# 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

动态规划（Dynamic Programming，DP）技术通常与记忆化（memoization）技术结合起来使用。

## 思想

动态规划（DP）的基本思想和策略：将待求解的问题分解为若干个子问题，按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。

复杂问题不能分解成几个子问题，而分解成一系列子问题；动态规划（DP）通常基于一个递推公式及一个(或多个)初始状态，当前子问题解由上一次子问题解推出。

适合于用动态规划求解的问题，经分解后得到的子问题往往**不是相互独立的**。

状态 状态转移方程 递推关系

动态规划算法的关键在于解决冗余，以空间换时间的技术，需要存储过程中的各种状态。可以看着是分治算法+解决冗余使用动态规划算法的问题的特征是子问题的重叠性，否则动态规划算法不具备优势。

**动态规划VS分治**

动态规划与分治的区别在于，分治法所要处理的那些子问题之间并没有依赖关系，而动态规划所要处理的子问题却是有所重叠的，因此，可以把已经解决的子问题保存到表格里，这就是记忆化技术。运用这种技术，算法可以把很多问题的复杂度由指数级别降至O(n2)、O(n3)这样的多项式级别。

**动态规划VS递归计数**

动态规划技术与（分治算法中的）递归计数相比，其区别在于，它会把已经解决的子问题放在表格中，以免去重复的计算；而分治算法所要递归解决的那些子问题，彼此之间不重复。由此可见，并非所有的问题都适合用动态规划技术来解决。

## 条件

动态规划不能解决所有问题，需要满足下面的条件：

1. 具备最优的子结构：整个问题的最佳解法可以由各个子问题的最佳解法所构成；
2. 具备相互重叠的子问题：在运用递归来解决问题的过程中，有几个问题会反复出现。

另外一种描述，能采用动态规划求解的问题一般具有三个性质：

1. 最优化原理：问题最优解包含的子问题的解也是最优解，称为最优子结构
2. 无后效性：某个阶段的状态一旦确定，就不再受该状态以后决策的影响，只与当前状态相关
3. 有重叠子问题：子问题之间不相互独立，一个子问题可能在后续的决策中多次被使用

## 基本步骤

1、划分问题/划分阶段

2、确定状态和状态变量

3、确定决策并写出状态转移方程（关键）

4、写出规划方程/寻找边界条件（找到递推式）

## 使用场合

### 字符串解码

题目要求：一个包含字母的消息被加密之后变成了一个只包含数字的字符串，但是我们现在知道加密的规则：

‘A’🡪1

‘B’🡪2

……

‘Z’🡪26

现在给定一个已经被加密的只包含数字的字符串，求出该字符串有多少种被解密的方法。例如“12”🡪AB或者12🡪L。

代码：

#include <iostream>

#include <string>

#include <vector>

using namespace std;

/\*

思路：

这是一个典型的DP问题，假设定义一个数组，dp[i]为到第i个字符所能够组成的所有编码方式的个数。那么对于dp[i+1]来说，肯定至少和dp[i]一样多，如果第i个字符和i+1个字符可合成一个字符，那么dp[i+1] += dp[i-1].

\*/

int Decode\_num(string& str)

{

vector<int> vec(str.size(),1);

if(str.size() <2)

return 1;

if(str[0]=='1'||(str[0]=='2'&& str[1]<='6'))

vec[1] =2;

int i;

int tmp;

for(i=2;i<str.size();i++)

{

if(str[i] >= '0' && str[i] <= '9')//判断是合法的字符

vec[i] = vec[i-1];

else

return 0;

tmp = str[i-1] -'0';

tmp = tmp\*10 + str[i]-'0';

if(str[i-1]!='0' && tmp <=26)

vec[i] += vec[i-2];

else

vec[i] = vec[i-1];

}

return vec[str.size()-1];

}

int main()

{

string str("1231725");

cout<<Decode\_num(str)<<endl;

return 0;

}

### 寻找最长的共同子序

### 路径数目&最小路径和

题目：求矩阵中从左上角到右下角的路径数目

求矩阵中左上角到右下角最小路径和

代码：

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

/\*

思路：对于某一点dp[i][j]的路径数目，是该点正上方和正左方路径数目之和

dp[i][j] = dp[i][j-1] + dp[i-1][j]; 但是对于特殊地方需要特殊考虑

\*/

int Unique\_path(int m,int n,int first,int second)

{

vector<vector<int> > dp(m);

int i,j;

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

dp[i].assign(n,0);

dp[0][0] =1;

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

{

for(j=0;j<dp[0].size();j++)

{

if(i!=0 || j!=0)

{

if(i == first && j == second)

dp[i][j] =0;

else

{

if(i == 0)

dp[i][j] = dp[i][j-1];

else if(j== 0)

dp[i][j] = dp[i-1][j];

else

dp[i][j] = dp[i][j-1]+dp[i-1][j];

}

}

}

}

return dp[m-1][n-1];

}

/\*

第二个问题，从左上角到右下角，寻找代价最小的路径

典型的动态规划问题，和上个问题类似

\*/

int MinPathSum(vector<vector<int> >& vec)

{

vector<vector<int> > dp(vec.size());

int i,j;

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

dp[i].assign(vec[i].size(),numeric\_limits<int>::max());

dp[0][0] = vec[0][0];

for(i=1;i<vec.size();i++)

dp[i][0] = vec[i][0]+dp[i-1][0];

for(j=1;j<vec[0].size();j++)

dp[0][j] = vec[0][j] + dp[0][j-1];

int tmp;

for(i=1;i<vec.size();i++)

{

for(j=1;j<vec[0].size();j++)

{

tmp = min(vec[i][j]+dp[i][j-1],vec[i][j]+dp[i-1][j]);

dp[i][j] = min(dp[i][j],tmp);

}

}

return dp[vec.size()-1][vec[0].size()-1];

}

int main()

{

// cout << Unique\_path(3,7,2,3)<<endl;

vector<vector<int> > vec(3);

int i,j;

int array[]={2,4,3,7};

int array1[]={5,3,2,1};

int array2[]={4,8,6,2};

vec[0].assign(array,array+4);

vec[1].assign(array1,array1+4);

vec[2].assign(array2,array2+4);

cout<<MinPathSum(vec)<<endl;

return 0;

}

### 最大子数组乘积

题目：给定一个整数数组，求乘积最大的子数组的值。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

最大子串乘积，由于可能出现负数。也是DP问题，也是局部最优和全局最优问题。

这里需要记录最小值，假设有两个数组，分别记录包括当前元素在内的子串所能构成的最大和最小值，然后根据这个再更新全局最大，至于当前最大，可能是之前最大乘以当前元素，也可能是前一个元素最小乘以当前元素，也可能是当前元素

\*/

int maxProduct(vector<int>& vec)

{

if(vec.size()==0)

return 0;

vector<int> maxcur(vec.size(),0);

vector<int> mincur(vec.size(),0);

maxcur[0]=vec[0];

mincur[0]=vec[0];

int maxproduct = vec[0];

int i,temp;

for(i=1;i<vec.size();i++)

{

maxcur[i] = max(vec[i],max(maxcur[i-1]\*vec[i],mincur[i-1]\*vec[i]));

mincur[i] = min(vec[i],min(mincur[i-1]\*vec[i],maxcur[i-1]\*vec[i]));

maxproduct = max(maxcur[i],maxproduct);

}

return maxproduct;

}

int main()

{

int array[] ={2,3,-2,4};

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

cout<<maxProduct(vec)<<endl;

return 0;

}

### 链式矩阵乘法

### 子集和问题

### 0/1背包问题

### 旅行推销员问题

### 最长递增子序列

最长递增子序列（LIS Longest Increasing Subsequence）

### 编辑距离

题目要求：给定两个字符串word1和word2，求出最少需要多少个步骤可以将word1转化为word2，其中每一个操作都被记为一步，每个操作可以是：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

代码：

#include <iostream>

#include <vector>

#include <string>

#include <limits>

using namespace std;

/\*

典型的DP问题，这里需要使用S来匹配T， 利用动态规划思想

使用dp[i][j]表示S与T的前i个字符与前j个字符的匹配子串个数。

1）初始条件：T为空串时，S为任意字符串都能匹配一次，所以dp[i][0]=1

S为空字符串，T不为空时，不能匹配，dp[0][j] =0

2)S[i] == T[j]，dp[i][j] = dp[i-1][j-1]+dp[i-1][j].表示当前字符可以保留也可以抛弃

3）S[i] != T[j]，dp[i][j] = dp[i-1][j-1]

\*/

int Distinct\_sub(string& src,string& dst)

{

vector<vector<int> > dp(src.size());

int i,j;

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

dp[i].assign(dst.size(),0);

if(src[0] == dst[0])

dp[0][0] =1;

for(j=1;j<dp.size();j++)

dp[j][0] = src[j]==dst[0]? dp[j-1][0]+1:dp[j-1][0];

for(i=1;i<dp.size();i++)

{

for(j=1;j<dp[0].size();j++)

{

if(src[i] == dst[j])

dp[i][j] = dp[i-1][j-1]+dp[i-1][j];

else

dp[i][j] = dp[i-1][j];

}

}

return dp[src.size()-1][dst.size()-1];

}

/\*

编辑距离问题

使用便利dp[i][j]记录包括word1[i]在内的字符串和word2[j]在内的字符串的编辑距离，如果word1[i+1] == word2[j+1] dp[i+1][j+1] = dp[i][j]，否则dp[i+1][j+1] = dp[i][j]+1,不过也有可能dp[i+1][j+1] = dp[i+1][j]+1或者dp[i][j+1]+1 取三者最小

\*/

int Edit\_distance(string& s1,string& s2)

{

vector<vector<int> > dp(s1.size());

int i,j;

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

dp[i].assign(s2.size(),numeric\_limits<int>::max());

if(s1[0]==s2[0])

dp[0][0] =0;

else

dp[0][0] =1;

for(i=1;i<dp[0].size();i++)

if(s1[0] == s2[i])

dp[0][i] = i;

else

dp[0][i] = dp[0][i-1]+1;

for(i=1;i<dp.size();i++)

if(s1[i] == s2[0])

dp[i][0] = i;

else

dp[i][0] = dp[i-1][0]+1;

for(i=1;i<dp.size();i++)

{

for(j=1;j<dp[0].size();j++)

{

if(s1[i] == s2[j])

dp[i][j] = dp[i-1][j-1];

else

dp[i][j] = min(dp[i-1][j-1]+1,min(dp[i][j-1]+1,dp[i-1][j]+1));

}

}

return dp[s1.size()-1][s2.size()-1];

}

int main()

{

string src("rabbbit");

string dst("rabbit");

cout<<Distinct\_sub(src,dst)<<endl;

string s1("abcd");

string s2("abc");

cout<<Edit\_distance(s1,s2)<<endl;

return 0;

}

### 盛水最大化

题目要求：给出一系列非负整数a1，a2，…，an，每一个数都代表数轴上的一个点(i,ai)，那么这n个垂直线中的任意两个都可以组成一个区间，然后和x轴可以构成一个容器，求出可以盛水最多的容器的两条边。

代码：

#include <iostream>

#include <vector>

#include <string>

#include <stack>

using namespace std;

/\*

夹逼方法

从数组的两段走起，每次迭代时判断左边点和右边点指向的数字哪个大如果左边点下，就意味着左移动右边点不可能使得结果变得更好因为瓶颈在左边点，移动右边点只会变小，所以这个时候我们选择左边点右移反之，选择右边点左移，在这个过程中一直维护最大的容积

\*/

int area(vector<int> &height, int i, int j)

{

int h = height[i]<height[j]?height[i]:height[j];

return h\*(j-i);

}

int maxArea(vector<int> &height)

{

int max=0;

for(int i=0;i<height.size();i++)

{

for(int j=i+1;j<height.size();j++)

{

int a = area(height,i,j);

if(a>max)

max=a;

}

}

return max;

}

/\*

第二种方法

\*/

int maxarea(vector<int>& vec)

{

int maxarea=0;

int first,second;

int i=0,j=vec.size()-1;

while( i<j)

{

if(min(vec[i],vec[j])\*(j-i) > maxarea)

{

maxarea = min(vec[i],vec[j])\*(j-i);

}

if(vec[i] < vec[j])

i++;

else

j--;

}

return maxarea;

}

/\*

|

| \_\_

| | |

| \_\_ | |\_\_ \_\_

| | | | | | | |

| \_\_ | |\_\_ \_\_ | | |\_\_| |\_\_

| | | | | | | | | | | | | |

|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_\_\_\_

1 2 3 4 5 6 7 8 8 9 10 11 12

上述给定的一个序列为[0,1,0,2,1,0,1,3,2,1,2,1]，每个元素代表柱子的高度

最后函数的返回值为6

思路：可以在这个序列中找到最高的柱子位置，那么从两头开始找可以

盛水的多少，假如从头开始遍历，需要遍历到柱子最高的位置，遍历到当前位置如果发现当前的柱子比之前记录的柱子高，那么更新如果没有之前记录的柱子高，那么就可以计算当前柱子相对之前的高柱子的盛水量

\*/

int TrapRainWater(vector<int>& vec)

{

int i,maxhigh;

maxhigh = 0;

int left=0,right = 0;

int sum =0;

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

if(vec[i] > vec[maxhigh])

maxhigh = i;

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

{

if(vec[i] < left)

sum +=(left-vec[i]);

else

left = vec[i];

}

for(i=vec.size()-1;i>maxhigh;i--)

{

if(vec[i]<right)

sum += (right-vec[i]);

else

right = vec[i];

}

return sum;

}

int main()

{

// int array[]={4,3,4,5,7,9,7,6,8,5,3,2};

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

// cout<<maxArea(vec)<<endl;

int array[]={0,1,0,2,1,0,1,3,2,1,2,1};

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

cout<<TrapRainWater(vec)<<endl;

return 0;

}

### 股票利润最大化

题目要求：给定一个整数数组，数组中的每个元素都是某支股票的当天的价钱，设计一个算法来找出这支股票的最大利润。你至少可以进行K次交易。

代码：

/\*

股票买卖最大利润

这里维护两个变量，一个是当前到达第i天可以最多进行j此交易，最好的利润是多少（global[i][j]）另一个是当前到达第i天，最多可以进行j此交易，并且最后一次交易在dangt卖出那么最好的利润是多少（local[i][j]）

递推公式

global[i][j] = ma(local[i][j],glboal[i-1][j]),

也就是取当前局部最好和过往全局最好的其中之一对于局部最好

local[i][j] = max(global[i-1][j-1]+maxdiff(diff,0),local[i-1][j]+diff);

\*/

/\*

在进行两次交易的利润最大化

\*/

int maxProfit(vector<int>& prices)

{

if(prices.size() <= 0)

return 0;

int global[3];

int local[3];

for(int i=0;i<prices.size()-1;i++)

{

int diff = prices[i+1]-prices[i];

for(int j=2;j>=1;j--)

{

local[j] = max(global[j-1]+(diff>0?diff:0),local[j]+diff);

global[j] = max(local[j],global[j]);

}

}

global[2];

}

/\*

多次交易之后

\*/

int helper(vector<int>& prices,int k)

{

int len = prices.size();

if(len == 0)

return 0;

int local[10][10];

int global[10][10];//临时申请的空间

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

{

int diff = prices[i]-prices[i-1];

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

{

local[i][j] = max(global[i-1][j-1]+max(diff,0),local[i-1][j]+diff);

glocal[i][j] = max(local[i][j],global[i-1][j]);

}

}

}

int maxProfit(vector<int>& prices)

{

return helper(prices,2);

}

# 贪心算法

贪心算法（greedy algorithm，贪婪算法）是分阶段执行的，每一个阶段都根据当前的情况来判断，而不考虑后续的发展。

一般来说，这种算法选出的解是局部最佳（local best）解。该算法预设了这样一个前提，就是认为全局最优解可以由局部最优解所推出。

即，贪心算法不追求最优解，只找到满意解。

## 条件

有待处理的问题必须满足下列两项条件，才能用贪婪算法求出最优的解：

1. 具备贪心选择性质（greedy choice property）
2. 具备最优子结构（optimal substructure）

贪心选择性质：该性质意味着全局最优解可以由局部最优解（也就是在贪心策略下所选出的解）所推出。贪婪算法在针对当前这一步做决定时，可以参考前面几步的决定，但是不会依赖后续的步骤。它总是会选出局部最优的解，并将原问题约简为更小的问题，然后在更小的问题上继续寻找其局部最优解，并继续约简。

最优子结构：如果某个问题的最优解可以由其各个子问题的最优解所构成，那么该问题就具备最优子结构。这意味着把子问题的解法拼合起来可以解决最初所要求解的那个大问题。

## 特点

优点：直观、易懂，实现简单。算法一旦做出决定，就不用回过头来去重新检查前面计算过的那些值。

缺点：并非所有问题都能那么解决，对于很多问题，在某个小范围内所做的最优决策，未必是整个问题的最优决策。

## 适用场合

排序：选择排序、拓扑排序

优先级队列：堆排序

霍夫曼编码压缩算法

Prim算法与Kruskal算法

加权图中的最短路径算法（Dijkstra算法）

用零钱换整钱的问题

分数背包（fractional knapsack）问题

按照大小及权重（或者说级别）来合并不相交的集合（disjoint set）

找零钱问题

作业调度算法

把贪婪算法当成一种近似算法，来解决某些复杂的问题

## 赫夫曼编码

## 其它案例

### 找回零钱问题

### 装箱问题

### 最大连续子数组和

题目要求：在一个整数数组中，求和最大的子数组的值。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

最大连续子数组和

\*/

int MaxSubarray(vector<int>& vec)

{

int sum = vec[0];

int curmax= vec[0];

int i;

for(i=1;i<vec.size();i++)

{

curmax += vec[i];

if(curmax < 0)

curmax = 0;

if(curmax > sum)

sum = curmax;

}

if(sum<0)

{

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

if(sum < vec[i])

sum = vec[i];

}

return sum;

}

int main()

{

int array[]={-1,1,-3,4,-1,2,1,-5,4};

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

cout<<MaxSubarray(vec)<<endl;

return 0;

}

### 分糖果

题目要求：现在为已经站成一排的小朋友分糖果，保证每个小朋友至少有一个糖果，同时保证各自比相邻小朋友高的所分的糖果要比他的邻居多，按照这样的分食方法，最少需要多少糖果？

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

进行两次扫描，一次从左向右，一次从右向左

第一次扫描的时候维护对于每一个小孩左边所需要最少的糖果数量

存入数组对应元素中，第二次扫描的时候维护右边所需的最少糖果数量

并且比较将左边和右边大的糖果数量存入结果数组对应元素中

\*/

int candy(vector<int> &ratings)

{

vector<int> candy(ratings.size(),1);

int sum,i;

for(i=1;i<ratings.size();i++)

{

if(ratings[i] > ratings[i-1])

candy[i] = candy[i-1]+1;

}

sum = candy[ratings.size()-1];

for(i=ratings.size()-2;i>=0;i--)

{

int cur =1;

if(ratings[i] > ratings[i+1])

cur = candy[i+1]+1;

sum += max(cur,candy[i]);

candy[i] = cur;

}

return sum;

}

/\*

更清晰的思路

\*/

int candy2(vector<int> &ratings)

{

vector<int> candy(ratings.size(),1);

int sum,i;

for(i=1;i<ratings.size();i++)

{

if(ratings[i] > ratings[i-1])

candy[i] = candy[i-1]+1;

}

sum = candy[ratings.size()-1];

for(i=ratings.size()-2;i>=0;i--)

{

int cur =1;

if(ratings[i] > ratings[i+1] && candy[i] <= candy[i+1])

candy[i] = candy[i+1]+1;

sum += candy[i];

}

return sum;

}

int main()

{

int array[]={4,2,6,8,5};

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

cout<<candy(vec)<<endl;

return 0;

}

### 跳远游戏

题目要求：给定一个整数数组，数组中的元素代表在当前位置能够向当前跳的最远距离，判断给定的这个跳远策略能否跳到最后的位置。

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

贪心思想，时刻计算当前位置和当前位置能跳的最远长度，并始终和界限比较

若在任意位置出现最大跳步为0，那么就无法继续跳下去

在任意位置出现最大跳步+当前位置 >界限，那么说明可以跳出去

\*/

bool canJump(vector<int>& vec)

{

if(vec.size() <=0)

return true;

int maxstep = vec[0];

for(int i=1;i<vec.size();i++)

{

if(maxstep == 0)

return false;

maxstep--;

if(maxstep < vec[i])

maxstep = vec[i];

if(maxstep+i >= vec.size()-1)

return true;

}

}

int main()

{

int array[]={2,3,1,1,4};

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

cout<<canJump(vec)<<endl;

return 0;

}

# 回溯法

也叫试探法。是一种选优搜索法，按照选优条件搜索，当搜索到某一步，发现原先选择并不优或达不到目标，就退回重新选择。

## 一般步骤

1、针对问题，定义解空间（ 这时候解空间是一个集合，且包含我们要找的最优解）

2、组织解空间，确定易于搜索的解空间结构，通常组织成树结构 或 图结构

3、深度优先搜索解空间，搜索过程中用剪枝函数避免无效搜索

回溯法求解问题时，一般是一边建树，一边遍历该树；且采用非递归方法。

## 八皇后问题

8x8的国际象棋棋盘上放置8个皇后，使得任何一个皇后都无法直接吃掉其他的皇后。任意2个皇后都不能处于同一个 横线，纵线，斜线上。

分析

任意2个皇后不能同一行，也就是每个皇后占据一行，通用的，每个皇后也要占据一列

一个斜线上也只有一个皇后

## 其它案例

### 迷宫问题

### 求一个序列中和为特定值的组合

题目要求：求一个序列元素之和等于固定值的特定组合。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

求一个序列中元素之和等于固定值的特定组合

\*/

void Combination\_helper(vector<int>& vec,int begin,int target,int& cur,vector<int>& path)

{

if(begin >=vec.size())

return ;

cur += vec[begin];

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

if(cur == target)

{

for(int i=0;i<path.size();i++)

cout<<path[i]<<endl;

cout<<"=========="<<endl;

}

Combination\_helper(vec,begin+1,target,cur,path);

path.pop\_back();

cur -= vec[begin];

int j;

for(j=begin+1;j<vec.size();)

{

if(vec[j] == vec[begin])

j++;

else

break;

}

Combination\_helper(vec,j,target,cur,path);

}

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

{

vector<int> path;

int cur = 0;

if(vec.size() == 0)

return ;

Combination\_helper(vec,0,target,cur,path);

}

int main()

{

vector<int> path;

int array[]={1,1,2,5,6,7,10};

int cur =0;

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

Combination(vec,8);

return 0;

}

### 求括号正确的组合方式

题目要求：

代码：

#include <iostream>

#include <string>

#include <vector>

using namespace std;

/\*

对于几对括号，有多少种正确的组合方式

思路：使用递归的方法，只不过在递归的时候

时刻需要保证左边的括号比右边的括号多

只有在左括号比右括号多的情况下才有可能保证整个序列为合法的括号匹配

\*/

void helper(vector<char>& str,int l,int r)

{

if(l == 0 && r == 0)

{

for(int i=0;i<str.size();i++)

{

cout<<str[i];

}

cout<<endl;

}

if(l>0)

{

str.push\_back('(');

helper(str,l-1,r);

str.pop\_back();

}

if(r>0 && l<r)

{

str.push\_back(')');

helper(str,l,r-1);

str.pop\_back();

}

}

void GenerateParenthess(int n)

{

if(n<=0)

return ;

vector<char> tmp;

helper(tmp,n,n);

}

int main()

{

GenerateParenthess(3);

return 0;

}

### 求九宫格中字符和数字的组合

题目要求：在九宫格手机键盘上，每个数字都对应着几个字符给出一个数字的字符串，找出所有对应的字符串的组合，而且每个数字只能有一个字符进行对应。

代码：

#include <iostream>

#include <string>

#include <vector>

using namespace std;

/\*

题目：在九宫格手机键盘上，每个数字都对应着几个字符

给出一个数字的字符串，找出所有对应的字符串的组合，每个

数字找一个字符进行对应

\*/

void helper(string& str,int begin,vector<string>& hash,vector<char>& vec)

{

int i;

if(begin > str.length())

return;

if(begin == str.length())

{

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

cout<<vec[i];

cout<<endl;

return;

}

for(i=0;i<hash[str[begin]-'0'].length();i++)

{

vec.push\_back(hash[str[begin]-'0'][i]);

helper(str,begin+1,hash,vec);

vec.pop\_back();

}

}

void Combination(string& str,vector<string>& hash)

{

if(str.length()==0)

return ;

vector<char> vec;

helper(str,0,hash,vec);

}

void LetterCom(string& str)

{

int i;

vector<string> hash(10); //这里假设有10个数字

hash[0]=" ";

hash[1]="-";

hash[2]="abc";

hash[3]="def";

hash[4]="ghi";

hash[5]="jkl";

hash[6]="mno";

hash[7]="pqrs";

hash[8]="tuv";

hash[9]="wxyz";

Combination(str,hash);

return ;

}

int main()

{

string str("23");

LetterCom(str);

return 0;

}

# 分治算法

分治（divide and conquer，D&C）算法将一个难以直接解决的大问题，分割成一些规模较小的相同问题，各个击破，分而治之。

分治算法常用递归实现：

1）问题缩小的小规模可以很容易解决

2) 问题可以分解为规模较小相同问题

3）子问题的解可以合并为该问题的解

4）各个子问题相互独立，(如果这条不满足,转为动态规划求解）

## 条件

分治技术不能解决所有的问题

## 特点

优点：

解决困难的问题

实现并行计算

有效利用缓存

缺点：

递归调用的速度比较慢

比迭代法更复杂

## 步骤

分治法的步骤：

1、划分（divide）：把问题划分成多个子问题，这些子问题和原问题同属一类，但规模较小

2、递归（recursion）：递归地解决这些子问题

3、治理（conquer）：将子问题的答案适当地合并起来

## 适用场合

二分搜索

归并排序

快速排序

找中位数

查找最小值与最大值

做矩阵乘法

查找各点中距离最近的两个点（closest pair problem，最接近的点对问题）

## 大整数乘法

如 26542123532213598\*345987342245553677884

## 其它案例

快速排序

归并排序

最大子数组和

二叉搜索算法

# 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