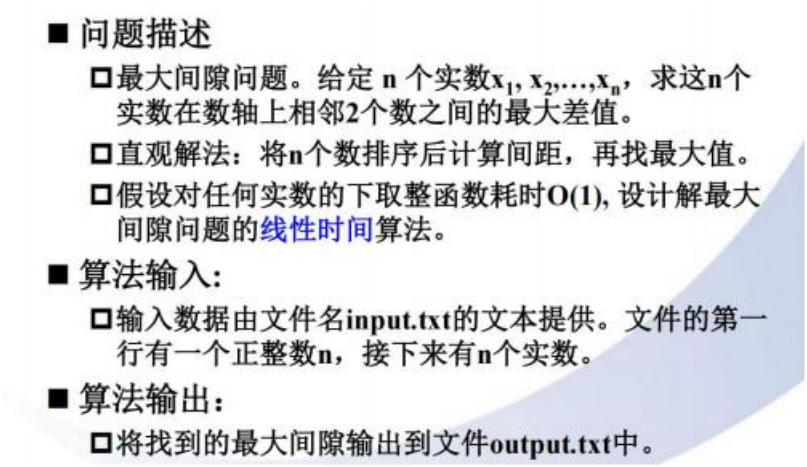
算法实现 1：



设计思路

- 考虑到派续后求间距最大值依然是最为直观的方式，同时又求设计一个$O(N)$复杂度的算法，因而自然而然地采用桶排序
- 考虑到如下规律：

$$L_{max} \geq \frac{V_{max} - V_{min}}{n-1}$$

其中L_{max}为最大间隔，V_{max}，V_{min}为读入数据的最大最小值，n为读入数据的数量，则若每个桶的大小控制在$\frac{V_{max}-V_{min}}{n-1}$，即可保证产生最大间距的两个点，一定在不同的桶中

- 因此对于每个桶，我们只需要知道桶内的最大最小值（第一个桶只需知道最大值，最后一个桶只需要知道最小值），然后比较相邻非空桶的最小值与最大值之差即可

复杂度分析

- 桶的数量为$\frac{数据最大差值}{桶大小}$，其中桶大小已经确定（见上），故数量为$n-1$
- 由于找一个数组中的最大最小值都是线性复杂度，故对每个桶内的最大最小值查询与桶中数据数有关，记桶编号为i，则其数据量记为t_i，有$\sum_{i=1}^{n-1} t_i = n$则有总计算复杂度为

$$\sum_{i=1}^{n-1} O(t_i) = O(n)$$

- 若在维护桶时只维护最大最小值而非链表，则遍历最大最小值的复杂度可以忽略，只需关注插入复杂度和后续对桶间间距比大小的复杂度，后两者显然为$O(n)$复杂度

```
import math

f = open("input.txt","r")
n = int(f.readline())
arr = f.readlines() #读取文件中的数据
f.close()

nums = []
for num in arr:
    nums.append(float(num.strip())) #将数据转存为数组

max_val = max(nums)
min_val = min(nums)
size = (max_val - min_val)/(n - 1) #求桶间距

bucket = [[None,None] for _ in range(n+1)]
for num in nums: #把每个数据分别放在对应的桶里
    b = bucket[math.floor((num - min_val) // size)]
    #求出每个桶里最大最小值
    b[0] = min(b[0], num) if b[0] else num
    b[1] = max(b[1], num) if b[1] else num
bucket = [b for b in bucket if b[0] is not None]
#计算最大间距
ans = max(bucket[i][0] - bucket[i-1][1] for i in range(1,len(bucket)))
print(ans)

#输出到文件中
f1 = open("output.txt","w")
f1.write(str(ans))
f1.close()
```

代码实现：

```cpp
#include <iostream>
#include <vector>
#include <string>
#include<fstream>
#include<sstream>
#include<windows.h>
#include<float.h>

using namespace std;

//1.txt  1 2 4 3 5 6 10 5 8.5
/*********readArray函数将txt文本中的数组读入array数组中*************/
void readArray(string abPath,vector<float>& array)//绝对路径
{
    SetConsoleOutputCP(65001);
    ifstream inTxt(abPath);
    string line;

    if(inTxt) // 有该文件
    {
        while (getline (inTxt, line)) // line中不包括每行的换行符
        {
            cout << line << endl;
        }
        istringstream in(line);
        string t;
        while(in>>t){
            array.push_back(stof(t));
        }
    }
    else // 没有该文件
    {
        cout <<"no such file" << endl;
    }
}

/*************writeRes函数将结果写入txt文件中*************************/
```

```cpp
void writeRes(float res){
    fstream f;

 f.open("D:\\code\\workspace\\c++_workspace\\algorithm\\lessons\\homework2\\result.txt",ios::out);
    f<<res;
    f.close();
}

/*******************核心功能函数maxGap计算最大间隙*************************************/
float maxGap(vector<float> array)
{
    vector<pair<float,float>>barrel;//桶的数据结构，barrel[].first是区间最小，barrel[].second是区间最大
    const int N = array.size()+1;//桶的数量
    barrel.resize(N, make_pair(FLT_MAX,FLT_MIN));//初始化一下桶

    //找出最大最小值
    float minElement = FLT_MAX;
    float maxElement = FLT_MIN;
    for(auto i:array)
    {
        if(i >= maxElement) maxElement = i;
        if(i <= minElement) minElement = i;
    }

    //桶的宽度
    float barrelApart = (maxElement-minElement)/N;

    //将每个数字放入对应的桶
    vector<int>noEmpty;
    for(auto i:array)
    {
        //找到最大最小，之后将每个值放入对应的桶里面,数字a应该所在的桶应该在a-min/barrelApart的向下取整里面
        int num = (int)(i-minElement)/barrelApart;//应该放在的桶的序号
        //处理边界问题
        if(i == maxElement) num = N-1;
        noEmpty.push_back(num);//记录一下有数据的桶号

        //只储存最大和最小值
        if(i >= barrel[num].second ) barrel[num].second = i;
        if(i <= barrel[num].first ) barrel[num].first = i;
    }


    //计算桶和桶之间的间距寻找最大间隙
    float minValue = FLT_MIN;
    float maxValue = FLT_MAX;
    float gap = 0;
    for(auto i : barrel)
    {
        if(i.first==FLT_MAX&&i.second==FLT_MIN)continue;
        if(i.first==FLT_MAX&&i.second!=FLT_MIN)i.first=i.second;
```

```
        if(i.first!=FLT_MAX&&i.second==FLT_MIN)i.second=i.first;
        //处理一下边界问题
        minValue = i.first;
        if(minValue-maxValue >= gap) gap = minValue-maxValue;
        maxValue = i.second;
    }
    return gap;
}
int main() {
    vector<float>myArray;


 readArray("D:\\code\\workspace\\c++_workspace\\algorithm\\lessons\\homework2\\input.txt",myArray);
    cout<< maxGap(myArray);

    writeRes(maxGap(myArray));
    return 0;
}

```

算法实现 2：

■ 棋盘覆盖问题

□在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。如图1所示，蓝色的为特殊方格：

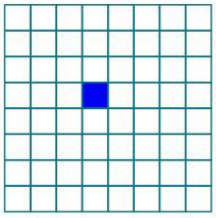


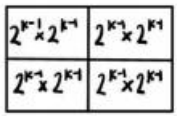
图1 特殊棋盘，蓝色的为特殊方格

图2 四种L形骨牌

- 思路

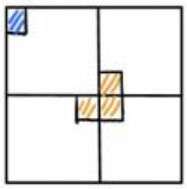
  这里的分治是将一个大棋盘分为多个小棋盘，使划分后的子棋盘的大小相同，并且每个子棋盘均包含一个特殊方格，从而将原问题分解为规模较小的棋盘覆盖问题。

  k>0时，可将$2^k \times 2^k$的棋盘划分为4个$2^{(k-1)} \times 2^{(k-1)}$的子棋盘。这样划分后，由于原棋盘只有一个特殊方格，所以，这4个子棋盘中只有一个子棋盘包含该特殊方格，其余3个子棋盘中没有特殊方格。



为了将这3个没有特殊方格的子棋盘转化为特殊棋盘，以便采用递归方法求解，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，从而将原问题转化为4个较小规模的棋盘覆盖问题。

递归地使用这种划分策略，直至将棋盘分割为1×1的子棋盘。



```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BOARD_SIZE 1024


//tile为骨牌序号，board为棋盘，二者为全局变量
int tile = 1;

int board[MAX_BOARD_SIZE][MAX_BOARD_SIZE];

void chessboard(int tr, int tc, int dr, int dc, int size) {
  //如果size为1，则意味着递归完成，直接退出
  if (size == 1) {
    return;
  }

  int t = tile++;
  int s = size / 2;
  //对于特殊点在分区内则直接进行递归，反之，若为左上分区则在右下处覆盖骨牌，若为右上分
  //在左下处覆盖骨牌，以此类推；覆盖将覆盖处记为特殊点进行下一轮递归
  if (dr < tr + s && dc < tc + s) {
    chessboard(tr, tc, dr, dc, s);
  } else {
    board[tr + s - 1][tc + s - 1] = t;
    chessboard(tr, tc, tr + s - 1, tc + s - 1, s);
  }
```

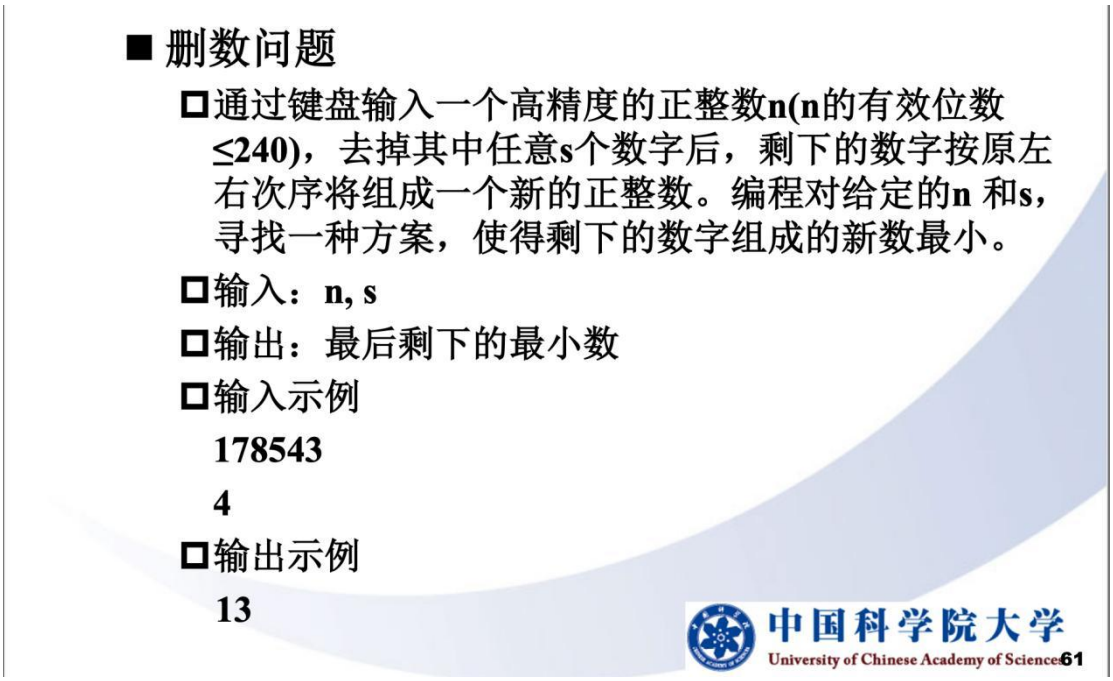
```c
  if (dr < tr + s && dc >= tc + s) {
    chessboard(tr, tc + s, dr, dc, s);
  } else {
    board[tr + s - 1][tc + s] = t;
    chessboard(tr, tc + s, tr + s - 1, tc + s, s);
  }

  if (dr >= tr + s && dc < tc + s) {
    chessboard(tr + s, tc, dr, dc, s);
  } else {
    board[tr + s][tc + s - 1] = t;
    chessboard(tr + s, tc, tr + s, tc + s - 1, s);
  }

  if (dr >= tr + s && dc >= tc + s) {
    chessboard(tr + s, tc + s, dr, dc, s);
  } else {
    board[tr + s][tc + s] = t;
    chessboard(tr + s, tc + s, tr + s, tc + s, s);
  }
}

int main() {
  int n;
  printf("Input the size of the board. \n");
  scanf("%d", &n);

  int boardSize = (int)pow(2, n);

  for (size_t i = 0; i < boardSize; i++) {
    for (size_t j = 0; j < boardSize; j++) {
      board[i][j] = 0;
    }
  }

  int uniqRow, uniqCol;
  printf("Input the unique row of the board. \n");
  scanf("%d", &uniqRow);
  printf("Input the unique col of the board. \n");
  scanf("%d", &uniqCol);

  chessboard(0, 0, uniqRow, uniqCol, boardSize);
```

```c
  for (size_t i = 0; i < boardSize; i++) {
    for (size_t j = 0; j < boardSize; j++) {
      printf("%2d\t", board[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

```
def chess(tr,tc,pr,pc,size):

    global mark
    global table
    if size==1:
        return     #递归终止条件
    mark+=1          #表示直角骨牌号
    count=mark
    half=size//2      #当size不等于1时，棋盘格规模减半，变为4个
    #小棋盘格进行递归操作
    #左上角
    if (pr<tr+half) and (pc<tc+half):
        chess(tr,tc,pr,pc,half)
    else:
        table[tr+half-1][tc+half-1]=count
        chess(tr,tc,tr+half-1,tc+half-1,half)
    #将[tr+half-1,tc+half-1]作为小规模棋盘格的特殊点，进行递归

    #右上角
    if (pr<tr+half) and (pc>=tc+half):
        chess(tr,tc+half,pr,pc,half)
    else:
        table[tr+half-1][tc+half]=count
        chess(tr,tc+half,tr+half-1,tc+half,half)
    #将[tr+half-1,tc+half]作为小规模棋盘格的特殊点，进行递归

    #左下角
    if (pr>=tr+half) and (pc<tc+half):
        chess(tr+half,tc,pr,pc,half)
    else:
        table[tr+half][tc+half-1]=count
        chess(tr+half,tc,tr+half,tc+half-1,half)
    #将[tr+half,tc+half-1]作为小规模棋盘格的特殊点，进行递归
```

```
def show(table):
    n=len(table)
    for i in range(n):
        for j in range(n):
            print(table[i][j],end='    ')
        print('')

if __name__=='__main__':
    mark = 0
    k = 8
    table=[[-1 for x in range(k)] for y in range(k)] #-1代表特殊格子
    chess(0,0,2,2,k)
    print('\n数字即为填充顺序')
    show(table)
```

算法实现 3：



算法思路

本算法的核心思路在于每次都最优地顺序删除局部最大值，起始从输入的首位开始扫描，若出现本位大于下一位的情况，则将本位删除，其余位左移；在完成一次数字位移后，若扫描位非首位，则将其前一一位（即移动到本次位移的第一位），不断循环，直到删除足够位数。

最优解的证明

本题的证明可以使用反证法，不妨假设本算法得到的解并非最优解，也即必然存在一次删除，使得被删除的数是应当保留的，不妨记其为 x，则不妨假设此时的数组为 ...axb...，且此时至少需要删除一个数字（此假设不失一般性），此时必然存在 x>b，若最终 b 同样未被删除，则说明 b 后必然有至少一位，显然将 b 后的数字中选一删除更小，即非最优解；若最终 b 被删除，则显然保留 b 而删除 x 更小，即非最优解，综上，本算法能够取得最优解。

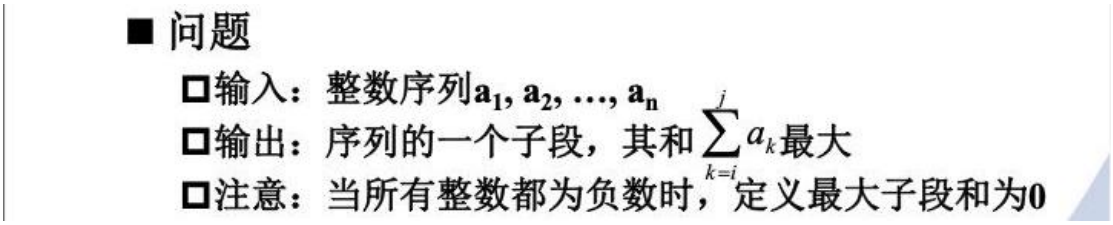
具体程序源码如下：

```
#include<iostream>
#include<string>
using namespace std;
int main() {
    string number;
    int k;
    cin >> number >> k;

    //有一个小问题。当要删除的数字个数 k 大于等于 n 的长度时，使用了
number.erase()语句直接将原始数字全部删除，
    // 但这会导致程序在接下来的操作中出错。
    //if (k >= number.size())
    //  number.erase();

    if (k >= number.size()) // 当要删除的数字个数 k 大于等于 n 的长度时，直接
输出 0
    {
        cout << 0 << endl;
        return 0;
    }
    else
        while (k>0) {
            int i;
            //for 循环中的条件判断语句是判断 number 数组中相邻两个元素的大小关
系，
            //如果前一个元素小于等于后一个元素，则继续循环，否则跳出循环。
            //循环结束后，i 的值为最后一个满足条件的元素的下标加 1。
            for (i = 0; (i < number.size() - 1) && (number[i] <=
number[i + 1]); ++i);
            number.erase(i, 1);
            k--;
        }
    //删去前导 0
    while (number.size() > 1 && number[0] == '0')
        number.erase(0, 1);
    cout << number << endl;
    return 0;
}
```

算法思路：为了尽可能逼近目标，选取的贪心策略为：每一步总选择一个使剩下的数最小的数删去。即从第一个数字往后搜索，如果数字递增，则删去最后一个数，否则删除第一个递减区间的第一个数。然后再回到串首，按上述规则再删除下一个数字，来回 s 次即可得到目标数字。存在最后剩下的数为 00X 的格式，对此进行处理，如果剩下的数中第一个数是 0，则移除 0；如果剩下的数全为 0，则 0 全被移除，需补一个 0。

```
print("请输入原数字")
n=str(input())
a=list(map(int,str(n)))
print("请输入您要删除的个数")
n=int(input())
def Delete(a,n):

    for i in range(0,n):
        j = 0
        while j<len(a)-1 and a[j]<=a[j+1]:
            j=j+1
        a.remove(a[j])
        for i in range(len(a)):
            if a[0]==0:
                a.remove(a[0])
        if len(a)==0:
            a.append(0)

Delete(a,n)
str2 = ''.join(str(i) for i in a)
print(str2)
```

算法实现 4：



求解思路：

当前状态只需要考虑，加上当前这个数，子序列和是否变得更大。

假设第i个数的最大子序列的和为$m[i]$，第i个数为$a[i]$，则状态转移方程为$m[i] = max(m[i-1] + a[i], a[i])$。

即，如果$a[i] < 0$，则$m[i] = a[i]$；如果$a[i] \geq 0$，则$m[i] = m[i-1] + a[i]$。

同样，如果$a[i] < 0$，则子序列起始坐标为i，终止坐标为i；如果$a[i] \geq 0$，则起始坐标为 ($m[i-1]$的起始坐标)，终止坐标为i。

当所有整数为负数时（求解的最大子序列和为负数），返回整个序列，且返回0。

```
#include <iostream>
#include <vector>
#include <sstream>

class Seq
{
public:
    Seq(){}
    Seq(int start_index, int end_index, int max_sum) :
        start_index(start_index),
        end_index(end_index),
        max_sum(max_sum){}
    int start_index;
    int end_index;
    int max_sum;
};

// e.g., -2 1 -3 4 -1 2 1 -5 4
// 思路：当前状态只需要考虑加上当前这个数子序列和是否变得更大
// 假设第i个数的最大子序列的和为m[i] 第i个数为a[i] 则状态转移方程为m[i]=max(m[i-1]+a[i],
a[i])
// 即，如果a[i]小于0 则 m[i] = a[i]； 如果a[i]大于等于0 则 m[i] = m[i-1]+a[i]
// 同样，如果a[i]小于0，子序列起始和终止坐标为i和i； 如果a[i]大于等于0，子序列起始和终止坐标为
(m[i-1]的起始坐标)和i
// 代码中 sequence_tmp对应m  sequence对应a
int get_maxsum_subsequences(std::vector<int> &sequence, std::vector<int>
&result_subsequence)
{
    int length = sequence.size();
    Seq sequence_tmp[length];

    int max_sum = 0;
    int max_sum_index = 0;
    for(int i=0;i<length;i++)
```

```
    {
        if(i==0)
        {
            sequence_tmp[i] = Seq(0,0,sequence[i]);
            max_sum = sequence[i];
        }
        else
        {
            if(sequence_tmp[i-1].max_sum < 0)
            {
                sequence_tmp[i] = Seq(i,i,sequence[i]);
            }
            else
            {
                sequence_tmp[i] = Seq(
                    sequence_tmp[i-1].start_index,
                    i,
                    sequence_tmp[i-1].max_sum + sequence[i]);
            }

            // 记录最大的和以及对应下标
            if (max_sum < sequence_tmp[i].max_sum)
            {
                max_sum = sequence_tmp[i].max_sum;
                max_sum_index = i;
            }
        }
    }

    // 处理结果
    if (max_sum <= 0)
    {
        result_subsequence.assign(sequence.begin(), sequence.end());
        return 0;
    }
    else
    {
        result_subsequence.assign(
            sequence.begin() + sequence_tmp[max_sum_index].start_index,
            sequence.begin() + sequence_tmp[max_sum_index].end_index + 1);
        return max_sum;
    }

}

void get_linedata_to_sequence(std::vector<int> &sequence)
{
    std::cout << "Input: ";
    std::string input_line;
    std::getline(std::cin, input_line);
    std::stringstream ss(input_line);
    int num;
    while (ss >> num)
        sequence.push_back(num);
}

void print_sequence(std::vector<int> &sequence)
{
```

```
    int length = sequence.size();
    std::cout << "[";
    for(int i=0;i<length;i++)
    {
        if(i!=length-1)
            std::cout << sequence[i] << ", ";
        else
            std::cout << sequence[i] << "]";
    }

}

int main(int argc, char** argv)
{
    // 从输入获得序列
    std::vector<int> seq;
    get_linedata_to_sequence(seq);

    // 求解最大和子序列
    std::vector<int> sub_seq;
    int max_sum = get_maxsum_subsequences(seq, sub_seq);

    // 打印
    std::cout << "Result subsequence: ";
    print_sequence(sub_seq);
    std::cout << std::endl;
    std::cout << "Max sum:";
    std::cout << max_sum;
    std::cout << " " << std::endl;

    return 0;
}
```

定义 dp 数组，记录当前位置子段最大和，状态转移方程 dp[i] = max(dp[i - 1] + lst[i], lst[i])，也就是如果前面 dp 加上当前输入这个数比当前单独这个数小的话，很明显，前面的数据我们不需要了，那么就是需要当前这个数，并且以他开始，所以我们此时的 dp 就是输入的数，后续的一直这样比较。计算出所有的 dp 后，找出最大值即为最大字段和，同时也能得到最大字段的结束位置，通过结束位置和最大字段和前推寻找子段起始位置。

```
print("请输入整数序列：")
## dp法：时间复杂度O(n)
lst = [int(i) for i in input().split(",")]
n = len(lst)

# 初始化dp数组，这里多给了10个空间也无伤大雅的
dp = [0] * (n + 10)
# 将dp[0]设为lst[0]，也就是指在lst[0]处，其问题的最优值很简单，就是lst[0]本身，这也是递推的第一个状态
dp[0] = lst[0]
# 注意是从1开始遍历
for i in range(1, n):
    dp[i] = max(dp[i - 1] + lst[i], lst[i])
# print(dp[0: n])
# dp数组全部计算完毕后，序列的最大连续子段和就是dp数组中的最大值
maximum = max(dp[0: n])

# 下面是求解最大连续子段和的最优解（最大连续子段的起始、终止下标）
# 求出maximum在dp中的下标，根据dp[]数组的定义可以推出最优解
# end就是指最优解的终点，start指最优解的起点，一开始也设为end
end = dp.index(maximum)
start = end
# 最大值小于0，说明输入序列全负，最大序列和为0
if maximum < 0:
    print("所有整数都为负数，最大子段和为0")
else:
# 设置一个temp变量准备做累加进行判断，值设为lst[end]
    temp = lst[end]
# 如果不是lst[end]本身，那便从end - 1开始向前遍历lst，并把遍历到的值加进temp
    for i in range(end - 1, -1, -1):
        temp += lst[i]
# 如果temp == maximum 表示找到了解。
# 但注意如果遍历没有结束，这个解可能不是最优解！
# 因为在还没有遍历到的那一段中有些值加起来可能为0，加上去不会影响最优值，
# 但是问题是要找“最大连续”，所以这个解不一定是最优解！
# 因而还要继续遍历。所以我把print()放在了循环外面。
        if temp == maximum:
            start = i
    print("最大子段为：", lst[start:end+1])
    print("最大子段和为：", maximum)
```

算法实现 5：

问题描述-算24点

- 几十年前全世界就流行一种数字游戏，至今仍有人乐此不疲．在中国我们把这种游戏称为“算24点”。您作为游戏者将得到4个1~9之间的自然数作为操作数，而您的任务是对这4个操作数进行适当的算术运算，要求运算结果等于24。
- 您可以使用的运算只有：+，-，*，/，您还可以使用（）来改变运算顺序。注意：所有的中间结果须是整数，所以一些除法运算是不允许的（例如，(2*2)/4是合法的，2*(2/4)是不合法的）。下面我们给出一个游戏的具体例子：
- 若给出的4个操作数是：1、2、3、7，则一种可能的解答是1+2+3*7=24。

中国科学院大学
University of Chinese Academy of Sciences 36

- 输入
  - 只有一行，四个1到9之间的自然数。
- 输出
  - 如果有解的话，只要输出一个解，输出的是三行数据，分别表示运算的步骤。其中第一行是输入的两个数和一个运算符和运算后的结果，第二行是第一行的结果和一个输入的数据、运算符、运算后的结果；第三行是第二行的结果和输入的一个数、运算符和“=24”。如果两个操作数有大小的话则先输出大的。
  - 如果没有解则输出“No answer!”

中国科学院大学
University of Chinese Academy of Sciences 37

- 输入样例
  - 1 2 3 7
- 输出样例
  - 2+1=3
  - 7*3=21
  - 21+3=24

求解思路：

采用回溯的思路解决问题，使用递归的方式实现。

对于求解的问题，有四个操作数。

通过循环选择两个不同的操作数x_1和x_2，其余两个记为x_3和x_4。

通过循环将x_1和x_2进行加减乘除之一的运算获得结果x。

将三个数x, x_3, x_4作为子问题的操作数求解。

通过这种方式，成功将大问题转换为小问题。

通过递归，使得问题规模不断变小，递归终止条件是：当只有一个操作数且该数为24。

代码：

```
#include <iostream>
#include <vector>
#include <sstream>

std::string num2string(int num)
{
    std::string num_str;
    if(num < 0)
        num_str = "(" + std::to_string(num) + ")";
    else
        num_str = std::to_string(num);
    return num_str;
}

bool solve_equation_to_24(std::vector<int> &array, std::vector<std::string>
&op_strings)
{

    if(array.size()==1)
    {
        // 因为中间结果都是整数 直接判断即可
        if(array[0] == 24)
            return true;
        return false;
    }

    for(int i=0;i<array.size();i++)
    {
        for(int j=0;j<array.size();j++)
```

```
        {
            // array[i]和array[j]为两个操作数
            if(i==j)
                continue;

            // 前一个操作数要比后一个大
            if(array[i] < array[j])
                continue;

            // 将其余的数放入数组
            std::vector<int> tmp_array;
            for(int k=0;k<array.size();k++)
            {
                if(i==k || j==k)
                    continue;
                tmp_array.push_back(array[k]);
            }

            // 缩小问题规模，对两个操作数执行运算，然后分别与其他数进行递归
            enum OP{ADD, SUB, MULTI, DIV};
            char op_str[4] = {'+', '-', '*', '/'};
            for (int op = 0; op < 4; op++)
            {
                int result;
                bool success = true;
                switch (op)
                {
                    case ADD:
                        result = array[i] + array[j];
                        break;
                    case SUB:
                        result = array[i] - array[j];
                        break;
                    case MULTI:
                        result = array[i] * array[j];
                        break;
                    case DIV:
                        // 分母不能为0
                        // 中间结果为整数 要能整除
                        if(array[j] == 0 || array[i] % array[j] != 0)
                            success = false;
                        else
                            result = array[i] / array[j];
                        break;
                }

                if(!success)
                    continue;

                tmp_array.push_back(result);
                if(solve_equation_to_24(tmp_array, op_strings))
                {
                    std::stringstream ss;
                    ss << array[i] << op_str[op] << array[j] << "=" << result;
                    op_strings.push_back(ss.str());

                    return true;
                }
```

```
                tmp_array.pop_back();


            }
        }
    }

    return false;
}

void get_linedata_to_4nums(std::vector<int> &nums)
{
    while(1)
    {
        std::cout << "Input: ";
        std::string input_line;
        std::getline(std::cin, input_line);
        std::stringstream ss(input_line);

        int num;
        bool success = true;
        for(int i=0;i<4;i++)
        {
            ss >> num;
            // 输入为1-9
            if(num < 1 || num > 9)
            {
                success = false;
                nums.clear();
                break;
            }
            nums.push_back(num);
        }

        if(success)
            break;
        else
            continue;
    }
}

int main(int argc, char** argv)
{
    // e.g., 1 2 3 7
    std::vector<int> nums;
    get_linedata_to_4nums(nums);

    std::vector<std::string> ops;
    if(!solve_equation_to_24(nums, ops))
    {
        std::cout << "No answer!" << std::endl;
    }
    else
    {
        // 需要反向打印
        for (auto it = ops.rbegin(); it != ops.rend(); ++it)
            std::cout << *it << std::endl;
    }


    return 0;
}
```

总共 4 个数，每次算两个数，算完两个得出一个结果，然后和剩下没有计算的数合成新数组。直到数组中只剩一个数时，判断这个数是否等于 24，如果等于 24，返回 true，否则返回 false。设置变量 answers 用于存储成功时每一步的计算过程，如果成功，answers 的长度大于 0，逆序输出即为运算顺序，如果不成功，answers 的长度为 0，则输出“No answer!”

```
def jP24(nums):
    if len(nums)==1:
        return abs(24-nums[0])<=10**(-10)
    # 每次把计算结果 和之前nums数组中其他元素组成的列表concate起来 作为新的nums 向下递归
    for i in range(len(nums)-1):
        for j in range(i+1,len(nums)):
            if jP24([nums[i]+nums[j]]+nums[0:i]+nums[i+1:j]+nums[j+1:]):
                answers.append(str(nums[i])+"+"+str(nums[j])+"="+str(nums[i]+nums[j]))
                return True
            if jP24([nums[i]*nums[j]]+nums[0:i]+nums[i+1:j]+nums[j+1:]):
                answers.append(str(nums[i])+"*"+str(nums[j])+"="+str(nums[i]*nums[j]))
                return True
            if jP24([nums[i]-nums[j]]+nums[0:i]+nums[i+1:j]+nums[j+1:]):
                answers.append(str(nums[i])+"-"+str(nums[j])+"="+str(nums[i]-nums[j]))
                return True
            if jP24([nums[j]-nums[i]]+nums[0:i]+nums[i+1:j]+nums[j+1:]):
                answers.append(str(nums[j])+"-"+str(nums[i])+"="+str(nums[j]-nums[i]))
                return True
            if nums[j]!=0 and jP24([nums[i]/nums[j]]+nums[0:i]+nums[i+1:j]+nums[j+1:]):
                answers.append(str(nums[i])+"/"+str(nums[j])+"="+str(nums[i]/nums[j]))
                return True
            if nums[i]!=0 and jP24([nums[j]/nums[i]]+nums[0:i]+nums[i+1:j]+nums[j+1:]):
                answers.append(str(nums[j])+"/"+str(nums[i])+"="+str(nums[j]/nums[i]))
                return True
    # 走到这一步 说明之前都不行
    return False

print("请输入4个1-9的整数：")
nums = [int(i) for i in input().split()]
answers = []
jP24(nums)
if(len(answers)==0):
    print("No answer!")
else:
    for i in range(len(answers)-1,-1,-1):
        print(answers[i])
```