# 复杂度

## 时间复杂度

### 概念

时间复杂度是总运算次数表达式中受n的变化影响最大的那一项(不含系数)

比如：一般总运算次数表达式类似于这样：

a\*2^n+b\*n^3+c\*n^2+d\*n\*lg(n)+e\*n+f

a!=0时，时间复杂度就是O(2^n);

a=0,b<>0 =>O(n^3);

a,b=0,c<>0 =>O(n^2)依此类推

举例：

(1) for(i=1;i<=n;i++) //循环了n\*n次，当然是O(n^2)

for(j=1;j<=n;j++)

s++;

(2) for(i=1;i<=n;i++)//循环了(n+n-1+n-2+...+1)≈(n^2)/2，因为时间复杂度是不考虑系数的，所以也是O(n^2)

for(j=i;j<=n;j++)

s++;

(3) for(i=1;i<=n;i++)//循环了(1+2+3+...+n)≈(n^2)/2,当然也是O(n^2)

for(j=1;j<=i;j++)

s++;

(4) i=1;k=0;

while(i<=n-1){

k+=10\*i;

i++;

}

//循环了

n-1≈n次，所以是O(n)

(5) for(i=1;i<=n;i++)

for(j=1;j<=i;j++)

for(k=1;k<=j;k++)

x=x+1;

//

循环了(1^2+2^2+3^2+...+n^2)=n(n+1)(2n+1)/6(这个公式要记住哦)≈(n^3)/3，不考虑系数，自然是O(n^3)

另外，在时间复杂度中，log(2,n)(以2为底)与lg(n)(以10为底)是等价的，因为对数换底公式：

log(a,b)=log(c,b)/log(c,a)

所以，log(2,n)=log(2,10)\*lg(n),忽略掉系数，二者当然是等价的

### 计算方法

1、一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。

一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)。

2、一般情况下，算法的基本操作重复执行的次数是模块n的某一个函数f（n），因此，算法的时间复杂度记做：T（n）=O（f（n））。随着模块n的增大，算法执行的时间的增长率和f（n）的增长率成正比，所以f（n）越小，算法的时间复杂度越低，算法的效率越高。

在计算时间复杂度的时候，先找出算法的基本操作，然后根据相应的各语句确定它的执行次数，再找出T（n）的同数量级（它的同数量级有以下：1，Log2n ，n ，nLog2n ，n的平方，n的三次方，2的n次方，n！），找出后，f（n）=该数量级，若T(n)/f(n)求极限可得到一常数c，则时间复杂度T（n）=O（f（n））。

3、常见的时间复杂度

按数量级递增排列，常见的时间复杂度有：

常数阶O(1), 对数阶O(log2n), 线性阶O(n), 线性对数阶O(nlog2n), 平方阶O(n^2)， 立方阶O(n^3),...， k次方阶O(n^k), 指数阶O(2^n) 。

其中，

1、O(n)，O(n^2)， 立方阶O(n^3),...， k次方阶O(n^k) 为多项式阶时间复杂度，分别称为一阶时间复杂度，二阶时间复杂度。

2、O(2^n)，指数阶时间复杂度，该种不实用

3、对数阶O(log2n), 线性对数阶O(nlog2n)，除了常数阶以外，该种效率最高

例：算法：

for（i=1;i<=n;++i）

{

for(j=1;j<=n;++j)

{

c[ i ][ j ]=0; //该步骤属于基本操作 执行次数：n^2

for(k=1;k<=n;++k)

c[ i ][ j ]+=a[ i ][ k ]\*b[ k ][ j ];

//该步骤属于基本操作 执行次数：n^3

}

}

则有 T（n）= n^2+n^3，根据上面括号里的同数量级，我们可以确定 n^3为T（n）的同数量级

则有f（n）= n^3，然后根据T（n）/f（n）求极限可得到常数c

则该算法的 时间复杂度：T（n）=O（n^3)

### 定义

如果一个问题的规模是n，解这一问题的某一算法所需要的时间为T(n)，它是n的某一函数 T(n)称为这一算法的“时间复杂性”。

当输入量n逐渐加大时，时间复杂性的极限情形称为算法的“渐近时间复杂性”。

我们常用大O表示法表示时间复杂性，注意它是某一个算法的时间复杂性。大O表示只是说有上界，由定义如果f(n)=O(n)，那显然成立f(n)=O(n^2)，它给你一个上界，但并不是上确界，但人们在表示的时候一般都习惯表示前者。

此外，一个问题本身也有它的复杂性，如果某个算法的复杂性到达了这个问题复杂性的下界，那就称这样的算法是最佳算法。

“大O记法”：在这种描述中使用的基本参数是 n，即问题实例的规模，把复杂性或运行时间表达为n的函数。这里的“O”表示量级 (order)，比如说“二分检索是 O(logn)的”,也就是说它需要“通过logn量级的步骤去检索一个规模为n的数组”记法 O ( f(n) )表示当 n增大时，运行时间至多将以正比于 f(n)的速度增长。

这种渐进估计对算法的理论分析和大致比较是非常有价值的，但在实践中细节也可能造成差异。例如，一个低附加代价的O(n2)算法在n较小的情况下可能比一个高附加代价的 O(nlogn)算法运行得更快。当然，随着n足够大以后，具有较慢上升函数的算法必然工作得更快。

拓展：O(1)

Temp=i;i=j;j=temp;

以上三条单个语句的频度均为1，该程序段的执行时间是一个与问题规模n无关的常数。算法的时间复杂度为常数阶，记作T(n)=O(1)。如果算法的执行时间不随着问题规模n的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是O(1)。

2.1. 交换i和j的内容

sum=0； （一次）

for(i=1;i<=n;i++) （n次 ）

for(j=1;j<=n;j++) （n^2次 ）

sum++； （n^2次 ）

解：T(n)=2n^2+n+1 =O(n^2)

2.2.

for (i=1;i<n;i++)

{

y=y+1; ①

for (j=0;j<=(2\*n);j++)

x++; ②

}

解：语句1的频度是n-1，语句2的频度是(n-1)\*(2n+1)=2n^2-n-1，f(n)=2n^2-n-1+(n-1)=2n^2-2，该程序的时间复杂度T(n)=O(n^2)

2.3.

a=0;

b=1;

解：时间复杂度 O(1)

## 空间复杂度

# 分类

可以按照很多标准来给算法分类，例如：

1. 按实现方式划分

**递归算法与迭代算法**

**过程式算法与声明式（非过程式）算法**

**串行算法、并行算法、分布式算法**

**确定性算法与非确定性算法**

**精确算法与近似算法**

1. 按设计方式划分

**贪婪算法**

**分治算法**

**动态规划算法**

**线性规划算法**

**规约（转换并治理）算法**

1. 按其他方式划分

**按研究领域划分**

**按复杂度划分**

**随机化的算法**

**分支定界与回溯**

# 迭代法

迭代法是一种不断用旧值递推新值的过程，分精确迭代和近视迭代。是用来求方程和方程组近似根的方法。

迭代变量

迭代关系， 迭代关系选择不合理，会导致迭代失败

迭代过程控制，也就是迭代什么时候结束，不能无休止进行下去

# 排列

## 实现

## 应用

### 递归方法求解序列的全排列

题目要求：使用递归方法求解一个字符串序列的全排列。

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

全排列问题

以字符串abc为例

首先固定a，求后面两个字符bc的全排列，结束之后。

把第一个字符a和后面的b交换，得到bac，借助固定第一个字符b，求后面两个字符ac的排列，然后把c放到第一位置....

\*/

int count =0;//记录排列个数

void Permutation(char\* str,int begin)

{

if(str[begin] == '\0')

{

cout<<str<<endl;

count++;

}

for(int i= begin;str[i] != '\0';i++)

{

swap(str[i],str[begin]);//交换当前位置和第一个位置

Permutation(str,begin+1);//求除第一位置之外的字符串的排列

swap(str[i],str[begin]);//回归原始状态

}

}

/\*

去除重复的全排列的实现

如果一个数和后面的数相同，那么这两个数就不交换

也就是说，去除重复的全排列就是从第一个元素起每个元素分别与他后面非

重复出现的元素交换

\*/

bool isValid(vector<int>& vec,int begin,int index)

{

int i;

for(i = begin;i<index;i++)

{

if(vec[i] == vec[index])

return false;

}

return true;

}

void Permutation(vector<int>& vec,int begin,int end)

{

int i=0;

if(begin >= end)

{

for(i=0;i<vec.size();i++)

cout<<vec[i]<<" ";

cout<<endl;

return ;

}

for(i = begin;i<= end;i++)

{

if(isValid(vec,begin,i))

{

swap(vec[begin],vec[i]);

Permutation(vec,begin+1,end);

swap(vec[begin],vec[i]);

}

}

}

int main()

{

vector<int> vec(3);

int i;

for(i=0;i<vec.size();i++)

vec[i] = i+1;

vec[1]=1;

int pos =0;

Permutation(vec,pos,2);

char str[]="abcd";

// Permutation(str,0);

// cout<<count<<endl;

return 0;

}

### 非递归方法求解序列的全排列

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

非递归的方法实现全排列

思路：使用STL中的思想，从后往前找一对相邻的数，在一对相邻的数中

第一个元素小于第二个元素，记做\*i < \*ii,从后往前找到第一个元素\*j，使得\*j > \*i,然后交换\*i和\*j,然后交换从\*ii开始（包括\*ii在内）到数组末尾的序列

\*/

void Rserve(vector<int>& vec,int begin,int end)

{

while(begin <= end)

{

swap(vec[begin],vec[end]);

begin++;

end--;

}

}

bool FindPair(vector<int>& vec,int& first,int& second)

{

int last = vec.size()-1;

for(;last>0;last--)

{

second = last;

first = last-1;

if(vec[first] < vec[second])

return true;

}

return false;

}

bool next\_premutation(vector<int>& vec)

{

int first,second,index;

if(FindPair(vec,first,second))

{

index = vec.size()-1;

for(;index>=first;index--)

{

if(vec[index] > vec[first])

break;

}

swap(vec[index],vec[first]);

Rserve(vec,second,vec.size()-1);

return true;

}

return false;

}

int main()

{

vector<int> vec(3);

int i;

for(i=0;i<vec.size();i++)

vec[i] = i+1;

vec[1] =1;

do{

int i;

for(i=0;i<vec.size();i++)

cout<<vec[i]<<" ";

cout<<endl;

}while(next\_premutation(vec));

return 0;

}

### 八皇后问题

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

题目要求：

在8\*8的国际象棋上摆放着八个皇后，使其不能互相攻击

也就是说，任意两个皇后不得处在同一行、同一列或者同一对角线上请问有多少种方法

\*/

/\*

八皇后问题

\*/

int g\_number=0;

void Print(int ColumnIndex[] , int length)

{

cout<<g\_number<<endl;

for(int i = 0 ; i < length; ++i)

cout<<ColumnIndex[i]<<" ";

cout<<endl;

}

bool Check(int ColumnIndex[] , int length)

{

int i,j;

for(i = 0 ; i < length; ++i)

{

for(j = i + 1 ; j < length; ++j)

{

if( i - j == ColumnIndex[i] - ColumnIndex[j] || j - i == ColumnIndex[i] - ColumnIndex[j]) //在正、副对角线上

return false;

}

}

return true;

}

void Permutation(int ColumnIndex[] , int length , int index)

{

if(index == length)

{

if( Check(ColumnIndex , length) )

//检测棋盘当前的状态是否合法

{

++g\_number;

Print(ColumnIndex , length);

}

}

else

{

for(int i = index ; i < length; ++i) //全排列

{

swap(ColumnIndex[index] , ColumnIndex[i]);

Permutation(ColumnIndex , length , index + 1);

swap(ColumnIndex[index] , ColumnIndex[i]);

}

}

}

void EightQueen( )

{

const int queens = 8;

int ColumnIndex[queens];

for(int i = 0 ; i < queens ; ++i)

ColumnIndex[i] = i; //初始化

Permutation(ColumnIndex , queens , 0);

}

int main()

{

EightQueen();

return 0;

}

# 组合

## 实现

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

/\*

组合的另一种实现

\*/

void helper(vector<int>& vec,int begin,int& num,vector<int>& subset)

{

if(begin >= vec.size() || num<0 )

return ;

subset.push\_back(vec[begin]);

num--;

if(num == 0)

{

int i;

for(i=0;i<subset.size();i++)

cout<<subset[i]<<" ";

cout<<endl;

}

helper(vec,begin+1,num,subset);

subset.pop\_back();

num++;

helper(vec,begin+1,num,subset);

}

void Combination(vector<int>& vec,int k)

{

if(vec.size()==0 || k <0)

return ;

vector<int> subset;

sort(vec.begin(),vec.end());

helper(vec,0,k,subset);

}

void Subsets(vector<int>& vec)

{

int i;

for(i=0;i<=vec.size();i++)

Combination(vec,i);

}

/\*

另一种方法

\*/

void generate(vector<int> res, vector<int> &S, int i)

{

if(i == S.size())

{

for(int j=0;j<res.size();j++)

cout<<res[j]<<" ";

cout<<endl;

//return;

}

else

{

generate(res, S, i+1);

res.push\_back(S[i]);

generate(res, S, i+1);

}

}

void SubsetSecond(vector<int>& vec)

{

if(vec.size()<=0)

return;

sort(vec.begin(),vec.end());

vector<int> result;

generate(result,vec,0);

}

int main()

{

int array[]={2,1,3};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

SubsetSecond(vec);

return 0;

}

## 应用

### 求一个字符串的所有组合

/\*

输入一个字符串，输出该字符串中字符的所有组合

\*/

#include <iostream>

#include <vector>

using namespace std;

/\*

思路：在长度为n的字符串中求m个字符的组合

从头扫描字符串得第一个字符，针对第一个字符，有两种选择

把这个字符放到组合中去，接下来我们需要在剩下的n-1个字符中选取m-1个字符

不把这个字符放到组合中去，需要在剩下的n-1个字符中选取m个字符

\*/

void Combination(char\* string, int number, vector<char>& result)

{

if(number == 0)

{

vector<char>::iterator iter = result.begin();

for(; iter < result.end(); ++ iter)

cout<<(\*iter);

cout<<endl;

return;

}

if(\*string == '\0')

return;

result.push\_back(\*string);

Combination(string + 1, number - 1, result);

result.pop\_back();

Combination(string + 1, number, result);

}

void Combination(char\* string)

{

if(string == NULL)

return;

int length = strlen(string);

vector<char> result;

for(int i = 1; i <= length; ++ i)

{

Combination(string, i, result);

}

}

int main()

{

char str[] ="abc";

Combination(str);

return 0;

}

### 整数数组之和

题目要求：在一个整数数组中，找到合适的四个数，使其和为0。

代码：

#include <iostream>

#include <vector>

#include <algorithm>

#include <numeric>

using namespace std;

/\*

在一个数组中，找到合适的四个数，使其和为0

\*/

void helper(vector<int>& vec,int index,int num,vector<int>& target)

{

if(index >= vec.size())

return ;

target.push\_back(vec[index]);

num--;

if(num ==0)

{

int i=0;

int sum=0;

sum =accumulate(target.begin(),target.end(),sum);

if(sum == 0)

{

for(i=0;i<target.size();i++)

cout<<target[i]<<" ";

cout<<endl;

}

}

helper(vec,index+1,num,target);

target.pop\_back();

num++;

helper(vec,index+1,num,target);

}

void FourSum(vector<int>& vec)

{

if(vec.size()==0)

return ;

vector<int> target;

helper(vec,0,4,target);

}

int main()

{

int array[]={1,0,-1,0,-2,2};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

FourSum(vec);

return 0;

}

# 递归

递归是一种设计和描述算法的有力工具。递归算法执行过程分递推和回归两个阶段

在递推阶段，将大的问题分解成小的问题在回归阶段，获得最简单问题的解后，逐级返回，依次得到稍微复杂情况的解，知道获得最终的结果

1）确定递归公式

2）确定边界条件

## 斐波那契数列

fib(n)=fib(n-1)+fib(n-2)

递归实现

非递归实现

## 其它案例

阶乘计算

梵塔问题 （三根针1，2，3表示，1号从小到大n个盘子，先要都移到3号上，不能出现大盘压小盘，找出移动次数最少的方案）

快速排序

递归运行效率较低，因为有函数调用的开销，递归多次也可能造成栈溢出。

# 穷举搜索法

或者叫蛮力法。对可能的解的众多候选按照某种顺序逐一枚举和检验。典型的问题如选择排序和冒泡排序。

## 背包问题

给定n个重量为 w1,w2,...,wn,定价为 v1,v2,...,vn 的物品，和一个沉重为W的背包，求这些物品中一个最有价值的子集，且能装入包中。

## 其它案例

选择排序

冒泡排序

# 动态规划

# 贪心算法

# 回溯法

# 分治算法

# 总结

贪心法、分治法、动态规划都是将问题归纳为根小的、相似的子问题，通过求解子问题产生全局最优解。

贪心法

分治法

动态规划