# 递归

## 概述

凡是会调用自身的函数，都称为递归（recursive）函数。

所谓递归法，就是把原问题简化，并套用解决原问题的思路，来解决简化之后的问题。这个简化并套用的步骤，称为递归步骤（recursive step）；执行完这一步以后，有可能还会发生更深层次的递归调用（recursive call）。

递归法的要点在于**必须确保整个递归过程能够终止**。也就是说，每次调用所要解决的问题，都应该比上一次调用时所要解决的问题简单一些，到了最后，可以把所有待解决的问题，化成最为基本的情况，从而无需继续往下递归。

## 使用场景

用递归法写成的代码一般要比用迭代法写出来的更加短小、更加易读。

循环逻辑在编译或解释的时候，通常也会化为递归函数。

如果待解决的问题可以划分成多个**与原问题相似**的小问题，那么这种任务特别适合用递归来做。例如，搜索、排序以及遍历等任务用递归做起来比较简单。

**每次递归都要在内存里把原方法复制一份（实际上复制的只是变量而已）**。当这个小的递归方法执行完毕，也就是返回了某种数据之后，系统会将其从内存中删除。递归函数看上去简单，但是回溯起来还是需要费些功夫的（因为需要不断复制变量，所以占用内存多有可能导致栈溢出）。

## 递归与分治

问题的分解肯定不是一步到位的，需要反复使用分治手段，在多个层次上层层分解，这种分解的方法很自然地导致了递归方式的使用。

从算法实现的角度看，分治法得到的子问题和原问题是相同的，当然可以使用相同的函数来解决，区别在于子问题的规模和范围不同。通过特定的函数参数安排，使得同一个函数可以解决不同规模的相同问题，这就是递归方法的基础。

## 递归与迭代

递归式的方法可能更接近于解题时所用的思路，而且对于那些无法直接看出答案的问题来说，这种办法比迭代法简单。不过它的缺点在于，每次递归调用都需要增加程序的开销，因此必须开辟栈帧（stack frame）来保存相关的数据。

**递归：**

1. 遇到基本条件时会终止；
2. 每次递归调用都要开辟栈帧并耗费内存；
3. 如果无休止地递归下去，那么程序可能会耗尽内存，并发生**栈溢出（stack overflow）**错误；
4. 有些问题用递归法解决起来更容易一些。

**迭代：**

1. 会在相关问题不成立时终止；
2. 执行每一轮迭代时，不需要开辟额外的内存；
3. 由于不需要耗费额外的内存，因此可以一直迭代下去；
4. 某些问题的迭代式解法，可能不如递归式解法那样直观。

## 注意事项

## 递归算法举例

斐波那契数列、阶乘

归并排序、快速排序

二分搜索/查找

树的遍历，以及树的相关问题

图的遍历

一些动态规划问题

分治算法

汉诺塔

回溯算法

## 其他案例

### 生成括号

分析：递归设计

# 回溯

## 背景

采用穷举搜索会遭遇两个主要困难：

1. 产生所有可能解的机制。对于有些问题而言，这些可能解会形成一种具备良好结构的集合。但还有很多问题，可能解无法形成具备如此规模结构的集合。
2. 第二个苦难，也是更加根本的困难在于需要生成和处理的可能解数量。一般地，该集合的规模至少随着问题规模呈指数增长。所以，穷举搜索只在这类题目很小规模的谜面上才可行。

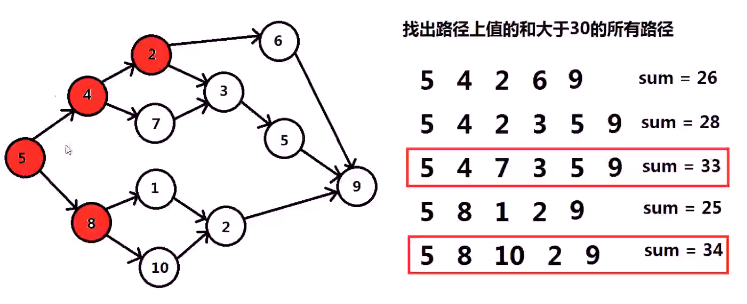
回溯法是对穷举搜索锁采取的蛮力做法的一种重要改进。它给出了一种生成可能解的方便方法，这样就可以避免生成不必要的可能解了。其核心思想在于，采用一次添加一个组件的方法来构造解，并且如下评估可能解的“半成品”：如果这个构造到一半的解可以再向前推进异步而不违反题设的约束，则选择第一个合法选项作为下一个部件。如果找不到合法选项作为下一个部件，那么就不再需要去考虑任何其余部件了。在这种情况下，算法就要执行回溯，把当前构造到一半的解的最后一个部件替换成该部件可选的下一个合法选项。

一般地，回溯法总是涉及一定数量的错误选项撤销动作。这个数量越小，算法找到解的速度就越快。尽管在最差情况下，某个回溯算法可能与穷举搜索一样，最终生成了所有的可能解，但是这种情况很罕见。

## 概述

回溯，是一种采用分治法来实现穷举式探索，也叫试探法。

回溯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。



**可以采用pruning（修剪、剪枝）技术来提升穷举式搜索的速度**。

回溯法的基本思想是：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根节点触发深度优先搜索解空间树。当搜索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去；如果该结点不包含问题的解，那就说明以该结点为根节点的子树一定不包含问题的最终解，因此要跳过对以该结点为根的子树的系统搜索，逐层向其祖先结点回溯。这个过程叫做解空间树的“剪枝”操作。

如果应用回溯法求解问题的所有解，要回溯到解空间树的树根，这样根节点的所有子树都被探索到才结束。如果只要求解问题的一个解，那么在探索解空间树时，只要搜索到问题的一个解就可以结束了。

## 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。

## 回溯算法举例

二进制串：生成含有n个二进制位的所有二进制串

生成由一定数量的字符所构成的全部字符串，这些字符可以从k种值里选

背包问题

生成一般的字符串

哈密尔顿回路

图染色问题

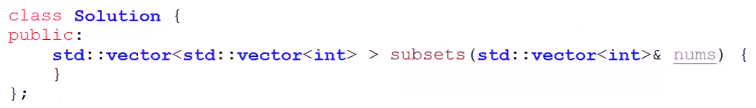
## 其它案例

### 求子集I

**题目：**已知一组数（其中无重复元素），求这组数可以组成的所有子集。结果中不可用无重复的子集。

例如：nums[]=[1,2,3]

结果为：[[],[1],[1,2],[1,2,3],[1,3],[2],[2,3],[3]]



注：Leetcode 78

**分析：**回溯法、位运算法

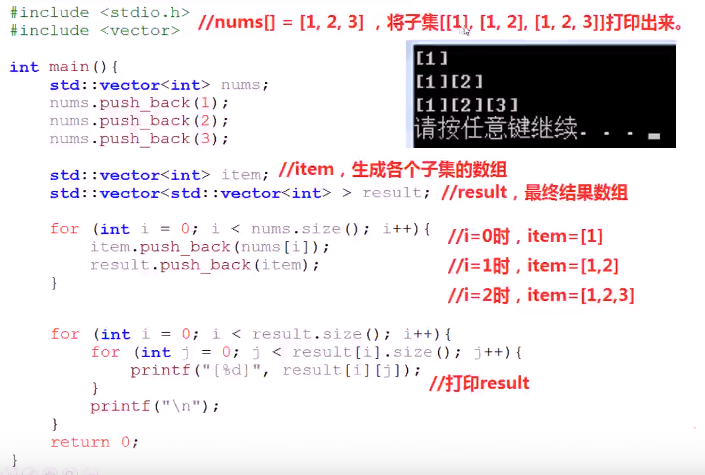
**方法一：回溯法**

在所有子集中，生成各个子集，[[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]，即是否选[1]，是否选[2]，是否选[3]的问题。

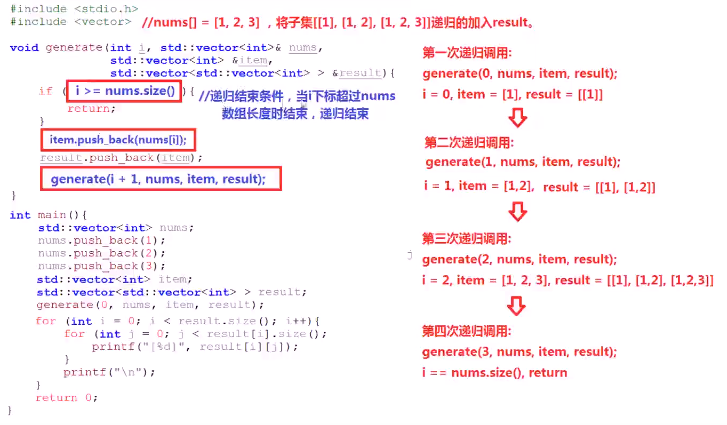
如果只使用循环，则使用循环程序难以直接模拟是否选某一元素的过程。

如果只是生成[1],[1,2],[1,2,3]三个子集，如何做？

**循环实现：**



**递归实现：**



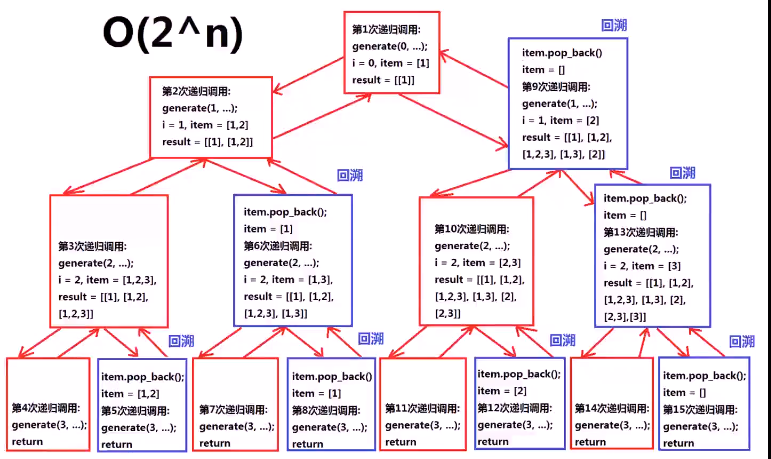
利用回溯方法生成子集，即对于每个元素，都有试探放入或不放入集合找那个的两个选择：

选择放入该元素，递归的进行后续元素的选择，完成放入该元素后续所有元素的试探；之后将其拿出，即再进行一次选择不放入该元素，递归的进行后续元素的选择，完成不放入该元素后续所有元素的试探。

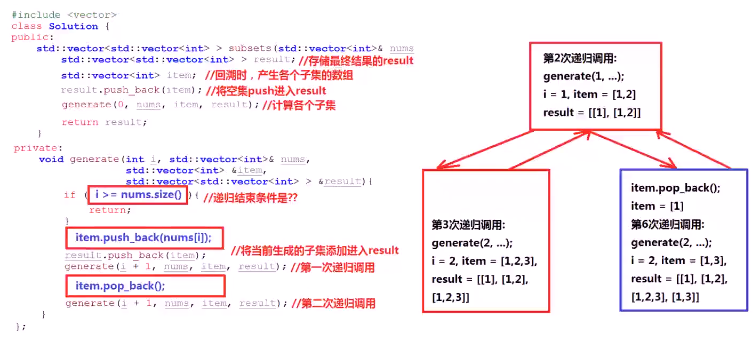
本来选择放入，再选择一次不放入的这个过程，称为回溯试探法。

例如：

元素数组：nums=[1,2,3,4,5,…]，子集生成数组item[]=[]。对于元素1，选择放入item，item=[1]，继续递归处理后续[2,3,4,5,…]元素；item=[1,…]；选择不放入item，item=[]，继续递归处理后续[2,3,4,5…]元素。



**代码：**



**方法二：**位运算

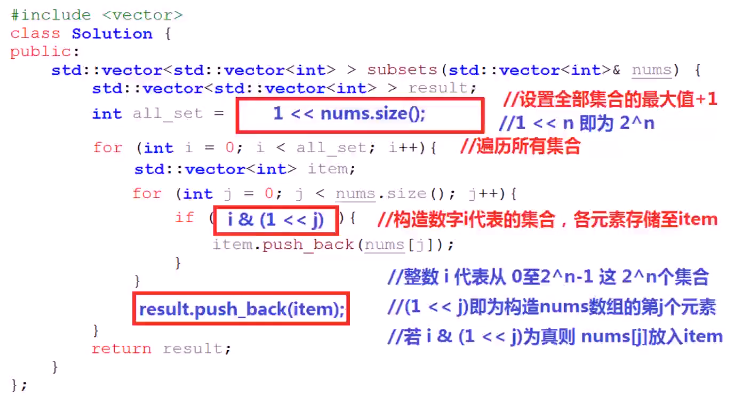
**分析：**



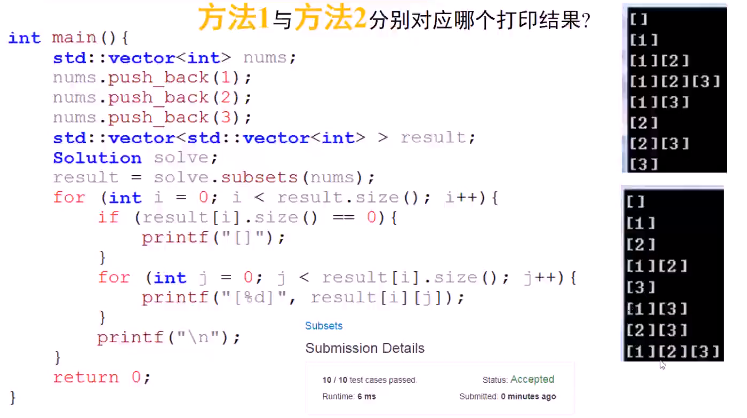




**代码：**



**测试：**



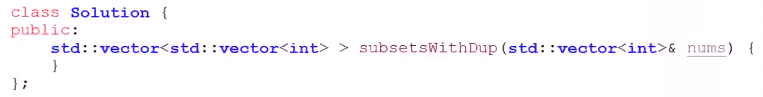
### 求子集II

**题目：**已知一组数（其中有重复元素），求这组数可以组成的所有子集。结果中无重复的子集。

例如：nums[]=[2,1,2,2]

结果为：[[],[1],[1,2],[1,2,2],[1,2,2,2],[2],[2,2],[2,2,2]]

注意：[2,1,2]与[1,2,2]是重复的集合！



注：Leetcode90

分析：回溯法

有两种重复原因：

1. 不同位置的元素组成的集合是同一个子集，顺序相同：

例如：[2,1,2,2]

选择第1,2,3个元素组成的子集：[2,1,2]

选择第1,2,4个元素组成的子集：[2,1,2]

1. 不同位置的元素组成的集合是同一个子集，虽然顺序不同，但仍然代表了同一个子集，因为集合中的元素是无序的。

例如：[2,1,2,2]

选择第1,2,3个元素组成的子集：[2,1,2]；

选择第2,3,4个元素组成的子集：[1,2,2]。

### 划分为k个相等的子集

**题目：**给定一个整数数组  nums 和一个正整数 k，找出是否有可能把这个数组分成 k 个非空子集，其总和都相等。

示例 1：

输入： nums = [4, 3, 2, 3, 5, 2, 1], k = 4

输出： True

说明： 有可能将其分成 4 个子集（5），（1,4），（2,3），（2,3）等于总和。

注意:

1 <= k <= len(nums) <= 16

0 < nums[i] < 10000

### 组合数之和

分析：回溯法、剪枝

### 迷宫问题

### 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：回溯法

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

### 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;

}