C/C++从入门到精通-高级程序员之路

【算法篇】

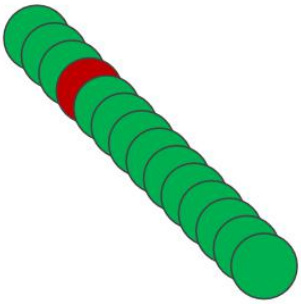
五大常规算法

第 1 节 分治法

一个装有 16 枚硬币的袋子，16 枚硬币中有一个是伪造的，伪造的硬币和普通硬币从表面上看不出有任何差别，但是那个伪造的硬币比真的硬币要轻。现有给你一台天平，请你在尽可能最短的时间内找出那枚伪造的硬币。

常规思维：

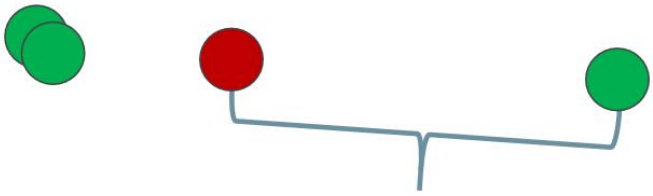
每次从待比较的硬币中取两枚进行计较，如果天平平衡（相等）就继续取剩下的硬币进行比较



待检测的硬币

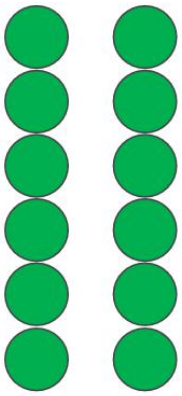
使用天平检测硬币

继续以上过程，直到找到硬币。



待检测的硬币

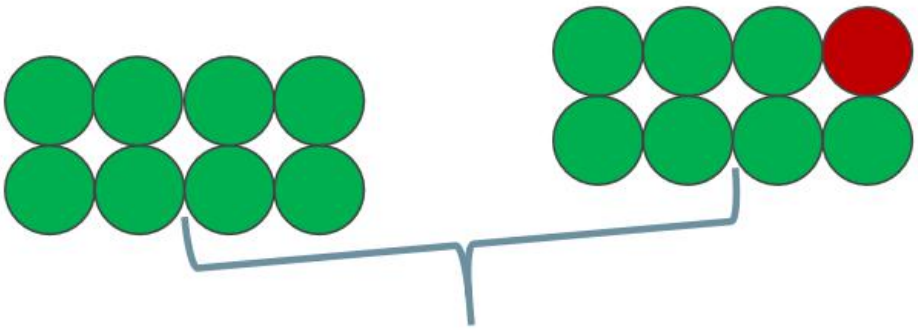
使用天平检测到硬币



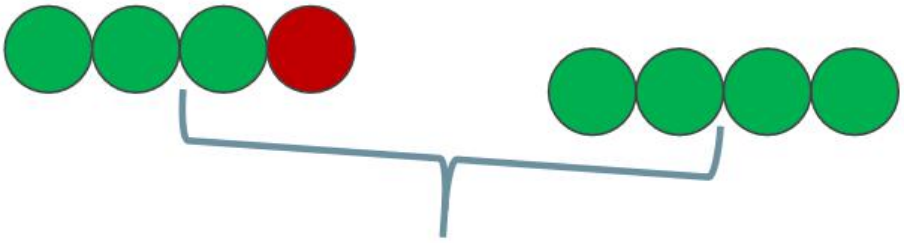
已检测过的硬币

强者思维:

我们先将 16 枚硬币分为左右两个部分，各为 8 个硬币，分别称重，必然会有一半轻一半重，而我们要的就是轻的那组,重的舍去。接下来我们继续对轻的进行五五分，直至每组剩下一枚或者两枚硬币，这时我们的问题自然就解决了，下面用一张图进行更好的理解。



第一次检测



第二次检测

第三次检测

第四次检测，找到结果

分治法——见名思义，即分而治之，从而得到我们想要的最终结果。分治法的思想是将一个规模为 N 的问题分解为 k 个较小的子问题，这些子问题遵循的处理方式就是互相独立且与原问题相同。

两部分组成

分（divide）：递归解决较小的问题

治（conquer）：然后从子问题的解构建原问题的解

三个步骤

1、分解（Divide）：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

2、解决（Conquer）：若子问题规模较小而容易被解决则直接解决，否则递归地解各个子问题；

3、合并（Combine）：将各个子问题的解合并为原问题的解。

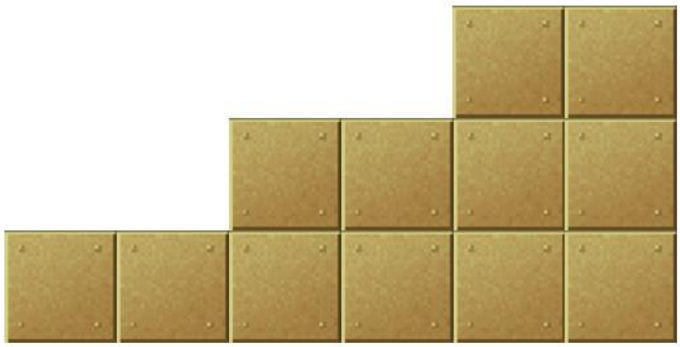
例程: 二分查找算法实现

```
#include <stdio.h>
#include <stdlib.h>


/*递归实现二分查找
参数:
     arr    - 有序数组地址 arr
     minSub - 查找范围的最小下标 minSub
     maxSub - 查找范围的最大下标 maxSub
     num    - 带查找数字

返回:找到则返回所在数组下标，找不到则返回-1
*/
```

```
int BinarySearch(int* arr,int minSub,int maxSub,int num){

    if(minSub>maxSub){
        return -1;//找不到 num 时,直接返回
    }

    int mid=(minSub+maxSub)/2;

    if(num<arr[mid]){
        return BinarySearch(arr,minSub,mid-1, num);
    }
    else if(num>arr[mid]){
        return BinarySearch(arr,mid+1,maxSub, num);
    }
    else{
        return mid;//找到 num 时直接返回
    }
}


int main(void){
    int  arr[]={1, 3, 7, 9, 11};

    int index = BinarySearch(arr, 0, 4, 8);
    printf("index: %d\n", index);

    system("pause");
    return 0;
}
```

第 2 节 动态规划算法

人工智能时代，各国都在大力研究机器人技术，也制造出各种各样的机器人，比如：为了解决男女失衡而制造的美女机器人，假如你参与了某美女机器人的研发，你在这个项目中要求实现一个统计算法：如果美女机器人一次可以上 1 级台阶，也可以一次上 2 级台阶。求美女机器人走一个 n 级台阶总共有多少种走法。



咋一看，无从下手，不急，我们不是讲了分治法嘛？这不是可以乘机表演一下？

启发性思考：

分治法核心思想：从上往下分析问题，大问题可以分解为子问题，子问题中还有更小的子问题

比如总共有 5 级台阶,求有多少种走法；由于机器人一次可以走两级台阶，也可以走一级台阶，所以我们可以分成两个情况

- 机器人最后一次走了两级台阶，问题变成了“走上一个 3 级台阶，有多少种走法？”
- 机器人最后一步走了一级台阶，问题变成了“走一个 4 级台阶，有多少种走法？”

我们将求 n 级台阶的共有多少种走法用 f(n) 来表示，则

f(n) = f(n-1) + f(n-2);

由上可得 f(5) = f(4) + f(3);

f(4) = f(3) + f(2);

f(3) = f(2) + f(1);

边界情况分析

走一步台阶时，只有一种走法，所以 f(1)=1

走两步台阶时，有两种走法，直接走 2 个台阶，分两次每次走 1 个台阶，所以 f(2)=2

走两个台阶以上可以分解成上面的情况

这符合我们讲解的分治法的思想：　　**分而治之**

代码实现

```
#include <stdio.h>
#include <stdlib.h>


/*递归实现机器人台阶走法统计
参数:
    n    - 台阶个数
返回: 上台阶总的走法
*/
int WalkCout(int n){
    if(n<0) return 0;

    if(n==1) return 1;       //一级台阶，一种走法
    else if(n==2) return 2; //二级台阶，两种走法
    else {                   //n 级台阶，  n-1 个台阶走法 + n-2 个台阶的
走法
        return WalkCout(n-1) + WalkCout(n-2);
    }

}

int main(void){

    for(int i=1; i<=6; i++){
        printf("%d 台阶共有 %d 种走法\n", i, WalkCout(i));
    }
    system("pause");
    return 0;
}
```

但是，如果细心的同学是否会注意到，上面的代码中存在很多重复的计算？

比如：　f(5) = f(4) + f(3)　　计算分成两个分支：

f(4) = f(3)+f(2) = f(2) + f(1) + f(2);

f(3) = f(2) + f(1);

上面阴影的部分就是重复计算的一部分！

有没有办法避免重复计算的部分？

其实我们可以从下向上分析推断问题。

f(1) = 1

f(2) = 2

f(3) = f(1) + f(2) = 3

f(4) = f(3) + f(2) = 3 + 2 = 5

f(5) = f(4) + f(3) = 5 + 3 = 8

。。。依次类推 。。。

实际求解过程

```
#include <stdio.h>
#include <stdlib.h>


int WalkCount2(int n) {
    int ret = 0;

    //无意义的情况
    if(n <= 0)
        return 0;
    if(n == 1)
        return 1;
    if(n == 2)
        return 2;

    //数组用于存储走 n 个台阶的走法数
    int* value = new int[n + 1];
    value[0] = 0;
    value[1] = 1;
```

```
    value[2] = 2;

    for(int i = 3; i <= n; i++) {
        value[i] = value[i - 1] + value[i - 2];
    }

    ret = value[n];
    delete value;

    return ret;
}


int main(void){

    for(int i=1; i<=6; i++){
        printf("%d 台阶共有 %d 种走法\n", i, WalkCount2(i));
    }
    system("pause");
    return 0;
}
```

这就是动态规划法 !!!

动态规划也是一种分治思想，但与分治算法不同的是，分治算法是把原问题分解为若干子问题，自顶向下，求解各子问题，合并子问题的解从而得到原问题的解。动态规划也是自顶向下把原问题分解为若干子问题，不同的是，然后自底向上，先求解最小的子问题，把结果存储在表格中，在求解大的子问题时，直接从表格中查询小的子问题的解，避免重复计算，从而提高算法效率。

什么时候要用动态规划？

如果要求一个问题的最优解（通常是最大值或者最小值），而且该问题能够分解成若干个子问题，并且小问题之间也存在重叠的子问题，则考虑采用动态规划。

怎么使用动态规划？

五步曲解决：

1. 判题题意是否为找出一个问题的最优解
2. 从上往下分析问题，大问题可以分解为子问题，子问题中还有更小的子问题
3. 从下往上分析问题 ，找出这些问题之间的关联（状态转移方程）
4. 讨论底层的边界问题
5. 解决问题（通常使用数组进行迭代求出最优解）

课后习题： 给你一根长度为 n 的金条，请把金条剪成 m 段（m 和 n 都是整数，n>1 并且 m>1)每断金条的长度记为 k[0],k[1],...,k[m].请问 *k[0] k[1]*...*k[m]可能的最大乘积是多少?

第 3 节 回溯算法

回溯的基本原理

在问题的解空间中，按深度优先遍历策略，从根节点出发搜索解空间树。算法搜索至解空间的任意一个节点时，先判断该节点是否包含问题的解。如果确定不包含，跳过对以该节点为根的子树的搜索，逐层向其祖先节点回溯，否则进入该子树，继续深度优先搜索。

回溯法解问题的所有解时，必须回溯到根节点，且根节点的所有子树都被搜索后才结束。回溯法解问题的一个解时，只要搜索到问题的一个解就可结束。

回溯的基本步骤

1. 定义问题的解空间

2. 确定易于搜索的解空间结构

3. 以深度优先搜索的策略搜索解空间，并在搜索过程中尽可能避免无效搜索

名企面试题：

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如在下面的 3×4 的矩阵中包含一条字符串“bfce”的路径（路径中的字母用下划线标出）。但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

A B T G
C F C S
J D E H

解题思路：

首先，在矩阵中任选一个格子作为路径的起点。如果路径上的第 i 个字符不是待搜索的目标字符 ch，那么这个格子不可能处在路径上的第 i 个位置。如果路径上的第 i 个字符正好是 ch，那么往相邻的格子寻找路径上的第 i+1 个字符。除在矩阵边界上的格子之外，其他格子都有 4 个相邻的格子。重复这个过程直到路径上的所有字符都在矩阵中找到相应的位置。

由于路径不能重复进入矩阵的格子，还需要定义和字符矩阵大小一样的布尔值矩阵，用来标识路径是否已经进入每个格子。 当矩阵中坐标为（row, col）的格子和路径字符串中相应的字符一样时，从 4 个相邻的格子(row,col-1),(row-1,col),(row,col+1)以及(row+1,col)中去定位路径字符串中下一个字符，如果 4 个相邻的格子都没有匹配字符串中下一个的字符，表明当前路径字符串中字符在矩阵中的定位不正确，我们需要回到前一个，然后重新定位。

源码实现：

```
#include <stdio.h>
#include <string>

using namespace std;

bool hasPathCore(const char* matrix, int rows, int cols, int row, int col,
const char* str, int& pathLength, bool* visited);

/*****************************************
功能：  查找矩阵中是否含有 str 指定的字串
参数说明：
     matrix 输入矩阵
     rows   矩阵行数
     cols   矩阵列数
     str    要搜索的字符串
返回值：  是否找到 true 是，false 否
*****************************************/
bool hasPath(const char* matrix, int rows, int cols, const char* str)
{
    if(matrix == nullptr || rows < 1 || cols < 1 || str == nullptr)
        return false;

    bool *visited = new bool[rows * cols];
    memset(visited, 0, rows * cols);

    int pathLength = 0;
    //遍历矩阵中每个点，做为起点开始进行搜索
    for(int row = 0; row < rows; ++row)
    {
        for(int col = 0; col < cols; ++col)
        {
            if(hasPathCore(matrix, rows, cols, row, col, str,
                pathLength, visited))
            {
                return true;
            }
        }
    }

    delete[] visited;

    return false;
}
```

```
/*探测下一个字符是否存在*/
bool hasPathCore(const char* matrix, int rows, int cols, int row,
    int col, const char* str, int& pathLength, bool* visited)
{
    if(str[pathLength] == '\0')
        return true;

    bool hasPath = false;
    if(row >= 0 && row < rows && col >= 0 && col < cols
        && matrix[row * cols + col] == str[pathLength]
        && !visited[row * cols + col])
    {
        ++pathLength;
        visited[row * cols + col] = true;

        hasPath = hasPathCore(matrix, rows, cols, row, col - 1,
            str, pathLength, visited)
            || hasPathCore(matrix, rows, cols, row - 1, col,
                str, pathLength, visited)
            || hasPathCore(matrix, rows, cols, row, col + 1,
                str, pathLength, visited)
            || hasPathCore(matrix, rows, cols, row + 1, col,
                str, pathLength, visited);

        if(!hasPath)
        {
            --pathLength;
            visited[row * cols + col] = false;
        }
    }

    return hasPath;
}

/*单元测试代码*/
void Test(const char* testName, const char* matrix, int rows, int cols,
const char* str, bool expected)
{
    if(testName != nullptr)
        printf("%s begins: ", testName);

    if(hasPath(matrix, rows, cols, str) == expected)
        printf("Passed.\n");
    else
        printf("FAILED.\n");
}
```

```
//ABTG
//CFCS
//JDEH

//BFCE
void Test1()
{
    const char* matrix = "ABTGCFCSJDEH";
    const char* str = "BFCE";

    Test("功能测试 1", (const char*) matrix, 3, 4, str, true);
}

//ABCE
//SFCS
//ADEE

//SEE
void Test2()
{
    const char* matrix = "ABCESFCSADEE";
    const char* str = "SEE";

    Test("功能测试 2", (const char*) matrix, 3, 4, str, true);
}

//ABTG
//CFCS
//JDEH

//ABFB
void Test3()
{
    const char* matrix = "ABTGCFCSJDEH";
    const char* str = "ABFB";

    Test("功能测试 3", (const char*) matrix, 3, 4, str, false);
}

//ABCEHJIG
//SFCSLOPQ
//ADEEMNOE
//ADIDEJFM
//VCEIFGGS

//SLHECCEIDEJFGGFIE
void Test4()
```

```
{
    const char* matrix = "ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS";
    const char* str = "SLHECCEIDEJFGGFIE";

    Test("功能测试 4", (const char*) matrix, 5, 8, str, true);
}

//ABCEHJIG
//SFCSLOPQ
//ADEEMNOE
//ADIDEJFM
//VCEIFGGS

//SGGFIECVAASABCEHJIGQEM
void Test5()
{
    const char* matrix = "ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS";
    const char* str = "SGGFIECVAASABCEHJIGQEM";

    Test("功能测试 5", (const char*) matrix, 5, 8, str, true);
}

//ABCEHJIG
//SFCSLOPQ
//ADEEMNOE
//ADIDEJFM
//VCEIFGGS

//SGGFIECVAASABCEEJIGOEM
void Test6()
{
    const char* matrix = "ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS";
    const char* str = "SGGFIECVAASABCEEJIGOEM";

    Test("功能测试 6", (const char*) matrix, 5, 8, str, false);
}

//ABCEHJIG
//SFCSLOPQ
//ADEEMNOE
//ADIDEJFM
//VCEIFGGS

//SGGFIECVAASABCEHJIGQEMS
void Test7()
{
    const char* matrix = "ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS";
```

```
    const char* str = "SGGFIECVAASABCEHJIGQEMS";

    Test("功能测试 7", (const char*) matrix, 5, 8, str, false);
}

//AAAA
//AAAA
//AAAA

//AAAAAAAAAAAA
void Test8()
{
    const char* matrix = "AAAAAAAAAAAA";
    const char* str = "AAAAAAAAAAAA";

    Test("边界值测试 8", (const char*) matrix, 3, 4, str, true);
}

//AAAA
//AAAA
//AAAA

//AAAAAAAAAAAAA
void Test9()
{
    const char* matrix = "AAAAAAAAAAAA";
    const char* str = "AAAAAAAAAAAAA";

    Test("边界值测试 9", (const char*) matrix, 3, 4, str, false);
}

//A

//A
void Test10()
{
    const char* matrix = "A";
    const char* str = "A";

    Test("边界值测试 10", (const char*) matrix, 1, 1, str, true);
}

//A

//B
void Test11()
{
```

```
    const char* matrix = "A";
    const char* str = "B";

    Test("边界值测试 11", (const char*) matrix, 1, 1, str, false);
}

void Test12()
{
    Test("特殊情况测试 12", nullptr, 0, 0, nullptr, false);
}

int main(int argc, char* argv[])
{
    Test1();
    Test2();
    Test3();
    Test4();
    Test5();
    Test6();
    Test7();
    Test8();
    Test9();
    Test10();
    Test11();
    Test12();

    system("pause");

    return 0;
}
```

第 4 节 贪心算法

贪婪算法(贪心算法)是指在对问题进行求解时，在每一步选择中都采取最好或者最优(即最有利)的选择，从而希望能够导致结果是最好或者最优的算法。

请看下面案例，假设有如下课程，希望尽可能多的将课程安排在一间教室里：

课程	开始时间	结束时间
美术	9:00	10:00
英语	9:30	10:30
数学	10:00	11:00
计算机	10:30	11:30
音乐	11:00	12:00

这个问题看似要思考很多，实际上算法很简单:

1. 选择结束最早的课，便是要在这教室上课的第一节课
2. 接下来，选择第一堂课结束后才开始的课，并且结束最早的课，这将是第二节在教室上的课。

重复这样做就能找出答案，这边的选择策略便是结束最早且和上一节课不冲突的课进行排序，因为每次都选择结束最早的，所以留给后面的时间也就越多，自然就能排下越多的课了。

每一节课的选择都是策略内的局部最优解(留给后面的时间最多)，所以最终的结果也是近似最优解(这个案例上就是最优解)。

贪婪算法所得到的结果往往不是最优的结果(有时候会是最优解)，但是都是相对近似(接近)最优解的结果。

贪婪算法并没有固定的算法解决框架，算法的关键是贪婪策略的选择，根据不同的问题选择不同的策略。

基本思路

其基本的解题思路为：

- 建立数学模型来描述问题
- 把求解的问题分成若干个子问题
- 对每一子问题求解，得到子问题的局部最优解
- 把子问题对应的局部最优解合成原来整个问题的一个近似最优解

钱币找零问题

假设1元、2元、5元、10元、20元、50元、100元的纸币分别有c0，c1，c2，c3，c4，c5，c6张。现在要用这些钱来支付K元，至少要用多少张纸币？

解题思路：

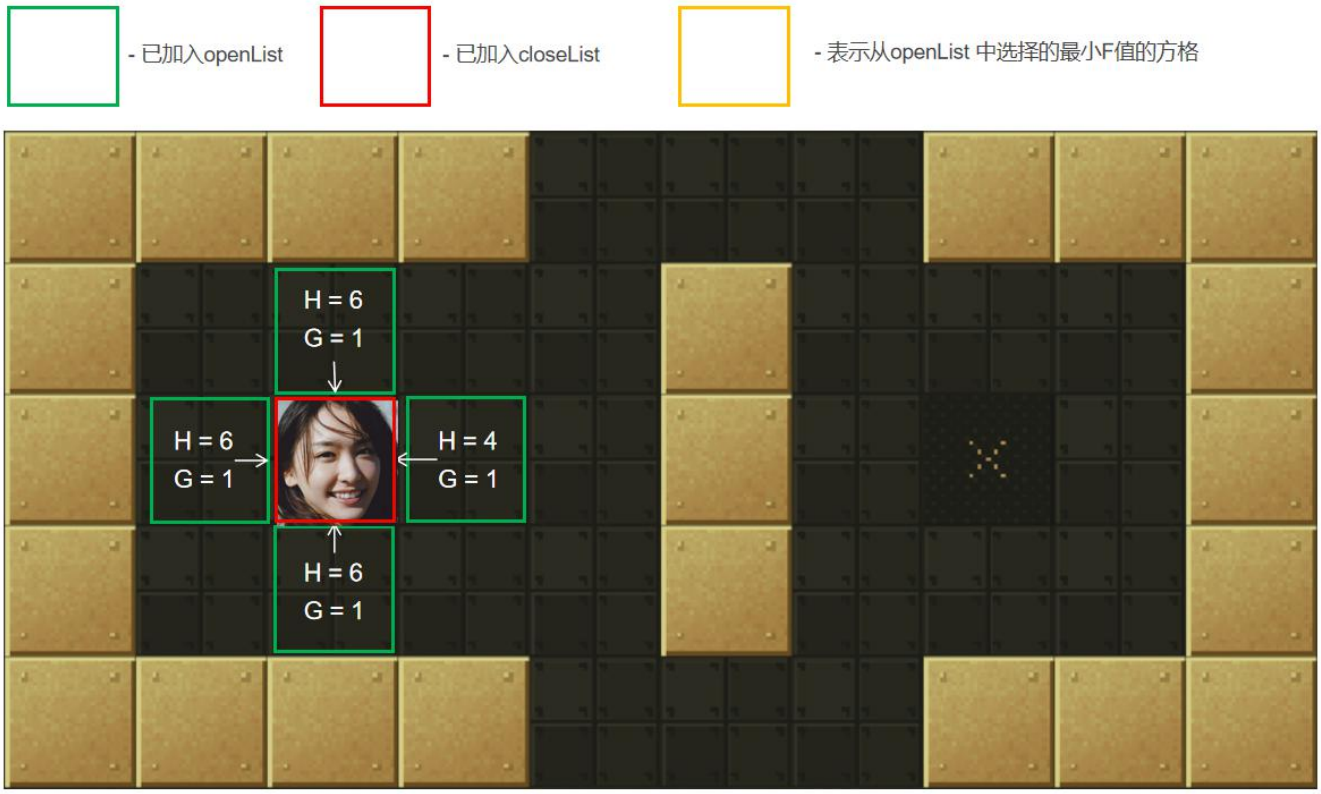
用贪心算法的思想，很显然，每一步尽可能用面值大的纸币即可。在日常生活中我们自然而然也是这么做的。在程序中已经事先将Value按照从小到大的顺序排好

```
#include <stdio.h>
#include <stdlib.h>

#define N  7

//int value[N]={1, 2, 5, 10, 20, 50, 100};  //800
//int count[N]={10, 2, 3, 1 ,2 ,3 , 5};

int value[N]={1, 2, 5, 10, 20, 50, 100};  //800
int count[N]={0, 0, 0, 0 ,3 ,1 , 0};


/**************************
 *对输入的零钱数，找到至少要用的纸币的数量
 *参数：
 *      money - 要找/支付的零钱数
 *返回：
 *    至少要用的纸币的数量，-1 表示找不开
 **************************/

int solve(int money){
    int num = 0;
    int i = 0;

    for(i=N-1; i>=0; i--){
        int j = money/value[i];  //120 100 1   220 100 2
        int c = j>count[i]?count[i]:j;

        printf("需要用面值 %d 的纸币 %d 张\n", value[i], c);
```

```
        money -= c*value[i];
        num += c;

        if(money==0) break;

    }

    if(money > 0) num = -1;

    return num;
}



int main(void){
    int money = 0;
    int num = 0;

    printf("请输入要支付的零的数目：\n");
    scanf_s("%d",  &money);

    num = solve(money);

    if(num==-1){
        printf("对不起，找不开\n");
    }else {
        printf("成功的使用至少%d 张纸币实现找零/支付！\n", num);
    }

    system("pause");
    return 0;
}
```

第 5 节 分支定界法

分支定界 (branch and bound) 算法是一种在问题的解空间上搜索问题的解的方法。但与回溯算法不同，分支定界算法采用广度优先或最小耗费优先的方法搜索解空间树，并且，在分支定界算法中，每一个活结点只有一次机会成为扩展结点。



利用分支定界算法对问题的解空间树进行搜索，它的搜索策略是：

1 ．产生当前扩展结点的所有孩子结点；

2 ．在产生的孩子结点中，抛弃那些不可能产生可行解（或最优解）的结点；

3 ．将其余的孩子结点加入活结点表；

4 ．从活结点表中选择下一个活结点作为新的扩展结点。

如此循环，直到找到问题的可行解（最优解）或活结点表为空。

从活结点表中选择下一个活结点作为新的扩展结点，根据选择方式的不同，分支定界算法通常可以分为两种形式：

1 ． FIFO(First In First Out) 分支定界算法：按照先进先出原则选择下一个活结点作为扩展结点，即从活结点表中取出结点的顺序与加入结点的顺序相同。

2 ．最小耗费或最大收益分支定界算法：在这种情况下，每个结点都有一个耗费或收益。假如要查找一个具有最小耗费的解，那么要选择的下一个扩展结点就是活结点表中具有最小耗费的活结点；假如要查找一个具有最大收益的解，那么要选择的下一个扩展结点就是活结点表中具有最大收益的活结点。