continue;
                        }
                        printf("Please input an age: ");
                        scanf("%d",&(item.age));
```

```
                        while(getchar()!='\n'){
                                continue;
                        }
                        if(Push(&item,&stack)){
                                puts("Push success!");
                        }else{
                             puts("Push fail!");
                        }
                }
                if(cSelect=='s'){
                        if(IsEmpty(&stack)){
                                puts("The stack is empty!");
                        }else{
                                item=Top(&stack);
                                printf("The                    Top's
name: %s\n",item.name);
                                printf("The                    Top's
age: %d\n",item.age);
                        }
                }
                if(cSelect=='d'){
                        if(IsEmpty(&stack)){
                                puts("The stack is empty");
                        }else{
                                if(Pop(&stack)){
                                        puts("Pop success!");
                                }else{
                                        puts("Pop fail!");
                                }
                        }
                }
                if(cSelect=='e'){
                        MakeEmpty(&stack);
                        puts("Pop finish!");
                }
        }
        puts("\nBye!");
        puts(" * ******************************************");
        return 0;
}
```

结果演示：

```
/* ************************************
        ---Stack---
[q] quit  [a] push an Item  [s] Top an Item  [d] Pop an Item  [e] empty Stack

Please input a command: a
Please input a name: bin
Please input an age: 24
Push success!

Please input a command: a
Please input a name: chao
Please input an age: 20
Push success!

Please input a command: a
Please input a name: lvcaimei
Please input an age: 18
Push success!

Please input a command: s
The Top's name: lvcaimei
The Top's age: 18

Please input a command: d
Pop success!

Please input a command: s
The Top's name: chao
The Top's age: 20

Please input a command: e
Pop finish!

Please input a command: a
Please input a name: qin
Please input an age: 30
Push success!

Please input a command: s
The Top's name: qin
The Top's age: 30
```

```
Please input a command: e
Pop finish!

Please input a command: q

Bye!
 * ******************************************
[lvhongbin@MiWiFi-R3G-srv ~/Desktop/cStudy]$
```

3.7 栈 ADT 的数组实现

建立一个结构体，记录最大容量 Capacity，当前的栈顶元素位置 TopOfStack 和动态数组首地址 array。空数组时 TopOfStack=-1，而每次添加新元素时 TopOfStack++，每次删除元素时 TopOfStack--。

```
#typedef struct stack{
    int Capacity;
    int TopOfStack;
    int *array
} Stack;
```

有一个地方需要数以的是，错误检测，因为对于空战的 Pop 和满栈的 Push 都会超出数组的界限从而引起崩溃。

3.8 栈的应用

1) 平衡符号；
2) 四则混合运算顺序，将一个中缀表达式转换为后缀表达式，这种记法叫后缀（postfix）或者逆波兰（reverse Polish）
   需要 Stack 和 Output 两个栈
3) 函数调用

3.9 队列

1) 跟栈差不多，也可以使用数组来实现：

```
Typedef struct queue{
    int Capacity;
    int Size;
    int Front;
    int Rear;
    int *Array;
```

}Queue;

四、 树

4.1 预备知识

1) 定义树的一种自然的方式是递归的方法，一棵树由一个称作根 root 的节点 r 以及 0 个至多个非空子树 T_1，T_2，T_3…T_k 组成，每一棵子树的根叫做根 r 的儿子（child），而 r 则是各子树的根的父亲（parent），。没有儿子的节点称为树叶（leaf），具有相同父亲的节点称为兄弟（sibling）
2) 一棵树由 N 个节点和 N-1 条线组成
3) 从节点 n_1 到 n_k 的路径（Path）定义为节点 n_1，n_2,…,n_k 的一个序列，使得对于 $1 \leqslant i<k$，节点 n_i 是 n_{i+1} 的父亲，该路径的长（length）为该路径上边的条数，即 k-1。
4) 一棵树从根到每一个节点的路径是唯一的；
5) 对任意节点的深度（Depth）为从根（root）到 n_i 的唯一路径长。
6) 如果存在一条从 n_1 到 n_k 的路径，则称 n_1 是 n_k 的祖先（ancestor），n_k 是 n_1 的后裔（descendant），如果 $n_1 \neq n_k$，则称 n_1 是 n_k 的真祖先（proper ancestor），n_k 是 n_1 的真后裔（proper descendant）
7) 其 C 语言描述为

```
typedef struct trnode{
    Item item;
    struct trnode * FirstChild;
    struct trnode * NextSibling;
}Trnode;
```

4.2 树的遍历

1) 先序遍历(Preorder traversal)

对节点的处理工作是在对它的诸儿子节点被处理之前进行的，在程序中表现为先进行关键操作，再进行递归；
2) 中序遍历(Inorder traversal)

3) 后序遍历(Postorder traversal)

跟先序遍历相反，对节点的处理工作是在对它的诸儿子节点被处理之后进行的，在程序中表现为先进行递归，再进行关键操作；

4.3 二叉树（binary tree）

二叉树是一个树，但是每个节点的数量不能超过 2，平均深度为 $O(N^{0.5})$
其 C 语言描述为：

```
typedef struct trnode{
    Item item;
    struct trnode *Left;
    struct trnode *Right;
}Trnode;
```

4.4 二叉查找树(binary search tree)

1) 性质：对于树中的每个节点 X，它的左子树中所有关键字小于 X 的关键字值，它的右子树中所有关键字大于 X 的关键字值。
2) 懒惰删除：给被删除的元素添加一个删除标记，但是不马上释放其内存，下次如果再用的话，检查此位置是否有被删除的标记，如果有，则不需要重新分配内存，这样做的话可以节省空间，不过可能会添加一个 O(1)时间。
3) 删除：删除右子树的最小元素（这跟 C primer Plus 里面的做法不一样— ^ —）
4) 除了 MakeEmpty 外，其他所有的操作都是 O(d)，其中 d 是包含所访问关键字的节点的深度。
5) 内部路径长（internal path length）：一棵树所有节点的深度的和，那怎么求呢？可以使用递归式，就是去除根部后，左右树各称为单独的树，但是路径长每个节点（除开根节点）都要加 1：

$$\mathrm{D(N)} = \mathrm{D(i)} + \mathrm{D(N-1-i)} + N - 1$$

假设 i 出现的概率相等，则$\mathrm{D(i)} + \mathrm{D(N-1-i)} = \frac{2}{N}\sum_{i=0}^{N-1}\mathrm{D(i)}$

所以又可以改写成

$$\mathrm{D(N)} = \frac{2}{N}\sum_{i=0}^{N-1}\mathrm{D(i)} + N - 1$$

得到平均期望深度为

$$\mathrm{D(N)} = O(N\log N)$$

6) 由于这是内部路经长，平摊到每个节点后，节点的期望深度为$O(\log N)$
7) 但是每次执行删除操作，都会用右子树代替，这样使得二叉查找树严重左倾不平衡。这种不平衡可能会导致平均运行时间大于$O(\log N)$。当然了，这可以通过随机选取右子树的最小元素或者左子树的最大元素替代被删除的元素以消除这种不平衡。若不执行删除或者懒惰删除的操作，所有子树出现的概率相同，则可以断言平均运行时间为$O(\log N)$
8) 如果一棵树全都没有左儿子，那就退化成了链表，而链表的代价非常大，时间就变成是线性的啦，为了避免这种情况的出现，任何节点的深度均不得过深。一般有两种做法：

9) 一是比较古老的方法，一般需要附加一个称为平衡（Balance）的条件，比如AVL 树，平衡查找树。
10) 二是放弃平衡条件，允许树有任意的高度，但是每次操作完后需要使用一个规则进行调整，这种类型的数据结构也叫做自调整（self-adjusting）类结构。如伸展树（splay tree）

4.5 AVL 树

1) AVL(Adelson-Velskii 和 Landis)树是带有平衡条件的二叉查找树。
2) 平衡条件有三种：
   - 最宽松：根节点的左右子树具有相同的高度；
   - 最严格：每个节点的左右节点都必须具有同样的高度；
   - 比较实际的做法：每个节点的左子树和右子树的高度最多相差 1，空树的高度定义为-1。为例做到这一点，需要在每个节点的结构体中添加该节点的高度信息。可以证明，这样的 AVL 树的高度最多为$1.44 \times \log(N+2) - 1.328$，理论上只比$\log(N)$多一点。
3) 旋转：事实上，当我们采取插入操作的时候，就很容易破坏之前平衡的性质，所以外来维持平衡性质，需要在插入完成之前执行旋转操作。
4) 单旋转：往高度低的方向旋转 90° g’g

五、 优先队列（堆）

5.1 模型

允许至少有以下两种操作的数据结构：
- Insert（插入）
- DeleteMin(删除最小者)
  找出，返回和删除优先队列中的最小元素
- 其他的操作属于拓展操作，不属于基本模型

一些实现方案：
- 使用简单链表，执行插入操作，花费 O(1)的时间，然后花费 O(N)的时间寻找最小元并执行删除操作；
- 使用简单链表，花费 O(N)的时间顺序插入，使队列保持有序，然后花费 O(1)的时间执行删除操作；
- 使用二叉查找树，对这两种操作的时间均为 O(logN)，但是经常删除最小元，会使得左子树挖空，右子树过于茂盛，影响二叉树的平衡。而且使用二叉查找树有些过分了，因为它具体很多我们并不需要的操作，关键的是他还需要使用指针。

我们设想寻找一种具有 O(logN)时间复杂度，而且若没有删除干扰，能以线性时间建立一个具有 N 项的优先队列，而且不需要使用指针。

5.2 二叉堆

堆的两个性质：

- 结构性
- 堆序性

六、 分治策略

6.1 总体思路

```
int DivideStrategy(char *a, int left, int right){
    if(left==right){
        return …
    }else{
        int middle=(left+ right)/2;
        left= DivideStrategy(a, left, middle);
        right= DivideStrategy(a, middle+1, right);
        //其他操作；
        return …
    }
}
```

6.2 简单的分治排序策略

$$\mathrm{T}(1) = 1$$
$$\mathrm{T}(N) = 2T(N/2) + N$$

上边两式同时除以 N 得：

$$\frac{\mathrm{T}(N)}{N} = \frac{T(N/2)}{N/2} + 1$$
$$\frac{\mathrm{T}(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$
$$\cdots\cdots$$
$$\frac{\mathrm{T}(N)}{N} = \frac{T(1)}{1} + \log N$$

所以 $$\mathrm{T}(N) = O(N \log N)$$

6.3 计算矩阵乘法的 Strassen 算法

1) 计算 n*n 规模的矩阵乘法，常规的平凡算法 SQUARE-MATRIX-MULTIPLE 时间复杂度是 $\Theta(n^3)$；
2) 使用简单的分治递归法也是 $\Theta(n^3)$的时间复杂度；
3) 使用 Strassen 算法可以使得时间复杂度降低为$\Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.807})$
   据说在 1969 年公布这个算法的时候引起了很大的轰动，因为当时人们并没有料想到竟存在时间复杂度优于 SQUARE-MATRIX-MULTIPLE 的算法，目前 Strassen 算法经过优化的最佳的时间复杂度为$\Theta(n^{2.376})$，显然下界是$\Theta(n^2)$
4) 从使用的角度分析，Strassen 算法通常不是解决矩阵乘法最佳选择，原因有四个：
   - 隐藏在 Strassen 算法的运行时间中的常数因子比过程 SQUARE-MATRIX-MULTIPLE 的 $\Theta(n^3)$时间的常数因子大；
   - 对于稀疏矩阵，专用算法更快；
   - Strassen 算法的数值稳定性不如 SQUARE-MATRIX-MULTIPLE 算法；
   - 递归过程中生成的子矩阵消耗内存空间；

七、 排序和顺序统计量

7.1 相关概念与排序算法比较

一个排序算法描述的是一个具有确定有序次序的方法；
原址的（in place）：如果输入数组中仅有常数个元素需要在排序过程中存储在数组之外；
顺序统计量：一个 n 个数的集合的第 i 个顺序统计量就是集合中第 i 小的数。

算法	类型	最坏运行时间	平均运行时间	是否原址
插入排序	比较排序	$\Theta(n^2)$	$\Theta(n^2)$	是
归并排序	比较排序	$\Theta(n\log n)$	$\Theta(n\log n)$	否
堆排序	比较排序	$O(n\log n)$	$--$	
快速排序	比较排序	$\Theta(n^2)$	$\Theta(n\log n)$	
计数排序		$\Theta(k+n)$	$\Theta(k+n)$	
基数排序		$\Theta(d(k+n))$	$\Theta(d(k+n))$	
桶排序		$\Theta(n^2)$	$\Theta(n)$	

用决策树模型可以证明，比较排序法排序 n 个元素的最坏情况运行时间下界为$\Omega(n\log n)$，从而证明堆排序和归并排序是渐进最优的比较排序算法。

7.2 堆排序

二叉堆：是一个数组，可以被看成是一个近似的完全二叉树：

两个属性： A .length 数组元素的个数； A.heap-size 数组实际堆元素的个数，0<A.heap-size< A .length。
给定一个结点的下标，很容易计算得到它的 ParentNode, LeftNode, RightNode

```
ParentNode(i){
  Return i/2;        //右移一位
}
LeftNode(i){
  Return 2i;         //左移一位
}
RightNode(i){
  Return 2i+1;       //左移一位再在低位加 1；
}
```

最大堆性质：除根以外所有的结点 i 都要满足：A[Parent(i)]≥A[i]，也就是说某个结点的值最多与其父节点一样大，因此最大的元素放在根节点中；
最小堆性质：除根以外所有的结点 i 都要满足：A[Parent(i)]≤A[i]，也就是说某个结点的值最多与其父节点一样小，因此最小的元素放在根节点中；
维护堆的性质：MAX-HEAPIFY，在该函数的执行过程中，将 A[i]，A[left]和 A[right]三者中挑选出最大者，并将其下表储存在在 largest 中

7.3 分治排序

1) 源代码

```
        *
*****************************************************************
        *          Filename: 03_SortDivide.c
        *     Description:
        *           Version: 1.0
        *           Created: 2017/01/27
        *          Revision: none
        *          Compiler: gcc
        *            Author: Lv Hongbin
        *           Company: Shanghai JiaoTong Univerity
        *
*****************************************************************/

        #include<stdio.h>
        #include<stdlib.h>
        #include<time.h>
        #include<sys/time.h>
        #include<string.h>
        void Merge(int *a, int p, int q, int r );
        void SortDivid(int *a, int p, int r);
```

```
void Merge1(int *a,int *b, int p, int q, int r );
void SortDivid1(int *a, int *b,int p, int r);

int main(){

        printf("\n/* ************************************\n");
        printf("Insert Sort\n");
        int N=0;
        printf("Please input N(N<1000) numbers to sort\nN=");
        scanf("%d",&N);
        while(getchar()!='\n'){
                continue;
        }
        char cDirection;
        printf("Please select the direction\na:large->small;\tb:small->large;\nDirection=");
        cDirection=getchar();
        while(getchar()!='\n'){
                continue;
        }
        int i=0,j=0,k=0,tem=0;
        int a[1000],b[1000];
        for (i=0;i<N;i++){
                printf("No.%d:",i);
                scanf("%d",&a[i]);
                while(getchar()!='\n'){
                        continue;
                }
        }
        char cSelect;
        struct timeval StartTime, EndTime;
        double cost;
        while(1){
             puts("\nPlease select one method to sort\n---\t[d] SortDivid\t[D] SortDivid Plus\t[i] Old Insertion\t[q] quit");
    printf("Your selection: ");
                        cSelect=getchar();
                        while(getchar()!='\n'){
                                continue;
                        }
                        if(cSelect=='q'){
                                break;
```

```
                }else if(cSelect=='d'){
                        memset(b,'\0',sizeof(b));
                        for(i=0;i<N;i++){
                                b[i]=a[i];
                        }
                        gettimeofday(&StartTime,NULL);
                        SortDivid(b,0,N-1);
                        gettimeofday(&EndTime,NULL);
                        cost=(EndTime.tv_sec-
StartTime.tv_sec)*1000000+(EndTime.tv_usec-StartTime.tv_usec);
                        if(cDirection=='a'){
                                printf("Sort from large one to samll
one: \n");
                                for(i=0;i<N;i++){
                                        printf("%d\t",b[i]);
                                }
                        }else{
                                printf("Sort from samll one to large
one: \n");
                                for(i=0;i<N;i++){
                                        printf("%d\t",b[N-i-1]);
                                }
                        }
                        printf("\nThe time cost is %.f\n",cost);
                }else if(cSelect=='D'){
                        memset(b,'\0',sizeof(b));
                        for(i=0;i<N;i++){
                                b[i]=a[i];
                        }
                        int *tem=malloc(1000*sizeof(int));
                        gettimeofday(&StartTime,NULL);
                        SortDivid1(b,tem,0,N-1);
                        gettimeofday(&EndTime,NULL);
                        free(tem);
                        cost=(EndTime.tv_sec-
StartTime.tv_sec)*1000000+(EndTime.tv_usec-StartTime.tv_usec);
                        if(cDirection=='a'){
                                printf("Sort from large one to samll
one: \n");
                                for(i=0;i<N;i++){
                                        printf("%d\t",b[i]);
                                }
```

```
                }else{
                        printf("Sort from samll one to large one: \n");
                        for(i=0;i<N;i++){
                                printf("%d\t",b[N-i-1]);
                        }
                }
                printf("\nThe time cost is %.f\n",cost);
        }else if(cSelect=='i'){
                memset(b,'\0',sizeof(b));
                for(i=0;i<N;i++){
                        b[i]=a[i];
                }
                gettimeofday(&StartTime,NULL);
                for(j=1;j<N;j++){
                        tem = b[j];
                        for(i=0;i<j;i++){
                                if(b[i]<tem){
                                        for(k=j;k>i;k--){
b[k]=b[k-1];
                                        }
                                        b[i]=tem;
                                        break;
                                }
                        }
                }
                gettimeofday(&EndTime,NULL);
                cost=(EndTime.tv_sec-StartTime.tv_sec)*1000000+(EndTime.tv_usec-StartTime.tv_usec);
                if(cDirection=='a'){
                        printf("Sort from large one to samll one: \n");
                        for(i=0;i<N;i++){
                                printf("%d\t",b[i]);
                        }
                }else{
                        printf("Sort from samll one to large one: \n");
                        for(i=0;i<N;i++){
                                printf("%d\t",b[N-i-1]);
                        }
```

```
                        }
                        printf("\nThe time cost is %.f\n",cost);

                }
        }
        printf("\n* ************************************/\n");
        return 0;
}

s_gets(char a[],int N){

        int i=0;
        if(fgets(a,N,stdin)!=NULL){
                while(a[i]!='\n' && a[i]!='\0')
                        i++;
                if(a[i]=='\n')
                        a[i]='\0';
                else{
                        while(getchar()!='\n')
                                continue;
}
        }

}

void SortDivid(int *a, int p, int r){

        int q=(p+r)/2;
        if(p<r){
                SortDivid(a,p,q);
                SortDivid(a,q+1,r);
                Merge(a,p,q,r);
        }

}

void Merge(int *a, int p, int q, int r ){

        int i=0, j=0, k=0;
        int *b=malloc((r-p+1)*sizeof(int));
        for(k=0;k<r-p+1;k++){
                if(i>q-p){
```

```
                        b[k]=a[q+1+j];
                        j++;
                }else if(j>r-q-1){
                        b[k]=a[p+i];
                        i++;
                }else{
                        if(a[p+i]>a[q+1+j]){
                                b[k]=a[p+i];
                                i++;
                        }else{
                                b[k]=a[q+1+j];
                                j++;
                        }
                }
        }
        for(k=0;k<r-p+1;k++){
                a[p+k]=b[k];
        }
        free(b);
}

void SortDivid1(int *a, int *b, int p, int r){

        int q=(p+r)/2;
        if(p<r){
                SortDivid1(a,b,p,q);
                SortDivid1(a,b,q+1,r);
            Merge1(a,b,p,q,r);
       }

}

void Merge1(int *a,int *b, int p, int q, int r ){

        int i=0, j=0, k=0;
        for(k=0;k<r-p+1;k++){
                if(i>q-p){
                        b[k]=a[q+1+j];
                        j++;
                }else if(j>r-q-1){
                        b[k]=a[p+i];
                        i++;
```

```
                }else{
                        if(a[p+i]>a[q+1+j]){
                                b[k]=a[p+i];
                                i++;
                        }else{
                                b[k]=a[q+1+j];
                                j++;
                        }
                }
        }
        for(k=0;k<r-p+1;k++){
                a[p+k]=b[k];
    }
```

2) 有三种分治的方法
   a) 每个递归调用局部都会声明一个临时数组，那么在任意时刻都会有 logN 个临时数组处于活动期，这对于小内存的机器而言是致命的
   b) 每个 Merge 例程都动态分配并释放最小量的临时内存，那么由 malloc 占用的时间会很多。严密测试指出，当 merge 例程处于最后一行的时候，每个时刻只需要一个临时的数组活动，而且可以使用该临时数组的任意部分
   c) 先动态创建一个与跟原数组等长的内存，最后再释放。

   所以我们使用第三种方法，节省时间和空间。
   调用的时间如下所示：

   比较两种分治方法的

```
[lvhongbin@MiWiFi-R3G-srv ~/Desktop/cStudy]$ ./03_SortDivide.out

/* ************************************
Insert Sort
Please input N(N<1000) numbers to sort
N=20
Please select the direction
a:large->small;   b:small->large;
Direction=a
No.0:1
No.1:3
No.2:5
No.3:7
No.4:9
No.5:2
No.6:4
No.7:6
```

```
No.8:8
No.9:0
No.10:1
No.11:3
No.12:5
No.13:7
No.14:9
No.15:2
No.16:4
No.17:6
No.18:8
No.19:0

Please select one method to sort
---  [d] SortDivid      [D] SortDivid Plus     [i] Old Insertion  [q] quit
Your selection: d
Sort from large one to samll one:
9   9   8   8   7   7   6   6   5   5   4   4   3   3   2   2   1   1
0   0
The time cost is 49

Please select one method to sort
---  [d] SortDivid      [D] SortDivid Plus     [i] Old Insertion  [q] quit
Your selection: d
Sort from large one to samll one:
9   9   8   8   7   7   6   6   5   5   4   4   3   3   2   2   1   1
0   0
The time cost is 3

Please select one method to sort
---  [d] SortDivid      [D] SortDivid Plus     [i] Old Insertion  [q] quit
Your selection: d
Sort from large one to samll one:
9   9   8   8   7   7   6   6   5   5   4   4   3   3   2   2   1   1
0   0
The time cost is 3

Please select one method to sort
---  [d] SortDivid      [D] SortDivid Plus     [i] Old Insertion  [q] quit
Your selection: D
Sort from large one to samll one:
9   9   8   8   7   7   6   6   5   5   4   4   3   3   2   2   1   1
```

0 0
The time cost is 2

Please select one method to sort
--- [d] SortDivid [D] SortDivid Plus [i] Old Insertion [q] quit
Your selection: D
Sort from large one to samll one:
9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
0 0
The time cost is 1

Please select one method to sort
--- [d] SortDivid [D] SortDivid Plus [i] Old Insertion [q] quit
Your selection: D
Sort from large one to samll one:
9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
0 0
The time cost is 2

Please select one method to sort
--- [d] SortDivid [D] SortDivid Plus [i] Old Insertion [q] quit
Your selection: i
Sort from large one to samll one:
9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
0 0
The time cost is 1

Please select one method to sort
--- [d] SortDivid [D] SortDivid Plus [i] Old Insertion [q] quit
Your selection: i
Sort from large one to samll one:
9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
0 0
The time cost is 1

Please select one method to sort
--- [d] SortDivid [D] SortDivid Plus [i] Old Insertion [q] quit
Your selection: i
Sort from large one to samll one:
9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
0 0
The time cost is 1

```
Please select one method to sort
---  [d] SortDivid      [D] SortDivid Plus     [i] Old Insertion  [q] quit
Your selection: q

* ************************************/
```

八、 概率分析和随机算法

九、 数组

1) 数组的声明和定义
   类型说明符 数组标识符[常量表达式]
   如：int array[5];
2) ；
3) ；
4) ；
5)