# 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

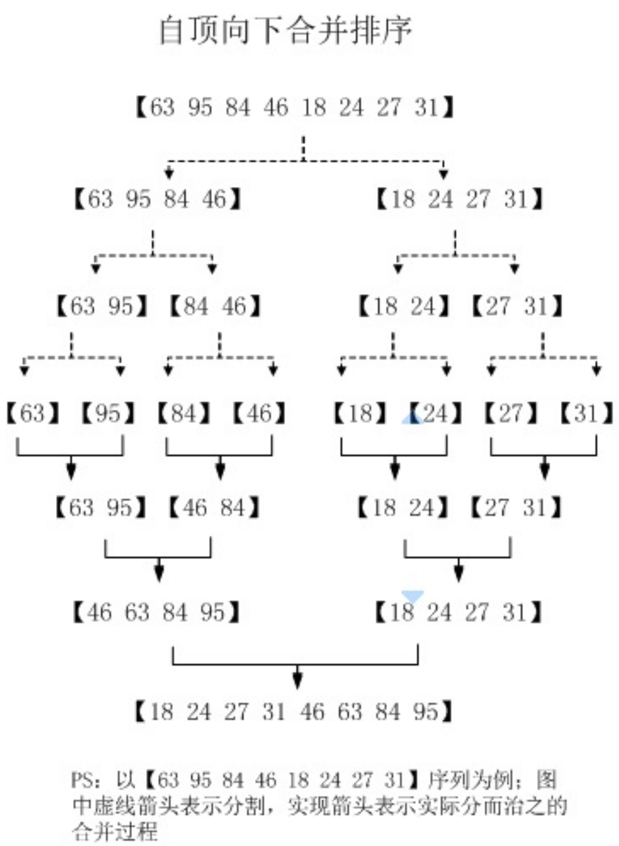
- 将一个新的数据放到原有有序的数列中——O(N)
- 动态缓慢地加入新的数据

## 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




- 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*

```c
#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：

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

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

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]