实验题第 1 题

(1) 设计思路

给定一个 m 行 n 列的矩阵，从左上角开始每次只能向右或向下移动，最后到达右下角的位置，移动路径上的数字之和作为这条路径的路径和。本题要求**设计一个算法求所有路径和中的最小路径和**。

- 根据动态规划程序的设计步骤，本题着重设计两个步骤：一是子结构的划分方法，二是建立优化函数，列出递推方程及边界条件。
- 1. 子结构的划分方法：
  - 问题要求从左上角 (0,0) 到右下角 (n,n) 所有路径中长度最大的路径，且每次只可走一格。则可划分为求到 (n-1,n) 的最短路径或 (n,n-1) 的最短路径的问题。取其中较小值，加上 m(n,n) 即为终解。
- 2. 优化函数、递推方程和边界条件：
  - 如果用$f(a,b)$表示从(0,0)到(a,b)的所有路径中最小路径和，不难得出优化函数

$$f(a,b) = \min(f(a,b-1), f(a-1,b)) + m(a,b).$$

  - 若用 L(n,n)表示从左上到 (n,n) 的最优路径，则可得递推方程为：

$$L(n,n) = \min(L(n,n-1), L(n-1,n)) + m(n,n).$$

  - 易得边界条件为左上(0,0)到自己本身要走的长度，即 L(0,0) = m(0,0).
- 根据以上划分方式以及递推方程，可得动态规划程序如下：

(2) 源代码

MatrixSP::MatrixSP(int m, int r, int c) //构造函数

```
MatrixSP::MatrixSP(int **m, int r, int c){
    this->m = m;
    this->r = r;
    this->c = c;
    //初始化动态规划矩阵
    for(int i = 0; i < r; i++){
        for(int j = 0; j < c; j++){
            dp[i][j] = 0;}}
}
```

void MatrixSP::Solve() //动态规划求解

```
void MatrixSP::Solve(){
    //动态规划求解
    for(int i = 0; i < r; i++){
        for(int j = 0; j < c; j++){
            if(i == 0 && j == 0){
                dp[i][j] = m[i][j];
                tag[i][j].first = 0;
                tag[i][j].second = 0;
            }
            else
            if(i == 0 && j != 0){
                dp[i][j] = dp[i][j-1] + m[i][j];
                tag[i][j].first = i;
```

```
                tag[i][j].second = j-1;
            }
            else
            if(i != 0 && j == 0){
                dp[i][j] = dp[i-1][j] + m[i][j];
                tag[i][j].first = i-1;
                tag[i][j].second = j;
            }
            else
            {
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + m[i][j];
                if (dp[i-1][j] < dp[i][j-1]){
                    tag[i][j].first = i-1;
                    tag[i][j].second = j;
                }
                else{
                    tag[i][j].first = i;
                    tag[i][j].second = j-1;
                }
            }
        }
    }
}

int MatrixSP::GetMinSum(){
    return dp[r-1][c-1];
}
```

void MatrixSP::ShowPath()　　//展示路径

```
void MatrixSP::ShowPath(int i, int j) {
    if (i == 0 && j == 0) {
        cout << m[i][j] << " ";
        return;
    }
    ShowPath(tag[i][j].first, tag[i][j].second);
    if(i == r-1 && j == c-1)
        cout << m[i][j];
    else
        cout << m[i][j] << " ";
}
void MatrixSP::ShowPath(){
    ShowPath(r-1,c-1);
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得注意的是，由于 dp 数组是从左上到右下构建的，所以标记数组 tag 必定是由更深处（右下）的点指向更浅处（左上）的点的。又因为最后输出需要从左上到右下输出路径，所以可以利用递归先回溯到最深处，再逐层回溯输出，以保证顺序正确。

实验题第 2 题

(1) 设计思路

本题要求设计一个算法，求解使不合法的括号序列成为合法的括号序列至少需要添加的括号数目。

- 根据动态规划程序的设计步骤，本题依旧着重设计两个步骤：一是子结构的划分方法，二是建立优化函数，列出递推方程及边界条件。
- 1. 子结构的划分方法：
  - 问题要求使不合法括号序列变为合法序列需要添加的最少括号数。则可把问题划分为求解最外层两个括号合法化需要的括号数，再求解内层子串合法化需要的最少括号数，最后相加的问题。
- 2. 优化函数、递推方程和边界条件：
  - 如果用$f(a,b)$表示合法化 str(a)和 str(b)这一组（两个）括号所需要的括号数，则不难得出优化函数

$$f(a,b)=\begin{cases} f(a+1,b-1), & \text{IsMatching(a,b)} = \text{true} \\ \min(f(a+1,b), f(a,b-1))+1 & \text{IsMatching(a,b)} \neq \text{true} \end{cases}$$

  - 若用 L(0,n-1)表示有 n 个括号的括号字符串合法化需要的括号数，则可得递推方程为：

$$L(0,n-1)=\begin{cases} L(0,n-1), & \text{IsMatching(0,n-1)} = \text{true} \\ \min(L(1,n-1), L(0,n-2))+1 & \text{IsMatching(0,n-1)} \neq \text{true} \end{cases}$$

  - 易得边界条件为字串只剩一个字符时需要添加的括号数，即 L(a,a) = 1.
- 根据以上划分方式以及递推方程，可得动态规划程序如下：

(2) 源代码

MinBracketsProblem::MinBracketsProblem(string str) //构造函数

```
MinBracketsProblem::MinBracketsProblem(string str){
    this->str = str;
    this->len = str.length();
    sol = 0;
    //初始化动态规划矩阵
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
            if (i == j) dp[j][j] = 1;  //只剩一个字符，dp 值为 1
            else dp[i][j] = 0;         //初始化 dp 值为 0
        }
    }
}
```

bool MinBracketsProblem::IsMatching(char left, char right) //判断是否配对辅助函数

```
bool MinBracketsProblem::IsMatching(char left, char right) {
    return (left == '(' && right == ')') ||
           (left == '[' && right == ']') ||
           (left == '{' && right == '}');
}
```

void MinBracketsProblem::Solve() //动态规划求解

```
void MinBracketsProblem::Solve(){
    if (len == 0) return;

    for (int length = 2; length <= len; length++) {      // 子串长度(从 2 开始增长)
        for (int i = 0; i <= len - length; i++) {        // 子串起点
            int j = i + length - 1;                      // 子串终点
            if (IsMatching(str[i], str[j])) {
                dp[i][j] = dp[i+1][j-1];
            } else {
                dp[i][j] = min(dp[i+1][j], dp[i][j-1]) + 1;
            }
        }
    }
    sol = dp[0][len-1];
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

需要注意的是，由于本题最终要求的是 L(0,n-1)，且根据优化函数，可得 dp 矩阵的构造是从中心向右上角进行的。则应当把主对角线作为边界，设定好 dp 值（1）并逐步向右上求解动态规划举证，否则可能出现错误。

实验题第 3 题

(1) 设计思路

已知一辆汽车加满油之后可以行驶 d 公里，而旅途中有 n 个加油站。本题要求**设计一个算法求出应在哪些加油站停靠加油，可以使得加油次数最少**。

- 贪心思想要从局部最优的角度出发，构造出整体最优解。设计步骤一般为分解、求解、合并三步。
- 本题要求可以抵达终点的最少加油次数，根据贪心思想，设计出贪心策略如下：
  - 只要还可以走完下一段路程，就不加油，超出上限的加油不能累积，只能溢出，意味着浪费。当剩余里程走不完下一段路程，在此加油站加油。如果满油状态还走不完，则返回 false。
- 可以定义一个变量 s，表示加完油后走完的里程。则 s+d[i+1]则表示到下个加油站预期要走的里程。将之与 m 比较，则可判断出发后，在下一段路上累计里程是否会超出限度，即油箱耗尽。如果会超出，则在此处加油。
- 综上，结合模拟和贪心的思想，可以得到代码如下：

(2) 源代码

RefuelProblem::RefuelProblem(int m, int n, int* d) //构造函数

```
RefuelProblem::RefuelProblem(int m, int n, int* d){
    this->m = m;    //满油可以行驶的里程
    this->n = n;    //加油站的数目
    this->d = new int[n];
    memcpy(this->d, d, n * sizeof(int));
    //初始化答案数组
    for(int i = 0; i < 100; i++){
        ans[i] = -1;
    }
}
```

bool RefuelProblem::Solve() //贪心思想求解

```
bool RefuelProblem::Solve(){
    //贪心法求解
    cnt = 0;
    int s = d[1];
    for(int i = 0; i < n; i++){
        if(m < d[i]) return false; //贪心小车偶遇超长公路，加油站偏僻远如怪物，拼尽全力未能抵达

        if(s + d[i+1] <= m){  //如果下一次的预测总里程还没有超过最大可行里程
            s += d[i];        //消耗里程累加
        }
        else{
            s = 0 + d[i+1];    //加油，重新计算里程
            ans[cnt] = i;      //加油站信息计入
            cnt ++;          //加油计数+1
        }
    }
    return true;
}
```

```
int *RefuelProblem::GetSolution(){
    return ans;
}

int RefuelProblem::GetMinValue(){
    return cnt;
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

需要注意的是，如果正好可以走完下一段路，同样不需要加油，到下一站再加油。所以判断条件为 $(s+d[i+1] \leq m)$ 而不是 $(s+d[i+1]<m)$.

实验题第 4 题

(1) 设计思路

给定一个 n 位的正整数 d，删除其中任意的 k（k≤n）个数字之后，剩下的数字按照原次序排列构成一个新的正整数。本题要求**于给定的 n 位正整数 d 和正整数 k，设计一个算法，使得删除之后剩下的数字构成的整数最小**。

- 贪心思想要从局部最优的角度出发，构造出整体最优解。设计步骤一般为分解、求解、合并三步。
- 通过数学归纳法可得贪心策略如下：
  - 在不删除到 0 在最前端的情况下，不会造成数位的变化。因此，只要从最的数开始删，直到删完 k 个数，剩下的数即是最小；
  - 在删除完 0 前面数的情况下，数位会发生变化。由于降低一个数位相当于多减少了一个数，就算删除一个 9，效果也不如降低一个数位好。因此，如果 k 个数的限额足够删除掉 0 之前的所有数，则因先删除 0 之前的所有数，使数据可以进行额外的减少。
- 根据以上贪心思想，设计算法如下，用-1 来表示删除的数，与前驱 0 区分开，方便记录信息。最后处理完后，统一把前驱 0 置为-1，防止输出。

(2) 源代码

int DelDigit(int v, int k)　　//贪心算法求解

```
int DelDigit(int v, int k){
  int res = 0;
  int i,j = 0;

  //把 value 转换为数组
  int t = v;
  int n = 0;
  while(t != 0){
    t /= 10;
    n ++;
  }
  int a[10010] = {0};
  for(i = n, j = 0; i > 0; i--, j++){
    a[j] = (v/(static_cast<int>(pow(10,i-1))))%10;
  }

  //贪心
  bool iszero = false;
  int zcnt=0,dcnt=0;

  for(i = 0; i < n; i++){
    if(a[i] == 0){
      iszero = true;
      break;
    }
  }
  if(iszero){
    i = 0;
    while(a[i] != 0){
      zcnt++;
      i++;
    }
```

```
    if (zcnt <= k){
      for(i = 0; i < zcnt; i++){
        a[i] = -1;
        dcnt++;
      }
    }
  }
  for(; dcnt<k; dcnt++){
    a[findMax(a,n)] = -1;
  }

  //后处理，防止0在开头输出
  i = 0;
  while(a[i] == 0 || a[i] == -1){
    if(a[i] == 0) a[i] = -1;
    i++;
  }

  //输出结果
  for(i = 0; i < n; i++){
    if(a[i] != -1){
      res = res * 10 + a[i];
    }
  }

  return res;
}
```

(3)说明

上述代码可以应用测试程序进行测试，但无法得到所有正确结果，有一组数据错误。

判断原因是“优先删除最大值”的贪心策略出错。除此之外，还应当考虑优先删除位数大的较大值。但由于提交时间迫近，已来不及再更改。初步判断，可以利用栈的数据结构重新设计贪心策略，以达整体最优。