4-1　会场安排问题。

问题描述：假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场。设计一个有效的贪心算法进行安排。（这个问题实际上是著名的图着色问题。若将每个活动作为图的一个顶点，不相容活动间用边相连。使相邻顶点着有不同颜色的最小着色数，相当于要找的最小会场数。）

算法设计：对于给定的 k 个待安排的活动，计算使用最少会场的时间表。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 k，表示有 k 个待安排的活动。接下来的 k 行中，每行有 2 个正整数，分别表示 k 个待安排的活动的开始时间和结束时间。时间以 0 点开始的分钟计。

结果输出：将计算的最少会场数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 1 23 12 28 25 35 27 80 36 50	3

答：本题可采用贪心算法求解。核心思路是按照活动的开始时间对活动进行排序，然后遍历活动，对于每个活动，检查是否能安排在已有的会场中（即当前活动的开始时间不早于某个会场中最后一个活动的结束时间），如果不能则新开一个会场。这样能保证在满足活动时间要求的前提下，使用最少的会场。

代码实现如下：

```
int greedy(vector<point> x) {
    int sum = 0, curr = 0, n = x.size();
    sort(x.begin(), x.end());
    for(int i = 0; i < n; i++) {
        if(x[i].leftend())
            curr++;
        else
            curr--;
        if((i == n - 1 || x[i] < x[i + 1]) && curr > sum)   // 处理 x[i]=x[i+1]的情况
            sum = curr;
    }
    return sum;
```

}

4-2　最优合并问题。

问题描述：给定 k 个排好序的序列 $s_1, s_2, \cdots, s_k$，用 2 路合并算法将这 k 个序列合并成一个序列。假设采用的 2 路合并算法合并 2 个长度分别为 m 和 n 的序列需要 $m+n-1$ 次比较。试设计一个算法确定合并这个序列的最优合并顺序，使所需的总比较次数最少。

为了进行比较，还需要确定合并这个序列的最差合并顺序，使所需的总比较次数最多。

算法设计：对于给定的 k 个待合并序列，计算最多比较次数和最少比较次数合并方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 k，表示有 k 个待合并序列。接下来的 1 行中，有 k 个正整数，表示 k 个待合并序列的长度。

结果输出：将计算的最多比较次数和最少比较次数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4	78 52
5 12 11 2	

答：本题可利用贪心算法来求解。

最少比较次数

要使合并的总比较次数最少，每次应选择长度最小的两个序列进行合并。因为合并两个序列的比较次数是两序列长度之和减 1，优先合并短的序列，能使后续合并时每次涉及的序列长度增长相对较慢，从而使总的比较次数最少。可以使用小顶堆（优先队列）来维护序列长度，每次取出堆中最小的两个元素进行合并，将合并后的长度再放回堆中，重复此过程直到堆中只剩一个元素。

最多比较次数

要使合并的总比较次数最多，每次应选择长度最大的两个序列进行合并。优先合并长的序列，会使后续合并时每次涉及的序列长度快速增长，进而使总的比较次数最多。可以使用大顶堆（优先队列）来维护序列长度，每次取出堆中最大的两个元素进行合并，将合并后的长度再放回堆中，重复操作直到堆中只剩一个元素。

算法正确性证明:

最少比较次数

假设存在一个最优合并顺序不是每次选择最小的两个序列进行合并。设某次合并时，没有选择最小的两个序列，而是选择了其他序列进行合并。那么，原本应该先合并的最小的两个序列，在后续合并中会与更大的序列合并，导致比较次数增加。所以，每次选择最小的两个序列进行合并能得到最少的总比较次数。

最多比较次数

假设存在一个合并顺序能使总比较次数更多，而不是每次选择最大的两个序列进行合并。若不优先合并长的序列，就会使长序列在后续合并中与较短序列合并，导致整体

合并过程中比较次数减少。所以，每次选择最大的两个序列进行合并能得到最多的总比较次数。

时间复杂度分析

无论是计算最少比较次数还是最多比较次数，每次操作都涉及从堆中取出两个元素并插入一个元素，堆操作的时间复杂度为 O(logk)，一共需要进行 k−1 次合并操作，所以总的时间复杂度为 O(klogk) 。

代码实现如下：

```
// 计算最少比较次数
Function MinCompare(k, lengths):
    minHeap = 构建小顶堆(lengths) // 将 k 个序列长度放入小顶堆
    minComparisons = 0
    While minHeap.size > 1:
        a = minHeap.pop() // 取出堆中最小元素
        b = minHeap.pop() // 再取出堆中最小元素
        mergeLength = a + b
        minComparisons = minComparisons + mergeLength - 1
        minHeap.push(mergeLength) // 将合并后的长度放入堆中
    End While
    Return minComparisons
End Function

// 计算最多比较次数
Function MaxCompare(k, lengths):
    maxHeap = 构建大顶堆(lengths) // 将 k 个序列长度放入大顶堆
    maxComparisons = 0
    While maxHeap.size > 1:
        a = maxHeap.pop() // 取出堆中最大元素
        b = maxHeap.pop() // 再取出堆中最大元素
```

```
        mergeLength = a + b
        maxComparisons = maxComparisons + mergeLength - 1
        maxHeap.push(mergeLength) // 将合并后的长度放入堆中
    End While
    Return maxComparisons
End Function

// 主函数
Function Main():
    从文件 input.txt 读取 k
    从文件 input.txt 读取长度数组 lengths
    minCount = MinCompare(k, lengths)
    maxCount = MaxCompare(k, lengths)
    将 maxCount 和 minCount 写入文件 output.txt
End Function
```

4-4 磁盘文件最优存储问题。

问题描述：设磁盘上有 n 个文件 $f_1, f_2, \cdots, f_n$，每个文件占用磁盘上的 1 个磁道。这 n 个文件的检索概率分别是 $p_1, p_2, \cdots, p_n$ 且 $\sum_{i=1}^{n} p_i = 1$。磁头从当前磁道移到被检信息磁道所需的时间可用这两个磁道之间的径向距离来度量。如果文件 f_i 存放在第 i（$1 \leqslant i \leqslant n$）道上，则检索这 n 个文件的期望时间是 $\sum_{1 \leqslant i \leqslant j \leqslant n} p_i p_j d(i,j)$。式中，$d(i,j)$ 是第 i 道与第 j 道之间的径向距离 $|i-j|$。

磁盘文件的最优存储问题要求确定这 n 个文件在磁盘上的存储位置，使期望检索时间达到最小。试设计一个解此问题的算法，并分析算法的正确性与计算复杂性。

算法设计：对于给定的文件检索概率，计算磁盘文件的最优存储方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行是正整数 n，表示文件个数。第 2 行有 n 个正整数 a_i，表示文件的检索概率。实际上第 k 个文件的检索概率应为 $a_k \Big/ \sum_{i=1}^{n} a_i$。

结果输出：将计算的最小期望检索时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	0.547396
33 55 22 11 9	

答：**算法思路：**

本题可采用贪心算法来解决。贪心策略是将检索概率大的文件尽量放在相邻位置，这样可以减少磁头移动的距离，从而使期望检索时间最小。具体做法是先对文件的检索概率进行从大到小排序，然后按照排序后的顺序依次将文件存储在相邻磁道上。

算法正确性证明：

假设存在一个最优存储方案，其中有两个检索概率分别为 pi 和 pj（pi>pj）的文件，它们所在磁道的距离相对较远。如果将这两个文件调整为相邻位置，对于其他文件与这两个文件组成的文件对，由于 pi>pj，移动后会使这部分的期望检索时间减少（因为距离变小了，且概率大的文件权重高），而其他文件之间的期望检索时间不变。所以将检索概率大的文件尽量放在相邻位置能得到最优解。

时间复杂度分析

排序操作使用 qsort 函数，平均时间复杂度为 O(nlogn)，其中 n 是文件个数。

计算期望检索时间的函数 calculateExpectedTime 中有两层循环，时间复杂度为 O(n2)。

整体算法的时间复杂度主要由计算期望检索时间的部分决定，为 O(n2)。

代码实现如下：

```
Read n from input.txt

Read array a[1..n] from input.txt

sum = 0

for i from 1 to n:

    sum += a[i]

end for

创建文件数组 files，元素包含索引和概率

for i from 1 to n:

    files[i].probability = a[i] / sum

    files[i].index = i

end for

对 files 按 probability 降序排序

expected_time = 0.0

for i from 1 to n:
```

```
    for j from 1 to n:
        distance = |files[i].index - files[j].index|
        expected_time += files[i].probability * files[j].probability * distance
    end for
end for
```

将 expected_time 输出到 output.txt，保留六位小数

4-6　最优服务次序问题。

问题描述：设有 n 个顾客同时等待一项服务，顾客 i 需要的服务时间为 t_i（$1 \leqslant i \leqslant n$）。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小？平均等待时间是 n 个顾客等待服务时间的总和除以 n。

算法设计：对于给定的 n 个顾客需要的服务时间，计算最优服务次序。

数据输入：由文件 input.txt 给出输入数据。第 1 行是正整数 n，表示有 n 个顾客。接下来的 1 行中，有 n 个正整数，表示 n 个顾客需要的服务时间。

结果输出：将计算的最小平均等待时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10	532.00
56 12 1 99 1000 234 33 55 99 812	

答：贪心策略为：将服务时间短的顾客优先进行服务，这样可以使总的等待时间最小，进而使平均等待时间最小。因为先服务时间短的顾客，后面等待的顾客等待的总时长就会相对较短。

代码如下：

```
double greedy(vector<int>x){
    int n=x.size();
    sort(x.begin(), x.end());
    for(int i=1;i<n ;i++)
    x[i] += x[i-1];
    double t=0;
    for(i=0;i< n; i++)
        t += x[i];
```

t /= n;

return t;}

算法正确性证明

假设存在一个最优服务序列，其中有两个相邻顾客 i 和 i + 1，顾客 i 的服务时间为 ti，顾客 i + 1 的服务时间为 Ti+1，且 Ti>Ti+1。若交换这两个顾客的顺序，原来顾客 i + 1 的等待时间为 Ti+1（包含前面顾客的服务时间累加），顾客 i 的等待时间为 $T_{i+1}+T_{i+1}$；交换后顾客 i 的等待时间变为 Ti+1，顾客 i + 1 的等待时间变为 Ti+1+ti。总等待时间的变化为 $(T_{i+1}+T_i)+T_{i+1}-(T_{i+1}+T_i+1)-(T_{i+1}+T_i)=T_i-T_{i+1}>0$，即交换后总等待时间减少。所以将服务时间短的顾客优先服务能得到最优解。

4-8 d 森林问题。

问题描述：设 T 是一棵带权树，树的每条边带一个正权，S 是 T 的顶点集，T/S 是从树 T 中将 S 中顶点删去后得到的森林。如果 T/S 中所有树的从根到叶的路长都不超过 d，则称 T/S 是一个 d 森林。

① 设计一个算法求 T 的最小顶点集 S，使 T/S 是 d 森林（提示：从叶向根移动）。

② 分析算法的正确性和计算复杂性。

③ 设 T 中有 n 个顶点，则算法的计算时间复杂性应为 $O(n)$。

算法设计：对于给定的带权树，计算最小分离集 S。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示给定的带权树有 n 个顶点，编号为 $1, 2, \cdots, n$。编号为 1 的顶点是树根。接下来的 n 行中，第 i+1 行描述与 i 个顶点相关联的边的信息。每行的第 1 个正整数 k 表示与该顶点相关联的边数。其后 $2k$ 个数中，每 2 个数表示 1 条边。第 1 个数是与该顶点相关联的另一个顶点的编号，第 2 个数是边权值。k=0，表示相应的结点是叶结点。文件的最后一行是正整数 d，表示森林中所有树的从根到叶的路长都不超过 d。

结果输出：将计算的最小分离集 S 的顶点数输出到文件 output.txt。如果无法得到所要求的 d 森林则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
4	1
2 2 3 3 1	
1 4 2	
0	
0	
4	

答：**算法设计思路：**

采用自底向上（从叶向根）的贪心策略。从叶子节点开始，沿着树向上遍历，对于每个节点，检查其到叶子节点的最长路径长度（可以通过递归计算其子节点的最长路径长度加上边权得到）。如果某个节点的最长路径长度超过了 d，则将该节点加入分离集 S，并截断其与子树的连接（即认为从该节点以下的子树已被处理）。这样不断向上处理，最终得到满足 d 森林要求的最小顶点集 S。

① 代码实现如下：

```
for (i = 1; i <= n; i++) {//读入数据
    fscanf(input_file, "%d", &k);
    for (j = 0; j < k; j++) {
        fscanf(input_file, "%d %d", &v, &w);
        addEdge(i, v, w);
        addEdge(v, i, w);
    }
}
```

```
#define MAX_N 1000
// 边的结构体
typedef struct Edge {
    int to;
    int weight;
    struct Edge* next;
} Edge;
Edge* adj[MAX_N + 1];// 邻接表存储图
int inSet[MAX_N + 1];// 记录节点是否在分离集S中
int maxPath[MAX_N + 1];// 记录节点到叶子节点的最长路径长度
//添加边
void addEdge(int u, int v, int w) {
    Edge* newEdge = (Edge*)malloc(sizeof(Edge));
    newEdge->to = v;
    newEdge->weight = w;
    newEdge->next = adj[u];
    adj[u] = newEdge;
}
int processNode(int u, int d) {//递归计算节点到叶子节点的最长路径长度，并判断是否需要将节点加入分离集
    if (adj[u] == NULL) {
        maxPath[u] = 0;
        return 0;
    }
    int maxChildPath = 0;
    Edge* e = adj[u];
    while (e != NULL) {
        if (!inSet[e->to]) {
            int childPath = processNode(e->to, d);
            if (childPath + e->weight > maxChildPath) {
                maxChildPath = childPath + e->weight;
            }
        }
        e = e->next;
    }
    maxPath[u] = maxChildPath;
    if (maxChildPath > d) {
        inSet[u] = 1;
        return 0;
    }
    return maxChildPath;
}
```

② **算法正确性分析：**

贪心选择性质：在每一步选择中，当发现某个节点到叶子节点的最长路径长度超过 d 时，将该节点加入分离集 S。因为如果不选择这个节点，其下方子树中无论选择哪个节点都无法改变从该节点到叶子节点的路径长度超过 d 的事实，而且选择这个节点能最大程度地截断较长路径，使得剩余子树更容易满足 d 森林的要求。所以这种选择是最优的局部选择。

最优子结构性质：对于整棵树的问题，当确定了一个节点加入分离集 S 后，剩下的子树问题仍然是求满足 d 森林要求的最小顶点集 S 的子问题。也就是说，原问题的最优解包含了子问题的最优解。因此，通过不断地进行局部最优选择，最终可以得到全局最优解。

③ 计算复杂性分析：

时间复杂度：算法主要通过递归遍历树的节点来计算路径长度和确定分离集。每个节点最多被访问一次，在访问每个节点时，对其邻接边的遍历操作次数与边的数量成正比。总的时间复杂度为 $O(n)$，其中 n 是树的节点数。因为树中边的数量是 $n-1$，在遍历节点和边的过程中，操作次数与节点数和边数之和相关，所以时间复杂度为 $O(n)$。

4-9　虚拟汽车加油问题。

问题描述：一辆虚拟汽车加满油后可行驶 n km。旅途中有若干加油站。设计一个有效算法，指出应在哪些加油站停靠加油，使沿途加油次数最少。并证明算法能产生一个最优解。

算法设计：对于给定的 n 和 k 个加油站位置，计算最少加油次数。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k，表示汽车加满油后可行驶 n km，且旅途中有 k 个加油站。接下来的 1 行中有 $k+1$ 个整数，表示第 k 个加油站与第 $k-1$ 个加油站之间的距离。第 0 个加油站表示出发地，汽车已加满油。第 $k+1$ 个加油站表示目的地。

结果输出：将计算的最少加油次数输出到文件 output.txt。如果无法到达目的地，则输出“No Solution”。

输入文件示例	输出文件示例
input.txt	output.txt
7　7 1 2 3 4 5 1 6 6	4

答：采用贪心算法求解。从出发地开始，在汽车当前油量能到达的范围内，总是选择距离最远的加油站加油，这样可以保证在满足行驶需求的前提下，加油次数最少。

算法最优性证明：

假设存在一个更优的加油方案，即加油次数比贪心算法得到的方案更少。考虑贪心算法选择的加油站，在汽车当前油量能到达的范围内，贪心算法总是选择最远的加油站加油。如果存在更优方案，意味着在某个位置不选择最远的加油站加油也能达到最少加油次数，但这样可能会导致后续需要更多的加油次数来弥补，因为不选择最远的加油站，后续可选择的加油站范围会变小，更有可能在后续遇到无法到达下一个加油站的情况，所以贪心算法得到的方案就是最优解。

代码实现如下：

```
int minRefuelStops(int n, int k, int* stations) {
    int count = 0;  // 加油次数
    int current = 0;  // 当前位置
    int next = 0;  // 下一个能到达的最远加油站位置
    int i = 0;

    while (current + n < stations[k]) {  // 不能到达目的地时循环
        next = current;
        for (; i < k && stations[i] <= current + n; i++) {
            if (stations[i] > next) {
                next = stations[i];
            }
        }
        if (next == current) {  // 无法前进，不能到达目的地
            return -1;
        }
        current = next;
        count++;
    }
    return count;
}
```

4-11　删数问题。

问题描述：给定 n 位正整数 a，去掉其中任意 $k \leqslant n$ 个数字后，剩下的数字按原次序排列组成一个新的正整数。对于给定的 n 位正整数 a 和正整数 k，设计一个算法找出剩下数字组成的新数最小的删数方案。

算法设计：对于给定的正整数 a，计算删去 k 个数字后得到的最小数。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是 1 个正整数 a。第 2 行是正整数 k。

结果输出：将计算的最小数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
178543 4	13

答：利用贪心算法，每次删除使得剩下数字组成的数最小的数字。从左到右遍历数字序列，若当前数字大于其右边数字，删除当前数字能使剩下数字组成的数更小；若当前数字不大于右边数字，则继续往后遍历，直到删除 k 个数字。

代码实现如下：

```c
void removeKdigits(char* num, int k) {
    int len = strlen(num);
    if (k >= len) {
        printf("0\n");
        return;
    }
    int top = -1;
    char stack[len];
    for (int i = 0; i < len; i++) {
        while (top != -1 && k > 0 && stack[top] > num[i]) {
            top--;
            k--;
        }
        stack[++top] = num[i];
    }
    //如果k还有剩余，从栈顶删除剩余的数字
    while (k > 0) {
        top--;
        k--;
    }
    //处理前导0
    int start = 0;
    while (start <= top && stack[start] == '0') {
        start++;
    }
    if (start > top) {
        printf("0\n");
    }
    else {
        for (int i = start; i <= top; i++) {
            printf("%c", stack[i]);
        }
        printf("\n");
    }
}
```

4-15　最优分解问题。

问题描述：设 n 是一个正整数。现在要求将 n 分解为若干互不相同的自然数的和，且使这些自然数的乘积最大。

算法设计：对于给定的正整数 n，计算最优分解方案。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n。

结果输出：将计算的最大乘积输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10	30

答：思路要使分解后的自然数乘积最大，尽量让分解出的数尽可能多且尽量小，同时这些数要互不相同。从最小的自然数 2 开始逐步累加，直到不能再分解为止。

从 2 开始尝试将数 n 进行分解。在每次循环中判断，如果剩余的 n 减去当前的数 i 小于 i 或者 n - i 已经在已有的分解因子列表 factors 中，说明不能再继续用 i 来分解，此时把剩余的 n 作为一个因子加入列表。

代码如下:

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

long long optimal_decomposition(int n) {
    int i = 2;
    int factors[1000];  // 假设最多有1000个因子，可根据实际情况调整
    int factor_count = 0;
    while (n > 0) {
        if (n - i < i || n - i == 0) {
            factors[factor_count++] = n;
            n = 0;
        }
        else {
            factors[factor_count++] = i;
            n -= i;
            i++;
        }
    }
    long long result = 1;
    for (int j = 0; j < factor_count; j++) {
        result *= factors[j];
    }
    return result;
}

int main() {
    FILE* input_file, * output_file;
    int n;
    input_file = fopen("input.txt", "r");
    if (input_file == NULL) {
        perror("无法打开输入文件");
        return EXIT_FAILURE;
    }
    if (fscanf(input_file, "%d", &n) != 1) {
        fprintf(stderr, "输入文件格式错误\n");
        fclose(input_file);
        return EXIT_FAILURE;
    }
    fclose(input_file);
    long long max_product = optimal_decomposition(n);
    output_file = fopen("output.txt", "w");
    if (output_file == NULL) {
        perror("无法打开输出文件");
        return EXIT_FAILURE;
    }
    fprintf(output_file, "%lld", max_product);
    fclose(output_file);

    return EXIT_SUCCESS;
}
```