# 基本理论:

## (1)递归是一种编程技巧，方法是分治，推荐使用递归法好理解，推荐在几十到百层的数量级推荐使用递归。

## (2)使用cache（缓存）进行减少重复计算,很多题都可以用递归+cache的方法。

## (3)广度(BFS)一般用的少，深度(DFS)可以解决很多问题。与深度强相关的是回溯，属于一种编程技巧或者思路。

A:回溯有2种写法，循环控制回溯路径，但是回溯推荐用递归+cache的方法，因为特别容易看懂。visit记得要抹除

B:套路话但很容易的写法，深度搜索上其实也有2个概念。深度对于每个题不一样，广度也不同。

如在井字格上深度（step）定义是往前走一步，广度(option)是怎么走，广度是必须在一个step中需要全部完成的。

### 1.基本井字格

4行3列的格子，从（0，0）到端点的（4，3）不走远路的话可以有多少种走法？



（1）分治法，使用的是递归编程技巧

设f(m,n)为到达（m,n）的走法种树则：f(m,n) = f(m-1,n) + f(m,n-1)

int f(m,n)

{

if (m < 0 || n < 0) {

return 0;

}

if (m == 0 && n == 0) {

return 1;

}

return f(m-1, n) + f(m, n-1);

}

缓存减少重复计算，解决性能问题

int cache[100][100] = {0};

int f(m,n)

{

if (m < 0 || n < 0) {

return 0;

}

if (m == 0 && n == 0) {

return 1;

}

if (cache[m][n] != 0 ) {

return cache[m][n];

}

cache[m][n] = f(m-1, n) + f(m, n-1);

return cache[m][n];

}

（2）动态规划法

A:把显而易见的东西写上去

B:按照路径去计算

int dp[100][100] = {0};

int f\_dp(int m,int n)

{

int i = 0, j = 0;

dp[0][0] = 1;

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

dp[0][i] = 1;

}

for (i = 1; i <= m; i++) {

dp[i][0] = 1;

}

for (i = 1; i <= m; i++) {

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

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

}

}

return dp[m][n];

}

4行3列的格子，从（0，0）到端点的（4，3）可以走远路的话可以有多少种走法？

（3）深度搜索（DFS）

广度(BFS)一般用的少，深度(DFS)可以解决很多问题。与深度强相关的事回溯，属于一种编程技巧或者思路。

A:回溯有2种写法，循环控制回溯路径，但是推荐用递归+cache的方法，因为特别容易看懂。

B:套路话但很容易的写法，深度搜索上其实也有2格概念。深度对于每个题不一样，广度也不同。

如在井字格上深度（step）是往前走一步，广度(option)是怎么走，广度是必须在一个step中需要全部完成的。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*也是DFS模板,可用与其他DFS的问题\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int step(int x, int y, int m, int n)

{

int sum = 0;

if (x == m && y == n) {

return 1; // 走到目的地

}

if (x < 0 || y <0 || x > COL || y > ROW) {

return 0; // 判断边界

}

if (visited[x][y] == 1) {

return 0;

}

if (visited[x][y] == 1) {

return 0;

}

visited[x][y] = 1; //踩脚印

sum += step(x - 1,y,m,n); //向左

sum += step(x + 1,y,m,n); //向右

sum += step(x,y + 1,m,n); //向上

sum += step(x,y - 1,m,n); //向下

visited[x][y] = 0; //抹脚印

return sum;

}

### 2.八/N皇后问题

描述：4\*4的格子，皇后要在格子上，不能互相攻击。可以攻击的规则是左右上下，及45度角。求有多少中状态？



其实不用二维数组，用一个queue\_pos的一维数组，表示每一列皇后所在的行号。那么用深度搜索，step在这里定义就是部署一个皇后；广度是必需在每个step必须完成的，在这里就是部署在index列的某一行。

#define N 8

int queue\_pos[N] = {0};

static int g\_kind = 0;

void step(int index)

{

int i = 0;

int j = 0;

//printf("?\n");

// 部署到第index个皇后，就认为是是一种OK的状态

if (index == N + 1) {

g\_kind++;

}

// option广度上需要判断是否合理,从index列的0行到N行都试一下

for (i = 0 ; i < N; i++) {

//尝试部署一个

queue\_pos[index] = i;

// 判断是否符合不攻击的状态，第一个判断是这个位置不能与前面的位置出现在同一行

for (j = 0; j < index; j++) {

// 不能在同一行

if (queue\_pos[j] == queue\_pos[index]) {

break;

}

// 不能在45度

if (abs(queue\_pos[index] - queue\_pos[j]) == abs(index - j)) {

break;

}

}

if (j >= index) {

step(index + 1);

}

}

}

### 3.数独问题

9\*9的格子，即9个9宫格。

这里step的定义其实是填格子，options是填1-9的9格数字。探索只管前面。

### 4.括号匹配问题

题目描述：

给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。

例如，给出 n = 3，生成结果为：

[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

/\*这里step的定义就是放一个括号，直到放满2\*box个盒子；option是放左还是放右;

放左的条件是只要左边的没放完，就可以一直放;

\*/

void step(char\* ans, int left, int right, int box)

{

if (left + right == box) {

memcpy(res[g\_count], ans, strlen(ans) + 1);

g\_count++;

return;

}

// 只要左括号没有达到最大值就都可以放

if (left \* 2 < box) {

ans[left + right] = '(';

step(ans, left +1, right, box);

}

// 只要右括号比左括号少就都可以放

if (right < left) {

ans[left +right] = ')';

step(ans, left, right +1, box);

}

}

char \*\* generateParenthesis(int n, int\* returnSize){

char\* ans;

int i;

g\_count = 0;

if (n == 0) {

\*returnSize = 0;

return res;

}

if (n == 1) {

\*returnSize = 1;

res = (char\*\*)malloc(sizeof(char\*)\*2);

res[0] = (char\*)malloc(sizeof(char) \* 3);

memset(res[0], 0, 3);

memcpy(res[0], "()",3);

return res;

}

ans = (char\*)malloc(sizeof(char) \* (2 \* n) + 1);

memset(ans, 0, sizeof(char) \* (2 \* n) + 1);

res = malloc(sizeof(char\*) \* RES\_SIZE\_MAX);

for (i = 0; i < RES\_SIZE\_MAX; i++) {

res[i] = malloc(sizeof(char) \* (2 \* n + 1));

memset(res[i], 0 , sizeof(char) \* (2 \* n + 1));

}

step(ans, 0, 0 , 2 \* n);

\*returnSize = g\_count;

free(ans);

ans = NULL;

return res;

}

### 5.通配符号匹配问题

text = "ab.....def" pattern = "a?\*bf",写一个程序判断text是否与patter你是否匹配。？可以匹配任意单个字符，\*可以匹配0个或多个任意字符。

这个问题上step定义为在patter方向，一个字符在text匹配上了，如step一步匹配了一个pattern中的a;

自己没做出来，用上面的办法。

### 6. 划分为k个相等的子集【回溯递归】

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。

回溯问题分为“盒子”问题和“迷宫”问题。

本题即为盒子问题。

本题思路：这里的step其实是去往盒子里放可以和为target的数，盒子放完即可。Option其实是从len(nums)个数中选择。

，

#define true 1

#define false 0

#define bool int

int my\_cmp(const void\* a, const void\* b)

{

int \*pa = (int\*)a;

int \*pb = (int\*)b;

return \*pa - \*pb;

}

int dfs(int\* nums, int numsSize, int target, int\* flag, int start, int index, int k)

{

int i = 0;

if (k == 0)

return true;

for (i = start; i < numsSize && (nums[i] + index <= target); ++i) {

if (flag[i] == 0 ) {

flag[i] = 1;

if (nums[i] + index == target) {

if (dfs(nums, numsSize, target, flag, 0, 0, k - 1))

return true;【不能直接写成return dfs(nums, numsSize, target, flag, 0, 0, k - 1)否则构不成回溯，因为一旦失败后从这里返回就无法对flag清0了】

}

else if (dfs(nums, numsSize, target, flag, i + 1,nums[i] + index, k))

return true;

flag[i] = 0;

}

}

return false;

}

bool canPartitionKSubsets(int\* nums, int numsSize, int k)

{

int sum = 0;

int i, target;

int flag[40] ={0};

int max = -1;

for (i = 0; i <numsSize; i++) {

sum += nums[i];

if (max < nums[i]) {

max = nums[i];

}

}

if (sum % k != 0) {

return false;

}

target = sum / k;

if ( max > target) {

return false;

}

qsort(nums, numsSize, sizeof(int), my\_cmp);

return dfs(nums, numsSize, target,flag,0, 0, k);

}

### 7. [字母大小写全排列](https://leetcode-cn.com/problems/letter-case-permutation/)【回溯递归】

给定一个字符串S，通过将字符串S中的每个字母转变大小写，我们可以获得一个新的字符串。返回所有可能得到的字符串集合。

**示例:**

**输入:** S = "a1b2"

**输出:** ["a1b2", "a1B2", "A1b2", "A1B2"]

本题其实也是回溯类问题的盒子模型。

思路：将S中的字母个数num算出，就可以得到结果集的总数（2^num）。num其实就是盒子，那么在这里step的定义就是填一个盒子，而option就是2个，分别大小写字母。

1. vis[i][0] 表示字母本身，为1表示字母已经排列过，vis[i][1]表示对应的大或小写
2. res用于报存整个题目的结果，ans用于保存每次step过程中的结果，如果直接用res[g\_count][i]可能会造成越界，因此以后尽量step过程中另外使用数组保存过程的结果
3. option中用index从start开始第一是可以优化性能，第二有时会导致问题重复（比如二进制时钟问题中必须用start开始）

#define MAX 13

int vis[MAX][2] = {0};

int g\_count = 0;

void Step(char\*\* res, char\* S, char\* ans, int box, int index, int len)

{

    int i;

    if (box == 0) {

        memcpy(res[g\_count], ans, len + 1);

        g\_count++;

        return;

    }

    for (i = index; i < len; i++) {

        if (vis[i][0] == 1 && vis[i][1] == 1) {

            ans[i] = S[i];

            continue;

        }

        if (vis[i][0] != 1) {

            vis[i][0] = 1;

            ans[i] = S[i];

            Step(res, S, ans, box - 1, i + 1, len);

            vis[i][0] = 0;

        }

        if (vis[i][1] != 1) {

            vis[i][1] = 1;

            ans[i] = (S[i] >= 'a' && S[i] <= 'z') ? (S[i] - 32) : (S[i] + 32);

            Step(res, S, ans, box - 1, i + 1, len);

            vis[i][1] = 0;

        }

    }

}

char \*\* letterCasePermutation(char \* S, int\* returnSize){

    int len, i;

    int box = 0;

    int sum = 1;

    char \*\*res;

    char\* ans ;

    len = strlen(S);

    g\_count = 0;

    // 计算总共的大小，并标记非字母的已经处理

    for (i = 0; i < len; i++) {

        if ((S[i] >= 'A' && S[i] <= 'Z') || (S[i] >= 'a' && S[i] <= 'z')) {

            sum \*= 2;

            box++;

            vis[i][0] = 0;

            vis[i][1] = 0;

        }

        else {

            vis[i][0] = 1;

            vis[i][1] = 1;

        }

    }

    // 如果没有字母直接返回原串

    if(box == 0) {

        \*returnSize = 1;

        res = (char\*\*)malloc(sizeof(char\*) \* (\*returnSize));

        res[0] = (char\*)malloc(sizeof(char) \* (len + 1));

        memcpy(res[0], S, len + 1);

        return res;

    }

    // 分配空间

    \*returnSize = sum;

    res = (char\*\*)malloc(sizeof(char\*) \* (\*returnSize));

    for (i = 0; i < \*returnSize; i++) {

        res[i] = (char\*)malloc(sizeof(char) \* (len + 1));

        memset(res[i], 0 , sizeof(char) \* (len + 1));

    }

    // ans用于保存每次step的结果，如果直接用res[g\_count][i]会导致越界。这里赋值，可以规避像1ab2的串第一次只返回1ab的问题

    ans = (char\*)malloc(len + 1);

    memcpy(ans, S, len + 1);

    Step(res, S, ans, box, 0, len);

    return res;

}

### 8. [401. 二进制手表](https://leetcode-cn.com/problems/binary-watch/)【回溯递归】

题目：二进制手表顶部有 4 个 LED 代表**小时（0-11）**，底部的 6 个 LED 代表**分钟（0-59）**。每个 LED 代表一个 0 或 1，最低位在右侧。

思路：其实也是属于盒子问题，这里step的定义其实就是num个灯，option就是9个选择，当然要用index为start开始搜索。

/\*这里的step定义为放一个灯；option可以定义为从0-10个灯选择位置\*/

/\*这里的step定义为放一个灯；option可以定义为从0-10个灯选择位置\*/

1. 每个结果最多6个字符串，并且用全局的res减少step函数的参数
2. cal[]数组可以减少计算，并且可以在参数中用hour与min减少计算
3. 注意

// 用于计数

int g\_count = 0;

char \*\*res = NULL;

int cal[] = {8,4,2,1,32,16,8,4,2,1};

#define MAX 0xfffff

// 每个字符串最长为6 如12:00

#define COL\_MAX 6

void step(int\* vis, int hour, int min, int start, int num)

{

int i;

if (num == 0) {

sprintf(res[g\_count], "%d:%02d", hour,min);

g\_count++;

return;

}

for (i = start; i <= 9; i++) {

if (vis[i] != 0) {

continue;

}

vis[i] = 1;

if (i >= 6 && hour + cal[i] >= 12) {

vis[i] = 0;

continue;

}

if (i < 6 && min + cal[i] >= 60) {

vis[i] = 0;

continue;

}

if (i >= 6) {

step(vis, hour + cal[i], min, i + 1, num - 1);

vis[i] = 0;

}

else {

step(vis, hour, min + cal[i], i + 1, num - 1);

vis[i] = 0;

}

}

}

char \*\* readBinaryWatch(int num, int\* returnSize){

int vis[10] = {0};

int i;

res = (char\*\*)malloc(sizeof(char\*) \* MAX);

for (i = 0; i < MAX; i++) {

res[i] = (char\*)malloc(sizeof(char) \* COL\_MAX);

memset(res[i], 0, sizeof(char) \* COL\_MAX);

}

g\_count = 0;

step(vis, 0 ,0, 0, num);

\*returnSize = g\_count;

return res;

}

### 9. [17. 电话号码的字母组合](https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number/)【深搜递归不算回溯】

题目描述：给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



**输入：**"23"

**输出：**["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

思路：在本题其实也是盒子模型。盒子的大小其实就是输入字符串的长度。因此step的定义就是放一个盒子（字母），option就是从3或者4个字母中选择。

可以制作一个表：

static char\* g\_table[] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

方便计算放第box个盒子时有多少种选择strlen(g\_table[digits[box] – ‘0’])。

很好理解的是本地不需要回溯，或者说标记。如果标记了，那么对于有重复数字的输入就会少了几种选择。

#define MAX 9

static char\* g\_table[] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

char \*\*res = NULL;

char\* ans = NULL;

int g\_count = 0;

int vis[10][4] = {0};

void step(char \*digits, int box, int box\_max)

{

int i;

if (box == box\_max) {

memcpy(res[g\_count], ans, strlen(ans));

g\_count++;

return;

}

for (i = 0; i < strlen(g\_table[digits[box] - '0']); i++) {

//if (vis[digits[box] - '0'][i] == 0) {

//vis[digits[box] - '0'][i] = 1;不能做标记

ans[box] = g\_table[digits[box] - '0'][i];

step(digits, box + 1, box\_max);

//vis[digits[box] - '0'][i] = 0;

//}

}

}

char \*\* letterCombinations(char \* digits, int\* returnSize){

int sum = 1;

int i, j, len;

g\_count = 0;

len = strlen(digits);

if (len == 0) {

res = (char\*\*)malloc(sizeof(char\*));

res[0] = malloc(sizeof(char) \* 1);

memset(res[0], 0, sizeof(char) \* MAX);

\*returnSize = 1;

return res;

}

// 计算总共可以返回多少个字符串

for (i = 0; i < len; i++) {

sum \*= strlen(g\_table[digits[i] - '0']);

}

\*returnSize = sum;

// 分配空间

res = (char\*\*)malloc(sum \* sizeof(char\*));

for (i = 0; i < sum; i++) {

res[i] = malloc(sizeof(char) \* MAX);

memset(res[i], 0, sizeof(char) \* MAX);

}

for (i = 0; i < 10; i++) {

for (j = 0; j < 4; j++) {

vis[i][j] = 0;

}

}

ans = (char\*)malloc(MAX);

memset(ans, 0 , MAX);

step(digits, 0, len);

return res;

}