贪心法的核心思想

贪心算法通过一系列选择来得到问题的解，所做的选择都是当前状态下局部最好选择，即贪心选择。局部最优并不总能获得整体最优解，找出全局最优解需要满足最优子结构和最优贪心选择属性。

问题 4-4

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 $a_k \Big/ \sum_{i=1}^{n} a_i$。

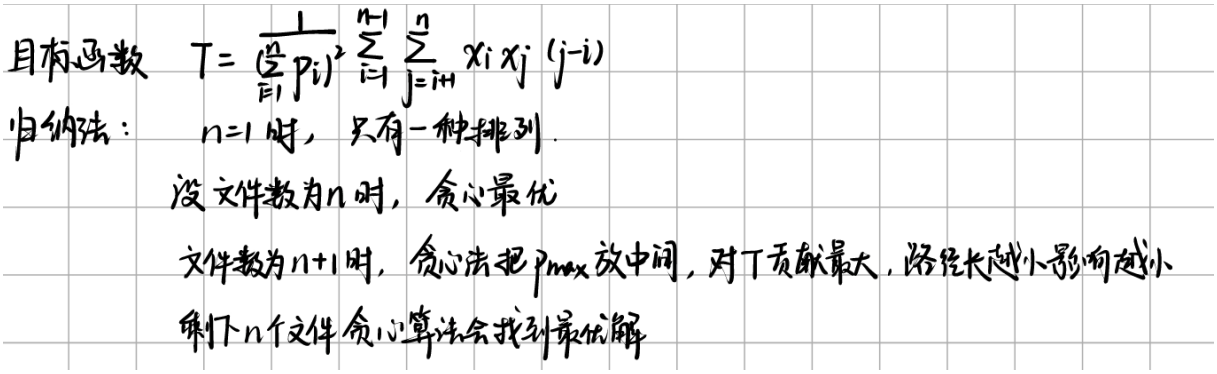
结果输出：将计算的最小期望检索时间输出到文件 output.txt。

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

思路：

1. 将检索概率最大的文件放在中心位置，以最小化高检索概率文件的平均路径长度；
2. 两边交替填充剩余文件，尽量使检索概率高的文件均匀分布在靠近中心的位置。

贪心法可得到最优解的证明：



伪代码：

核心模块

Function greedy(n, p)

```
sort p in ascending order
let x be an array of size n
let k = n / 2
//将最大检索概率放在中心位置
x[k] = p[n - 1]
// 从中心向右两边交替填充
for i = k + 1 to n - 1 do
    x[i] = p[n - 2 * (i - k)]
end for
// 从中心向左两边交替填充
for i = k - 1 down to 0 do
    x[i] = p[n - 2 * (k - i) - 1]
end for
let m = 0
```

```
let t = 0
for i = 0 to n - 1 do
    m = m + p[i]
    for j = i + 1 to n - 1 do
        t = t + x[i] * x[j] * (j - i)
    end for
end for
t = t / (m * m)
return t
```

End Function

时间复杂度：$O(n^2)$

对数组p进行升序排序，时间复杂度为O(nlogn);

数组填充两个for循环，O(n);

期望时间计算外层循环n次，内层循环n-i-1次，最坏$O(n^2)$;

空间复杂度：O(n)

问题 4-15

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。

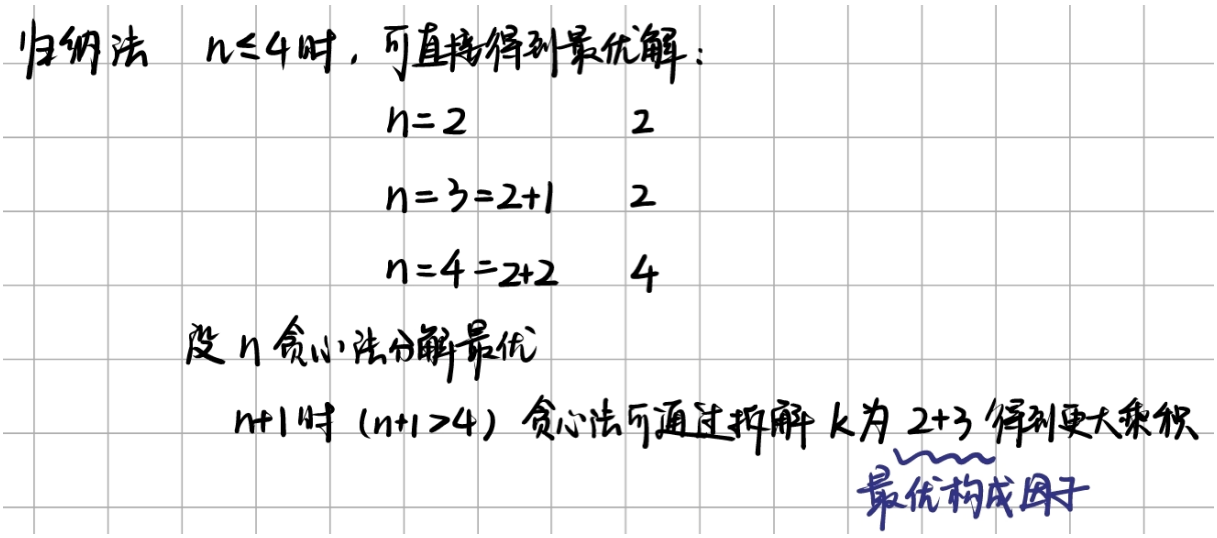
输入文件示例	输出文件示例
input.txt	output.txt
10	30

思路：

若a+b=const，则|a-b|越小，ab越大

将n分成从2开始的连续自然数的和。如果最后剩下一个数，将此数在后项优先的方式下均匀地分给前面各项。

贪心法可得到最优解的证明：



伪代码：

核心模块

Function dicomp(n)

```
k = 2
a = array of size n/2
index = 0
while n > 0{
    if n - k >= 0 then
        a[index] = k
        index = index + 1
        n = n - k
        k = k + 1
    end if
    else
        a[index] = n
        n = 0
    end if
}
remain = a[index]
for i = index - 1 down to 0 do
    if remain > 0 then
        a[i] = a[i] + 1
        remain = remain - 1
    end if
end for
res = 1
```

```
    for i = 0 to index - 1 do
        res = res * a[i]
    end for
    return res
End Function
```

时间复杂度：$O(n^{1/2})$

主循环：k从2开始逐渐增大，直到n为0，因此循环执行次数大约是$O(n^{1/2})$；

分配剩余数：最坏情况下，index是$O(n^{1/2})$；

最大乘积计算：循环的次数与index成正比，最坏情况下为$O(n^{1/2})$；

空间复杂度：O(n)