目录

c基础

数据类型说明

数据类型	16位平台	32位平台	64位平台
char	1 字节	1 字节	1 字节
pointer	2 字节	4 字节	8 字节

数据类型	16位平台	32位平台	64位平台
short	2 字节	2 字节	2 字节
int	2 字节	4 字节	4 字节
float	4 字节	4 字节	4 字节
double	8 字节	8 字节	8 字节
long	4 字节	4 字节	8 字节
long long	8 字节	8 字节	8 字节

volatile

volatile 指出变量是随时可能发生变化的，每次使用它的时候必须从变量的地址中读取，因而编译器生成的汇编代码会重新从变量的地址读取数据。

1. 并行设备的硬件寄存器，
2. 一个中断服务子程序中会访问到的非自动变量（Non-automatic variables),可以使用关键区保护
3. 多线程应用中被几个任务共享的变量，可以关闭系统调度

指针

函数指针

```
int (*fun)(int *a);
```

函数指针数组

```
int (*fun[10])(int *data, int size);
```

使用方法：

```
int (*sort_fun[5])(int *data, int size) = {
    quick_sort,     /* 快速排序 */
    insert_sort,    /* 插入排序 */
    bubble_sort,    /* 冒泡排序 */
    heap_sort,      /* 堆排序 */
    selection_sort  /* 选择排序 */
};
// 或者
sort_fun[0] = quick_sort;
sort_fun[1] = insert_sort;
```

指针数组

```
int *a[10];
```

数组指针

```
/* a为指向含10个元素的一维数组的指针变量，
 * ()优先级高，说明a是一个指针，指向一个整型的一维数组.
 * a+1时，a要跨过10个整型数据的长度 */
int (*a)[10];
```

指针的指针

```
int **a;
```

main函数的返回值

1. 0 表示程序正常退出
2. 负数表示程序异常退出

const

const T

定义一个常量，声明的同时必须进行初始化。一旦声明，这个值将不能被改变。

```
const int constInt = 10;        //正确
constInt = 20;                  //错误，常量值不可被改变
const int constInt3;            //错误，未被初始化
```

const T*

指向常量的指针，不能用于改变其所指向的对象的值。

```
const int i = 5;
const int i2 = 10;
const int* pInt = &i;           //正确，指向一个const int对象，即i的地址
//*pInt = 10;                   //错误，不能改变其所指缶的对象
pInt = &i2;                     //正确，可以改变pInt指针本身的值,此时pInt指向的是i2的
地址
const int* p2 = new int(8);     //正确，指向一个new出来的对象的地址
delete p2;                      //正确
//int* pInt = &i;               //错误，i是const int类型，类型不匹配，不能将const int
* 初始化为int *
```

```
int nValue = 15;
const int * pConstInt = &nValue;     //正确，可以把int *赋给const int *，但是pConstInt不能改变其所指向对象的值，即nValue
*pConstInt = 40;                     //错误，不能改变其所指向对象的值
```

const int* 与int* const的区别

指针本身就是一种对象，把指针定义为常量就是常量指针，也就是int* const的类型，也可以写成int *const，声明时必须初始化。

```
int nValue = 10;
int* const p = &nValue;
int *const p2 = &nValue;
//const int* 指针指向的对象不可以改变，但指针本身的值可以改变；int* const 指针本身的值不可改变，但其指向的对象可以改变。
int nValue1 = 10;
int nValue2 = 20;
int* const constPoint = &nValue1;
//constPoint = & nValue2;            //错误，不能改变constPoint本身的值
*constPoint = 40;                    //正确，可以改变constPoint所指向的对象，此时nValue1 = 40


const int nConstValue1 = 5;
const int nConstValue2 = 15;
const int* pPoint = &nConstValue1;
//*pPoint  = 55;                     //错误，不能改变pPoint所指向对象的值
pPoint = &nConstValue2;              //正确，可以改变pPoint指针本身的值，此时pPoint邦定的是nConstValue2对象，即pPoint为nConstValue2的地址
```

const int* const 是一个指向常量对象的常量指针，即不可以改变指针本身的值，也不可以改变指针指向的对象。

```
const int nConstValue1 = 5;
const int nConstValue2 = 15;
const int* const pPoint = &nConstValue1;
//*pPoint  = 55;                     //错误，不能改变pPoint所指向对象的值
//pPoint = &nConstValue2;            //错误，不能改变pPoint本身的值
```

const助记方法

把一个声明从右向左读。(* 读成 pointer to)

```
char * const cp;  // cp is a const pointer to char
const char * p;   // p is a pointer to const char;
char const * p;   // 同上,因为C++里面没有const*的运算符，所以const只能属于前面的类型。
```

C++标准规定，const关键字放在类型或变量名之前等价的。

结论：

```
char * const cp     // 定义一个指向字符的指针常数，即const指针
const char* p       // 定义一个指向字符常数的指针
char const* p       // 等同于const char* p
const char **       // 是一个指向指针的指针，那个指针又指向一个字符串常量。
char **             // 也是一个指向指针的指针，那个指针又指向一个字符串变量。
```

浮点数存储方式

1. float 占用4个字节，32bits

符号位	指数位	尾数部分
1 bits	8 bits	23 bits

2. double 占用8字节，64bits

符号位	指数位	尾数部分
1 bits	11 bits	52 bits

c题目

- printf返回值
- enum枚举类型
- 可变参数函数

printf返回值

```
#include <stdio.h>
int main() {
  int i = 43;
  printf("%d\n", printf("%d", printf("%d", i)));
  return 0;
}
```

输出：4321 printf返回值是输出字符的个数（不包括字符串结尾\x00）

enum枚举类型

```
#include <stdio.h>
int main() {
```

```
  enum color{
    RED,
    BLUE,
    GREEN = -2,
    YELLOW,
    PINK
  };
  printf("%d %d\n", BLUE, PINK);
  return 0;
}
```

输出：1 0 enum默认是从0开始的，所以RED = 0， BLUE = 1， GREEN = -2， YELLOW = -1， PINK = 0；

可变参数函数

```
#include "stdarg.h"

char buf[512] = {0};

int func(const char *fmt, ...)
{
  va_list args;
  va_start(args, fmt);
  vsprintf(buf, fmt, args);
  va_end(args);
}
```

大小端区分

```
union data {
    int a;
    char b;
};

struct def {
    union data mine;
};

struct def endian;

int main(int argc, char **argv)
{
    endian.mine.a = 0x12345678;
    printf("%02X\n", endian.mine.b);

    return 0;
}
/*打印 78 说明是小端， 12 说明是大端 */
```

链表

```
//////////////////////////////////////////
//单链表的初始化，建立，插入，查找，删除。//
//Author:Wang Yong                      //
//Date: 2010.8.19                       //
//////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
typedef int ElemType;
//////////////////////////////////////////
//定义结点类型
typedef struct Node
{
    ElemType data;              //单链表中的数据域
    struct Node *next;          //单链表的指针域
}Node,*LinkedList;
//////////////////////////////////////////
//单链表的初始化
LinkedList LinkedListInit()
{
    Node *L;
    L = (Node *)malloc(sizeof(Node));   //申请结点空间
    if(L == NULL)                       //判断是否有足够的内存空间
        printf("申请内存空间失败/n");
    L->next = NULL;                 //将next设置为NULL,初始长度为0的单链表
}
//////////////////////////////////////////
//单链表的建立1，头插法建立单链表
LinkedList LinkedListCreatH()
{
    Node *L;
    L = (Node *)malloc(sizeof(Node));   //申请头结点空间
    L->next = NULL;                      //初始化一个空链表

    ElemType x;                         //x为链表数据域中的数据
    while(scanf("%d",&x) != EOF)
    {
        Node *p;
        p = (Node *)malloc(sizeof(Node));   //申请新的结点
        p->data = x;                        //结点数据域赋值
        p->next = L->next;                      //将结点插入到表头L-->|2|-->|1|--
>NULL
        L->next = p;
    }
    return L;
}
//////////////////////////////////////////
//单链表的建立2，尾插法建立单链表
LinkedList LinkedListCreatT()
{
    Node *L;
```

```
    L = (Node *)malloc(sizeof(Node));   //申请头结点空间
    L->next = NULL;                    //初始化一个空链表
    Node *r;
    r = L;                             //r始终指向终端结点，开始时指向头结点
    ElemType x;                            //x为链表数据域中的数据
    while(scanf("%d",&x) != EOF)
    {
        Node *p;
        p = (Node *)malloc(sizeof(Node));   //申请新的结点
        p->data = x;                        //结点数据域赋值
        r->next = p;                   //将结点插入到表头L-->|1|-->|2|-->NULL
        r = p;
    }
    r->next = NULL;

    return L;
}
//////////////////////////////////////////
//单链表的插入，在链表的第i个位置插入x的元素
LinkedList LinkedListInsert(LinkedList L,int i,ElemType x)
{
    Node *pre;                         //pre为前驱结点
    pre = L;
    int tempi = 0;
    for (tempi = 1; tempi < i; tempi++)
        pre = pre->next;                   //查找第i个位置的前驱结点
    Node *p;                                  //插入的结点为p
    p = (Node *)malloc(sizeof(Node));
    p->data = x;
    p->next = pre->next;
    pre->next = p;

    return L;
}
//////////////////////////////////////////
//单链表的删除，在链表中删除值为x的元素
LinkedList LinkedListDelete(LinkedList L,ElemType x)
{
    Node *p,*pre;                      //pre为前驱结点，p为查找的结点。
    p = L->next;
    while(p->data != x)                //查找值为x的元素
    {
        pre = p;
        p = p->next;
    }
    pre->next = p->next;           //删除操作，将其前驱next指向其后继。
    free(p);
    return L;
}
//////////////////////////////////////////
int main()
{
    LinkedList list,start;
/*  printf("请输入单链表的数据：");
```

```
    list = LinkedListCreatH();
    for(start = list->next; start != NULL; start = start->next)
        printf("%d ",start->data);
    printf("/n");
*/  printf("请输入单链表的数据：");
    list = LinkedListCreatT();
    for(start = list->next; start != NULL; start = start->next)
        printf("%d ",start->data);
    printf("/n");
    int i;
    ElemType x;
    printf("请输入插入数据的位置：");
    scanf("%d",&i);
    printf("请输入插入数据的值：");
    scanf("%d",&x);
    LinkedListInsert(list,i,x);
    for(start = list->next; start != NULL; start = start->next)
        printf("%d ",start->data);
    printf("/n");
    printf("请输入要删除的元素的值：");
    scanf("%d",&x);
    LinkedListDelete(list,x);
    for(start = list->next; start != NULL; start = start->next)
        printf("%d ",start->data);
    printf("/n");

    return 0;
}
```

排序算法

排序算法有：

选择排序　快速排序　希尔排序　冒泡排序　插入排序　堆排序　归并排序

排序算法比较：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

选择排序

选择排序是一种简单直观的排序算法，无论什么数据进去都是 $O(n^2)$ 的时间复杂度。所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

时间复杂度

```
平均: O(n2) 最好: O(n2) 最坏:O(n2)
```

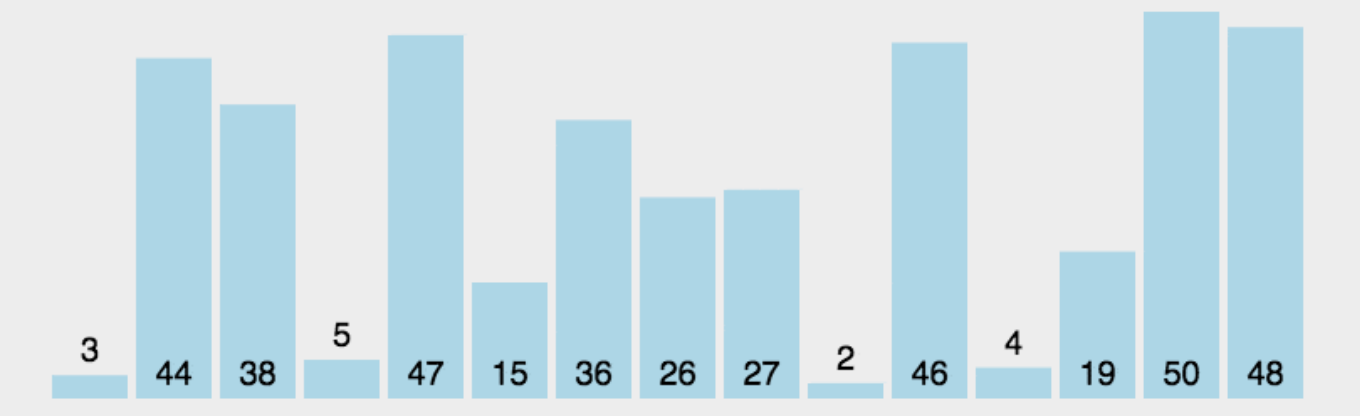
空间复杂度

```
O(1)
```

算法步骤

1. 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
2. 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
3. 重复第二步，直到所有元素均排序完毕。

动图演示




```
/* 直接选择排序 */
void selection_sort(int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (x[j] < x[min])
            {
                min = j;
            }
        }
        if (min != i)
        {
            swap(x,i,min);
        }
    }
}
```

插入排序

插入排序的代码实现虽然没有冒泡排序和选择排序那么简单粗暴，但它的原理应该是最容易理解的了，因为只要打过扑克牌的人都应该能够秒懂。插入排序是一种最简单直观的排序算法，它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。 插入排序和冒泡排序一样，也有一种优化算法，叫做拆半插入。

时间复杂度

```
平均：O(n2) 最好：O(n) 最坏：O(n2)
```

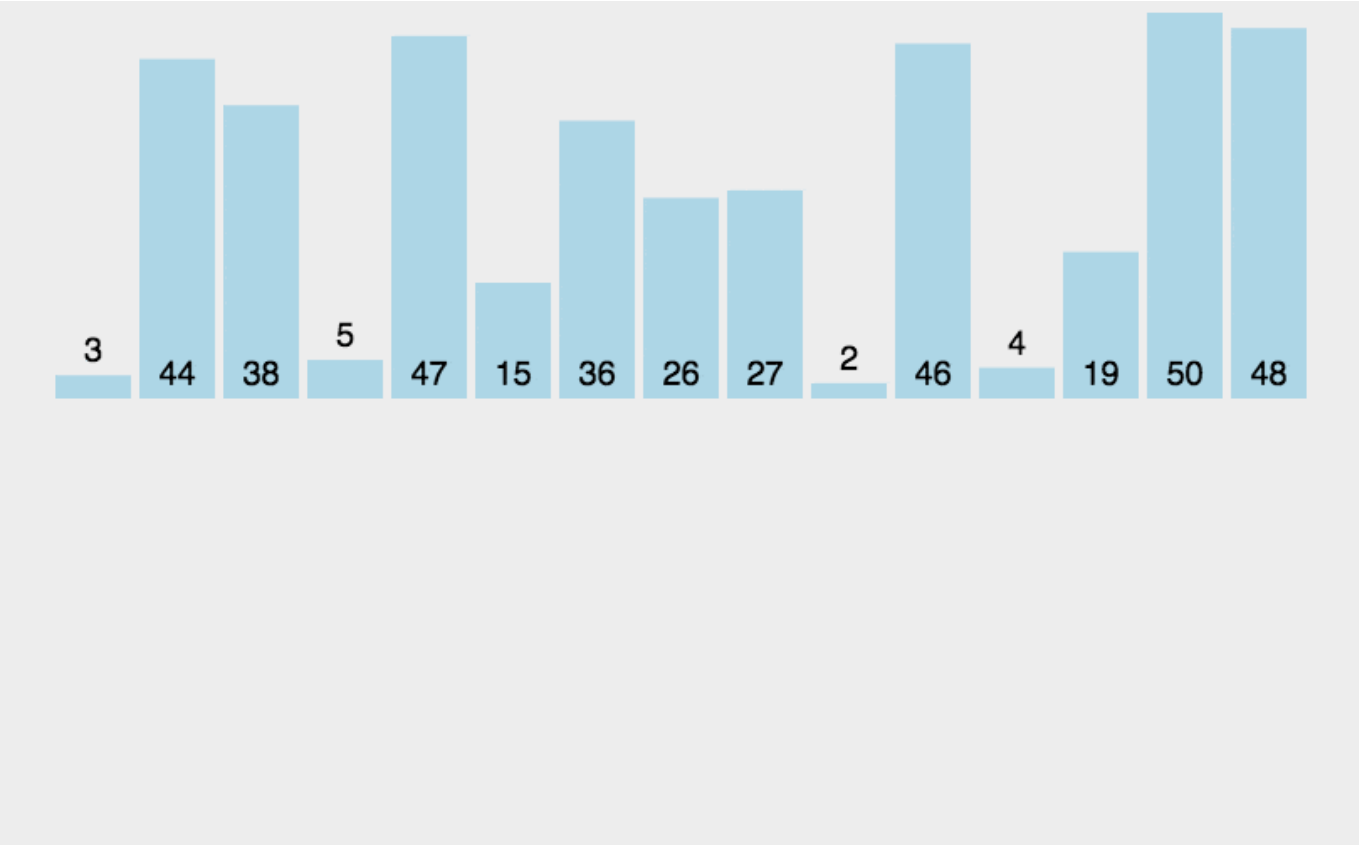
空间复杂度

```
O(1)
```

算法步骤

1. 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
2. 从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

动图演示




```
/* 直接插入排序 */
void insertion_sort(int n)
{
    for(int i = 1; i < n; i++)
    {
        int t = x[i];
        int j;
        for (j = i; j > 0 && x[j-1] > t ; j--)
        {
            x[j] = x[j - 1];
        }
        x[j] = t;
    }
}
```

希尔排序

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法。 希尔排序是基于插入排序的以下两点性质而提出改进方法的：

- 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率；
- 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位； 希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。

时间复杂度

```
平均：O(n1.5) 最好：O(n) 最坏O(n2)
```

空间复杂度

```
O(1)
```

算法步骤

1. 选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；
2. 按增量序列个数 k，对序列进行 k 趟排序；
3. 每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```
/* 希尔排序 */
void shell_sort(int n)
{
    for (int inc = n/3; inc >= 1; inc /= 3 )
    {
        for (int i = inc; i < n; i++)
        {
            int t = x[i];
            int j;
            for (j = i; j >= inc && x[j - inc] > t ; j -= inc)
            {
                x[j] = x[j - inc];
            }
            x[j] = t;
        }
    }
}
```

冒泡排序

时间复杂度

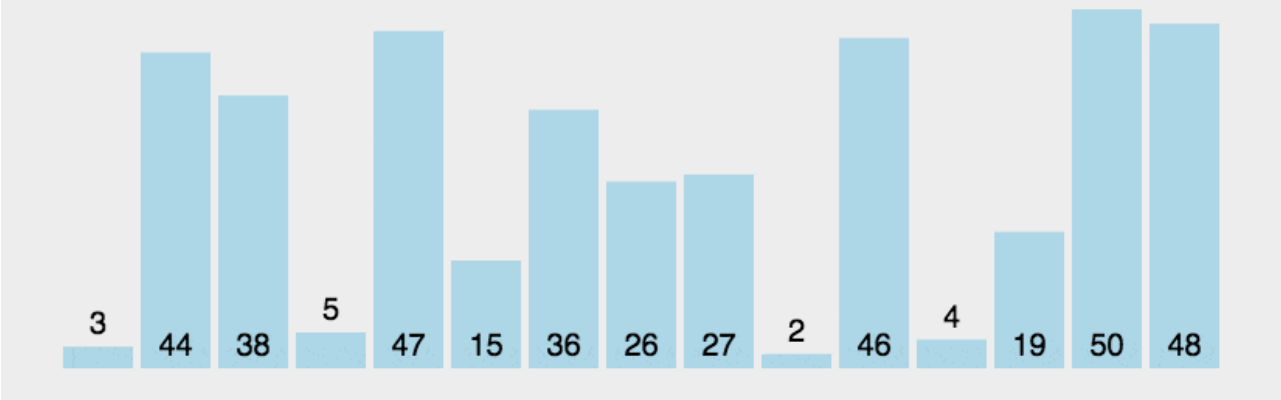
```
平均：O(n2) 最好：O(n) 最坏：O(n2)
```

空间复杂度

```
O(1)
```

算法步骤

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。 动图演示




```c
/* 冒泡排序 */
void bubble_sort(int n)
{
    bool change = true;
    for (int i = n-1; i >= 1 && change; i--)
    {
        change = false;
        for (int j = 0; j < i; j++)
        {
            if(x[j] > x[j + 1])
            {
                swap(x, j, j + 1);
                change = true;
            }
        }
    }
}
```

快速排序

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个项目要 O(nlogn) 次比较。在最坏状况下则需要 O(n2) 次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 O(nlogn) 算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。 快速排序使用分治法（Divide and conquer）策略来把一个串行（list）分为两个子串行（sub-lists）。

快速排序又是一种分而治之思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法。

快速排序的名字起的是简单粗暴，因为一听到这个名字你就知道它存在的意义，就是快，而且效率高！它是处理大数据最快的排序算法之一了。虽然 Worst Case 的时间复杂度达到了 $O(n^2)$，但是人家就是优秀，在大多数情况下都比平均时间复杂度为 O(n logn) 的排序算法表现要更好，可是这是为什么呢，我也不知道。好在我的强迫症又犯了，查了 N 多资料终于在《算法艺术与信息学竞赛》上找到了满意的答案：

> 快速排序的最坏运行情况是 $O(n^2)$，比如说顺序数列的快排。但它的平摊期望时间是 O(nlogn)，且 O(nlogn) 记号中隐含的常数因子很小，比复杂度稳定等于 O(nlogn) 的归并排序要小很多。所以，对绝大多数顺序性较弱的随机数列而言，快速排序总是优于归并排序。

时间复杂度

```
平均: O(nlogn) 最好: O(nlogn) 最坏:O(n2)
```

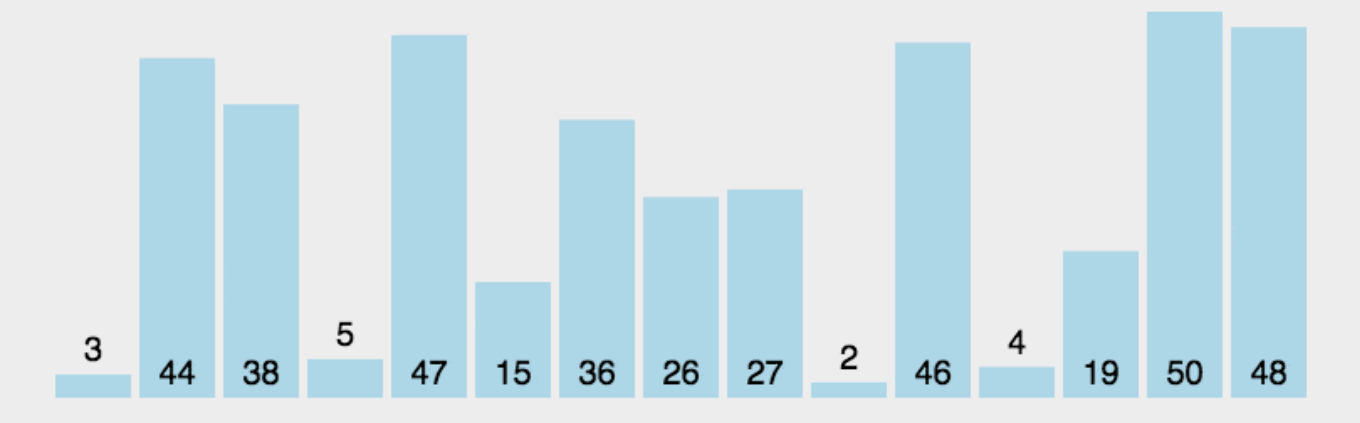
空间复杂度

```
O(nlogn)    用于方法栈
```

算法步骤

1. 从数列中挑出一个元素，称为 “基准”（pivot）；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序； 递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会退出，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

动图演示




```
/* 快速排序 */
void quick_sort(int l, int u)
{
    if (l >= u)
        return;
    int m = l;
    for (int i = l+1; i<= u; i++)
    {
        if(x[i] < x[l])
            swap(x, ++m, i);
    }
    swap(x, l, m);
    quick_sort(l, m-1);
    quick_sort(m+1, u);
}
```

hash算法

hash构造方法

1. 直接寻址法：取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a?key + b，其中a和b为常数（这种散列函数叫做自身函数）
2. 数字分析法：分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址。
3. 平方取中法：取关键字平方后的中间几位作为散列地址。
4. 折叠法：将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。
5. 随机数法：选择一随机函数，取关键字的随机值作为散列地址，通常用于关键字长度不同的场合。
6. 除留余数法：取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p, p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取素数或m，若p选的不好，容易产生同义词。

hash冲突及解决

冲突原因：

1. 与散列函数有关，一个好的散列函数的值应尽可能平均分布。
2. 与解决冲突的哈希冲突函数有关。
3. 与负载因子的大小。太大不一定就好，而且浪费空间严重，负载因子和散列函数是联动的。

解决冲突的办法

- 开放定址法：线性探查法、平方探查法、伪随机序列法、双哈希函数法。
- 拉链法：把所有同义词，即hash值相同的记录，用单链表连接起来。

常用的hash函数

```
unsigned int RSHash(char* str, unsigned int len)
{
    unsigned int b    = 378551;
    unsigned int a    = 63689;
    unsigned int hash = 0;
    unsigned int i    = 0;

    for(i = 0; i < len; str++, i++)
    {
        hash = hash * a + (*str);
        a    = a * b;
    }

    return hash;
}
/* End Of RS Hash Function */

unsigned int JSHash(char* str, unsigned int len)
{
    unsigned int hash = 1315423911;
    unsigned int i    = 0;

    for(i = 0; i < len; str++, i++)
    {
        hash ^= ((hash << 5) + (*str) + (hash >> 2));
    }

    return hash;
}
/* End Of JS Hash Function */

unsigned int ELFHash(char* str, unsigned int len)
{
    unsigned int hash = 0;
    unsigned int x    = 0;
    unsigned int i    = 0;

    for(i = 0; i < len; str++, i++)
    {
        hash = (hash << 4) + (*str);
```

```
        if((x = hash & 0xF0000000L) != 0)
        {
            hash ^= (x >> 24);
        }
        hash &= ~x;
    }

    return hash;
}
/* End Of ELF Hash Function */

unsigned int DJBHash(char* str, unsigned int len)
{
    unsigned int hash = 5381;
    unsigned int i    = 0;

    for(i = 0; i < len; str++, i++)
    {
        hash = ((hash << 5) + hash) + (*str);
    }

    return hash;
}
/* End Of DJB Hash Function */
```

javascript数据类型

字符串 数字 布尔 对象 数组 undefined null

> js只有一种数字类型 undefined：当声明的变量没有初始化的话，那这个变量的值就是undefined null表示尚未存在的对象。