中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 1-2 课程作业

2022 年 9 月 11 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

试确定下述程序的关键操作数, 该函数实现一个 $m \times n$ 矩阵与一个 $n \times p$ 矩阵之间的乘法:

```
1  template <class T>
2  void Mult(T **a, T **b, int m, int n, int p) {
3      //m×n 矩阵 A 与 n×p 矩阵 B 相成得到 m×p 矩阵 C
4      for(int i = 0; i < m; i++) {
5          for(int j = 0; j < p ;j++) {
6              T sum = 0;
7              Tfor(int k = 0; k < n; k++)
8              Tsum += a[i][k]*b[k][j];
9              C[i][j] = sum;
10         }
11     }
12 }
```

Solution:

此算法的关键操作为**第 8 行**, 该语句的操作数为 2 (1 次加法和 1 次乘法). 则三层循环的总关键操作数为:

$$T = \sum_{i=0}^{m-1} \sum_{j=0}^{p-1} \sum_{k=0}^{n-1} 2 = 2mnp$$

故该算法的时间复杂度为

$$T(m,n,p) = O(2mnp) = O(mnp)$$

Problem 2

函数 MinMax 用来查找数组 $a[0:n-1]$ 中的最大元素和最小元素, 以下给出两个程序. 令 n 为实例特征. 试问: 在各个程序中, a 中元素之间的比较次数在最坏情况下各是多少?

```cpp
/* 找最大最小元素 (方法一)*/
template <class T>
bool MinMax(T a[], int n, int& Min, int& Max) {
    //寻找 a[0:n-1] 中的最小元素与最大元素
    //如果数组中的元素数目小于 1, 则返回 false
    if(n<1) return false;
    Min=Max=0; //初始化
    for(int i=1; i<n; i++) {
        if(a[Min]>a[i]) Min=i;
        if(a[Max]<a[i]) Max=i;
    }
    return true;
}
```

```cpp
/* 找最大最小元素 (方法二)*/
template <class T>
bool MinMax(T a[], int n, int& Min, int& Max) {
    //寻找 a[0:n-1] 中的最小元素与最大元素
    //如果数组中的元素数目小于 1, 则返回 false
    if(n<1) return false;
    Min=Max=0; //初始化
    for(int i=1; i<n; i++) {
        if(a[Min]>a[i]) Min=i;
        else if(a[Max]<a[i]) Max=i;
    }
    return true;
}
```

Solution:

不论数组 a 是单调递增还是单调递减, for 循环内部的两次判断都得执行, 所以方法 1 在任何情况下的元素比较次数都为 $2 \times (n-1)$; 而对于方法 2 来说, 当数组 a 单调递减 (最好情况) 时, for 循环内部的第一个判断条件一定满足, 那么 else if 这个判断自然就不用执行了, 即此情形下的元素比较次数为 $1 \times (n-1)$. 但当数组 a 单调递增 (最坏情况) 时, 第一个循环条件在各轮循环中都不能满足, 所以紧跟着需要执行后边的 else if 判断, 即此情形下的元素比较次数为 $2 \times (n-1)$.

Problem 3

证明以下关系式不成立: (1). $10n^2 + 9 = O(n)$; (2). $n^2 \log n = \Theta\left(n^2\right)$.

Solution:

证明. **(1).** 考虑极限 $\lim\limits_{n\to\infty} \dfrac{10n^2+9}{n} = +\infty > c \ \ (\forall c > 0)$, 故根据大 O 比率定理可知该式不成立;

(2). 考虑极限 $\lim\limits_{n\to\infty} \dfrac{n^2 \log n}{n^2} = +\infty > c \ \ (\forall c > 0)$, 故根据 Θ 比率定理可知该式不成立. □

Problem 4

按照渐近阶从低到高的顺序排列以下表达式:

$$4n^2, \log n, 3^n, 20n, n^{2/3}, n!$$

Solution:

根据 Stirling 公式:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \ (\text{as } n \to +\infty)$$

可知有渐进阶的顺序为

$$\log n \ll n^{2/3} \ll 20n = \Theta(n) \ll 4n^2 = \Theta\left(n^2\right) \ll 3^n \ll n!$$

Problem 5

(1). 假设某算法在输入规模是 n 时为 $T(n) = 3 * 2^n$. 在某台计算机上实现并完成该算法的时间是 t 秒. 现有另一台计算机, 其运行速度为第一台的 64 倍. 那么, 在这台计算机上用同一算法在 t 秒内能解决规模为多大的问题?

(2). 若上述算法改进后的新算法的时间复杂度为 $T(n) = n^2$, 则在新机器上用 t 秒时间能解决输入规模为多大的问题?

(3). 若进一步改进算法, 最新的算法的时间复杂度为 $T(n) = 8$, 其余条件不变, 在新机器上运行, 在 t 秒内能够解决输入规模为多大的问题?

Solution:

(1). 设问题规模为 M, 则新机器上的求解时间为 $t = 3 \times 2^M/64$, 老机器的求解时间 $t = 3 \times 2^n$, 即解得 $M = n + 6$;

(2). 同理设问题规模为 M, 则新机器上的求解时间为 $t = M^2/64 = n^2$, 老机器的求解时间 $t = n^2$, 即解得 $M = 8n$;

(3). 因为该算法的时间复杂度是常数阶的, 也就意味着问题规模不影响求解时间 (当问题规模很大时), 所以在任何 (可以运行该算法的) 机器上, t 秒内可以解决任意规模的问题.

Problem 6

Fibonacci 数有递推关系:

$$F(n)=\begin{cases}1, & n=0\\ 1, & n=1\\ F(n-1)+F(n-2), & n>1\end{cases}$$

试求出 $F(n)$ 的表达式.

Solution:

解法 1 (数学解法): 根据递推关系可知其对应的特征方程为

$$r^2=r+1\Rightarrow r_1=\frac{1+\sqrt{5}}{2}, r_2=\frac{1-\sqrt{5}}{2}$$

根据二阶常系数齐次差分方程解的结构定理可知解形如 $F\left(n\right)=C_1\left(\frac{1+\sqrt{5}}{2}\right)^n+C_2\left(\frac{1-\sqrt{5}}{2}\right)^n$.

又由于 $F(0)=F(1)=1$, 所以 $C_1=\frac{1}{\sqrt{5}}, C_2=-\frac{1}{\sqrt{5}}$, 即方程解为

$$F\left(n\right)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

解法 2 (矩阵的快速幂解法): 考虑如下

$$\begin{pmatrix}F_{n+1}\\F_n\end{pmatrix}=\begin{pmatrix}F_n+F_{n-1}\\F_n\end{pmatrix}=\begin{pmatrix}1&1\\1&0\end{pmatrix}^1\begin{pmatrix}F_n\\F_{n-1}\end{pmatrix}=\cdots=\begin{pmatrix}1&1\\1&0\end{pmatrix}^n\begin{pmatrix}F_1\\F_0\end{pmatrix}$$

矩阵 $\begin{pmatrix}1&1\\1&0\end{pmatrix}$ 的特征值为 $\lambda_1=\frac{1-\sqrt{5}}{2}, \lambda_2=\frac{1+\sqrt{5}}{2}$. 对角矩阵 Λ 和相似变换矩阵 P 分别为

$$\Lambda=\begin{pmatrix}\frac{1-\sqrt{5}}{2}&0\\0&\frac{1+\sqrt{5}}{2}\end{pmatrix},\quad P=\begin{pmatrix}\frac{1-\sqrt{5}}{2}&\frac{1+\sqrt{5}}{2}\\1&1\end{pmatrix}$$

所以

$$\begin{pmatrix}F_{n+1}\\F_n\end{pmatrix}=\begin{pmatrix}1&1\\1&0\end{pmatrix}^n\begin{pmatrix}F_1\\F_0\end{pmatrix}=P\Lambda^nP^{-1}\begin{pmatrix}1\\1\end{pmatrix}=\begin{pmatrix}\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]\\\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n}-\left(\frac{1-\sqrt{5}}{2}\right)^{n}\right]\end{pmatrix}$$

Problem 7

下面的无向图以邻接链表存储, 而且在关于每个顶点的链表中与该顶点相邻的顶点是按照字母顺序排列的. 试以此图为例描述讲义中算法 DFNL 的执行过程.

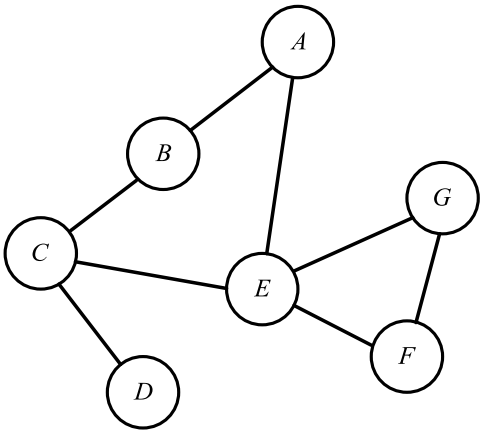


图 1: 一个无向图 G

Solution:

先利用深度优先搜索 (**DFS**) 获取深度优先搜索树 T(深索数 DFN 也顺便求出来了), 再利用后根遍历来搜索 T, 最后**从下到上、从右往左**依次计算出各顶点的最低深索数. 此过程可以如下模拟:

Step 1. 出入栈的模拟:

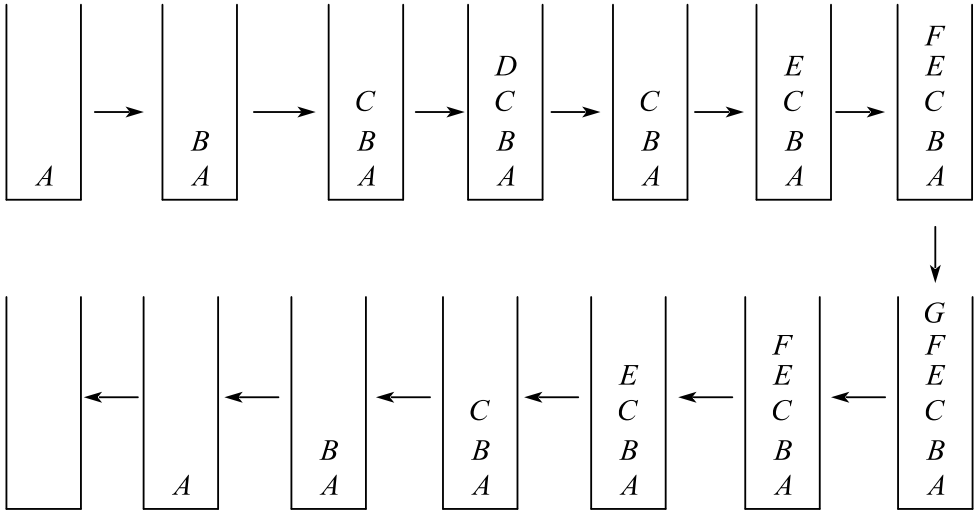


图 2: 出入栈模拟

Step 2. 求出深度搜索树 T 和余边, 并利用后根遍历来搜索 T, 最后从下到上、从右往左依次计算出各顶点的最低深索数: 该步骤的模拟过程见如下 (右图为深度优先搜索树 T, 左图的红边为余边 (对应到搜索树 T 的红色虚线), 红色圈数字为最低深索数).

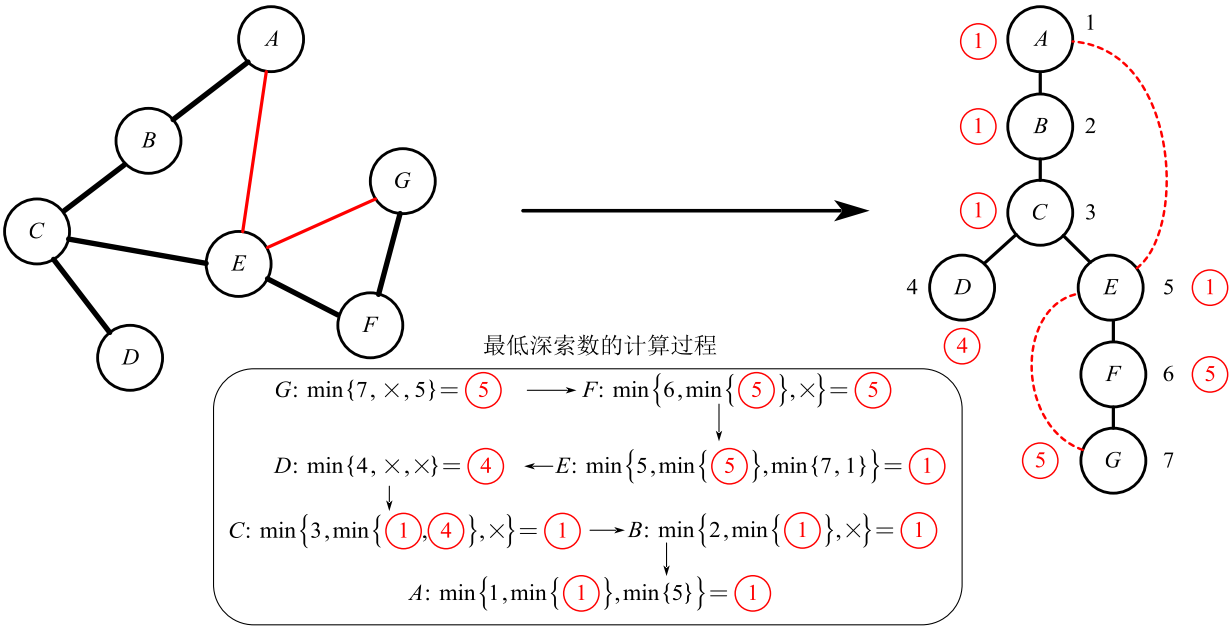


图 3: 第二步骤的模拟

Step 3. 求出割点以及 2-连通分支: 右图搜索树 T 的根节点 A 由于没有至少两个子节点, 故不可能是割点; 而 D, G 作为叶子节点更不可能是割点; 通过非根节点的割点判别充要条件[1]可知: E, C 均为割点, 于是可以简单得出如下 2-连通分支 (下图中红色的即为 2-连通分支, 图中分叉只是为了体现出拆解过程):

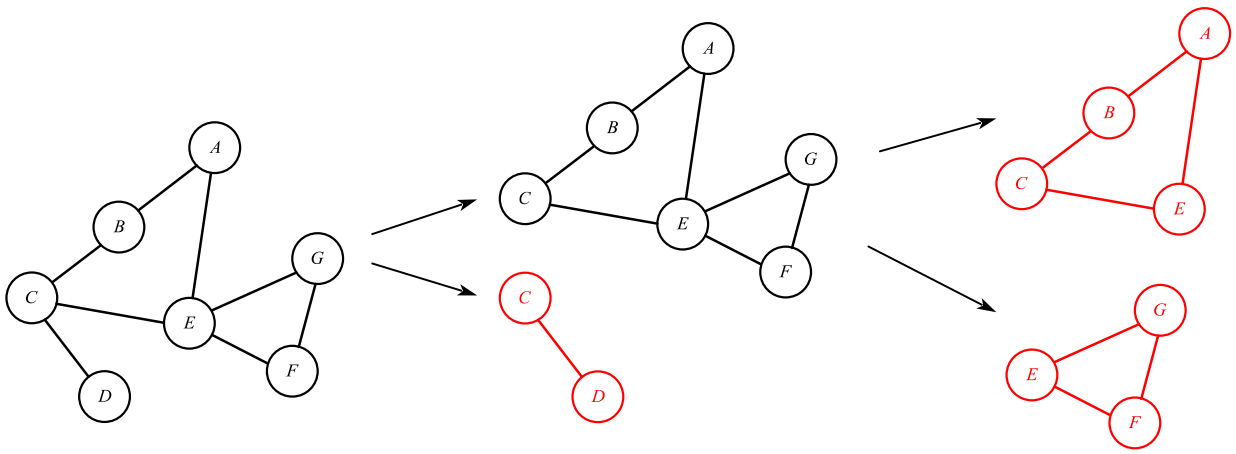


图 4: 第三步骤的模拟

至此, 此例的 DFNL 算法模拟执行完毕.

[1]对于非根节点 u 是割点 $\Leftrightarrow$ 节点 u 存在儿子 w, 使得 $L(w) \geq \mathrm{DFN}(u)$.

Problem 8

考虑下述选择排序算法1所示:

Algorithm 1 选择排序

Input: n 个不等整数的数组 $A[1..n]$
Output: 按递增次序排序的 A

```
1: for i := 1 to n do
2:     for j := i + 1 to n do
3:         if A[j] < A[i] then
4:             A[i] ↔ A[j]
5:         end if
6:     end for
7: end for
8: 输出排序后的数组 A
```

问: (1) 最坏情况下做多少次比较运算?
(2) 最坏情况下做多少次交换运算? 在什么输入时发生?

Solution:

(1). 任意情况下要比较 $(n-1)+(n-2)+\cdots+1=\frac{1}{2}n(n-1)$ 次. 再者, 此处不存在所谓的最好或最坏情况, 不论数组 A 是逆序还是升序排列, 计算机不可能因为提前得知 A 的所有情况而不执行判断语句, 即每次循环都需要进行比较运算;

(1). 最坏情况: 输入数组内元素为降序排列. 此时做 $(n-1)+(n-2)+\cdots+1=\frac{1}{2}n(n-1)$ 次交换运算. 当数组 A 内元素为升序排列时, 则交换次数为 0 (即为最好情况).

Problem 9

考虑下面的每对函数 $f(n)$ 和 $g(n)$, 比较他们的阶.
(1). $f(n)=\frac{1}{2}\left(n^2-n\right),\ g(n)=6n$; (2). $f(n)=n+2\sqrt{n},\ g(n)=n^2$;
(3). $f(n)=n+n\log n,\ g(n)=n\sqrt{n}$; (4). $f(n)=\log(n!),\ g(n)=n^{1.05}$;

Solution:

(1). $f(n)=\Theta\left(n^2\right)\gg\Theta(n)=g(n)$; **(2).** $f(n)=\Theta(n)\ll\Theta\left(n^2\right)=g(n)$;
(3). $f(n)=\Theta(n\log n)\ll\Theta\left(n^{1.5}\right)=g(n)$; **(4).** 根据**斯特林公式**可知:

$$\log(n!)\sim\log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)=\frac{1}{2}\log(2\pi n)+n(\log n-\log e)\sim n\log n\ (\text{as}\, n\to\infty)$$

所以有 $f(n)=\Theta(n\log n)\ll\Theta\left(n^{1.05}\right)=g(n)$.

Problem 10

在表1中填入 true 或 false.

	$f(n)$	$g(n)$	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$	$f(n) = \Theta(g(n))$
1	$2n^3 + 3n$	$100n^2 + 2n + 100$			
2	$50n + \log n$	$10n + \log\log n$			
3	$50n \log n$	$10n \log\log n$			
4	$\log n$	$\log^2 n$			
5	$n!$	5^n			

表 1: 原始表格

Solution:

解答如下表2所示:

	$f(n)$	$g(n)$	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$	$f(n) = \Theta(g(n))$
1	$2n^3 + 3n$	$100n^2 + 2n + 100$	False	True	False
2	$50n + \log n$	$10n + \log\log n$	True	True	True
3	$50n \log n$	$10n \log\log n$	False	True	False
4	$\log n$	$\log^2 n$	True	False	False
5	$n!$	5^n	False	True	False

表 2: 解答

Problem 11

用迭代法求解下列递推方程:

(1). $\begin{cases} T(n) = T(n-1) + n - 1 \\ T(1) = 0 \end{cases}$; (2). $\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + n - 1 \\ T(1) = 0 \end{cases}, n = 2^k$

Solution:

(1). 易知

$$\begin{cases} T(2) - T(1) = 1 \\ T(3) - T(2) = 2 \\ \vdots \\ T(n) - T(n-1) = n - 1 \end{cases} \xrightarrow{\text{各式相加}} T(n) = 1 + 2 + \cdots + n - 1 = \frac{1}{2} n(n-1) = \Theta\left(n^2\right)$$

(2). 令 $n = 2^k$, 则易知有如下:

$$\begin{aligned} T\left(2^k\right) &= 2T\left(2^{k-1}\right) + 1 \cdot 2^k - 1 \\ &= 2^2 T\left(2^{k-2}\right) + 2 \cdot 2^k - (1+2) \\ &= 2^3 T\left(2^{k-3}\right) + 3 \cdot 2^k - \left(1 + 2 + 2^2\right) \\ &\vdots \\ &= 2^k T(1) + k \cdot 2^k - \left(1 + 2 + \cdots + 2^{k-1}\right) \\ &= k \cdot 2^k + \left(1 - 2^k\right) = (k-1) \cdot 2^k + 1 \\ &= n \log n - n + 1 = T(n) = \Theta(n \log n) \end{aligned}$$

至此, Chap 1-2 的作业解答完毕.

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 3 课程作业解答

2022 年 9 月 20 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

编写程序实现归并排序算法 **MergeSort** 和快速排序算法 **QuickSort**;

Solution: 以下是手撕归并排序算法 **MergeSort** 的 C++ 程序.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
void Merge(vector<int>& nums, int low, int mid, int high) {
    int i = low, j = mid + 1, k = 0;
    vector<int> temp(high - low + 1);
    while(i <= mid && j <= high) {
        if(nums[i] <= nums[j])
            temp[k++] = nums[i++];
        else
            temp[k++] = nums[j++];
    }
    while(i <= mid) temp[k++] = nums[i++];
    while(j <= high) temp[k++] = nums[j++];
    for(int i = low; i <= high; i++) {
        nums[i] = temp[i - low];
    }
}
void MergeSort(vector<int>& nums, int low, int high) {
    if(low < high) {
        int mid = (high + low) >> 1;
        MergeSort(nums, low, mid);
        MergeSort(nums, mid + 1, high);
        Merge(nums, low, mid, high);
    }
}
int main() {
    int n;
    cin >> n;//输入数组的长度
    vector<int> a(n);//定义数组
    for (int i = 0; i < n; i++) {//对数组初始化
        a[i]= -1 * (n + 1) + rand()%(2 * n + 2); //在 [-n-1,n+1] 随机产生数组
        cout << a[i] << " ";  //打印该数组
    }
    printf("\n");
    clock_t startTime = clock();//计时开始
    MergeSort(a, 0, n - 1); //手撕归并排序, 20000000 的数据量耗时 6.016s
    clock_t endTime = clock();//计时结束
    for(int i = 0; i < n; i++) {
        cout << a[i] << " "; //输出排序后的数组
    }
    printf("\n");
    cout << " 归并排序算法的运行时间为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

Solution: 以下是手撕快速排序算法 **QuickSort** 的 C++ 程序.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int partition(vector<int>& nums, int low, int high) {
    int index = low + rand()%(high - low + 1);
    swap(nums[low], nums[index]);
    int pivot = nums[low];
    while (low < high) {
        while (low < high && nums[high] >= pivot) high--;
        swap(nums[low], nums[high]);
        while (low < high && nums[low] <= pivot) low++;
        swap(nums[low], nums[high]);
    }
    return low;
}
void QuickSort(vector<int>& nums, int low, int high) {
    if(low < high) {
        int loc = partition(nums, low, high);
        QuickSort(nums, low, loc - 1);
        QuickSort(nums, loc + 1, high);
    }
}
int main() {
    int n;
    cin >> n;//输入数组的长度
    vector<int> a(n);//定义数组
    for (int i = 0; i < n; i++) {//对数组初始化
        a[i]= -1 * (n + 1) + rand()%(2 * n + 2); //在 [-n-1,n+1] 随机产生数组
        cout << a[i] << " ";  //打印该数组
    }
    printf("\n");
    clock_t startTime = clock();//计时开始
    QuickSort(a, 0, n - 1); //手撕归并排序, 20000000 的数据量耗时 8.165s
    //sort(a.begin(), a.end()); //调 STL 库, 20000000 的数据量耗时 3.775s
    clock_t endTime = clock();//计时结束
    for(int i = 0; i < n; i++) {
        cout << a[i] << " "; //输出排序后的数组
    }
    printf("\n");
    cout << " 快速排序算法的运行时间为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

Problem 2

用长分别为 10000、30000、50000、80000、100000、200000 的 6 个数组 (可用机器随机产生) 的排列来统计这两种算法的时间复杂性;

Solution:

在上述两段代码中, 分别输入 $n = 10000, 30000, 50000, 80000, 100000, 200000$, 可得到归并排序算法的运行时间为 2ms, 6ms, 12ms, 19ms, 23ms, 49ms. 而对应手撕的快速排序算法的运行时间为 1ms, 4ms, 6ms, 10ms, 12ms, 26ms. STL 库中 sort 算法的运行时间为 1ms, 5ms, 6ms, 11ms, 14ms, 29ms. 由此可见, 当数据量比较大时, 快速排序算法会比归并排序更快! 值得一提的是, 当数据量为 20000000 时, 手撕快排、归并排序以及 STL 的 sort 算法的用时分别为 8.325s, 5.953s, 3.743s, 由此可见 STL 标准库的算法优化是很成熟的.

Problem 3

讨论归并排序算法 **MergeSort** 的空间复杂性.

Solution:

归并排序的递归调用过程需要 $O(h)$ 的栈空间 (h 为递归树的高度), 而整个递归树的高度 (即递归调用的最深层数) 为 $\log n$, 在合并过程中也需要额外 $O(n)$ 空间的 temp 数组 (而快速排序却不需要). 故归并排序和快速排序的空间复杂度分别为 $O(n + \log n) = O(n), O(\log n)$.

Problem 4

证明算法 **PartSelect** 的平均时间复杂性为 $O(n)$.

证明. 假定数组中的元素各不相同, 且第一次划分时划分元素 v 是第 i 小元素的概率为 $1/n$. 由于一趟快排的时间复杂度为 $O(n)$(因为是双指针解法), 所以不妨设一趟快排用时为 cn. 设 $C_A^k(n)$ 表示在数组 A 的 n 个元素中寻找第 k 小元素的平均时间复杂度. 若 $i = 1, \cdots, k-1$(即 $i < k$), 则子问题的平均用时分别为 $C_A^{k-i}(n-i)$; 若 $i = k+1, \cdots, n$(即 $i > k$), 则子问题的平均用时分别为 $C_A^k(i-1)$[1]. 于是, $C_A^k(n)$ 作为随机变量 v 的函数, 其数学期望表达式为 (即将概率与随机变量的函数取值相乘求和)

$$\begin{aligned} C_A^k(n) &\le cn + \frac{1}{n}\cdot C_A^{k-1}(n-1) + \cdots + \frac{1}{n}\cdot C_A^1(n-k+1) + \frac{O(1)}{n} + \frac{1}{n}\cdot C_A^k(k) + \cdots + \frac{1}{n}\cdot C_A^k(n-1) \\ &= cn + \frac{1}{n}\sum_{i=1}^{k-1} C_A^{k-i}(n-i) + \frac{1}{n}\sum_{i=k+1}^{n} C_A^k(i-1) \end{aligned}$$

令 $R(n) = \max\limits_k \left\{C_A^k(n)\right\}$, 下面通过数学归纳法来证明 $R(n) \le 4cn$:

- 当 $n = 1$ 时, 保证 $C \le 4c$, 即可满足 $R(1) = \max\limits_k \left\{C_A^k(1)\right\} = C \le 4c$;

[1] 当 $i = k$ 时, 显然子问题的平均用时就是 $O(1)$(即一下就找到了). 并且随机变量的函数取值就是子问题的平均用时.

- 设 $\forall m < n, R(m) \leq 4cm$ 都成立, 且不妨设 $R(n) = \max\limits_{k}\left\{C_A^k(n)\right\} = C_A^{k_n}(n)$, 数学期望不等式两边同时关于 k 取 max 即有如下[2]:

$$R(n) \leq cn + \frac{1}{n}\cdot\{R(n-1)+\cdots+R(n-k_n+1)\} + \frac{1}{n}\cdot\{R(k_n)+\cdots+R(n-1)\} \tag{1}$$

$$\leq cn + \frac{4c}{n}\cdot\{(n-1)+\cdots+(n-k_n+1)\} + \frac{4c}{n}\cdot\{k_n+\cdots(n-1)\} \tag{2}$$

$$\leq cn + \frac{4c}{n}\cdot\left\{\frac{(2n-k_n)(k_n-1)}{2} + \frac{(n+k_n-1)(n-k_n)}{2}\right\} \tag{3}$$

$$= cn + \frac{4c}{n}\cdot\left\{-k_n^2 + (1+n)k_n + \frac{n^2-3n}{2}\right\} \leq cn + \frac{4c}{n}\cdot\{\cdots\}_{k_n=\frac{n+1}{2}} \tag{4}$$

$$= cn + \frac{4c}{n}\cdot\left\{\frac{3n^2-4n+1}{4}\right\} \leq cn + c\cdot\{3n-3\} \leq 4cn \tag{5}$$

综上所述, 根据数学归纳法可得知 $\forall n \geq 1$, 都有 $R(n) \leq 4cn$, 即 **PartSelect** 算法的平均时间复杂度为 $O(n)$. □

Problem 5

改进插入排序算法 (第三章 ppt No.6), 在插入元素 $a[i]$ 时使用二分查找代替顺序查找, 将这个算法记做 **BinarySort**, 估计算法在最坏情况下的时间复杂度.

Solution:

先写出 **BinarySort** 算法的伪代码, 如下所示:

Algorithm 1 二分插入排序 **BinarySort** 算法

Input: 长度为 n 的数组 $A[0,\cdots,n-1]$
Output: 按递增次序排序的 A

```
1:  for i := 1 to n − 1 do
2:      int temp = A[i];                ▷ 其实第 3 行也可这样: int low = upperbound(A, 0, i − 1, temp);
3:      int low = upper_bound(A.begin(), A.begin() + i, temp) − A.begin();   ▷ 源于 C++ 的 STL 标准库
4:      if low ! = i then               ▷ 若 low=i, 说明 A[0,··· ,i − 1] 中没有 temp 的插入位置
5:          for j from i − 1 by −1 to low do
6:              A[j + 1] := A[j];
7:          end for
8:          A[low] = temp;
9:      end if
10: end for
11: end {BinarySort};
```

其中 **upper_bound** 是 STL 标准库里的函数, 其作用是在一段有序数组中寻找第一个严格大于 target 的元素位置[3]. 其具体算法的伪代码也如下所示:

[2]注意 (4) 式中第二项是关于 k_n 的二次函数 (开口向下), 所以可以放缩到他的最大值.
[3]是个迭代器, 若想得到下标, 则需要减去 A.begin()

Algorithm 2 二分查找 **upperbound** 算法

Input: 长度为 n 的升序数组 $A[0, \cdots, n-1]$, 目标元素 target
Output: 第一个大于 target 的元素下标

```
int low = 0, high = n − 1;
while low <= high do
    int mid = (low + high) >> 1;
    if A[mid] <= target then          ▷ 若是 lowerbound 算法, 此处判断条件则是 A[mid] < target
        low = mid + 1;
    else
        high = mid − 1;
    end if
end while
return low;                           ▷ 返回所要求的下标
end {upperbound}
```

所用到的 **upperbound** 函数和 **BinarySort** 函数编码如下:

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int upperbound(vector<int>& nums, int low, int high, int target) {
    while(low <= high) {
        int mid = (low + high) >> 1;
        if(nums[mid] <= target) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}
void BinarySort(vector<int>& nums) {
    int len = nums.size();
    for(int i = 1; i < len; i++) {
        int temp = nums[i];
        int low = upperbound(nums, 0, i - 1, temp); //自己手撕的二分查找 upperbound
        //int low = upper_bound(nums.begin(), nums.begin() + i, temp) - nums.begin(); //直接
↪   调用 STL 标准库的二分查找 upper_bound
        if(low != i) {
            for(int j = i - 1; j >= low; j--) {
                nums[j + 1] = nums[j];
            }
            nums[low] = temp;
        }
    }
}
```

紧接着主函数的编码如下 (经过调试, 读者可亲自运行):

```cpp
int main()
{
    int n;
    scanf("%d", &n);
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        a[i]= -1 * (n + 1) + rand()%(2 * n + 2);
        cout << a[i] << " ";
    }
    printf("\n");
    printf("\n");
    clock_t startTime = clock();
    BinarySort(a);
    clock_t endTime = clock();
    for(int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    printf("\n");
    printf("\n");
    cout << " 运行时间为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

接下来分析最坏情况下的时间复杂度:

证明. 最坏情况显然是逆序的数组. 不管是用二分查找还是顺序查找, 都只能在查找位置上节约时间, 但是算法的**关键操作**是数组遍历和元素后移, 而需要遍历 $1+2+\cdots+(n-1)=\frac{1}{2}n(n-1)=\Theta(n^2)$. □

Problem 6

设 A 是 n 个非 0 实数构成的数组, 设计一个算法重新排列数组中的数, 使得负数都排在正数前面, 要求算法复杂度为 $O(n)$.

Solution:

方法 1 (时空复杂度均为 $O(n)$): 最简单的做法是, 直接对数组做一次遍历, 将负数放入数组 temp1、将正数放入 temp2, 最后将 temp1 数组和 temp2 进行合并即可. 其算法伪码如下[4]: 可以看出, 该算法需要借助长度综合为 n 的临时数组, 所以空间复杂度为 $O(n)$, 算法只对数组 A 做了一次遍历, 因此时间复杂度也为 $O(n)$. 并且该算法不仅可以将负数放在左边、把正数放在右边, 还可以把 0 放在中间 (虽然功能多余了). 因此, 该问题可以拓展为: 设 A 是一个长度为 n 的数组, 要求设计时间复杂度为 $O(n)$ 的算法, 使得在数组 A 中**原地交换元素**[5]来使得数组 A 的负数排在左边、0 排在中间、正数排在右边. 该问题就是著名的荷兰三色国旗问题, 该问题的算法求解思想可以借助了 (三路) 快速排序 (即二路随机快排排序的进一步算法优化) 中的一趟快排操作.[6]

[4]由于该伪代码所对应的 C++ 程序基本一样, 所以此处就不给出对应的 C++ 代码了.

[5]即不允许算法借助其他临时数组, 即要求算法的空间复杂度为 $O(1)$.

[6]双路随机快速排序的缺点是: 当在数组中碰到大量的重复数元素时, 双路快排的许多次交换操作就显得很多余了.

Algorithm 3 双色旗问题的 **doublecolor** 算法

Input: n 个非 0 实数构成的数组 $A[0,\cdots,n-1]$
Output: 负数排在正数前面的数组 $A[0,\cdots,n-1]$

```
vector<int> temp1, temp2, temp3;
for int i = 0; i < A.size(); i++ do
    if A[i] < 0 then
        temp1.push_back(A[i]);
    else if A[i] = 0 then
        temp2.push_back(A[i]);
    else
        temp3.push_back(A[i]);
    end if
end for
copy(temp1.begin(), temp1.end(), A.begin());
copy(temp2.begin(), temp2.end(), A.begin() + temp1.size());
copy(temp3.begin(), temp3.end(), A.begin() + temp1.size() + temp2.size());
return A;
end {doublecolor}
```

方法 2 (时空复杂度分别为 $O(n), O(1)$): 我们先给出三路快速排序的算法伪码和 C++ 代码.

Algorithm 4 三路快速排序 **QuickSort3** 算法

Input: 数组 $A[0,\cdots,n-1]$ 的子段 $A[\text{left}, \text{right}]$
Output: 排序后的数组子段 $A[\text{left}, \text{right}]$

```
if left >= right then
    return ;
end if
int rand_index = left + rand()%(right - left + 1);
swap(A[left], A[rand_index]);
int pivot = A[left];
int lt = left − 1, i = left, gt = right + 1;
while i < gt do
    if A[i] == pivot then
        i++;
    else if A[i] > pivot then
        swap(A[i], A[gt − 1]), gt- -;
    else if A[i] < pivot then
        swap(A[lt + 1], A[i]), lt++, i++;
    end if
end while
QuickSort3(A, left, lt);
QuickSort3(A, gt, right);
end {QuickSort3}
```

三路快速排序的算法的 C++ 代码描述如下, 在主函数里输入 `QuickSort3(a, 0, n − 1)` 即可调用以排序数组. 一趟三路快速排序后的数组情况: 左边全是小于 pivot, 中间等于 pivot, 右边大于 pivot. 因此针对

荷兰三色国旗问题, 我们可以直接设置 pivot 为 0, 并将算法4的 1,2,3,4,5,17,18 行删去, 即可写出三色问题的求解算法, 具体见算法5.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
void QuickSort3(vector<int>& nums, int left, int right) {
    if(left >= right) {
        return;
    }
    int rand_index = left + rand()%(right - left + 1);
    swap(nums[left], nums[rand_index]);
    int pivot = nums[left];
    int lt = left - 1, i = left, gt = right + 1;
    while(i < gt) {
        if(nums[i] == pivot) {
            i++;
        }
        else if(nums[i] > pivot) {
            swap(nums[i], nums[gt - 1]);
            gt--;
        }
        else if(nums[i] < pivot) {
            swap(nums[lt + 1], nums[i]);
            lt++, i++;
        }
    }
    QuickSort3(nums, left, lt);
    QuickSort3(nums, gt, right);
}
```

Algorithm 5 三色国旗问题的 **ThreeColor** 算法

Input: n 个实数构成的数组 $A[0,\cdots,n-1]$

Output: 负数排在正数前面且 0 排在中间的数组 $A[0,\cdots,n-1]$

int pivot = 0;
int lt = -1, i = 0, gt = n;
while i < gt **do**
  if $A[i]$ == pivot **then**
    i++;
  else if $A[i]$ > pivot **then**
    swap($A[i]$, $A[\text{gt}-1]$), gt- -;
  else if $A[i]$ < pivot **then**
    swap($A[\text{lt}+1]$, $A[i]$), lt++, i++;
  end if
end while
end {ThreeColor}

在此页的 C++ 程序上做对应的修改即可得到该算法的 C++ 程序, 由于作业空间限制, 此处就不予展示了.

Problem 7

Hanoi 塔问题: 图中有 A, B, C 三根柱子, 在 A 柱上放着 n 个圆盘, 其中小圆盘放在大圆盘的上边. 从 A 柱将这些圆盘移到 C 柱上去, 在移动和放置时允许使用 B 柱, 但不能把大盘放到小盘的下面. 设计算法解决此问题, 分析算法复杂度.

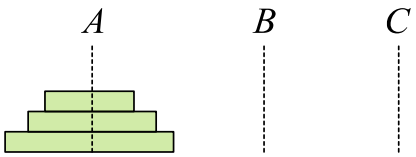


图 1: 汉诺塔问题

Solution:

该问题即为著名的汉诺塔问题, 递归式的求解算法描述为: 先将 A 上面的 $n-1$ 个盘子移到 B, 再将 A 中最下边的盘子移动到 C, 再将 B 中的 $n-1$ 个盘子移动到 C 上即可. 伪码描述为算法6:

Algorithm 6 汉诺塔问题的递归算法 **Hanoi**(A, C, n)

Input: n 个盘子从上往下、从小到大放在 A 柱
Output: 将 A 柱的圆盘移到 C 柱上

```
1: if n == 1 then
2:     move (A, C);
3: else
4:     Hanoi(A, B, n − 1);
5:     move (A, C);
6:     Hanoi(B, C, n − 1);
7: end if
8: end {Hanoi}
```

易知 $T(1) = 1$, 根据上述伪码可知时间复杂度有递推方程

$$T(n) = 2T(n-1) + 1 \tag{6}$$

于是通过递推得到

$$\begin{aligned} T(n) &= 2T(n-1) + 1 = 2\left(2T(n-2) + 1\right) + 1 \\ &= 2^2 T(n-2) + 1 + 2^1 \\ &= 2^3 T(n-3) + 1 + 2^1 + 2^2 \\ &= 2^{n-1} T(1) + 1 + 2 + \cdots + 2^{n-2} \\ &= 2^n - 1 \end{aligned}$$

于是 $T(n) = \Theta(2^n)$. 而且可以证明的是, 汉诺塔问题不存在多项式时间算法, 因此是一个难解的问题.

Problem 8

给定含有 n 个不同数的数组 $L = \{x_1, x_2, \cdots, x_n\}$, 若 L 中存在 x_i, 使得 $x_1 < x_2 < \cdots < x_{i-1} < x_i > x_{i+1} > \cdots > x_n$, 则称 L 是单峰的, 并称 x_i 是 L 的峰顶. 假设 L 是单峰的, 设计一个优于 $O(n)$ 的算法找到 L 的峰顶.

Solution:

算法思路描述: 对区间 $[0, n-1]$ 进行二分, 不妨设中点为 mid$= \lfloor (n-1)/2 \rfloor$.
观察中点的左右邻点:

- **case1:** 若 $L[\text{mid}-1] < L[\text{mid}] < L[\text{mid}+1]$, 则显然峰顶在右半区间 $[\text{mid}+1, n-1]$, 在该区间继续二分搜索即可;
- **case2:** 若 $L[\text{mid}-1] > L[\text{mid}] > L[\text{mid}+1]$, 则显然峰顶在左半区间 $[0, \text{mid}-1]$, 在该区间继续二分搜索即可;
- **case2:** 若 $L[\text{mid}-1] < L[\text{mid}] > L[\text{mid}+1]$, 则显然峰顶就是 $L[\text{mid}]$, 至此搜索完毕.

对应的算法伪代码见如下[7]:

Algorithm 7 单峰数组的二分搜索算法 **peakIndex**(L)

Input: n 个不同数的数组 $L[0, n-1]$ 且 L 为单峰数组
Output: L 的峰顶

```
1:  if n == 3 then                          ▷ 因为已知 L 为单峰数组, 因此 n ≥ 3
2:      return L[1];                        ▷ 若 n = 3, 则显然 L[1] 为峰顶
3:  else
4:      int low = 0, high = n − 1;
5:      while low <= high do
6:          int mid = (low + high) >> 1;
7:          if L[mid − 1] < L[mid] > L[mid + 1] then
8:              return L[mid];
9:          else if L[mid − 1] < L[mid] < L[mid + 1] then
10:             low = mid + 1;
11:         else
12:             high = mid − 1;
13:         end if
14:     end while
15: end if
16: end {peakIndex}
```

现在来分析算法的时间复杂度 $T(n)$: 因为每一次二分都只需要做两次 (常数次) 比较, 所以 $T(n)$ 的递归方程为 $T(n) = T(n/2) + O(1)$, 根据主定理 ($a = 1, b = 2, d = 0$) 可解得 $T(n) = O(\log n)$.

[7] 具体的 C++ 程序代码见下一页

寻找单峰数组峰顶的 C++ 程序 (已在对应的**LeetCode 题目**上全部通过 42 个测试样例):

```cpp
class Solution {
public:
    int peakIndexInMountainArray(vector<int>& arr) {
        int n = arr.size(), res = 0;
        if(n == 3) {
            res = 1;
        }
        else {
            int low = 1, high = n - 2;  //起始指针不要放两边, 否则会越界
            while(low <= high) {
                int mid = (low + high) >> 1;
                if(arr[mid - 1] < arr[mid]) {
                    low = mid + 1;
                }
                else if(arr[mid - 1] > arr[mid]) {
                    high = mid - 1;
                }
            }
            res = high;  //此时 low > high, L[high] 就是对应的峰顶
        }
        return res;
    }
};
```

Problem 9

设 A 是 n 个不同元素组成且排好序的数组, 给定数 L 和 U, $L < U$, 设计一个优于 $O(n)$ 的算法, 找到 A 中满足 $L < x < U$ 的所有数 x.

Solution:

算法思路描述: 我们需要分类讨论:

- 当 $L >= A[n-1]$ 或 $U <= A[0]$ 时, 显然数集 $x = \emptyset$;
- 当 $L < A[0] < U < A[n-1]$ 时, 用下述的**lowerbound 算法**找到数组 A 中第一个大于等于 U 的元素索引 q, 则 $x = \{A[0], A[1], \cdots, A[q-1]\}$;
- 当 $L = A[0] < U < A[n-1]$ 时, 则 $x = \{A[1], A[2], \cdots, A[q-1]\}$;
- 当 $A[0] < L < U < A[n-1]$ 时, 则先用 **Problem 5** 的**upperbound 算法**找到数组 A 中第一个大于 L 的元素索引 p, 再用**lowerbound 算法**找到数组 A 中第一个大于等于 U 的元素索引 q, 这样就有

$$x = \{A[p], A[p+1], \cdots, A[q-1]\}$$

- 当 $A[0] < L < U = A[n-1]$ 时, 则用**upperbound 算法**找到数组 A 中第一个大于 L 的元素索引 p, 于是 $x = \{A[p], A[p+1], \cdots, A[n-2]\}$;
- 当 $A[0] < L < A[n-1] < U$ 时, 则用**upperbound 算法**找到数组 A 中第一个大于 L 的元素索引 p, 则 $x = \{A[p], A[p+1], \cdots, A[n-1]\}$;

Algorithm 8 二分查找 **lowerbound** 算法

Input: 长度为 n 的升序数组 $A[0, \cdots, n-1]$, 目标元素 target
Output: 第一个大于等于 target 的元素下标

```
1:  int low = 0, high = n − 1;
2:  while low <= high do
3:      int mid = (low + high) >> 1;
4:      if A[mid] < target then
5:          low = mid + 1;
6:      else
7:          high = mid − 1;
8:      end if
9:  end while
10: return low;                                  ▷ 返回所要求的下标
11: end {lowerbound}
```

此题主要在于分类讨论和第 4 种情况的求解, 其对应的 C++ 程序非常简单 (直接用一下 STL 的 lower_bound 和 upper_bound 二分查找函数即可), 故此处就不再罗列了. 而本文所构造的算法主要用到了两个二分查找的函数, 显然该算法的时间复杂度为 $O(\log n)$, 是优于 $O(n)$ 的.

Problem 10

在 n 枚 ($n \geq 3$) 硬币中有一枚重量不合格的硬币 (过轻或过重), 如果只有一架天平可以用来称重且称重的硬币数没有限制. 设计一个算法, 找出这枚不合格的硬币, 使得称重的次数最少 (优于 $O(n)$).

(提示: 分成 $n/3$ 或 $n/4$ 份, 至少两份数量相等)

Solution:

算法思想描述: 采用分治策略, 如果剩下的硬币个数小于 3(即 $n < 3$), 则将其逐个与正常硬币 (可以从拿走的硬币中随便选一个) 相比较, 不等的那枚就是不合格的硬币; 若 $n \geq 3$(即剩下的硬币个数至少是 3), 则将这些硬币分成 3 份, 其中两份个数一样 ($k = \lfloor n/3 \rfloor$), 另一份个数为 $n - 2k$. 将前两份称重对比, 若天平失衡则这两份中包含着异常硬币, 否则异常硬币在剩下的一份中. 递归进行划分处理, 直到 $n < 3$ 为止. 具体的算法伪代码见如下:

Algorithm 9 硬币检验算法 **CoinDet**(A, n)

Input: 共 n 枚硬币 (且含有 1 枚异常硬币) 的集合 A
Output: 1 枚异常硬币

```
1:  k := ⌊n/3⌋;
2:  if n < 3 then
3:      将其逐个与正常硬币 (从拿走的硬币中选一个) 相比较并找到异常硬币;        ▷ 递归终止条件
4:  else
5:      将集合 A 分成 3 份 X, Y, Z, 其中 |X| = |Y| = k, |Z| = n − 2k;
6:      if W(X) ≠ W(Y) then                                  ▷ W(X), W(Y) 分别为 X, Y 的重量
7:          A := X ∪ Y;
8:      else
9:          A := Z;
10:     end if
11:     n := |A|;
12:     CoinDet(A, n);
13: end if
14: end {CoinDet}
```

根据上述伪代码我们可以写出最坏情形下的时间复杂度 $T(n)$ 的递归方程[8]:

$$T(n) = T(2n/3) + O(1) \tag{7}$$

显然 $a = 1, b = 3/2, d = 0$, 故根据主定理可知 $T(n) = O(\log n)$.

最好情形下的时间复杂度 $T(n)$ 的递归方程为

$$T(n) = T(n/3) + O(1) \tag{8}$$

显然最好情形下 $a = 1, b = 3, d = 0$, 根据主定理可求得对应的时间复杂度也是 $O(\log n)$. 故不论最好最坏情形, 算法的时间复杂度都是 $O(\log n)$, 显然优于 $O(n)$.

[8] 子问题的规模要么是 Z 的规模 $n/3$(最好情形), 要么是 $X \cup Y$ 的规模 $2n/3$(最坏情形)

Problem 11

设 $A = \{a_1, a_2, \cdots, a_n\}$, $B = \{b_1, b_2, \cdots, b_m\}$ 是整数集合, 其中 $m = O(\log n)$, 设计一个优于 $O(nm)$ 的算法找出集合 $C = A \cap B$.

Solution:

方法 1 (基于哈希集合): 算法思想描述: 先用一个哈希集合存储数组 A 的元素[9], 然后遍历数组 B, 如果 B 中的元素在 A 对应的哈希集合中可以找到, 那么将该元素放进哈希集合 res 中, 最后将 res 转换为 vector 数组即可. 算法对应的 C++ 代码如下 (已在对应的**LeetCode T349**上全部通过 55 个测试样例):

```cpp
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> res;  //res 为哈希集合是为了对结果进行去重
        unordered_set<int> nums(nums1.begin(), nums1.end());
        for(auto num : nums2) {
            if(nums.find(num) != nums.end()) {  //如果找到了, 就放到 res 里
                res.insert(num);
            }
        }
        return vector<int>(res.begin(), res.end());  //将去重后得到的结果转为 vector 数组
    }
};
```

现在来分析一下此方法的时空复杂度: 使用一个集合存储数组 A 中的元素需要 $O(n)$ 的时间, 遍历集合 B 并判断元素是否在 A 的哈希集合中需要 $O(m)$ 的时间, 因此总的时间复杂度为 $O(m+n) = O(n)$. A 的哈希集合所占用的空间为 $O(n)$, 故空间复杂度为 $O(n)$.

方法 2 (排序 + 二分查找): 算法思想描述: 由于数组 B 比较短, 所以先对其进行排序, 然后遍历数组 A 的元素, 在排序后的 B 中使用二分查找来检索该元素, 若找到则放入 C 中. 算法伪码描述如下:

Algorithm 10 数组交集算法 **Intersection**

Input: 数组 $A[0, \cdots, n-1]$, $B[0, \cdots, m-1]$
Output: 数组 C, 其中 $C = A \cap B$

```
vector<int> C;
sort(B.begin(), B.end());                          ▷ 对数组 B 进行原地排序
for i = 0; i < n; i++ do
    bool flag = binary_search(B.begin(), B.end(), A[i]);
    if flag == true then
        C.push_back(A[i]);
    end if
end for
return C;
end {Intersection}                                 ▷ 算法的 C++ 代码跟伪代码非常相似, 就不列出了
```

现在来分析一下此方法的时空复杂度: 对 B 排序需要 $O(m \log m)$ 的时间, 遍历 + 二分查找需要消耗 $O(n \times \log m)$ 的时间, 所以总的时间复杂度为 $O(m \log m) + O(n \log m) = O((m+n) \log m) = O(n \log m) = O(n \log \log n)$; 而数组 B 排序所用到的栈空间为 $O(\log m)$, 对 B 二分查找所需的栈空间为 $O(\log m)$, 所以算法的空间复杂度为 $O(\log m) = O(\log \log n)$.

[9]哈希集合中的元素查找的时间复杂度是 $O(1)$, 且天然起到一个去重的作用

Problem 12

设 S 是 n 个不等的正整数的集合, n 为偶数, 给出一个算法将 S 划分为子集 S_1 和 S_2, 使得 $|S_1| = |S_2|$ 且 $\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$ 达到最大, 即两个子集元素之和的差达到最大 (要求时间复杂度 $T(n) = O(n)$).

Solution:

算法思想描述: 先利用 **PartSelect 算法**选取数组 S 的第 $n/2+1$ 小的元素, 并将该元素作为 pivot 并利用 **Partition 算法**来对数组 S 进行**一次划分**, 低区元素全部进入 S_2, 高区元素和 pivot 都进入到 S_1(由于 n 为偶数, 所以能够保证 $|S_1| = |S_2|$), 这样就能够确保 $\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$ 达到最大. 算法伪码见如下:

Algorithm 11 最大化子集和差算法 **MaxSubtract**

Input: 数组 $S[0,\cdots,n-1]$ ▷ n 为偶数, S 的元素彼此互异都为正整数

Output: $\max\limits_{|S_1|=|S_2|}\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$

```
vector<int> S1(n/2), S2(n/2);
int pivot = PartSelect(S, 0, n-1, n/2+1);   ▷ 求数组 S 第 n/2+1 小的元素, 可以认为是" 中位数"
int low = 0, high = n-1;                    ▷ 对撞双指针做一次划分
while low < high do
    while low < high && S[high] >= pivot do
        high--;
    end while
    swap(S[low], S[high]);
    while low < high && S[low] <= pivot do
        low++;
    end while
    swap(S[low], S[high]);
end while
int loc = low;                              ▷ 此时的 loc 即为 pivot 所处的最终下标
copy(S.begin(), S.begin() + loc, S2.begin());                    ▷ 低区进入 S2
copy(S.begin() + loc, S.begin() + (n - loc), S1.begin());        ▷ 高区和 pivot 进入 S1
return S1, S2;                              ▷ 注意到 loc 其实就是 n/2, 因此 n - loc 即为 n/2
return accumulate(S1.begin(), S1.end(), 0) - accumulate(S2.begin(), S2.end(), 0);
end {MaxSubtract}
```

现在来分析一下算法的时间复杂度: 调用 **PartSelect 算法**最坏需要 $O(n)$ 的时间, **Partition 算法** (核心是对撞双指针) 需要 $O(n)$ 的时间, 所以总的时间复杂度为 $T(n) = O(n) + O(n) = O(n)$.

Problem 13

考虑第三章 PPT NO.17 **Select**(A,k) 算法:

(1). 如果初始元素分组 $r=3$, 算法的时间复杂度如何? (2). 如果初始元素分组 $r=7$, 算法的时间复杂度如何?

Solution:

(1). 若 $r=3$, 则 3 个元素一组的中间值 u 是该数组的第 2 小元素, 此数组至少有 2 个小于等于 u; $\lfloor n/3 \rfloor$ 个中间值中至少有 $\lceil \lfloor n/3 \rfloor /2 \rceil$ 个小于等于这些中间值的中间值 v. 因此, 数组 A 中至少有 $2*\lceil \lfloor n/3 \rfloor /2 \rceil \geq \lfloor n/3 \rfloor \geq n/3-1$ 个元素小于等于 v. 即 A 中至多有 $n-(n/3-1)=2n/3+1$ 个元素大于 v. 同理, 至多有 $2n/3+1$ 个元素小于 v. 这样, 以 v 为划分元素所产生的新数组中至多有 $2n/3+1$ 个元素. 既可以认为子问题的规模为 $2n/3$. 求中位数的中位数所递归调用的规模为 $n/3$, 一趟快排和插入排序的所需时间为 cn. 综上, 时间复杂度的递推方程为

$$T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+cn \tag{9}$$

画出 $T(n)$ 的递归树, 见如下图2:

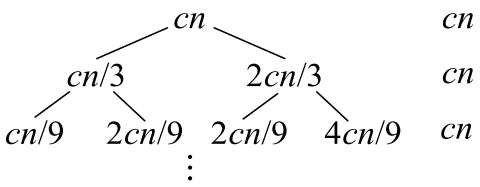


图 2: 递归树-1

而递归树的深度为 $\log n$, 而每一层的操作都是 cn, 所以时间复杂度 $T(n)=O(cn\log n)=O(n\log n)$;

(2). 若 $r=7$, 则 7 个元素一组的中间值 u 是该数组的第 4 小元素, 此数组至少有 4 个小于等于 u; $\lfloor n/7 \rfloor$ 个中间值中至少有 $\lceil \lfloor n/7 \rfloor /2 \rceil$ 个小于等于这些中间值的中间值 v. 因此, 数组 A 中至少有 $4*\lceil \lfloor n/7 \rfloor /2 \rceil \geq 2*\lfloor n/7 \rfloor \geq 2(n/7-1)$ 个元素小于等于 v. 即 A 中至多有 $n-2(n/7-1)=5n/7+2$ 个元素大于 v. 同理, 至多有 $5n/7+2$ 个元素小于 v. 这样, 以 v 为划分元素所产生的新数组中至多有 $5n/7+2$ 个元素. 既可以认为子问题的规模为 $5n/7$. 求中位数的中位数所递归调用的规模为 $n/7$, 一趟快排和插入排序的所需时间为 cn. 综上, 时间复杂度的递推方程为

$$T(n)=T\left(\frac{n}{7}\right)+T\left(\frac{5n}{7}\right)+cn \tag{10}$$

画出 $T(n)$ 的递归树, 见如下图3:

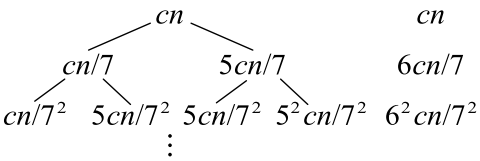


图 3: 递归树-2

故总的时间复杂度为:

$$T(n)=cn\left(1+\frac{6}{7}+\left(\frac{6}{7}\right)^2+\cdots\right)\leq cn\cdot\frac{1}{1-\frac{6}{7}}=7cn=O(n) \tag{11}$$

Problem 14

在 Internet 上的搜索引擎经常需要对信息进行比较, 比如可以通过某个人对一些事物的排名来估计他对各种不同信息的兴趣. 对于不同的排名结果可以用逆序来评价他们之间的差异. 考虑 $1, 2, \cdots, n$ 的排列 $i_1, i_2, ..., i_n$, 如果其中存在 i_j, i_k, 使得 $j < k$ 但 $i_j > i_k$, 那么就称 (i_j, i_k) 是这个排列的一个逆序. 一个排列含有逆序的个数称为这个排列的逆序数.

例如: 排列 $(2, 6, 3, 4, 5, 1)$ 含有 8 个逆序: (2,1),(6,3),(6,4),(6,5),(6,1),(3,1),(4,1),(5,1), 它的逆序数就是 8. 一个由 $1, 2, \cdots, n$ 组成的所有 $n!$ 个排列中, 最小的逆序数是 0, 对应的排列是 $(1, 2, 3, 4, \cdots, n)$, 最大的逆序数是 $n(n-1)/2$, 对应的排列是 $(n, n-1, \cdots, 2, 1)$. 逆序数越大的排列与原始排列的差异度越大. 利用二分归并排序算法设计一个计数给定排列逆序数的分治算法, 并对算法的时间复杂度进行分析.

Solution:

算法思想描述: 在二分归并排序算法中**顺带做了**计数逆序的工作, 在递归调用算法分别对子数组 L_1, L_2 排序时, 分别计算每个子数组内部的逆序; 在归并排好序的子数组 L_1, L_2 的过程中, 计算 L_1 的元素与 L_2 的元素之间产生的逆序. 具体来说, 合并两个有序数组所采用的是双指针算法, 每当遇到当前左指针 i 所指元素 $L_1[i] >$ 当前右指针 j 所指元素 $L_2[j]$ 时, 意味着 L_1 中从指针 i 到末尾的所有元素都与 $L_2[j]$ 构成逆序对, 于是此情形的逆序对增量为 $\text{mid} - i + 1$; 而当 $L_1[i] <= L_2[j]$ 时, 则此情形对于逆序对增量的贡献为 0(即没有逆序对). 算法伪代码见如下:

Algorithm 12 双指针合并算法 **Merge**(low, mid, high)

Input: 排序后的子数组 $A[\text{low}, \cdots, \text{mid}]$ 和 $A[\text{mid}+1, \cdots, \text{high}]$ ▷ 双指针法合并两个有序数组
Output: 合并好的升序数组 $A[\text{low}, \cdots, \text{high}]$ 及逆序数

```
1:  global count = 0;                                          ▷ count 是 int 型全局变量
2:  int k = 0, i = low, j = mid + 1;          ▷ i, j 是拣取游标 (即双指针), k 是向 B 存放元素的游标
3:  vector<int> B(high − low + 1);                             ▷ 借用临时数组 B
4:  while i ≤ mid && j ≤ high do                               ▷ 当两个集合都没有取尽时
5:      if A[i] ≤ A[j] then
6:          B[k++] := A[i++];                                  ▷ 此情形对逆序数的增量是没有贡献的
7:      else
8:          B[k++] := A[j++];
9:          count += mid − i + 1;    ▷ 此情形下, L1 中从指针 i 到末尾的所有元素都与 L2[j] 构成逆序对
10:     end if
11: end while
12: while i ≤ mid do                                  ▷ 当第二子组元素被取尽, 而第一组元素未被取尽时
13:     B[k++] := A[i++];
14: end while
15: while j ≤ high do                                 ▷ 当第一子组元素被取尽, 而第二组元素未被取尽时
16:     B[k++] := A[j++];
17: end while
18: copy(B.begin(), B.end(), A.begin() + low)          ▷ 将临时数组 B 中的元素拷贝给数组 A[low, ··· , high]
19: end{Merge}
```

Algorithm 13 归并排序主程序伪码 **MergeSort**(low, high)

Input: 待排序的数组 A 及下标 low,high
Output: 排序后的数组 $A[\text{low}, \cdots, \text{high}]$ 及 A 的逆序数

```
1: if low < high then
2:     int mid := ⌊(low + high)/2⌋          ▷ 求当前数组的分割点
3:     MergeSort(low, mid);                  ▷ 递归处理前半部分子数组
4:     MergeSort(mid + 1, high);             ▷ 递归处理后半部分子数组
5:     Merge(low, mid, high);                ▷ 归并两个排序后的子数组并计算逆序数
6: end if
7: end{MergeSort}
```

该算法对应的 C++ 程序 (已在对应的**LeetCode 题目**上全部通过 139 个测试样例) 为:

```cpp
class Solution {
public:
    int count; //定义为全局变量
    int reversePairs(vector<int>& nums) {
        count = 0;
        MergeSort(nums, 0, nums.size() - 1);
        return count;
    }

    void Merge(vector<int>& nums, int low , int mid, int high) {
        int i = low, j = mid + 1, k = 0;
        vector<int> temp(high - low + 1);
        while(i <= mid && j <= high) {
            if(nums[i] <= nums[j])
                temp[k++] = nums[i++];  //此情形下逆序数没有增量
            else {
                temp[k++] = nums[j++];
                count += mid - i + 1;  //此情形下逆序数才能有增量
            }
        }
        while(i <= mid) temp[k++] = nums[i++];  //把剩下的 nums 左半数组接上
        while(j <= high) temp[k++] = nums[j++];  //把剩下的 nums 右半数组接上
        copy(temp.begin(), temp.end(), nums.begin() + low);  //把临时数组的元素拷贝回 nums
        vector<int>().swap(temp);  //清除容器并最小化它的容量
    }

    void MergeSort(vector<int>& nums, int low, int high) {
        if(low < high) {
            int mid = (low + high) >> 1;
            MergeSort(nums, low, mid);
            MergeSort(nums, mid + 1, high);
            Merge(nums, low, mid, high);
        }
    }
};
```

显然该逆序数计算的算法伪码和 C++ 代码跟归并排序算法几乎一模一样, 就比归并排序在第 9 行多了个 count 的加法运算, 但算法的关键操作还是比较. 所以此算法的时间复杂度递推公式跟归并排序

一样:

$$T(n) = 2T(n/2) + O(n) \tag{12}$$

易知 $a = 2, b = 2, d = 1$, 故根据主定理可知 $T(n) = O(n \log n)$.

Problem 15

对玻璃瓶做强度试验, 设地面高度为 0, 从 0 向上有 n 个高度, 记为 $1, 2, \cdots, n$, 其中任何两个高度之间的距离都相等. 如果一个玻璃瓶从高度 i 落到地上没有摔碎, 但从高度 $i+1$ 落到地上摔碎了, 那么就将玻璃瓶的强度记为 i.

(1). 假设每种玻璃瓶只有 1 个测试样品, 设计算法来测试出每种玻璃瓶的强度. 以测试次数作为算法的时间复杂度, 估计算法的复杂度;

(2). 假设每种玻璃瓶有足够多的相同的测试样品, 设计算法使用最少的测试次数来完成测试;

(3). 假设每种玻璃瓶只有 2 个相同的测试样品, 设计次数尽可能少的算法完成测试.

Solution:

(1). 顺序从下到上测试, 一次一个高度, 最坏情况下时间复杂度为 $T(n) = O(n)$;

(2). 因为高度越高, 玻璃瓶越容易碎, 其实可以理解为" 升序数组". 因此我们可以考虑用二分法: 先在高度 $n/2$ 测试玻璃瓶, 如果摔碎了, 则玻璃瓶的强度位于 $[1, n/2-1)$(在该区间继续二分搜索即可); 若没摔碎, 则玻璃瓶的强度位于 $[n/2+1, n]$(在该区间继续二分搜索即可). 显然, 该二分搜索的时间复杂度为 $T(n) = O(\log n)$;

(3). 不失一般性, 不妨设 $\sqrt{n}$ 为整数, 则可以将 $1, 2, 3, \cdots, n$ 这些 n 个高度分成 $\sqrt{n}$ 组[10]. 那么第 j 组 $(j = 1, 2, \cdots, \sqrt{n})$ 所含有的高度有

$$(j-1)\sqrt{n}+1, (j-1)\sqrt{n}+2, \cdots, (j-1)\sqrt{n}+\sqrt{n}, \quad j = 1, 2, \cdots, \sqrt{n} \tag{13}$$

先拿第一个瓶子测试: 从下往上, 按照每组的最大高度 (即 $j\sqrt{n}, j = 1, 2, \cdots, \sqrt{n}$) 进行测试. 如果前 $j-1$ 组的测试中瓶子都没有碎, 而在第 j 组的测试中碎了, 则强度显然位于第 j 组的 $\sqrt{n}$ 个高度中. 于是, 至多经过 $\sqrt{n}$ 此测试, 待检查的高度范围就缩减到原来的 $\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}}$ 倍;

再拿第二个瓶子测试: 在第 j 组的 $\sqrt{n}$ 个高度中, 从下往上测试玻璃瓶的强度, 至多经过 $\sqrt{n}$ 次测试, 就可以得到玻璃瓶的强度.

现在来分析算法的时间复杂度: 显然第一个瓶子测验至多需要耗时 $O(\sqrt{n})$, 第二个瓶子测试也至多需要耗时 $O(\sqrt{n})$, 于是总的算法时间复杂度为

$$T(n) = O(\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n}) \tag{14}$$

[10] 如果 $\sqrt{n}$ 不是整数, 则取 $\lfloor \sqrt{n} \rfloor$ 个整组, 剩下的单独成一组

Problem 16

1. 使用主定理求解以下递归方程:

(1). $\begin{cases} T(n) = 9T(n/3) + n \\ T(1) = 1 \end{cases}$; (2). $\begin{cases} T(n) = 5T(n/2) + (n\log n)^2 \\ T(1) = 1 \end{cases}$; (3). $\begin{cases} T(n) = 2T(n/2) + n^2\log n \\ T(1) = 1 \end{cases}$

Solution:

(1). 易知 $a = 9, b = 3, d = 1, f(n) = n$, 由于 $f(n) = n = O\left(n^{2-\epsilon}\right)$, 故根据主定理可知: $T(n) = \Theta(n^2)$;

(2). 易知 $a = 5, b = 2, f(n) = n^2\log^2 n = O(n^{\log_2 5-\epsilon})$, 故根据主定理可知 $T(n) = \Theta\left(n^{\log_2 5}\right)$;

(3). 易知 $a = 2, b = 2, f(n) = n^2\log n = \Omega\left(n^{1+\epsilon}\right)$, 而且

$$af(n/b) = 2(n/2)^2\log(n/2) = n^2/2(\log n - 1) \leq 0.5n^2\log n \quad (c = 1/2 < 1)$$

故根据主定理可知 $T(n) = \Theta(f(n)) = \Theta(n^2\log n)$.

2. 使用递归树求解: $\begin{cases} T(n) = T(n/2) + T(n/4) + cn \\ T(1) = 1 \end{cases}$;

Solution:

递归树见如下图4:

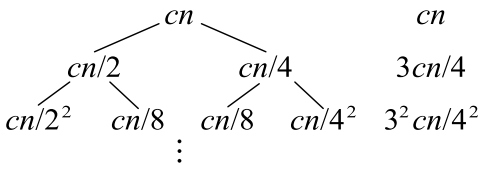


图 4: 递归树

故总的时间复杂度为:

$$T(n) = cn\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \cdots\right) \leq cn \cdot \frac{1}{1-\frac{3}{4}} = 4cn = O(n) \tag{15}$$

3. 使用迭代递归法求解: (1). $\begin{cases} T(n) = T(n-1) + \log 3^n \\ T(1) = 1 \end{cases}$; (2). $\begin{cases} T(n) = T(n-1) + 1/n \\ T(1) = 1 \end{cases}$.

Solution:

(1). 易知

$$T(n) = T(n-1) + \log 3^n = T(n-2) + \log 3^{n-1} + \log 3^n \tag{16}$$

$$\cdots = T(1) + \log 3^2 + \log 3^3 + \cdots + \log 3^n \tag{17}$$

$$= 1 + \log\left(3^{2+3+\cdots+n}\right) = 1 + \log\left(3^{(n+2)(n-1)/2}\right) = \Theta\left(n^2\right) \tag{18}$$

(2). 易知

$$T(n) = T(n-1) + \frac{1}{n} = T(n-2) + \frac{1}{n-1} + \frac{1}{n} \tag{19}$$

$$\cdots = T(1) + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n} = \sum_{i=1}^{n}\frac{1}{i} = \Theta(\gamma + \log n) = \Theta(\log n) \tag{20}$$

注意, 其中我们用到了 γ 常数的数学结论: $\gamma = \lim\limits_{n\to+\infty}\left(\sum\limits_{i=1}^{n}\frac{1}{i} - \ln n\right)$.

中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 4 课程作业解答

2022 年 10 月 7 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

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 $t_i, 1 \leq i \leq n$. 应该如何安排 n 个顾客的服务次序才能使总的等待时间达到最小? 总的等待时间是各顾客等待服务的时间的总和. 试给出你的做法的理由 (证明).

Solution: 我们使用贪心算法求解该问题, 具体的**贪心策略**为: **服务时间较短的优先安排**. 假设调度 f 的顺序为 $i_1, i_2, \cdots, i_n$, 那么 i_k 的等待时间为 $\sum_{j=1}^{k-1} t_{i_j}$, 总的等待时间为

$$T(f) = \sum_{i=1}^{n} (n-i)\, t_{i_j} \tag{1}$$

根据贪心策略, 需要先排序使得 $t_1 \leq t_2 \leq \cdots \leq t_n$, 按照 $1, 2, \cdots, n$ 的顺序安排服务. 则调度 f^* 的总等待时间为

$$T(f^*) = \sum_{i=1}^{n} (n-i)\, t_i \tag{2}$$

于是我们可以给出对应的算法伪码:

Algorithm 1 Service 算法

Input: 服务时间的数组 $T[1, \cdots, n] = [t_1, t_2, \cdots, t_n]$
Output: 调度 f, $f(i)$ 为第 i 个顾客的开始服务时刻, $1 \leq i \leq n$

```
sort(T.begin(), T.end());          ▷ 按照服务时间从小到大的顺序排列
f(1) := 0;
for i := 2 to n do
    f(i) := f(i − 1) + t_{i−1};
end for
return f;
end {Service}
```

由于**算法主要在于排序**, 故其最坏情况下的时间复杂度为 $O(n \log n)$.
下面证明: **对任何输入, 对服务时间短的顾客优先安排将得到最优解.**

证明. 使用**交换论证**的方法: 假设存在某个最优解 g 且存在 $i, j \in A, i < j$, 但是 i 在 j 之后得到服务. 那么在 g 的安排中一定存在相邻安排的顾客 i 和 j, 使得 $i < j$ 但 i 在 j 之后得到服务. 交换 i 和 j 得到新的服务顺序 g', 那么

$$T(g) - T\left(g'\right) = t_j - t_i \geq 0 \tag{3}$$

总的等待时间将不会增加, 因此 g' 也是最优解. 至多经过 $n(n-1)/2$ 次这样的交换, 就可以将 g 转换为算法的解 f^*, 从而证明了 f^* 是最优解. □

Problem 2

字符 $a \sim h$ 出现的频率分布恰好是前 8 个 Fibonacci 数, 它们的 Huffman 编码是什么? 将结果推广到 n 个字符的频率分布恰好是前 n 个 Fibonacci 数的情形. Fibonacci 数的定义为: $F_0 = 1, F_1 = 1, F_n = F_{n-2} + F_{n-1} (n \geq 1)$.

Solution: 对应的 Huffman 树如下图1所示. 故可知 Huffman 编码为

$$h : 0,\ g : 10,\ f : 110,\ e : 1110,\ d : 11110,\ c : 111110,\ b : 1111110,\ a : 1111111 \tag{4}$$

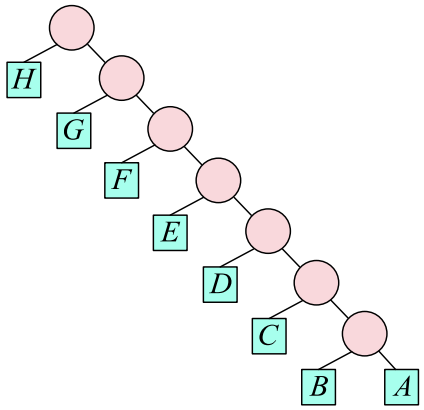


图 1: Huffman 树

为了推广, 需要先证明一个结论: 设 $f_i (i \geq 1)$ 为 Fibonacci 数列, 则

$$\sum_{i=1}^{k} f_i \leq f_{k+2} \tag{5}$$

证明. 采用数学归纳法: 当 $k = 1$ 时, 命题显然成立; 假设 $k = n$ 时命题成立, 则 $k = n + 1$ 时, 则有

$$\sum_{i=1}^{n+1} f_i = \sum_{i=1}^{n} f_i + f_{n+1} \leq f_{n+2} + f_{n+1} = f_{n+3} \tag{6}$$

于是 $\forall k \in \mathbb{N}^*$, 不等式 (5) 都成立. □

因此根据上述结论, 前 k 个字符合并后子树的根权值小于等于第 $k+2$ 个 Fibonacci 数. 根据 Huffman 算法, 他将继续参加与第 $k + 1$ 个字符的合并. 因此 n 个字符的 Huffman 编码按照频数从小到大依次为

$$\underbrace{11\cdots1}_{n-1个1},\ \underbrace{11\cdots10}_{n-2个1},\ \underbrace{11\cdots0}_{n-3个1},\ \cdots,\ 10,\ 0 \tag{7}$$

即第 $i (i > 1)$ 个字母的编码为 $\underbrace{11\cdots10}_{n-i个1}$.

Problem 3

设 $p_1, p_2, \cdots, p_n$ 是准备存放到长为 L 的磁带上的 n 个程序, 程序 p_i 需要的带长为 a_i. 设 $\sum_{i=1}^{n} a_i > L$, 要求选取一个能放在带上的程序的最大子集合 (即其中含有最多个数的程序)Q. 构造 Q 的一种贪心策略是按 a_i 的非降次序将程序计入集合.

(1). 证明这一策略总能找到最大子集 Q, 使得 $\sum_{p_i \in Q} a_i \leq L$;

(2). 设 Q 是使用上述贪心算法得到的子集合, 磁带的利用率可以小到何种程度?

(3). 试说明 (1) 中提到的设计策略不一定能得到使 $\sum_{p_i \in Q} a_i / L$(即磁带的利用率) 取最大值的子集合.

Solution:

(1).

证明. 还是采用**交换论证**: 易知只要存放程序名称相同 (不管次序) 的任何方法都是同样的解. 不妨设最优解为 $\mathrm{OPT} = \left\{i_1, i_2, \cdots, i_j\right\}, i_1 < i_2 < \cdots < i_j, j < n$. 如果

$$\left\{i_1, i_2, \cdots, i_j\right\} = \{1, 2, \cdots, j\} \tag{8}$$

那么算法的解就是最优解. 假设 $\left\{i_1, i_2, \cdots, i_j\right\} \neq \{1, 2, \cdots, j\}$, 设 $i_1 = 1, i_2 = 2, \cdots, i_{t-1} = t-1, i_t > t$. 用 t 替换 i_t, 那么得到的解 I^* 占用的存储空间与解 OPT 占用空间的差值为

$$S\left(I^*\right) - S\left(\mathrm{OPT}\right) = a_t - a_{i_t} \leq 0 \tag{9}$$

因此 I^* 也是最优解, 但是它比解 OPT 减少了一个标号不相等的程序. 对于解 OPT, 从第一个标号不等的程序开始, 至多经过 j 次替换, 就得到最优解 $\{1, 2, \cdots, j\}$. 显然算法的时间复杂度为

$$T(n) = O(n \log n) + O(n) = O(n \log n) \tag{10}$$

□

(2). 磁带的利用率最小可以小到 0, 比如 $\forall 1 \leq i \leq n, a_i > L$.

(3). 按照题中的贪心策略虽然能够保障所装的程序最多, 但对应的空间利用率不一定最大. 具体例子为: 设 $\{a_1, a_2, \cdots a_s\}$ 为 Q 的最大子集, 可能会有: 用 a_{s+1} 替换 a_s, 子集合变为 $\{a_1, a_2, \cdots a_{s-1}, a_{s+1}\}$ 并且满足 $\sum_{k=1}^{s-1} a_k + a_{s+1} < L$. 虽然程序个数仍为 s 个, 但利用率却增加了. 因此上述贪心策略并不一定能求得使利用率最大化的最优解.

(4). 如果要求磁带利用率最大, 这个问题的本质上时 0-1 背包问题, 每个程序相当于物品, 其重量和价值就是所需要的存储带长, 背包的重量限制等于磁带容量 L. 可以使用动态规划 (DP) 算法来解决: 设 $F_k(y)$ 表示考虑前 k 个程序, 磁带空间为 y 时的最大存储量. 递推方程为

$$F_k(y) = \begin{cases} \max\left\{F_{k-1}(y), F_{k-1}(y - a_k) + a_k\right\}, a_k \leq y \leq L \\ F_{k-1}(y), a_k > y \end{cases} \tag{11}$$

其中 $k > 0$ 且 $F_0(y) = 0\,(0 \leq y \leq L), F_k(0) = 0, F_k(y) = -\infty\,(y < 0)$. 可以设定如下标记函数 $i_k(y)$ 用于追踪解:

$$i_k(y) = \begin{cases} k, \text{若} F_{k-1}(y) \leq F_{k-1}(y - a_k) + a_k, a_k \leq y \leq L\,(k > 1) \\ i_{k-1}(y), \text{否则} \end{cases}, \; i_1(y) = \begin{cases} 1, \text{若} y \geq a_1 \\ 0, \text{否则} \end{cases} \tag{12}$$

该 DP 算法在最坏情形下的时间复杂度为 $T(n) = O(nL)$.

Problem 4

写出 Huffman 编码的伪代码, 并编程实现.

Solution: 伪代码见如下算法2:

Algorithm 2 HuffmanCode 算法

Input: 待编码的数组 $A[1, \cdots, n]$
Output: 数组 A 的 Huffman 编码

1: **local** h; ▷ 最小化堆, 内含元素为结点类型, 堆初始为空
2: **int** i;
3: **Node** p, q, r; ▷ 结点数据结构, 内含数值以及分别指向左、右儿子的两个指针
4: **for** $i = 1; i <= n; i$++ **do** ▷ 将数组 A 中的所有元素插入堆
5: **Insert**$(h, A[i])$;
6: **end for**
7: **while** $|h| > 1$ **do** ▷ h 元素个数大于 1
8: $p =$ **DeleteMin**(h); $q =$ **DeleteMin**(h); ▷ 移除最小的两个结点
9: $r = p + q$; r.left $= \min(p, q)$; r.right $= \max(p, q)$; ▷ 构造新的结点 r, 其值为 p, q 值之和
10: **Insert**(h, r); ▷ 将 r 插入堆 h 中
11: **end while**
12: $p =$ **DeleteMin**(h); ▷ 取出最后一个结点, 此节点即为 Huffman 树的根节点
13: **end {HuffmanCode}**

其对应的 C++ 程序代码见如下:

```cpp
#include<iostream>
#include<string>
using namespace std;

struct Node
{
    double weight;
    string ch;
    string code;
    int lchild, rchild, parent;
};

void Select(Node huffTree[], int *a, int *b, int n)//找权值最小的两个 a 和 b
{
    int i;
    double weight = 0; //找最小的数
    for (i = 0; i < n; i++)
    {
        if (huffTree[i].parent != -1)      //判断节点是否已经选过
            continue;
        else
        {
            if (weight == 0)
            {
                weight = huffTree[i].weight;
```

```
                *a = i;
            }
            else
            {
                if (huffTree[i].weight < weight)
                {
                    weight = huffTree[i].weight;
                    *a = i;
                }
            }
        }
    }
    weight = 0; //找第二小的数
    for (i = 0; i < n; i++)
    {
        if (huffTree[i].parent != -1 || (i == *a))//排除已选过的数
            continue;
        else
        {
            if (weight == 0)
            {
                weight = huffTree[i].weight;
                *b = i;
            }
            else
            {
                if (huffTree[i].weight < weight)
                {
                    weight = huffTree[i].weight;
                    *b = i;
                }
            }
        }
    }
    int temp;
    if (huffTree[*a].lchild < huffTree[*b].lchild)  //小的数放左边
    {
        temp = *a;
        *a = *b;
        *b = temp;
    }
}

void Huff_Tree(Node huffTree[], int w[], string ch[], int n)
{
    for (int i = 0; i < 2 * n - 1; i++) //初始过程
    {
        huffTree[i].parent = -1;
        huffTree[i].lchild = -1;
        huffTree[i].rchild = -1;
        huffTree[i].code = "";
    }
```

```
    for (int i = 0; i < n; i++)
    {
        huffTree[i].weight = w[i];
        huffTree[i].ch = ch[i];
    }
    for (int k = n; k < 2 * n - 1; k++)
    {
        int i1 = 0;
        int i2 = 0;
        Select(huffTree, &i1, &i2, k); //将 i1, i2 节点合成节点 k
        huffTree[i1].parent = k;
        huffTree[i2].parent = k;
        huffTree[k].weight = huffTree[i1].weight + huffTree[i2].weight;
        huffTree[k].lchild = i1;
        huffTree[k].rchild = i2;
    }
}

void Huff_Code(Node huffTree[], int n)
{
    int i, j, k;
    string s = "";
    for (i = 0; i < n; i++)
    {
        s = "";
        j = i;
        while (huffTree[j].parent != -1) //从叶子往上找到根节点
        {
            k = huffTree[j].parent;
            if (j == huffTree[k].lchild) //如果是根的左孩子, 则记为 0
            {
                s = s + "0";
            }
            else
            {
                s = s + "1";
            }
            j = huffTree[j].parent;
        }
        cout << " 字符 " << huffTree[i].ch << " 的编码: ";
        for (int l = s.size() - 1; l >= 0; l--)
        {
            cout << s[l];
            huffTree[i].code += s[l]; //保存编码
        }
        cout << endl;
    }
}
```

```
string Huff_Decode(Node huffTree[], int n,string s)
{
    cout << " 解码后为: ";
    string temp = "", str = "";//保存解码后的字符串
    for (int i = 0; i < s.size(); i++)
    {
        temp = temp + s[i];
        for (int j = 0; j < n; j++)
        {
            if (temp == huffTree[j].code)
            {
                str = str + huffTree[j].ch;
                temp = "";
                break;
            }
            else if (i == s.size()-1 && j == n-1 && temp != "")//全部遍历后没有
            {
                str = " 解码错误! ";
            }
        }
    }
    return str;
}

int main()
{
    //编码过程
    const int n = 5;
    Node huffTree[2 * n];
    string str[] = { "A", "B", "C", "D", "E"};
    int w[] = { 30, 30, 5, 20, 15 };
    Huff_Tree(huffTree, w, str, n);
    Huff_Code(huffTree, n);
    //解码过程
    string s;
    cout << " 输入编码: ";
    cin >> s;
    cout << Huff_Decode(huffTree, n, s) << endl;
    system("pause");
    return 0;
}
```

Problem 5

设有一条边远山区的道路 AB, 沿着道路 AB 分布着 n 所房子. 这些房子到 A 的距离分别是 $d_1, d_2, \cdots, d_n (d_1 < d_2 < \cdots < d_n)$. 为了给所有房子的用户提供移动电话服务, 需要在这条道路上设置一些基站. 为了保证通讯质量, 每所房子应该位于距离某个基站的 4km 范围内. 设计一个算法找基站的位置, 并且使得基站的总数最少, 并证明算法的正确性.

Solution: 使用贪心法, 令 $a_1, a_2, \cdots$ 表示基站的位置. 贪心策略为: 首先令 $a_1 = d_1 + 4$. 对 $d_2, d_3, \cdots, d_n$ 依次检查, 找到下一个不能被该基站覆盖的房子. 如果 $d_k \leq a_1 + 4$ 但 $d_{k+1} > a_1 + 4$, 那么第 $k+1$ 个房子不能被基站覆盖, 于是取 $a_2 = d_{k+1} + 4$ 作为下一个基站的位置. 照此下去, 直到检查完 d_n 为止. 伪代码见如下算法3:

Algorithm 3 Location 算法

Input: 距离数组 $d[1, \cdots, n] = [d_1, d_2, \cdots, d_n]$, 满足 $d[1] < d[2] < \cdots < d[n]$
Output: 基站位置的数组 a

```
1: a[1] := d[1] + 4; k := 1;
2: for j = 2; j <= n; j++ do
3:     if d[j] > a[k] + 4 then
4:         a[++k] := d[j] + 4;
5:     end if
6: end for
7: return a;
8: end {Location}
```

结论: 对任何正整数 k, 存在最优解包含算法前 k 步选的的基站位置.

证明. $k = 1$, 存在最优解包含 $a[1]$. 如若不然, 有最优解 OPT, 其第一个位置是 $b[1]$ 且 $b[1] \neq a[1]$, 那么 $d_1 - 4 \leq b[1] < d_1 + 4 = a[1]$. $b[1]$ 覆盖的是距离在 $[d_1, b[1] + 4]$ 之间的房子. $a[1]$ 覆盖的是距离在 $[d_1, a[1] + 4]$ 的房子. 因为 $b[1] < a[1]$, 且 $b[1]$ 覆盖的房子都在 $a[1]$ 覆盖的区域内, 故用 $a[1]$ 替换 $b[1]$ 得到的仍是最优解;

假设对于 k, 存在最优解 A 包含算法前 k 步选择的基站位置, 即

$$A = \{a[1], a[2], \cdots, a[k]\} \cup B \tag{13}$$

其中 $a[1], a[2], \cdots, a[k]$ 覆盖了距离为 $d_1, d_2, \cdots, d_j$ 的房子. 那么 B 是关于 $L = \{d_{j+1}, d_{j+2}, \cdots, d_n\}$ 的最优解. 否则, 存在关于 L 的更优解 B^*, 那么用 B^* 替换 B 就会得到 A^* 且 $|A^*| < |A|$, 这与 A 是最优解相矛盾. 根据归纳假设可得知 L 有一个最优解 $B' = \{a[k+1], \cdots\}, |B'| = |B|$. 于是

$$A' = \{a[1], a[2], \cdots, a[k]\} \cup B' = \{a[1], a[2], \cdots, a[k], a[k+1], \cdots\} \tag{14}$$

且 $|A'| = |A|$, 故 A' 也是最优解, 从而命题对于 $k+1$ 也成立. 故根据数学归纳法可知, 对任何正整数 k 命题都成立. □

算法的关键操作是 for 循环, 而循环体内部的操作都是常数时间, 因此算法在最坏情况下的时间复杂度为 $O(n)$.

Problem 6

有 n 个进程 $p_1, p_2, .., p_n$, 进程 p_i 的开始时间为 $s[i]$, 截止时间为 $d[i]$. 可以通过检测程序 Test 来测试正在运行的进程, Test 每次测试时间很短, 可以忽略不计, 即如果 Test 在时刻 t 测试, 那么它将对满足 $s[i] \leq t \leq d[i]$ 的所有进程同时取得测试数据. 问: 如何安排测试时刻, 使得对每个进程至少测试一次, Test 测试的次数达到最少? 设计算法并证明正确性, 分析算法复杂度.

Solution: 贪心策略: 将进程按照 ddl 进行排序. 取第 1 个进程的 ddl 作为第一个测试点, 然后顺序检查后续能够被这个测试点检测的进程 (这些进程的开始时间 $\leq$ 测试点), 直到找到下一个不能被测试到的进程为止. 伪码见如下算法4:

Algorithm 4 Test 算法

Input: 开始时间的数组 $s[1, \cdots, n]$, 截止时间的数组 $d[1, \cdots, n]$
Output: 数组 t: 顺序选定的测试点构成的数组

```
1: 将进程按照 d[i] 递增的顺序进行排序 (使得 d[1] ≤ d[2] ≤ ··· ≤ d[n]);
2: i := 1; t[i] := d[1]; j := 2                    ▷ 第一个测试点是最早结束进程的 ddl
3: while j ≤ n && s[j] ≤ t[i] do                   ▷ 检查进程 j 是否可以在时刻 t[i] 被测试
4:     j++;
5: end while
6: if j > n then
7:     return t;
8: else
9:     t[++i] := d[j++], goto 3;                   ▷ 找到待测进程中结束时间最早的进程 j
10: end if
11: end {Test}
```

结论: 对于任意正整数 k, 存在最优解包含算法前 k 步选择的测试点.

证明. $k = 1$ 时, 设 $S = \{t[i_1], t[i_2], \cdots\}$ 是最优解, 不妨设 $t[i_1] < t[1]$. 设 p_u 是在时刻 $t[i_1]$ 被测到的任意进程, 那么 $s(u) \leq t[i_1] \leq d[u]$, 从而有

$$s[u] \leq t[i_1] < t[1] = d[1] \leq d[u] \tag{15}$$

因此 p_u 也可以在 $t[1]$ 时刻被测试. 于是在 S 中用 $t[1]$ 替换掉 $t[i_1]$ 后也可得到一个最优解.

假设对于任意 k, 算法在前 k 步选择了 k 个测试点 $t[1], t[i_2], \cdots, t[i_k]$ 且存在最优解

$$T = \{t[1], t[i_2], \cdots, t[i_k]\} \cup T' \tag{16}$$

设算法前 k 步选择的测试点不能测到的进程构成集合 $Q \subseteq P$, 其中 P 为全体进程集合. 不难证明 T' 是子问题 Q 的最优解[1]. 根据归纳假设可得知, $\exists Q$ 的最优解 T^* 包含测试点 $t[i_{k+1}]$, 即

$$T^* = \{t[i_{k+1}]\} \cup T'' \tag{17}$$

因此有

$$\{t[1], t[i_2], \cdots, t[i_k]\} \cup T^* = \{t[1], t[i_2], \cdots, t[i_{k+1}]\} \cup T'' \tag{18}$$

也是原问题的最优解, 根据归纳法可知命题成立. □

算法的时间复杂度为 $T(n) = O(n \log n) + O(n) = O(n \log n)$.

[1] 反证法: 假设 T' 不是子问题 Q 的最优解, 则会推出 T 不是最优解, 显然矛盾.

Problem 7

设有作业集合 $J = \{1, 2, \cdots, n\}$, 每项作业的加工时间都是 1, 所有作业的截止时间是 D. 若作业 i 在 D 之后完成, 则称为被延误的作业, 需赔偿罚款 $m(i)(i = 1, 2, \cdots, n)$, 这里 D 和 $m(i)$ 都是正整数, 且 n 项 $m(i)$ 彼此不等. 设计一个算法求出使总罚款最小的作业调度算法, 证明算法的正确性并分析时间复杂度.

Solution: 贪心策略: 优先安排前 D 个罚款最多的作业. 正确性证明需要利用交换论证的方法, 先给出以下结论:

结论: 设作业调度 f 的安排次序是 $\langle i_1, i_2, \cdots, i_n \rangle$, 那么罚款为

$$F(f) = \sum_{k=D+1}^{n} m(i_k) \tag{19}$$

证明. 显然最优调度没有空闲时间, 不妨假设作业是连续安排的. 因为每项作业的加工时间都是 1, 再截止时间 D 之前可以完成 D 项作业. 只有在 D 之后安排的 $n - D$ 项作业 (即 $i_{D+1}, i_{D+2}, \cdots, i_n$ 都是被罚款的作业). □

根据上述结论可以推出: 令 S 是 $n - D$ 项罚款最少的作业构成的集合.

(1). 对于 S(或 $J \backslash S$) 中的作业 i 和 j, 交换 i, j 的加工顺序不影响总罚款;

(2). 对于作业 i 和 $j, m(i) < m(j)$, 调度 f 将 i 安排在 D 之前, j 安排在 D 之后, 那么交换作业 i 和 j 得到的调度 g, 则 g 的罚款会减少, 这是因为

$$F(g) - F(f) = m(i) - m(j) < 0 \tag{20}$$

根据上述分析可以看出, 把罚款最小的 $n - D$ 项作业安排在最后会使得总罚款金额达到最小.

于是可以设计出以下算法5

Algorithm 5 Work 算法

Input: 罚款数组 $m[1, \cdots, n]$, 作业集合 J

Output: 作业调度 f

1: 利用 **PartSelect** 算法从 $m(1), m(2), \cdots, m(n)$ 中选出第 $n - D$ 小的元素 (记作 m^*);
2: 用 m^* 与数组 m 中剩下的 $n - 1$ 个元素进行比较, 找出比 m^* 小的 $n - D - 1$ 个元素;
3: 将步骤 2 中的元素和 m^* 所对应的作业从 D 时刻开始以任意顺序进行加工;
4: 将剩下的 D 项作业以任意顺序安排在 $0, 1, \cdots, D - 1$ 时刻加工;
5: **end {Work}**

现在来分析一下这个算法的时间复杂度: 第 1 行调用了 **PartSelect** 算法, 最坏需要 $O(n)$ 的时间[2]; 算法中的第 2 步对剩余数组元素做一次遍历也需要 $O(n)$ 的时间, 故总的时间复杂度为 $T(n) = O(n) + O(n) = O(n)$. 其实也可以先将作业按照 $m(i)$ 由大到小进行排序, 然后直接安排前 D 项作业即可, 但是排序所需的时间复杂度为 $O(n \log n)$, 效率上显然不如上述的 **work** 算法.

[2]**改进后**的 PartSelect 算法在最坏情形下的时间复杂度为 $O(n)$.

Problem 8

举出反例证明: 本章开始例 1 贪心规则找零钱算法 (目标: 零币数量最少；规则: 尽量先找币值大的), 在零钱种类不合适时, 贪心算法结果不正确.

Solution: 比如, 如果提供找零的面值是 11,5,1, 找零 15. 使用贪心算法找零方式为 11+1+1+1+1, 需要五枚硬币. 而最优解为 5+5+5, 只需要 3 枚硬币.

至此, Chap 4 的作业解答完毕.

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 5 课程作业解答

2022 年 10 月 22 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

最大子段和问题: 给定整数序列 $a_1, a_2, \cdots, a_n$, 求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值:

$$\max\left\{0, \max_{1\le i\le i\le n}\sum_{k=i}^{j} a_k\right\}$$

(1). 已知一个简单算法如下:

```cpp
int Maxsum(int n, vector<int> a, int& besti, int& bestj) {
    int sum = 0;
    for(int i = 1; i <= n; i++) {
        int suma = 0;
        for(int j = i; j <= n; j++) {
            suma += a[j];
            if(suma > sum) {
                sum = suma;
                besti = i;
                bestj = j;
            }
        }
    }
    return sum;
}
```

试分析该算法的时间复杂性;

(2). 试用分治算法解最大子段和问题, 并分析算法的时间复杂性;

(3). 试说明最大子段和问题具有最优子结构性质, 并设计一个动态规划算法求解最大子段和问题, 分析算法的时间复杂度. (提示: 可令 $b(j) = \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k, j = 1, 2, \cdots, n$)

Solution: (1). 显然第 2 层 for 循环里面的操作都是常数次的 (记为 C), 所以算法总的关键操作数为

$$\sum_{i=1}^{n}\sum_{j=i}^{n} C = C\sum_{i=1}^{n}(n-i+1) = \frac{1}{2}C\left(n^2+n\right)$$

故显然时间复杂度为 $T(n) = O(n^2)$.

(2). 采用分治算法, 则考虑: 先将数组从中间 mid 切开. 此时, 最大和的子段可能出现在左半边, 也可能出现在右半边, 也有可能横跨左右两个子数组. 所以需要返回这三种情况下所分别对应的子问题解的最大值.

当最大和的子段出现在左半边 (右半边同理) 时, 继续分中点递归直至分解到只有一个数为止;

当最大和的子段横跨 mid 左右时, 只需分别求解左子数组的最优后缀和以及右子数组的最优前缀和. 这三种情形下的最大值即为整个数组的最大子段和. 具体分治算法的 C++ 代码见后页. 从 C++ 代码可以看出最坏情形下的时间复杂度的递推式和结果分别为

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) + O(n)$$

故根据主定理 ($\log_b(a) = \log_2(2) = 1 = d$) 可知 $T(n) = O(n\log n)$.

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        return maxSum(nums, 0, nums.size() - 1);
    }

    int maxSum(vector<int> nums, int left, int right) {
        if(left == right) {
            return nums[left];
        }
        int mid = (left + right) >> 1;
        int LeftRes = maxSum(nums, left, mid);  //可能情况 1
        int RightRes = maxSum(nums, mid + 1, right);  //可能情况 2
        /* 可能情况 3: 最大 (和) 子段出现横跨 mid(切分点) 两边 */
        int lbs = INT_MIN, sum1 = 0;
        for(int i = mid; i >= left; i--) { //一次向前扫描来求得最大后缀和
            sum1 += nums[i];
            if(sum1 > lbs) {
                lbs = sum1;
            }
        }
        int rbs = INT_MIN, sum2 = 0;
        for(int i = mid + 1; i <= right; i++) { //一次向后扫描来求得最大前缀和
            sum2 += nums[i];
            if(sum2 > rbs) {
                rbs = sum2;
            }
        }
        return max(LeftRes, max(RightRes, lbs + rbs)); //返回三种情形下的最大值
    }
};
```

(3). 先证明此问题具有最优子结构性质: 依次考虑 $(1 \leq i \leq n)$ 以 $a[i]$ **为结尾**的最大子段和 $C[i]$, 然后在这 n 个值当中取最大值即为原问题答案. 假设以 $a[i]$ 为结尾的最大 (和) 子段为 $\{a[k], \cdots, a[i]\}$, 那么 $\{a[k], \cdots, a[i-1]\}$ 一定是以 $a[i-1]$ 为结尾的最大 (和) 子段. 否则若 $\{a[m], \cdots, a[i-1]\}$ 为以 $a[i-1]$ 为结尾的最大 (和) 子段, 那么 $\{a[m], \cdots, a[i-1], a[i]\}$ 就是以 $a[i]$ 为结尾的最大 (和) 子段, 这显然与假设相矛盾, 也就是说该优化函数是满足优化原则的 (即此问题具有最优子结构性质).

现在来推导 $C[i]$ 的递推表达式: 当 $C[i-1] \leq 0$, 说明 $C[i-1]$ 对应的子段对于整体的贡献是没有的, 所以 $C[i] \leftarrow a[i]$; 当 $C[i-1] > 0$, 说明 $C[i-1]$ 对应的子段对于整体的是有贡献的, 于是 $C[i] \leftarrow a[i] + C[i-1]$. 两种可能情况 (对应两种决策) 取最大值即可:

$$\begin{cases} C\left[i\right] = \max\left\{a\left[i\right], C\left[i-1\right] + a\left[i\right]\right\}, 2 \leq i \leq n \\ C\left[1\right] = a\left[1\right] \end{cases}$$

最后返回数组 C 中的最大值 ($\max\limits_{1 \leq i \leq n} C\left[i\right]$) 即可. 计算 $C[i]$ 的过程需要消耗 $O(n)$ 的时间, 找出数组最大值也需要 $O(n)$ 的时间 (一次遍历), 所以算法的总时间复杂度为 $T(n) = O(n)$. 并且我们可以给出后页的 C++ 代码:

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <limits.h>
using namespace std;

int mostvalue(vector<int>& a) { //时间复杂度为 O(n), 空间复杂度为 O(n)
    int n = a.size();
    vector<int> dp(n);
    dp[0] = a[0];
    for(int i = 1; i < n; i++) {
        dp[i] = max(dp[i - 1] + a[i], a[i]);
    }
    int index = max_element(dp.begin(), dp.end()) - dp.begin();
    return dp[index];
}

int mostvalue2(vector<int>& a) {  //时间复杂度为 O(n), 空间复杂度为 O(1)
    int n = a.size();
    int dp = a[0];
    int res = INT_MIN;
    for(int i = 1; i < n; i++) {
        dp = max(dp + a[i], a[i]);  //使用滚动数组思想来优化空间
        res = max(res, dp); //迭代式更新求得最大值
    }
    return res;
}

int main() {
    int N;
    scanf("%d", &N);
    vector<int> a(N);
    for(int i = 0; i < N; i++){
        scanf("%d", &a[i]);
    }
    //int res = mostvalue(a); //不优化空间
    int res = mostvalue2(a); //优化空间
    cout << res;

}
```

Problem 2

设 $A=\{x_1,x_2,\cdots,x_n\}$ 是 n 个不等的整数构成的序列, A 的一个单调递增子序列是指序列 $\{x_{i_1},x_{i_2},\cdots,x_{i_k}\}, i_1<i_2<\cdots i_k$ 且 $x_{i_1}<x_{i_2}<\cdots<x_{i_k}$.(子序列包含 k 个整数). 例如, $A=\{1,5,3,8,10,6,4,9\}$, 他的长度为 4 的递增子序列是: $\{1,5,8,10\},\{1,5,8,9\},\cdots$. 设计一个算法, 求 A 的最长的单调递增子序列, 分析算法的时间复杂度. 对于输入实例 $A=\{2,8,4,-4,5,9,11\}$, 给出算法的计算过程和最后的解.

Solution: 定义 $dp[i]$ 是以 nums[i] **为结尾** (且考虑前 i 个元素) 的最长单增子序列的长度.

- 如果在索引范围 $[0,i-1]$ 当中能够找到 (若干个) 下标 j 使得 nums[j] < nums[i] 成立. 根据**大小关系的传递性**可知: 以 nums[j] 为结尾的最长递增子序列中的所有元素都 < nums[i], 也就是说我们找到了符合条件的位置. 那么在这些位置 j 中找到 $dp[j]$ 的最大值, 在此基础上加 1 即可使得以 nums[i] 为结尾的单增子序列长度最大 (即将 nums[i] 接到符合条件的、且最长的 nums[j] 后边).
- 如果在索引范围 $[0,i-1]$ 当中找不到下标 j 使得 nums[j] < nums[i] 成立 (即 nums$[0,\cdots,i-1]$ 都比 nums[i] 大), 那么以 nums[i] 为结尾的单增子序列长度就只能为 1(即 nums[i] 独立作为单增子序列).

故综合上述两种情况, 我们可以写出如下转移方程 ($i\geq 1$) 和初始条件:

$$dp\left[i\right]=\begin{cases}\max\limits_{0\leq j\leq i-1} dp\left[j\right]+1, & \exists j\in[0,i-1], s.t.\,\text{nums}\left[j\right]<\text{nums}\left[i\right]\\ 1 & \forall j\in[0,i-1], s.t.\,\text{nums}\left[j\right]>\text{nums}\left[i\right]\end{cases}, dp\left[0\right]=1$$

但是上述递推方程只能给出长度, 并不能给出具体的最优解, 所以我们需要借助一个数组 m 来对解进行回溯[1](即 $m[i]$ 记录 $dp[i]$ 是由哪个下标的状态转移而来的). 而要想算出**整个数组**的最长单增子序列长度, 则需要算好所有的 $dp[i]$ 值, 再对 dp 数组进行遍历, 由此得到最长单增子序列长度和对应下标[2]. 最后使用数组 m 进行回溯以取得答案. 可以看出, 算法的空间复杂度为 $O(n)$, 而时间复杂度显然为

$$T\left(n\right)=O\left(\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1\right)=O\left(\sum_{i=0}^{n-1}i\right)=O\left(\frac{n\left(n-1\right)}{2}\right)=O\left(n^2\right)$$

对于具体实例, 计算过程如下:

$$C\left[1\right]=1; C\left[2\right]=2, k\left[2\right]=1; C\left[3\right]=2, k\left[3\right]=1; C\left[4\right]=1, k\left[4\right]=0;$$
$$C\left[5\right]=3, k\left[5\right]=3; C\left[6\right]=4, k\left[6\right]=5; C\left[7\right]=5, k\left[7\right]=6$$

显然在数组 C 中的最大值为 $C[7]=5$, 即最长递增子序列长度为 5 且追踪过程为:

$$x_7, k\left[7\right]=6\Rightarrow x_6;\ k\left[6\right]=5\Rightarrow x_5;\ k\left[5\right]=3\Rightarrow x_3;\ k\left[3\right]=1\Rightarrow x_1$$

故 $A=\{2,8,4,-4,5,9,11\}$ 的最长单调递增子序列为 $\{x_1,x_3,x_5,x_6,x_7\}=\{2,4,5,9,11\}$.

我们将上述的最优值求解过程和解的回溯过程写成 C++ 代码, 并且已完全通过**LeetCode-T300**的所有测试样例, 具体如后页所示:

[1] 对于求**具体方案**的动态规划题目, 多开一个数组来记录状态的转移情况是**最常见**的手段.

[2] 因为整个数组的最长单增子序列**不一定**以 nums[$n-1$] 为结尾! 故才需要求得 dp 数组的最大值.

```cpp
#include <algorithm>
#include <ctime>
#include <iostream>
#include <vector>
using namespace std;
vector<int> LIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 0);
    dp[0] = 1;
    vector<int> m(n, 0); // m[0] = 0
    for (int i = 1; i <= n - 1; i++) {
        int prev = i, len = 1; //至少包含自身一个数, 因此起始长度为 1, 由自身转移而来
        for (int j = 0; j <= i - 1; j++) {
            if (nums[j] < nums[i]) { //找到满足条件的位置 j
                if (dp[j] + 1 > len) {
                    len = dp[j] + 1;
                    prev = j; //迭代式更新求得满足条件的 d[j]+1 的最大值和 dp[i] 的来源 (即 prev)
                }
            }
        }
        dp[i] = len, m[i] = prev; // m[i] 就是 nums[i] 的来源位置 (即 prev)!
    }
    int index = max_element(dp.begin(), dp.end()) - dp.begin(); //最大值对应的索引
    int MaxLen = dp[index]; //递增子序列的最大长度值
    vector<int> res; //注意整个数组的最长递增子序列是以 nums[index] 为结尾的!
    while (res.size() != MaxLen) {
        res.push_back(nums[index]);
        index = m[index]; //逐步地从后往前回溯索引, 并将对应元素写进 res 数组
    }
    reverse(res.begin(), res.end()); //reverse 一下更好看一些, 此步可以选择注释掉
    return res;
}
int main() {
    int n;
    cin >> n;             //输入数组的长度
    vector<int> nums(n); //定义数组
    cout << " 数组 nums 为:" << endl;
    for (int i = 0; i < n; i++) { //对数组初始化
        nums[i] = -1 * (n + 1) +
                  rand() % (2 * n + 2); //在 [-n - 1, n + 1] 随机产生长度为 n 的数组
        cout << nums[i] << " "; //打印该数组
    }
    cout << endl;
    clock_t startTime = clock(); //计时开始
    vector<int> res = LIS(nums);
    clock_t endTime = clock(); //计时结束
    cout << " 数组 nums 的最长单增子序列为:" << endl;
    for (int i = 0; i < res.size(); i++) {
        cout << res[i] << " ";
    }
    cout << endl;
    cout << " 算法耗时为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

在解决这个问题之后, 我们需要再给出另一个解决思路与此非常类似的问题以便进行类比学习: **最大整除子集问题**: 给定一组互异的正整数集合, 找到最大子集: 使得子集元素中的每一对 (S_i, S_j) 都满足 $S_i\%S_j = 0$ 或 $S_j\%S_i = 0$. 请返回最大子集的阶数 (即子集中的元素个数). **注意**: $S_i\%S_j = 0$ 意味着 S_i 可以被 S_j 整除.

由于整除子集中的每一对值都是倍数**或**约数关系, 因此为了后续解决问题的方便, 我们不妨先将数组 nums 进行排序 (耗时为 $O(n\log n)$, (后续可知) 不影响整个算法的复杂度), 以免还要具体考虑一对值**到底是**倍数关系还是约数关系 (思考起来会很乱), 并且对 nums 排好序有助于我们利用**整除关系的传递性**来进行动态规划!

于是我们定义: $dp[i]$ 是以 nums[i] **为结尾** (且考虑前 i 个元素) 的最大整除子集的阶数 (即该子集合的元素个数).

- 如果在索引范围 $[0, i-1]$ 当中能够找到 (若干个) 下标 j 使得 nums[i]%nums[j]==0 成立. 由于 nums[j] | nums[i] 且根据**整除关系的传递性**可知: 以 nums[j] 为结尾的最大整除子集中的所有元素都能整除 nums[i], 也就是说我们找到了符合条件的位置. 那么在这些位置 j 中找到 $dp[j]$ 的最大值, 在此基础上加 1 即可使得以 nums[i] 为结尾的整除子集的长度最大 (即将 nums[i] 接到符合条件的、且最长的 nums[j] 后边).
- 如果在索引范围 $[0, i-1]$ 当中找不到下标 j 使得 nums[i]%nums[j] $\neq 0$ 成立 (即 nums[i] 不能接在位置 i 之前的任何数的后面!), 那么以 nums[i] 为结尾的最大整除子集的长度就只能为 1(即 nums[i] 独立作为最大整除子集).

故综合上述两种情况, 我们可以写出如下转移方程 ($i \geq 1$) 和初始条件:

$$dp[i] = \begin{cases} \max\limits_{0\leq j\leq i-1} dp[j]+1, & \exists j \in [0,i-1], s.t.\,\text{nums}[i]\,\%\text{nums}[j] = 0 \\ 1 & \forall j \in [0,i-1], s.t.\,\text{nums}[i]\,\%\text{nums}[j] \neq 0 \end{cases}, dp[0] = 1$$

此时我们需要借助一个数组 m 来对解进行回溯 (即 $m[i]$ 记录 $dp[i]$ 是由哪个下标的状态转移而来的). 而要想算出**整个数组**的整除子集的最大长度, 则需要算好所有的 $dp[i]$ 值, 再对 dp 数组进行遍历, 由此得到最大的整除子集长度和对应下标. 最后使用数组 m 进行回溯以取得答案. 于是我们可以给出最大整除子集算法的 C++ 代码:

```cpp
class Solution {
public:
    vector<int> largestDivisibleSubset(vector<int>& nums) {
        sort(nums.begin(), nums.end());  //先对数组进行排序, 方便后续考虑递推表达式
        int n = nums.size();
        vector<int> dp(n, 0);
        dp[0] = 1;
        vector<int> m(n, 0); // m[0] = 0
        for (int i = 1; i <= n - 1; i++) {
            int prev = i, len = 1; //至少包含自身一个数, 因此起始长度为 1, 由自身转移而来
            for (int j = 0; j <= i - 1; j++) {
                if (nums[i] % nums[j] == 0) {
                    if (dp[j] + 1 > len) {
                        len = dp[j] + 1;
                        prev = j; //迭代式更新求得满足条件的 d[j]+1 的最大值和 dp[i] 的来源
↪   (即 prev)
                    }
                }
            }
            dp[i] = len, m[i] = prev; // m[i] 就是 nums[i] 的来源位置 (即 prev)!
        }
```

```
        int index = max_element(dp.begin(), dp.end()) - dp.begin(); //最大值对应的索引
        int MaxLen = dp[index]; //整除子集的最大长度值
        vector<int> res; //注意整个数组的最大整除子集是以 nums[index] 为结尾的!
        while (res.size() != MaxLen) {
            res.push_back(nums[index]);
            index = m[index]; //逐步地从后往前回溯索引, 并将对应元素写进 res 数组
        }
        reverse(res.begin(), res.end()); //reverse 一下更好看一些, 此步可以选择注释掉
        return res;
    }
};
```

上述 C++ 代码已完全通过**LeetCode-T368**的所有测试样例. 再来分析一下该算法的时空复杂度: 排序需要 $O(n\log n)$ 的时间, 算法主体显然需要消耗 $O(n^2)$ 的时间, 而其余操作 (如 reverse 操作、求 dp 最大值以及结果写入) 均只需要 $O(n)$ 的时间, 所以综合可得算法的时间复杂度为 $O(n^2)$. 由于借助了两个长度为 n 的中间数组, 所以算法的空间复杂度为 $O(n)$. 上述两个算法之所以正确, 本质上是利用了**二元关系的可传递性**. 比如最长递增子序列是利用了**大小关系“>”的可传递性**, 而最大整除子集利用了**整除关系“|”的可传递性**. 这两段代码基本一样, 所以可以将此记住作为解题模板, 以便应付其他具有传递性的二元关系的 dp 算法题.

Problem 3

考虑下面特殊的整数线性规划问题

$$\max \sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_i x_i \le b, x_i \in \{0,1,2\}, 1 \le i \le n$$

试设计一个解决此问题的动态规划算法, 并分析算法的时间复杂度.

Solution: 方法 1: 设 $y_i \in \{0,1\}, 1 \le i \le 2n$, 令 $x_i = y_i + y_{i+n}, 1 \le i \le n$, 则上述规划问题转化为

$$\max \sum_{i=1}^{2n} c_i y_i$$
$$\sum_{i=1}^{2n} a_i y_i \le b, y_i \in \{0,1\}, 1 \le i \le 2n$$

其中 $c_{i+n} = c_i, a_{i+n} = a_i, 1 \le i \le n$. 将 c_i 看作价值, a_i 看作重量, b 看作背包容量. 于是就将问题转化为了 0-1 背包问题 (物品数为 $2n$). 由于 n 件物品的 0-1 背包问题的动态规划算法的时间复杂度为 $O(2^n)$, 故此方法的时间复杂度为 $O\left(2^{2n}\right) = O(4^n)$.

方法 2: 可以看成是另一种背包问题. 即 b 为背包容量, $x_i \in \{0,1,2\}$ 为背包中可以装 0,1 或者 2 件物品, x_i 对应的价值为 c_i, 求在容量 b 一定的前提下, 背包所容纳物品的最大价值. 也就是参数完全相同的两个 0-1 背包问题, 它们同时制约于背包容量为 C 这个条件.

在设计算法时可以优先考虑 m_i, 也就是先判断背包剩下的容量能不能放进去 c_i, 若可以就再判断能否使 $p_i = 1$, 若可以那就再放入一个 c_i, 这样就间接地满足了 $x_i = m_i + p_i = 2$ 的条件. 根据单参数的

0-1 背包问题的动态规划算法, 于是可以类比写出该问题的递推公式

$$m(k,x)=\begin{cases}-\infty, & x<0\\ m(k-1,x), & 0\le x<w_k\\ \max\{m(k-1,x),m(k-1,x-w_k)+p_k\}, & w_k\le x<2w_k\\ \max\{m(k-1,x),m(k-1,x-w_k)+p_k,m(k-1,x-2w_k)+2p_k\}, & x\ge 2w_k\end{cases}$$

类似讲义中的推导过程可以得出该方法的时间复杂度为 $O(3^n)$.

Problem 4

可靠性设计: 一个系统由 n 级设备串联而成, 为了增强可靠性, 每级都可能并联了不止一台同样的设备. 假设第 i 级设备 D_i 用了 m_i 台, 该级设备的可靠性 $g_i(m_i)$, 则这个系统的可靠性是 $\Pi g_i(m_i)$. 一般来说 $g_i(m_i)$ 都是递增函数, 所以每级用的设备越多系统的可靠性越高. 但是设备都是有成本的, 假定设备 D_i 的成本是 c_i, 设计该系统允许的投资不超过 c. 那么, 该如何设计该系统 (即各级采用多少设备) 使得这个系统的可靠性最高. 试设计一个动态规划算法求解可靠性设计问题.

Solution: 问题描述为

$$\begin{aligned}&\max\prod_{i=1}^{n}g_i(m_i)\\ &\text{s.t.}\sum_{i=1}^{n}m_ic_i\le c, 1\le m_i\le 1+\left\lfloor\frac{c-\sum_{i=1}^{n}c_i}{c_n}\right\rfloor\end{aligned}$$

记 $G[k](x)$ 为第 k 级设备在可用投资为 x 时的系统可靠性最大值则有如下关系式

$$G[k](x)=\max_{1\le m_k\le\left\lfloor\frac{x}{c_k}\right\rfloor}\{g_k(m_k)\cdot G[k-1](x-c_km_k)\}$$

定义下列函数

$$G[k](x)=\begin{cases}-\infty, & x<\sum_{i=k}^{n}c_i\\ \max\limits_{1\le m_k\le\min\left\{\left\lfloor\frac{x}{c_1}\right\rfloor,\left\lfloor\frac{c}{c_k}\right\rfloor\right\}}\{g_k(m_k)\cdot G[k-1](x-c_km_k)\}, & x\ge\sum_{i=k}^{n}c_i\end{cases}$$

$$G[0](x)=\begin{cases}1, & \sum_{i=1}^{n}c_i\le x\le c\\ -\infty, & else\end{cases},\ G[1](x)=\begin{cases}-\infty, & x<\sum_{i=1}^{n}c_i\\ \max\limits_{1\le m_1\le\left\lfloor\frac{x}{c_1}\right\rfloor}\{g_1(m_1)\cdot G[0](x-c_1m_1)\}, & \sum_{i=1}^{n}c_i\le x\le c\\ \max\limits_{1\le m_1\le\left\lfloor\frac{c}{c_1}\right\rfloor}\{g_1(m_1)\cdot G[0](x-c_1m_1)\}, & x>c\end{cases}$$

初始计算 $G[0](c)$, 依次向后求解, 即可得到策略集.

Problem 5

(双机调度问题) 用两台处理机 A 和 B 处理 n 个作业. 设第 i 个作业交给机器 A 处理时所需要的时间是 a_i, 若由机器 B 来处理, 则所需要的时间是 b_i. 现在要求每个作业只能由一台机器处理, 每台机器都不能同时处理两个作业. 设计一个动态规划算法, 使得这两台机器处理完这 n 个作业的时间最短 (从任何一台机器开工到最后一台机器停工的总时间). 以下面的例子说明你的算法:

$$n = 6, (a_1, a_2, a_3, a_4, a_5, a_6) = (2, 5, 7, 10, 5, 2), (b_1, b_2, b_3, b_4, b_5, b_6) = (3, 8, 4, 11, 3, 4)$$

在完成前 k 个作业时, 设机器 A 工作了 x 时间 (注意 x 限制为正整数), 则机器 B 此时最小的工作时间是 x 的函数. 设 $F[k](x)$ 表示完成前 k 个作业时, 机器 B 最小的工作时间, 则有

$$F[k](x) = \min\{F[k-1](x) + b_k, F[k-1](x - a_k)\}$$

其中 $F[k-1](x) + b_k$ 对应的是第 k 个作业由机器 B 来处理, 此时完成前 $k-1$ 个作业时机器 A 的工作时间仍是 x, 则 B 在 $k-1$ 阶段用时为 $F[k-1](x)$; 而 $F[k-1](x-a_k)$ 对应第 k 个作业由机器 A 处理 (完成 $k-1$ 个作业, 机器 A 工作时间时 $x - a[k]$, 而 B 完成 k 阶段与完成 $k-1$ 阶段用时都为 $F[k-1](x-a_k)$). 于是完成前 k 个作业所需要的时间为 $T = \max\{x, F[k](x)\}$. 用题中的例子演示上述算法:

- 初始化第 1 个作业, 下标从 1 开始. 当处理第 1 个作业时, $a[1] = 2, b[1] = 3$, 机器 A 所用时间的可能值范围是 $0 \le x \le a[1]$, 于是会出现以下几种情况:
  - 当 $x < 0$ 时, 设 $F[1](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[1](x) = 3$, 则 $\max\{x, 3\} = 3$;
  - 当 $x \ge 2$ 时, $F[1](x) = 0$, 则 $\max\{1, x\} = 2$;

  从上面的 3 种情况可以看出, 当 $x = 2$ 时, 完成第一个作业两台机器花费的最少时间为 2, 此时机器 A 花费时间 2, 机器 B 花费 0 时间. 即前 1 个作业的安排为 (A)
- 再来看第 2 个作业: 首先 x 的取值范围是 $0 \le x \le a[1] + a[2] = 7$.
  - 当 $x < 0$ 时, 设 $F[2](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{3 + 8, \infty\} = 11$, 则 $\max\{x, 11\} = 11$;
  - 当 $2 \le x < 5$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{0 + 8, \infty\} = 8$, 则 $\max\{x, 8\} = 8$;
  - 当 $5 \le x < 7$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{0 + 8, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 5, 6);
  - 当 $x \ge 7$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{0 + 8, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 7);

  于是可以看出当 $x = 5$ 时, 完成前两个作业的两台机器所花费时间最少为 5, 此时机器 A 花费 5 时间, 机器 B 花费 3 时间, 即前 2 个作业的安排为 (B, A).
- 再来看第 3 个作业: 首先 x 的取值范围是 $0 \le x \le a[1] + a[2] + a[3] = 14$.
  - 当 $x < 0$ 时, 设 $F[3](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{11 + 4, \infty\} = 15$, 则 $\max\{x, 15\} = 15$;

- 当 $2 \leq x < 5$ 时,$F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{8 + 4, \infty\} = 12$, 则 $\max\{x, 12\} = 12$;
- 当 $5 \leq x < 7$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{3 + 4, \infty\} = 7$, 则 $\max\{x, 7\} = 7$;
- 当 $7 \leq x < 9$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 11\} = 4$, 则 $\max\{x, 4\} = x$(包含 7, 8);
- 当 $9 \leq x < 12$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 8\} = 4$, 则 $\max\{x, 4\} = x$(包含 9, 10, 11);
- 当 $12 \leq x < 14$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 12, 13);
- 当 $x \geq 14$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 14);

于是可以看出当 $x = 7$ 时, 完成前两个作业的两台机器所花费时间最少为 7, 此时机器 A 花费 7 时间, 机器 B 花费 7 时间 (也可以花费 4 时间). 即完成前 3 个作业有两种安排: (B, A, B)(对应 A 花费 7, B 花费 7) 和 (A, A, B)(对应 A 花费 7, B 花费 4).

- 再来看第 4 个作业: 首先 x 的取值范围是 $0 \leq x \leq a[1] + a[2] + a[3] + a[4] = 24$.
  - 当 $x < 0$ 时, 设 $F[4](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \leq x < 2$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{15 + 11, \infty\} = 26$, 则 $\max\{x, 26\} = 26$;
  - 当 $2 \leq x < 5$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{12 + 11, \infty\} = 23$, 则 $\max\{x, 23\} = 23$;
  - 当 $5 \leq x < 7$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{7 + 11, \infty\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $7 \leq x < 9$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{4 + 11, \infty\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $9 \leq x < 10$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{4 + 11, \infty\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $10 \leq x < 12$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{4 + 11, 15\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $12 \leq x < 14$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{3 + 11, 12\} = 12$, 则 $\max\{x, 12\} = x$(包含 12, 13);
  - 当 $14 \leq x < 15$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 12\} = 11$, 则 $\max\{x, 11\} = x$(包含 14);
  - 当 $15 \leq x < 17$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 15, 16);
  - 当 $17 \leq x < 19$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 17, 18);
  - 当 $19 \leq x < 22$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 19, 20, 21);
  - 当 $22 \leq x < 24$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 22, 23);

  - 当 $24 \leq x$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 24);

  于是可以看出当 $x = 12$ 时, 完成前两个作业的两台机器所花费时间最少为 12, 此时机器 A 花费 12 时间, 机器 B 花费 12 时间. 即完成前 3 个作业的最优安排为 (A, B, B, A).

- 再来看第 5 个作业: 首先 x 的取值范围是 $0 \leq x \leq a[1] + a[2] + a[3] + a[4] + a[5] = 29$.
  - 当 $x < 0$ 时, 设 $F[5](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \leq x < 2$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{26 + 3, \infty\} = 29$, 则 $\max\{x, 29\} = 29$;
  - 当 $2 \leq x < 5$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{23 + 3, \infty\} = 26$, 则 $\max\{x, 26\} = 26$;
  - 当 $5 \leq x < 7$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{18 + 3, 26\} = 21$, 则 $\max\{x, 21\} = 21$;
  - 当 $7 \leq x < 9$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{15 + 3, 23\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $9 \leq x < 10$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{15 + 3, 23\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $10 \leq x < 12$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{15 + 3, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $12 \leq x < 14$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{12 + 3, 15\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $14 \leq x < 15$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{11 + 3, 15\} = 14$, 则 $\max\{x, 14\} = x$(包含 14);
  - 当 $15 \leq x < 17$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{7 + 3, 15\} = 10$, 则 $\max\{x, 10\} = x$(包含 15, 16);
  - 当 $17 \leq x < 19$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{4 + 3, 12\} = 7$, 则 $\max\{x, 7\} = x$(包含 17, 18);
  - 当 $19 \leq x < 20$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{4 + 3, 11\} = 7$, 则 $\max\{x, 7\} = x$(包含 19);
  - 当 $20 \leq x < 22$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{4 + 3, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 20, 21);
  - 当 $22 \leq x < 24$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{3 + 3, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 22, 23);
  - 当 $24 \leq x < 27$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{0 + 3, 4\} = 3$, 则 $\max\{x, 3\} = x$(包含 24, 25, 26);
  - 当 $27 \leq x < 29$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{0 + 3, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 27, 28);
  - 当 $x \geq 29$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{0 + 3, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 29);

  于是可以看出当 $x = 14$ 时, 完成前两个作业的两台机器所花费时间最少为 14, 此时机器 A 花费 14 时间, 机器 B 花费 14 时间. 即完成前 5 个作业的最优安排为 (A, A, A, B, B).

- 再来看第 6 个作业: 首先 x 的取值范围是 $0 \le x \le a[1]+a[2]+a[3]+a[4]+a[5]+a[6] = 31$.
  - 当 $x < 0$ 时, 设 $F[6](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{29 + 4, \infty\} = 33$, 则 $\max\{x, 33\} = 33$;
  - 当 $2 \le x < 4$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{26 + 4, 29\} = 29$, 则 $\max\{x, 29\} = 29$;
  - 当 $4 \le x < 5$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{26 + 4, 26\} = 26$, 则 $\max\{x, 26\} = 26$;
  - 当 $5 \le x < 7$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{21 + 4, 26\} = 25$, 则 $\max\{x, 25\} = 25$;
  - 当 $7 \le x < 9$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 21\} = 21$, 则 $\max\{x, 21\} = 21$;
  - 当 $9 \le x < 10$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $10 \le x < 11$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $11 \le x < 12$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $12 \le x < 14$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{15 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $14 \le x < 15$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{14 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $15 \le x < 16$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{10 + 4, 15\} = 14$, 则 $\max\{x, 14\} = x$(包含 15);
  - 当 $16 \le x < 17$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{10 + 4, 14\} = 14$, 则 $\max\{x, 14\} = x$(包含 16);
  - 当 $17 \le x < 19$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 10\} = 10$, 则 $\max\{x, 10\} = x$(包含 17, 18);
  - 当 $19 \le x < 20$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 19);
  - 当 $20 \le x < 21$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 20);
  - 当 $21 \le x < 22$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 21);
  - 当 $22 \le x < 24$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{4 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 22, 23);
  - 当 $24 \le x < 26$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{3 + 4, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 24, 25);
  - 当 $26 \le x < 27$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{3 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 26);

- 当 $27 \leq x < 29$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{3 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 27, 28);
- 当 $29 \leq x < 31$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{0 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 29, 30);
- 当 $31 \leq x$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{0 + 4, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 31);

于是可以看出当 $x = 15$ 时, 完成前两个作业的两台机器所花费时间最少为 15, 此时机器 A 花费 15 时间, 机器 B 花费 14 时间. 即完成前 6 个作业的最优安排 (只是其中一种安排方案) 为 (B, A, B, A, B, B), 并且算法的时间复杂度为 $O(n)$.

Problem 6

有 n 项作业的集合 $J = \{1, 2, \cdots, n\}$, 每项作业 i 有加工时间 $t(i) \in \mathbb{Z}^+, t(1) \leq t(2) \leq \cdots \leq t(n)$, 效益值 $v(i)$, 任务的结束时间 $D \in \mathbb{Z}^+$, 其中 $\mathbb{Z}^+$ 表示正整数集合. 一个可行调度是对 J 的子集 A 中任务的一种安排, 对于 $i \in A$, $f(i)$ 是开始时间, 且满足下述条件:

$$\begin{cases} f(i) + t(i) \leq f(j) \text{或} f(j) + t(j) \leq f(i), \text{其中} j \neq i \text{且} i, j \in A \\ \sum_{k \in A} t(k) \leq D \end{cases}$$

设机器从 0 时刻开始启动, 只要有作业就不闲置, 求具有最大总效益的调度. 给出算法并分析其时间复杂度.

Solution: 与 0-1 背包问题相类似, 使用 DP 算法, 令 $N_j(d)$ 表示考虑作业集 $\{1, 2, \cdots, j\}$、结束时间为 d 的最优调度的效益, 那么有递推方程

$$N_j(d) = \begin{cases} \max\{N_{j-1}(d), N_{j-1}(d - t(j)) + v_j\}, & d \geq t(j) \\ N_{j-1}(d), & d < t(j) \end{cases}$$

并且边界 (初始) 条件为

$$N_1(d) = \begin{cases} v_1, & d \geq t(1) \\ 0, & d < t(1) \end{cases}, \quad N_j(0) = 0, \quad N_j(d) = -\infty \left(\text{其中} d < 0\right)$$

自底向上计算, 存储使用备忘录 (以存代算), 可以使用标记函数 $B(j)$ 记录使得 $N_j(d)$ 达到最大时是否

$$N_{j-1}(d - t(j)) + v_j > N_{j-1}(d)$$

如果是, 则 $B(j) = j$; 否则 $B(j) = B(j - 1)$.(换句话说, 如果装了作业 j, 那么就追踪其下标; 否则就不追踪更新)

伪代码如后页算法1中所示, 由此我们可以分析出时间复杂度: 得到最大效益 $N[n, D]$ 后, 通过对 $B[n, D]$ 的追踪就可以得到问题的解, 算法的主要工作在于第 7 行到第 16 行的 for 循环, 需要执行 $O(nD)$ 次, 循环体内的工作量是常数时间, 因此算法的总时间复杂度为 $O(nD)$. 显然该算法是**伪多项式时间**的算法[3].

[3]问题就在于如果 D 过大, 即 D 的 2 进制表示会很长, 且 $O(n \cdot D) = O\left(n \cdot 2^{\log(D)}\right) = O\left(n \cdot 2^{\text{输入长度}}\right)$, 是与输入长度相关的指数表达式, 这种复杂度形式的算法称之为**伪多项式时间算法**.

Algorithm 1 Homework 算法

Input: 加工时间 $t[1,\cdots,n]$, 效益 $v[1,\cdots,n]$, 结束时间 D
Output: 最优效益 $N[i,j]$, 标记函数 $B[i,j], i=1,2,\cdots,n, j=1,2,\cdots,D$

```
for d = 1; d ≤ t[1] − 1; d++ do
    N[1, d] ← 0, B[1] ← 0;
end for
for d = t[1]; d ≤ D; d++ do
    N[1, d] ← v[1], B[1] ← 1;
end for
for j = 2; j ≤ n; j++ do
    for d = 1; d ≤ D; d++ do
        N[j, d] ← N[j − 1, d];
        B[j, d] ← B[j − 1, d];
        if d ≥ t[j] && N[j − 1, d − t[j]] + v[j] > N[j − 1, d] then
            N[j, d] ← N[j − 1, d − t[j]] + v[j];
            B[j, d] ← j;
        end if
    end for
end for
end {Homework}
```

Problem 7

设 A 是顶点为 $1,2,\cdots,n$ 的凸多边形, 可以用不在内部相交的 $n-3$ 条对角线将 A 划分成三角形, 下图1中就是 5 边形的所有划分方案. 假设凸 n 边形的边及对角线的长度 d_{ij} 都是给定的正整数, 其中 $1 \le i < j \le n$. 划分后三角形 ijk 的权值等于其周长, 求具有最小权值的划分方案. 设计一个动态规划算法求解该问题, 并说明其时间复杂度 (提示: 参考矩阵连乘问题).

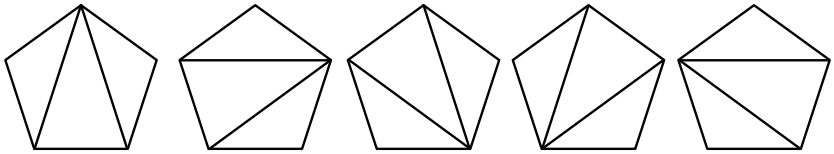


图 1: 5 边形的划分方案

如下图2所示, n 边形的顶点是 $1,2,\cdots,n$. 顶点 $i-1,i,\cdots,j$ 构成的凸多边形记作 $A[i,j]$, 于是原始问题就是 $A[2,n]$.

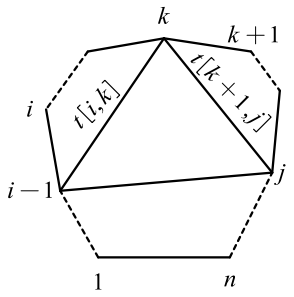


图 2: 子问题归约图

考虑子问题 $A[i,j]$ 的划分, 假设它的所有划分方案中最小权值为 $t[i,j]$. 从 $i, i+1, \cdots, j-1$ 中任选顶点 k, 它与底边 $(i-1)j$ 构成一个三角形 (图2中的三角形). 这个三角形将 $A[i,j]$ 划分成两个凸多边形: $A[i,k]$ 和 $A[k+1,j]$, 从而产生了两个子问题. 这两个凸多边形的划分方案的最小权值分别为 $t[i,k]$ 和 $t[k+1,j]$. 根据 DP 思想, $A[i,j]$ 相对于这个顶点 k 的划分方案的最小权值是

$$t[i,k] + t[k+1,j] + d_{(i-1)k} + d_{kj} + d_{(i-1)j}$$

其中 $d_{(i-1)k} + d_{kj} + d_{(i-1)j}$ 是三角形 $(i-1)kj$ 的周长, 于是得到递推关系

$$t[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k \le j-1} \left\{ t[i,k] + t[k+1,j] + d_{(i-1)k} + d_{kj} + d_{(i-1)j} \right\}, & i < j \end{cases}$$

显然这个递推关系与矩阵链乘积算法的递推式十分相似, 可以通过标记函数来得到最小权值对应顶点 k 的位置, 并且类比矩阵链乘积算法, 可知该划分算法最坏情况下的时间复杂度为 $O\left(n^3\right)$.

至此, Chap 5 的作业解答完毕.

中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 6&7 课程作业解答

2022 年 11 月 8 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

假设对称旅行商问题的邻接矩阵如下所示, 试用优先队列式分枝限界算法给出最短环游. 画出状态空间树的搜索图, 并说明搜索过程.

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ & \infty & 16 & 4 & 2 \\ & & \infty & 6 & 7 \\ & & & \infty & 12 \\ & & & & \infty \end{pmatrix}$$

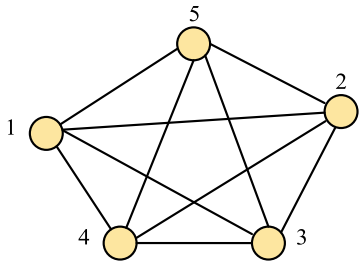


图 1: 旅行商问题

Solution: 状态空间树的搜索图如下图2中所示:

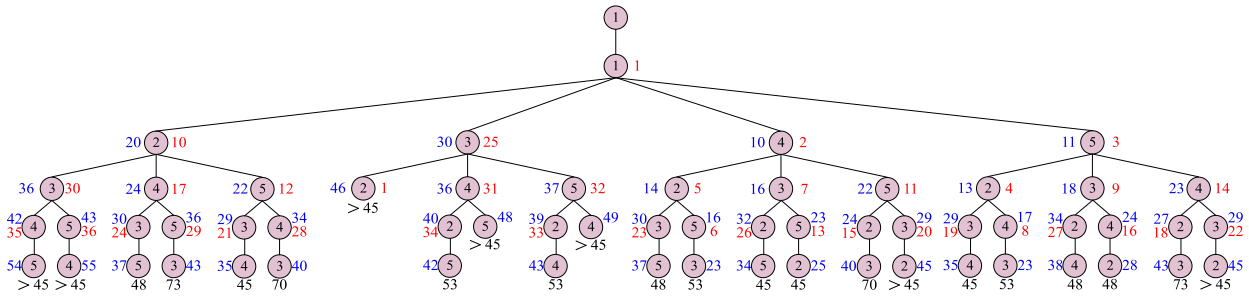


图 2: 解空间树搜索图

图中圆圈内为顶点序号, 蓝色数字为该路径到该节点的总路程, 红色数字表示该节点的搜索序数. 从顶点 1 开始搜索, 将其 4 个儿子节点放入队列. 然后优先访问队列中当前路程最小的节点, 再将它的儿子放入队列. 然后重复上述过程, 优先访问队列中当前路径最小的节点, 将其儿子放入队列. 第一个被访问到的叶子节点为 1-4-2-5-3-1 这条路径, 总路程为 53, 记录当前最短路径. 之后若某节点的路径大于最短路程, 则不再搜索该节点的子树. 若某叶子节点的总路程小于当前最短路径, 则更新之. 如此搜索, 当访问到 1-4-3-5-2-1 这条路径的叶子节点时, 其总路程为 45 $<$ 53, 将 45 更新为当前最短路径, 继续搜索. 最后得出最短环游为 1-2-5-3-4-1、1-4-3-2-5-1、1-4-3-5-2-1、1-5-2-3-4-1, 总路程均为 45.

Problem 2

最佳调度问题: 假设有 n 个任务要由 k 个可并行工作的机器来完成, 完成任务 i 需要的时间为 t_i. 试设计一个分枝限界算法, 找出完成这 n 个任务的最佳调度, 使得完成全部任务的时间 (从机器开始加工任务到最后停机的时间) 最短.

Solution: 设计的算法步骤如下:

- 先将任务由大到小排序;
- 计算 n 个任务需要的总时间和平均到 k 个机器上的时间;
- 将大于平均时间的任务各分配一个机器, 找到最大完成时间;
- 将其他任务顺序安排在一台机器上, 如果时间超出最大时间, 则把该任务交给下一个机器, 下一个任务继续在这台机器上试安排, 直到所有任务都不能在小于最大完成时间的情况下安排;
- 安排下一台机器直到所有任务安排完;
- 或有可能安排某些任务找不到小于最大完成时间, 那么重新扫描各台机器使再加上该任务后时间最小, 按此方法安排完所有任务.

Problem 3

分派问题: 给 n 个人分派 n 件工作, 给第 i 人分派第 j 件工作的成本是 $C(i,j)$, 试用分枝限界法求成本最小的工作分配方案.

Solution: 设 n 个人的集合是 $\{1,2,\cdots,n\}$, n 项工作的集合是 $\{1,2,\cdots,n\}$, 每个人恰好 1 项工作. 于是有

$$\text{把工作} j \text{分配给} i \Leftrightarrow x_i = j, \quad i,j = 1,2,\cdots n$$

设解向量为 $X = \langle x_1, x_2, \cdots, x_n\rangle$, 分配成本为 $C(X) = \sum_{i=1}^{n} C(i, x_i)$. 搜索空间是排列树. 部分向量 $\langle x_1, x_2, \cdots, x_k\rangle$ 表示已经考虑了人 $1,2,\cdots,k$ 的工作分配. 节点分支的约束条件为:

$$x_{k+1} \in \{1,2,\cdots,n\} \setminus \{x_1, x_2, \cdots, x_k\}$$

可以设立代价函数:

$$F(x_1, x_2, \cdots, x_k) = \sum_{i=1}^{k} C(i, x_i) + \sum_{i=k+1}^{n} \min\{C(i,t) : t \in \{1,2,\cdots,n\} \setminus \{x_1, x_2, \cdots, x_k\}\}$$

界 B 是已得到的最好可行解的分配成本. 如果代价函数大于界, 则回溯. 算法的时间复杂度为 $O(n \cdot n!)$.

Problem 4

如图3所示, 一个 4 阶 Latin 方是一个 4×4 的方格, 在它的每个方格内填入 $1, 2, 3$ 或 4, 并使得每个数字在每行、每列都恰好出现一次. 用回溯法求出所有第一行为 1,2,3,4 的所有 4 阶 Latin 方. 将每个解的第 2 行到第 4 行的数字从左到右写成一个序列. 如图3中的解是 $\langle 3, 4, 1, 2, 4, 3, 2, 1, 2, 1, 4, 3\rangle$. 给出所有可能的 4 阶 Latin 方.

1	2	3	4
3	4	1	2
4	3	2	1
2	1	4	3

图 3: Latin 方

通过 PPT 中的回溯算法可以求出共有 24 个 Latin 方, 具体如下图4所示:

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	3	4	1	2	3	4	1
3	4	1	2	3	4	2	1	4	3	1	2	4	3	2	1	3	4	1	2	4	1	2	3
4	3	2	1	4	3	1	2	3	4	2	1	3	4	1	2	4	1	2	3	3	4	1	2
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
2	4	3	1	2	4	1	3	3	1	4	2	3	1	4	2	3	4	1	2	3	4	1	2
3	1	4	2	4	3	2	1	2	4	1	3	4	3	2	1	2	1	4	3	2	3	4	1
4	3	2	1	3	1	4	2	4	3	2	1	2	4	1	3	4	3	2	1	4	1	2	3
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
3	4	1	2	3	4	1	2	3	4	2	1	3	4	2	1	4	1	2	3	4	1	2	3
4	1	2	3	4	3	2	1	2	1	4	3	4	3	1	2	2	3	4	1	3	4	1	2
2	3	4	1	2	1	4	3	4	3	1	2	2	1	4	3	3	4	1	2	2	3	4	1
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
4	3	1	2	4	3	1	2	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
2	1	4	3	3	4	2	1	2	1	4	3	2	4	1	3	3	1	4	2	3	4	1	2
3	4	2	1	2	1	4	3	3	4	1	2	3	1	4	2	2	4	1	3	2	1	4	3

图 4: Latin 方结果

中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 8 课程作业解答

2022 年 11 月 27 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

叙述二元可满足性问题, 并证明二元可满足性问题是 $\mathcal{P}$ 类问题.
Solution: 二元可满足性问题 (2SAT) 问题:

- **例:** 给定布尔变量的一个有限集合 $U = \{u_1, u_2, \cdots, u_n\}$ 以及定义其上的子句 $C = \{c_1, c_2, \cdots, c_m\}$, 其中 $|c_k| = 2, k = 1, 2, \cdots, m$.
- **问:** 是否存在 U 的一个真赋值, 使得 C 中所有的子句均被满足?

显然 2SAT 是 $\mathcal{P}$ 类问题. 我们采用数理逻辑中的“合取式”表达逻辑命题, 于是有

$$C = c_1 \wedge c_2 \wedge \cdots \wedge c_m = \prod_{k=1}^{m} c_k = \prod_{k=1}^{m} (x_k + y_k)$$

其中 $c_i \cdot c_j$ 表示逻辑与, $x_k + y_k$ 表示逻辑或, x_k, y_k 是某个 u_j 或 $\overline{u}_i$.

考虑表达式 $C = \prod_{k=1}^{m} (x_k + y_k)$, 如果有某个 $x_k + y_k = u_i + \overline{u}_i$, 那么在乘积式中可以去掉该子句, 即 $C' \leftarrow C \backslash (u_i + \overline{u}_i)$, 然后再接着做. 因此可见 C 和 C' 的可满足性是等价的. 所以我们可以假定 C 中不含有形如 $u_i + \overline{u}_i$ 的子句. 注意到此时 C 中的子句个数不会超过 $n(n-1)$.

如果逻辑变量 u_n 或者 $\overline{u}_n$ 在 C 中的某个子句出现了, 那么我们可以将 C 写为

$$C = G \cdot (u_n + y_1) \cdots (u_n + y_k) \cdot (\overline{u}_n + z_1) \cdots (\overline{u}_n + z_h) \tag{1}$$

其中 G 是 C 的一部分子句, 而且不会出现逻辑变量 u_i 或者 $\overline{u}_i$. 于是我们令

$$C' = G \cdot \prod_{1 \le i \le k} \prod_{1 \le j \le h} (y_i + z_j) \tag{2}$$

(2) 式中不再含有变量 u_n 和 $\overline{u}_n$ 了. 记 $U' = \{u_1, u_2, \cdots, u_{n-1}\}$, 如果存在 U 的真赋值, 使得 C 满足, 那么一定存在 U' 的真赋值使得 C' 满足.

这是因为若 u_n 取真值, 则所有的 z_j 都必然取真值; 若 u_n 取假值, 则所有的 y_i 都必然取真值. 因此, 这两种情况中 C' 的乘积部分一定为真值. 反过来看, 假设存在 U' 的真赋值使得 C' 满足.

- 此时若有某个 y_i 取假值, 则所有的 z_j 都必然取真值. 于是令 u_n 取真值来得到 U 的真赋值, 使得 C 满足;
- 然而若有某个 z_j 取假值, 那么令 u_n 取假值可得到 U 的真赋值来使得 C 满足;
- 如果所有的 y_i 和 z_j 都取真值, 那么令 u_n 取假值即可得到 U 的真赋值来使得 C 满足.

于是我们可以得出结论: **C 和 C' 的可满足性是等价的**. 但是 C' 涉及到的变量数比 C 对应的少 1, 子句数为 $m - (k + h) + kh$. 但是可以像前面那样简化掉所有形如 $u_i + \overline{u}_i$ 的子句, 因此可以假定 C' 中子句个数不超过 $(n-1)(n-2)$.

类似的, 我们一直持续上述过程, 一直进行到判定只含有 1 个逻辑变量的逻辑语句可满足性问题. 然而这只需要常数时间. 注意到上面每一步简化都可以在 (关于 n 的) 多项式时间内完成, 总共最多需要 $n-1$ 次化简. 因此在多项式时间内可以完成 **2SAT** 问题的判定, 即 **2SAT** 问题是 $\mathcal{P}$ 类问题, □.

Problem 2

碰撞集问题: 给定一组集合 $\{S_1, S_2, \cdots, S_n\}$ 和预算 b, 问是否存在一个集合 H, 其大小不超过 b, 且 H 和所有 $S_i (i = 1, 2, \cdots, n)$ 相交非空. 请证明碰撞集问题是 $\mathcal{NPC}$ 问题.

Solution:

- **先证明该问题是一个 $\mathcal{NP}$ 问题**: 假设给出集合 H 的所有元素, 显然可以在**多项式时间内**验证该集合 H 是否满足条件要求 (和 S_i 逐一比较是否有交集并检查规模是否超过 b), 所以该问题 $\in \mathcal{P}$ 问题 $\subseteq \mathcal{NP}$ 问题.
- **再利用一个已知的 $\mathcal{NPC}$ 问题, 将其 (在多项式时间内) 归约到目标问题**: 已知**图的顶点覆盖问题 (VC)** 是 $\mathcal{NPC}$ 问题, 只要找到一种把 VC 问题归约到碰撞集问题的多项式方法, 即可证明碰撞集问题是 $\mathcal{NPC}$ 问题. 具体的归约方式构造如下:

  假设有图 $G = (V, E)$, 则把该图的每一条边对应一个集合 S_i, 该边的两点作为该集合的元素, 即每个集合都有两个元素 (如 $S_1 = \{v_1, v_2\}$). 这样就可以构造出 $|E|$ 个集合 $\{S_1, S_2, \cdots, S_{|E|}\}$, 再将 VC 问题中覆盖的顶点数上限 K 作为预算 b, 并把图 $G = (V, E)$ 的顶点覆盖中的所有点作为集合 H 的元素. 另外还需要说明一下上述多项式变换的充分必要性:

  - **当碰撞集问题有解时, 则顶点覆盖问题就有解:** 只需要选取碰撞集的解 H 对应的所有点, 即为对应顶点覆盖问题的解;
  - **当顶点覆盖问题有解时, 则碰撞集问题就有解:** 当顶点覆盖问题有解 V 时, 则将 V 中的每个顶点对应到所生成的那一组集合 (即 $\{S_1, S_2, \cdots, S_{|E|}\}$) 中的元素, 从而得到集合 H, 即为碰撞集问题的解.

这样就把 VC 问题归约到碰撞集问题了, 而 VC 问题是 $\mathcal{NPC}$ 问题, 因此碰撞集问题是 $\mathcal{NPC}$ 问题, □.

Problem 3

0-1 整数规划问题: 给定 $m \times n$ 的矩阵 A, n 维整数列向量 c, m 维整数列向量 b 以及整数 D, 问是否存在 n 维 0-1 列向量 x, 使得 $Ax \leq b$ 且 $c^{\mathrm{T}}x \geq D$. 请证明 0-1 整数规划问题是 $\mathcal{NPC}$ 问题.

Solution:

- **先证明该问题是一个 $\mathcal{NP}$ 问题**: 用非确定性图灵机遍历所有可行解, 即可在多项式时间内把解给暴力求出来. 于是 0-1 整数规划问题即为 $\mathcal{NP}$ 问题.
- **再利用一个已知的 $\mathcal{NPC}$ 问题, 将其归约到目标问题**: 为了简化归约过程, 这里我们采用**图的团问题**作为已知的 $\mathcal{NPC}$ 问题[1]. 对于团问题的每一个实例 I: 无向图 $G = (V, E)$ 和非负整数 $J \leq |V|$, 其中 $V = \{v_1, v_2, \cdots, v_n\}$. 则在 0-1 整数规划中对应的实例 $f(I)$ 为:

$$\begin{cases} \sum_{i=1}^{n} x_i \geq J \\ x_i + x_j \leq 1, \quad (v_i, v_j) \notin E, i \neq j \\ x_i = 0, 1, \quad i = 1, 2, \cdots, n \end{cases}$$

  显然 f 是多项式时间内可计算的. 现在来证明一下多项式归约 f 的充要性:

[1] VC 到团的多项式归约变换 f: 对 VC 的每一个实例 I, 无向图 $G = (V, E)$ 和非负整数 $K \leq |V|$. 而团对应的实例 $f(I)$: 无向图 $\overline{G}(V, \overline{E})$ 和 $J = |V| - K$. 由于 VC 是 $\mathcal{NPC}$ 问题, 因此团就是 $\mathcal{NPC}$ 问题.

– 设 $V' \subseteq V$ 是 G 的一个团且 $|V'| \geq J$, 令

$$x_i = \begin{cases} 1, & \text{若} v_i \in V' \\ 0, & \text{否则} \end{cases}$$

于是当 $(v_i, v_j) \notin E$ 时, 则有 $x_i + x_j \leq 1$ 且 $\sum_{i=1}^{n} x_i = \left|V'\right| \geq J$;

– 反之, 取 $V' = \{v_i | x_i = 1\}$, 则 $\forall v_i, v_j \in V', x_i + x_j = 2$, 从而 $(v_i, v_j) \in E$.

又因为 $\left|V'\right| = \sum_{i=1}^{n} x_i \geq J$. 即 $V'(\subseteq V)$ 就是 G 的一个顶点数不小于 J 的团.

至此, 图的团问题就归约到 0-1 整数规划问题了, 由于图的团问题是 $\mathcal{NPC}$ 问题, 因此 0-1 整数规划问题也是 $\mathcal{NPC}$ 问题, □.

Problem 4

独立集问题: 任给一个无向图 $G = (V, E)$ 和非负整数 $J \leq |V|$, 问 G 中是否存在顶点数不小于 J 的独立集. 请证明独立集问题是 $\mathcal{NPC}$ 问题.

Solution:

- **先证明该问题是一个 $\mathcal{NP}$ 问题**: 用非确定性图灵机即可完成独立集问题的判定, 故独立集问题显然是 $\mathcal{NP}$ 问题;
- **再利用一个已知的 $\mathcal{NPC}$ 问题, 将其归约到目标问题**: 为了归约过程简单, 这里我们选取**图的顶点覆盖问题**作为参照物[2]. 归约过程为: 任给顶点覆盖问题的一个实例, 它是由无向图 $G = (V, E)$ 和非负整数 $K \leq |V|$ 组成, 对应独立集问题的实例由无向图 $G = (V, E)$ 和非负整数 $J = |V| - K$ 组成. 并且显然该变换是充要的, 因此图的顶点覆盖问题就归约到独立集问题了, 由于顶点覆盖问题是 $\mathcal{NPC}$ 问题, 因此独立集问题问题也是 $\mathcal{NPC}$ 问题, □.

Problem 5

证明无向图的 Hamilton 圈问题 (HC) 是 $\mathcal{NPC}$ 问题, 并以此为基础来证明 TSP 判定问题是 $\mathcal{NPC}$ 问题.

Solution: 利用非确定性图灵机可以得到 HC$\in \mathcal{NP}$. 我们以**有向 HC 问题**为参照物, 将其归约到 HC 问题. 任给一个有向图 $D = (V, E)$, 要构造一个无向图 $G = (V', E')$ 使得 D 有哈密顿回路当且仅当 G 有哈密顿回路. 因此关键在于如何用无向边来表示有向边. 故而把 D 的每一个顶点 v 替换成 3 个顶点 $v^{\text{in}}, v^{\text{mid}}, v^{\text{out}}$, 用边连接 v^{in} 和 v^{mid} 以及 v^{mid} 和 v^{out}. D 中的每一条有向边 $\langle u, v\rangle$ 在 G 中替换成 $\left(u^{\text{out}}, v^{\text{in}}\right)$, 即有

$$V' = \left\{v^{\text{in}}, v^{\text{mid}}, v^{\text{out}} | v \in V\right\}, \quad E' = \left\{\left(u^{\text{out}}, v^{\text{in}}\right) | \langle u, v\rangle \in E\right\} \cup \left\{\left(v^{\text{in}}, v^{\text{mid}}\right), \left(v^{\text{mid}}, v^{\text{out}}\right) | v \in V\right\}$$

显然上述的归约变换[3]是多项式时间的且 G 可以满足要求 (D 有哈密顿回路当且仅当 G 有哈密顿回路), 因此 HC 问题是 $\mathcal{NPC}$ 问题.

[2]不用像参考答案 (以团问题为参照物来归约到独立集问题) 那样复杂.

[3]此处使用的方法称为**局部替换法**, 当两个问题的结构类似时 (例如有向 HC 问题和 (无向)HC 问题), 往往可以通过这种方法构造多项式归约变换.

由于已知 TSP 判定问题 $\in \mathcal{NP}$, 所以我们考虑以 HC 问题为起点, 将其归约到 TSP 判定问题. 构造 HC 到 TSP 的多项式变换 f: 对 HC 的每一个实例 I, I 是一个无向图 $G = (V, E)$, TSP 对应的实例 $f(I)$ 为: 城市集合 V, 任意两个不同的城市 u 和 v 之间的距离为

$$d(u, v) = \begin{cases} 1, & \text{若 } (u, v) \in E \\ 2, & \text{若 } (u, v) \notin E \end{cases}$$

以及界限 $D = |V|$. 显然 f 是多项式时间内可计算的, 又因为 $f(I)$ 中每一个城市恰好经过一次的巡回路线有 $|V|$ 条边, 每条边的长度为 1 或 2. 故而巡回路线的长度至少为 D. 因此巡回路线的长度不超过 D 当且仅当巡回路线的长度为 D, 当且仅当它的每条边长度都为 1, 当且仅当它是 G 中的一条哈密顿回路. 综上所述, $I \in Y_{\text{HC}} \Leftrightarrow f(I) \in Y_{\text{TSP}}$, 即多项式规约变换 f 的充要性是满足的. 又因为 HC 问题是 $\mathcal{NPC}$ 问题, 故 TSP 判定问题也是 $\mathcal{NPC}$ 问题.

Problem 6

0-1 背包 (优化) 问题: 有 n 个物品, 他们各自的体积 w_i 和价值 p_i, 现有给定容量 M 的背包, 如何将背包装入的物品具有最大价值总和? 请说明 0-1 背包问题是 $\mathcal{NP}$ 困难问题, 但不是 $\mathcal{NPC}$ 问题.

Solution: 要验证一个可行解是否为下述问题的最优解

$$\begin{cases} \max \sum_{i=1}^{n} x_i p_i \\ \sum_{i=1}^{n} x_i w_i \le M \\ x_i = 0, 1 \ (i = 1, \cdots, n) \end{cases}$$

则需要**比较所有的可行解** $X = (x_1, x_2, \cdots, x_n), x_i \in \{0, 1\}$. 这最多有 2^n 种可能, 而基于比较的求最大值问题的复杂度下界为 $\Omega(M)$, 因此本问题对于“最优解”的正确性验证需要消耗的时间下界为 $O(2^n)$. 即不存在多项式时间算法, 使得其能够验证解的正确性. 也就是说, 0-1 背包 (优化) 问题不是 $\mathcal{NP}$ 的, 所以肯定就不是 $\mathcal{NPC}$ 的.

与此同时, **0-1 背包优化问题不会比 0-1 背包判定问题更容易**, 然而 0-1 背包判定问题是 $\mathcal{NPC}$ 的. 因此 0-1 背包优化问题是 $\mathcal{NPH}$ 的, 但不是 $\mathcal{NPC}$ 的.

Problem 7

$\mathcal{NPC}$ 问题一定是 $\mathcal{NPH}$ 问题么?

Solution: $\mathcal{NPC}$ 问题一定是 $\mathcal{NPH}$ 问题. $\mathcal{P}, \mathcal{NP}, \mathcal{NPC}, \mathcal{NPH}$ 问题的关系如下图1所示:

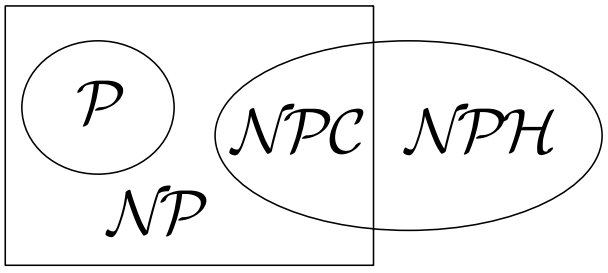


图 1: P-NP-NPC-NPH 问题关系图

Problem 8

对于给定 $x \neq 0$, 求 n 次多项式 $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ 的值.

- 设计一个最坏情况下时间复杂度为 $\Theta(n)$ 的求值算法;
- 证明任何求值算法的时间复杂度都是 $\Omega(n)$.

Solution: (1). 采用秦九韶算法, 递归计算如下:

$$\begin{aligned}
P_0(x) &= a_n \\
P_1(x) &= P_0(x) \cdot x + a_{n-1} \\
P_2(x) &= P_1(x) \cdot x + a_{n-2} \\
P_3(x) &= P_2(x) \cdot x + a_{n-3} \\
&\vdots \\
P_n(x) &= P_{n-1}(x) \cdot x + a_{n-3}
\end{aligned}$$

上述算法用一个 for 循环就可以实现, 并且循环体内部的操作就只有加法和乘法, 即循环体消耗常数时间并且循环次数为 $n+1$. 因此算法在最坏情形下的时间复杂度为 $\Theta(n)$.

(2). 多项式 $P(x)$ 有 $n+1$ 个系数, 对于任意一个算法 $\mathcal{A}$, 都需要对**每一个系数**至少做一次处理, 否则算法就可能得到错误的输出. 即任何正确的算法都至少需要 $n+1$ 次运算, 即最坏情况下的时间复杂度下界为 $\Omega(n)$.

Problem 9

写出下述优化问题对应的判定问题, 并证明这些判定问题 $\in \mathcal{NP}$:

- **最长回路优化问题**: 任给无向图 G, 在 G 中找到一条最长的初级 (即顶点不重复的) 回路.
- **图着色优化问题**: 任给无向图 $G = (V, E)$, 给 G 的每一个顶点涂一种颜色, 要求任一条边的两个端点的颜色都不相同. 如何用最少的颜色给 G 的顶点着色? 即求映射 $f : V \to \mathbb{Z}^+$ 满足条件 $\forall (u, v) \in E, f(u) \neq f(v)$, 且使 $\operatorname{card}\{f(u) | u \in V\}$ 最小.

Solution: 上述优化问题对应的判定问题为:

- **最长回路判定问题**: 任给无向图 $G = (V, E)$ 和正整数 $L \leq |V|$, 在 G 中能否找到一条长度 $\geq L$ 的初级 (即顶点不重复的) 回路?
- **最长回路判定问题**的非确定性多项式时间算法 $\mathcal{A}$: 猜想对 G 的任意一条初级回路 C, 若 C 的长度 $\geq L$, 则回答“yes”, 否则回答“no”. 显然该问题 $\in \mathcal{NP}$.
- **图着色判定问题**: 任给无向图 $G = (V, E)$ 和正整数 K, 给 G 的每一个顶点涂一种颜色, 要求任一条边的两个端点的颜色都不相同. 能否用不超过 K 种颜色给 G 的顶点着色? 即是否存在映射 $f : V \to \mathbb{Z}^+$ 满足条件 $\forall (u, v) \in E, f(u) \neq f(v)$, 且使 $\operatorname{card}\{f(u) | u \in V\} \leq K$?
- **图着色判定问题**的非确定性多项式时间算法 $\mathcal{B}$: 猜想用不超过 K 种颜色给 G 的每一个顶点涂色, 检查每一条边的两个端点的颜色是否都不相同. 若是, 则回答“yes”, 否则回答“no”. 即猜想一个单射 $f : V \to \{1, 2, \cdots, K\}$, 若满足条件 $\forall (u, v) \in E, f(u) \neq f(v)$, 则回答“yes”, 否则回答“no”. 于是可知该问题 $\in \mathcal{NP}$.

中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 9&10&11 课程作业

2022 年 12 月 1 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

设集合 $S = \{x_1, x_2, \cdots, x_n\}$, x_i 出现的概率为 $0 < p_i < 1, \sum_{i=1}^{n} p_i = 1$. 试设计一个算法, 按照 S 的概率分布对 S 的元素进行随机选择 (一个).

Solution: 可以设计出以下的随机算法, 代码有 C++ 版本、也有 python 版本 (对应的输出结果紧跟代码之后):

```cpp
#include <bits/stdc++.h> //万能头文件
using namespace std;

int random_pick(vector<pair<int, double>> nums_probs) {
    int res;
    random_device rd; //定义一个非确定性的真随机生成器
    uniform_real_distribution<double> U(0, 1); //随机数分布对象
    double x = U(rd); //在 [0,1) 内随机选取实数作为 x
    double cumu_prob = 0.0; //初始化累积概率
    for(auto num_prob : nums_probs) {
        cumu_prob += num_prob.second;
        if(x < cumu_prob) {
            res = num_prob.first;
            break;
        }
    }
    return res;
}

int main() {
    vector<pair<int, double>> nums_probs = {
        {1, 0.2}, {2, 0.1}, {3, 0.6}, {4, 0.1}
    };
    cout << " 自写算法的抽取结果: " << random_pick(nums_probs) << endl;
    random_device rd;
    discrete_distribution<> d({0.2, 0.1, 0.6, 0.1});
    cout << " 调用 C++11 中算法的抽取结果: " << d(rd) + 1 << endl;
}
```

上述 C++ 代码对应的输出结果为:

```
开始运行...
自写算法的抽取结果: 3
调用 C++11中算法的抽取结果: 3
运行结束.
```

```python
import random
import numpy as np

def random_pick(some_list, probabilities):
    x = random.uniform(0, 1)
    cumulative_probability = 0.0
    for item, item_probability in zip(some_list, probabilities):
        cumulative_probability += item_probability
        if x < cumulative_probability: break
    return item

if __name__ == '__main__':
    some_list = [1, 2, 3, 4]
    probabilities = [0.2, 0.1, 0.6, 0.1]
    print(" 自写算法的抽取结果: ", random_pick(some_list, probabilities))
    print(" 调用 numpy 中算法的抽样结果: ", np.random.choice(some_list, 1, probabilities))
```

上述 python 代码对应的输出结果为:

```
开始运行...
自写算法的抽取结果: 3
调用 numpy 中算法的抽样结果: [3]
运行结束.
```

显然该随机算法的复杂度主要取决于 for 循环, 因此上述算法的时间复杂度为 $O(n)$, 其中 n 为集合 S 中的元素个数. 而空间复杂度显然为 $O(1)$.

Problem 2

设计概率算法, 求解 $365!/\left(340!\cdot 365^{25}\right)$.

Solution: 该问题的数是概率论中著名的生日问题的解答. 在 k 个人中, 至少 2 个人有相同生日的概率为

$$1-\frac{365!}{(365-k)!\cdot 365^k}$$

根据斯特林公式以及近似公式:

$$n!\sim\sqrt{2\pi n}\left(\frac{n}{e}\right)^n(n\to\infty),\quad \ln(1+x)\sim x-\frac{x^2}{2}(x\to 0)$$

因此有:

$$\begin{aligned}\frac{n!}{(n-k)!\cdot n^k}&\sim\frac{\sqrt{2\pi n}}{\sqrt{2\pi(n-k)}}\cdot\frac{\left(\frac{n}{e}\right)^n}{\left(\frac{n-k}{e}\right)^{n-k}\cdot n^k}\\&\sim\left(\frac{n}{n-k}\right)^{n-k}\cdot e^{-k}=\exp\left[(n-k)\ln\left(1+\frac{k}{n-k}\right)-k\right]\\&\sim\exp\left\{(n-k)\left[\frac{k}{n-k}-\frac{\frac{k^2}{(n-k)^2}}{2}\right]-k\right\}\\&\sim\exp\left\{-\frac{k^2}{2(n-k)}\right\}\sim\exp\left\{-\frac{k^2}{2n}\right\}\end{aligned}$$

因此可得

$$\left.\frac{n!}{(n-k)!\cdot n^k}\right|_{n=365}^{k=25} = \frac{365!}{340!\cdot 365^{25}} \approx \exp\left\{-\frac{25^2}{730}\right\} = 0.424788$$

Problem 3

设计一个 Las Vegas 随机算法, 求解电路板布线问题. 将该算法与分支限界算法结合, 观察求解效率.

Solution: 该算法的设计核心是: **采用随机放置位置策略并结合分支限界算法**. 算法的 C++ 代码如下所示:

```cpp
#include <bits/stdc++.h> //万能头文件, 刷题时可以用, 大型项目千万不能用
using namespace std;

//表示方格位置上的结构体
struct position{
    int row;
    int col;
};

//分支限界算法
bool FindPath(
    position start, position finish, int &PathLen,
    position *&path, int **grid, int m, int n
) { //找到最短布线路径, 若找得到则返回 true, 否则返回 false
    //起点终点重合则不用布线
    if((start.row == finish.row) && (start.col == finish.col)) {
        PathLen = 0;
        return true;
    }

    //设置方向移动坐标值: 东南西北
    position offset[4];
    offset[0].row = 0;
    offset[0].col = 1; //右移
    offset[1].row = 1;
    offset[1].col = 0; //下移
    offset[2].row = 0;
    offset[2].col = -1; //左移
    offset[3].row = -1;
    offset[3].col = 0; //上移

    int NumNeighBlo = 4; //相邻的方格数
    position here, nbr;
    here.row = start.row; //设置当前方格, 即搜索单位
    here.col = start.col;
    //由于 0 和 1 用于表示方格的开放和封闭, 故距离: 2-0 3-1
    grid[start.row][start.col] = 0; //-2 表示强, -1 表示可行, -3 表示不能当作路线
    //队列式搜索, 标记可达相邻方格
    queue<position> q_FindPath;
```

```
    do {
        int num = 0; //方格未标记个数
        position selectPosition[5]; //保存选择位置
        for(int i = 0; i < NumNeighBlo; i++) {
            //到达四个方向
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == -1) { //该方格未标记
                grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
                if((nbr.row == finish.row) && (nbr.col == finish.col)) {
                    break;
                }
                selectPosition[num].row = nbr.row;
                selectPosition[num].col = nbr.col;
                num++;
            }
        }
        if(num > 0) { //如果标记，则在这么多个未标记个数中随机选择一个位置（本算法核心）
            q_FindPath.push(selectPosition[rand()%(num)]); //随机选一个入队
        }
        if((nbr.row == finish.row) && (nbr.col == finish.col)) {
            break; //是否到达目标位置 finish
        }
        //判断活结点队列是否为空
        if(q_FindPath.empty() == true) return false; // 无解
        //访问队首元素出队
        here = q_FindPath.front();
        q_FindPath.pop();
    } while (true);

    //构造最短布线路径
    PathLen = grid[finish.row][finish.col];
    path = new position[PathLen]; //路径
    //从目标位置 finish 开始向起始位置回溯
    here = finish;
    for(int j = PathLen - 1; j >= 0; j--) {
        path[j] = here;
        //找前驱位置
        for(int i = 0; i <= NumNeighBlo; i++) {
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == j) {//距离 +2 正好是前驱位置
                break;
            }
        }
        here = nbr;
    }
    return true;
}
```

```
int main() {
    cout << "---------分支限界算法之布线问题---------" << endl;
    int path_len, path_len1, m, n;
    position *path, *path1, start, finish, start1, finish1;
    cout << " 在一个 m*n 的棋盘上, 请分别输入 m 和 n, 代表行数和列数, 然后输入回车" << endl;
    cin >> m >> n;
    //创建棋盘格
    int **grid = new int * [m + 2], **grid1 = new int * [m + 2];
    for(int i = 0; i < m + 2; i++) {
        grid[i] = new int[n + 2];
        grid1[i] = new int[n + 2];
    }
    //初始化棋盘
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            grid[i][j] = -1;
        }
    }
    //设置方格阵列的围墙
    for(int i = 0; i <= n + 1; i++) {
        grid[0][i] = grid[m + 1][i] = -2; //上下的围墙
    }
    for(int i = 0; i <= m + 1; i++) {
        grid[i][0] = grid[i][n + 1] = -2; //左右的围墙
    }
    cout << " 初始化棋盘格和加围墙" << endl;
    cout << "------------------" << endl;
    for(int i = 0; i < m + 2; i++) {
        for(int j = 0; j < n + 2; j++) {
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }
    cout << "------------------" << endl;
    cout << " 请输入已经占据的位置 (行坐标, 列坐标), 代表此位置不能布线" << endl;
    cout << " 例如输入 2 2(然后输入回车), 表示坐标 (2,2) 不能布线;" <<
    " 当输入坐标为 0 0(然后输入回车) 表示结束输入" << endl;
    //添加已经布线的棋盘格
    while(true) {
        int ci, cj;
        cin >> ci >> cj;
        if(ci > m || cj > n) {
            cout << " 输入非法!";
            cout << " 行坐标 <" << m << ", 列坐标 <" << n << " 当输入的坐标为 0,0 时结束输入"
↪   << endl;
            continue;
        } else if (ci == 0 || cj == 0) {
            break;
        } else {
            grid[ci][cj] = -3;
        }
    }
```

```
//布线前的棋盘格
cout << " 布线前的棋盘格" << endl;
cout << "------------------" << endl;
for(int i = 0; i < m + 2; i++) {
    for(int j = 0; j < n + 2; j++) {
        cout << grid[i][j] << " ";
    }
    cout << endl;
}
cout << "------------------" << endl;
cout << " 请输入起点位置坐标" << endl;
cin >> start.row >> start.col;
cout << " 请输入终点位置坐标" << endl;
cin >> finish.row >> finish.col;
clock_t starttime, endtime;
starttime = clock(); //程序开始时间
srand((unsigned) time (NULL)); //初始化时间种子, 是随机选择的关键
int time = 0; //为假设运行次数
start1 = start, finish1 = finish, path_len1 = path_len, path1 = NULL; //初始值拷贝
for(int i = 0; i < m + 2; i++) {
    for(int j = 0; j < n + 2; j++) {
        grid1[i][j] = grid[i][j];
    }
}
bool result = FindPath(start1, finish1, path_len1, path1, grid1, m, n);
while(result == 0 && time < 1000) { //尝试次数最多为 1000 次
    //初始值拷贝
    start1 = start, finish1 = finish, path_len1 = path_len, path1 = NULL;
    for(int i = 0; i < m + 2; i++) {
        for(int j = 0; j < n + 2; j++) {
            grid1[i][j] = grid[i][j];
        }
    }
    time++;
    cout << endl;
    cout << " 没有找到路线, 第" << time << " 次尝试" << endl;
    result = FindPath(start1, finish1, path_len1, path1, grid1, m, n);
}
endtime = clock(); //程序结束时间

if(result == 1) {
    cout << "------------------" << endl;
    cout << "$ 代表围墙" << endl;
    cout << "# 代表已经占据的点" << endl;
    cout << "* 代表布线路线" << endl;
    cout << "= 代表还没有布线的点" << endl;
    cout << "------------------" << endl;
    for(int i = 0; i <= m + 1; i++) {
        for(int j = 0; j <= n + 1; j++) {
            if(grid1[i][j] == -2) cout << "$ ";
            else if(grid1[i][j] == -3) cout << "# ";
```

```
                else {
                    int r;
                    for(r = 0; r < path_len1; r++) {
                        if(i == path1[r].row && j == path1[r].col) {
                            cout << "* ";
                            break;
                        }
                        if(i == start1.row && j == start1.col) {
                            cout << "* ";
                            break;
                        }
                    }
                    if(r == path_len1) cout << "= ";
                }
            }
            cout << endl;
        }
        cout << "-------------------" << endl;
        cout << " 路径坐标和长度" << endl;
        cout << endl;
        cout << "(" << start1.row << "," << start1.col << ")" << " ";
        for(int i = 0; i < path_len1; i++) {
            cout << "(" << path1[i].row << "," << path1[i].col << ")" << " ";
        }
        cout << endl;
        cout << endl;
        cout << " 路径长度: " << path_len1 + 1 << endl;
        cout << endl;
        time++;
        cout << " 布线完毕, 共查找" << time << " 次" << endl;
        cout << " 算法运行时间为: " << (endtime - starttime) << "ms" << endl;
    } else {
        cout << endl;
        cout << " 经过多次尝试, 依然没有找到路线" << endl;
    }
    return 0;
}
```

上述代码的关键之处在于:

- P5 页的第 13 行代码, 这里是当前点相邻四个点是否可以放置, 如果可以放置用 selectPostion 保存下来, 并用 num 记录有多少个位置可以放置;
- P5 页的第 19 行代码, 这里利用上面保存的可以放置的点, 然后**随机选取其中一个加入队列**, 这就是 Las Vegas 算法的精髓;
- P7 页的第 17 行代码, 作用是初始化时间种子, 是伪随机生成器的关键, 即是随机选择的关键.

结果分析:

- 测试样例 1: 3×3 棋盘, 代码的交互输出过程如下:

```
开始运行...
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上，请分别输入 m 和 n，代表行数和列数，然后输入回车
3 3
初始化棋盘格和加围墙
-----------------
-2 -2 -2 -2 -2
-2 -1 -1 -1 -2
-2 -1 -1 -1 -2
-2 -1 -1 -1 -2
-2 -2 -2 -2 -2
-----------------
请输入已经占据的位置(行坐标，列坐标)，代表此位置不能布线
例如输入 2 2(然后输入回车)，表示坐标(2,2)不能布线;当输入坐标为 0 0(然后输入回车)表示结束输
↪  入
2 1
2 3
3 3
0 0
布线前的棋盘格
-----------------
-2 -2 -2 -2 -2
-2 -1 -1 -1 -2
-2 -3 -1 -3 -2
-2 -1 -1 -3 -2
-2 -2 -2 -2 -2
-----------------
请输入起点位置坐标
1 1
请输入终点位置坐标
3 1

没有找到路线，第 1 次尝试

没有找到路线，第 2 次尝试
-----------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
-----------------
$ $ $ $ $
$ * * = $
$ # * # $
$ * * # $
$ $ $ $ $
-----------------
路径坐标和长度
(1,1) (1,2) (2,2) (3,2) (3,1)
路径长度: 5
布线完毕，共查找 3 次
算法运行时间为: 39ms
运行结束.
```

- 测试样例 2: 5×5 棋盘, 代码的交互输出过程如下:

```
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上, 请分别输入 m 和 n, 代表行数和列数, 然后输入回车
5 5
请输入已经占据的位置(行坐标, 列坐标), 代表此位置不能布线
例如输入 2 2(然后输入回车), 表示坐标(2,2)不能布线;当输入坐标为 0 0(然后输入回车)表示结束输入
3 1
3 2
3 4
3 5
4 5
0 0
布线前的棋盘格
------------------
-2 -2 -2 -2 -2 -2 -2
-2 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -2
-2 -3 -3 -1 -3 -3 -2
-2 -1 -1 -1 -1 -3 -2
-2 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
5 2
没有找到路线, 第 1 次尝试
没有找到路线, 第 2 次尝试
没有找到路线, 第 3 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $ $ $
$ * = = = = $
$ * * * = = $
$ # # * # # $
$ = = * = # $
$ = * * = = $
$ $ $ $ $ $ $
------------------
路径坐标和长度
(1,1) (2,1) (2,2) (2,3) (3,3) (4,3) (5,3) (5,2)
路径长度: 8
布线完毕, 共查找 4 次
算法运行时间为: 61ms
```

- 测试样例 2: 10×10 棋盘, 代码的交互输出过程如下:

```
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上，请分别输入 m 和 n，代表行数和列数，然后输入回车
10 10
布线前的棋盘格
------------------
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -3 -3 -3 -3 -1 -1 -3 -3 -3 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
9 9

没有找到路线，第 1 次尝试
没有找到路线，第 2 次尝试
没有找到路线，第 3 次尝试
没有找到路线，第 4 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $ $ $ $ $ $ $ $
$ * = = = = = = = = = $
$ * * * = = = = = = = $
$ = = * * = = = = = = $
$ = = = * * = = = = = $
$ # # # # * = # # # = $
$ = = = = * = = = = = $
$ = = = = * * * = = = $
$ = = = = = = * = = = $
$ = = = = = = * * * = $
$ = = = = = = = = = = $
$ $ $ $ $ $ $ $ $ $ $ $
------------------
路径坐标和长度
(1,1) (2,1) (2,2) (2,3) (3,3) (3,4) (4,4) (4,5) (5,5) (6,5) (7,5) (7,6) (7,7) (8,7)
↪  (9,7) (9,8) (9,9)
路径长度: 17
布线完毕，共查找 5 次
算法运行时间为: 73ms
```

由此可见, 结合随机化和分支限界的 Las Vegas 算法的求解效率还是相当不错的.

Problem 4

判断正误:

- Las Vegas 算法不会得到不正确的解. ( )
- Monte Carlo 算法不会得到不正确的解. ( )
- Las Vegas 算法总能求得一个解. ( )
- Monte Carlo 算法总能求得一个解. ( )

Solution:

- 正确, 拉斯维加斯算法不会得到不正确的解. 一旦用拉斯维加斯算法找到一个解, 这个解就一定是正确解. 但有时用拉斯维加斯算法找不到解.
- 错误, Monte Carlo 算法每次都能得到问题的解, 但不保证所得解的正确性. 请注意, 可以在 Monte Carlo 算法给出的解上加一个验证算法, 如果正确就得到解, 如果错误就不能生成问题的解, 这样 Monte Carlo 算法便转化为了 Las Vegas 算法.
- 错误, Las Vegas 算法并不能保证每次都能得到一个解, 但是如果一旦某一次得到解, 那么就一定是正确的.
- 正确, Monte Carlo 算法每次运行都能给出一个解, 但正确性就不能保证了.

Problem 5

设 Las Vegas 算法获得解的概率为 $p(x) \geq \delta, 0 < \delta < 1$, 则调用 k 次算法后, 获得解的概率为:_____.
Solution: 不妨求一下调用 k 次算法后, 求解失败 (即 k 次调用都求解失败) 的概率:

$$P\left(\text{失败}\right) = (1 - p\,(x))^k \leq (1 - \delta)^k \Rightarrow P\left(\text{成功}\right) = 1 - P\left(\text{失败}\right) \geq 1 - (1 - \delta)^k$$

即获得解的概率至少为 $1 - (1 - \delta)^k \to 1$(当$k \to \infty$).

Problem 6

对于判定问题 Π 的 Monte Carlo 算法, 当返回 false(true) 时解总是正确的, 但当返回 true(false) 时解可能有错误, 该算法是 _____.

(A) .偏真的*Monte Carlo*算法　　(B) .偏假的*Monte Carlo*算法
(C) .一致的*Monte Carlo*算法　　(D) .不一致的*Monte Carlo*算法

Solution: 答案选 B, 只要将偏真的 Monte Carlo 算法的定义中的 true/false 互换即可得到偏真的 Monte Carlo 算法的定义.

Problem 7

判断正误:

- 一般情况下, 无法有效判定 Las Vegas 算法所得解是否正确. ( )
- 一般情况下, 无法有效判定 Monte Carlo 算法所得解是否正确. ( )
- 虽然在某些步骤引入随机选择, 但 Sherwood 算法总能求得问题的一个解, 且所求得的解总是正确的. ( )
- 虽然在某些步骤引入随机选择, 但 Sherwood 算法总能求得问题的一个解, 但一般情况下, 无法有效判定所求得的解是否正确. ( )

Solution:

- 错误, Las Vegas 算法并不能保证每次都能得到解, 但是如果一旦某一次得到解, 那么就一定是正确的.
- 错误, 虽然 Monte Carlo 算法每次运行都能给出一个解, 可能是错的也可能是对的, 但是可以通过检验解来有效判定其正确性. 判定解的正确性跟算法本身没有多大关系, 只要代进去验证即可. 特殊点在于, 只要 Las Vegas 算法求得解了, 那么就一定是正确的, 就不用再浪费时间来判定了; 但是对于 Monte Carlo 算法的所得解, 必须要进行正确性检验.
- 正确, Sherwood 算法总能求得问题的一个解, 且所求得的解总是正确的.
- 错误.

Problem 8

装箱问题: 任给 n 件物品, 物品 j 的重量为 $w_j, 1 \leq j \leq n$, 限制每只箱子装入物品的总重量不超过 B, 其中 B 和 w_j 都是正整数, 且 $w_j \leq B, 1 \leq j \leq n$. 要求用最少的箱子装入所有物品, 怎么装法?

考虑下述近似算法-**首次适合算法 (FF)**: 按照输入顺序装物品, 对每一件物品, 依次检查每一只箱子, 只要能装得下就把它装入, 只有在所有已经打开的箱子都装不下这件物品时, 才打开一个新箱子. 证明: **FF 算法**是 2-近似的, 即任给实例 I, 都有

$$\mathbf{FF}(I) < 2\mathrm{OPT}(I)$$

Solution: 当 $\mathbf{FF}(I) = 1$ 时, 显然 $\mathbf{FF}(I) = \mathrm{OPT}(I) = 1$. 如果 $\mathbf{FF}(I) > 1$, 记 $W = \sum_{i=1}^{n} w_i$. 因为任何两只箱子的重量之和大于 B.

- 因此当 $\mathbf{FF}(I)$ 为偶数时,
$$W > \frac{B}{2}\mathbf{FF}(I)$$
- 当 $\mathbf{FF}(I)$ 为奇数时, 设最重的箱子重量为 B_1, 则有
$$W > \frac{B}{2}\left(\mathbf{FF}\left(I\right) - 1\right) + B_1 > \frac{B}{2}\mathbf{FF}\left(I\right)$$

故总有 $W > \frac{B}{2}\mathbf{FF}(I)$, 即 $\mathbf{FF}(I) < \frac{2W}{B}$. 而显然 $\mathrm{OPT}(I) \geq \frac{W}{B}$, 因此证得 $\mathbf{FF}(I) < 2\mathrm{OPT}(I)$, □.

Problem 9

设无向图 $G = \langle V, E\rangle, V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset$, 称

$$(V_1, V_2) = \left\{(u, v) \,\middle|\, (u, v) \in E, \text{且} u \in V_1, v \in V_2\right\}$$

为 G 的割集, (V_1, V_2) 中的边称为割边, 不在 (V_1, V_2) 中的边称作非割边.

最大割集问题: 任给无向图 $G = \langle V, E\rangle$, 求 G 的边数最多的割集. 考虑下述求最大割集问题的**局部改进算法 (MCUT)**: 令 $V_1 = V, V_2 = \emptyset$. 如果存在顶点 u, 在 u 关联的边中非割边多于割边, 如果 $u \in V_1$, 则把 u 移到 V_2 中; 如果 $u \in V_2$, 则把 u 移到 V_1 中, 直到不存在这样的顶点为止, 取此时得到的 (V_1, V_2) 作为解.

证明: MCUT 是 **2-近似算法**, 即对任一实例 I, 都有

$$\mathrm{OPT}(I) \leq 2\mathbf{MCUT}(I)$$

Solution: 根据算法, 每一个顶点关联的割边数大于等于关联的非割边数. 对所有的顶点求和, 每条边出现 2 次, 故所有的割边数大于等于所有非割边数. 从而 $\mathbf{MCUT}(I) \geq \frac{|E|}{2}$. 又显然有 $\mathrm{OPT}(I) \leq |E|$, 于是证得 $\mathrm{OPT}(I) \leq 2\mathbf{MCUT}(I)$, □.[1]

[1] 最小割集问题是多项式时间可解的, 而最大割集问题是 $\mathcal{NPH}$ 的.

Problem 10

双机调度问题 (优化形式): 有 2 台相同的机器和 n 项作业 $J_1, J_2, \cdots, J_n$, 每一项作业可以在任一台机器上处理, 没有顺序的限制, 作业 J_i 的处理时间为正整数 $t_i, 1 \leq i \leq n$. 要求把 n 项作业分配给这 2 台机器使得完成时间最短, 即把 $\{1, 2, \cdots, n\}$ 划分为 I_1 和 I_2, 使得

$$\max\left\{\sum_{i\in I_1} t_i, \sum_{i\in I_2} t_i\right\}$$

最小.

令 $D = \left\lfloor \frac{1}{2}\sum_{i=1}^{n} t_i \right\rfloor$, $B(i) = \left\{ t \middle| t = \sum_{j\in S} t_j \leq D, S \subseteq \{1, 2, \cdots, i\} \right\}, 0 \leq i \leq n$. $B(i)$ 包括所有前 i 项作业中任意项 (可以是 0 项) 作业的处理时间之和, 只要这个和不超过所有作业处理时间之和的二分之一. 试给出关于 $B(i)$ 的递推公式, 并利用这个递推公式设计双机调度问题的伪多项式时间算法, 进而设计这个问题的完全多项式时间近似算法.

Solution: 递推公式如下所示:

$$B(0) = \{0\}, B(i) = B(i-1) \cup \left\{ t \middle| t - t_i \in B(i-1), t_i \leq t \leq D \right\}, i = 1, 2, \cdots, n$$

于是显然有

$$\mathrm{OPT}(I) = \sum_{i=1}^{n} t_i - \max B(n)$$

于是我们可以给出如下的DP算法:

Algorithm 1 DP 算法

Input: n 个作业的处理时间 $t_1, t_2, \cdots, t_n$
Output: 处理好的作业集 J

令 $D \leftarrow \left\lfloor \frac{1}{2}\sum_{i=1}^{n} t_i \right\rfloor$, $B(0) \leftarrow 0$;
for $i = 1; i \leq n; i$++ **do**
    令 $B(i) \leftarrow B(i-1)$;
    for $t = t_i; t \leq D; t$++ **do**
        if $t - t_i \in B(i-1)$ **then**
            $B(i) \leftarrow B(i) \cup \{t\}$;
        end if
    end for
end for
令 $t \leftarrow \max B(n), J \leftarrow \emptyset, i \leftarrow n$;
for $i = n; i \geq 1; i$-- **do**
    if $t - t_i \in B(i-1)$ **then**
        $J \leftarrow J \cup \{i\}, t \leftarrow t - t_i$;
    end if
    if $t \leq 0$ **then**
        输出 J, **break**;
    end if
end for
end {DP};

DP 的时间复杂度为 $O(nD) = O\left(n^2 t_{\max}\right)$, 其中 $t_{\max} = \max\left\{t_i \middle| i = 1, 2, \cdots, n\right\}$. 这是伪多项式时间算法. 进而设计出下述的完全多项式时间近似算法-**FPTAS**:

Algorithm 2 FPTAS 算法

Input: n 个作业的处理时间 $t_1, t_2, \cdots, t_n$ 和 $\epsilon > 0$
Output: 处理好的作业集 J

1: 令 $t_{\max} \leftarrow \max\{t_i | i = 1, 2, \cdots, n\}$;
2: 令 $b \leftarrow \max\left\{\left\lfloor t_{\max} \middle/ \left(1 + \frac{1}{\epsilon}\right) n \right\rfloor, 1\right\}$;
3: 令 $t_i' \leftarrow t_i / b, i = 1, 2, \cdots, n$;
4: 以 $t_i' (i = 1, 2, \cdots, n)$ 为输入并运行 DP 算法;
5: 输出处理好的作业集 J;
6: **end {FPTAS}**;

FPTAS 的时间复杂度为 $O\left(n^2 t'_{\max}\right) = O\left(n^2 t_{\max}/b\right) = O\left(n^3\left(1 + \frac{1}{\epsilon}\right)\right)$.

当 $b = 1$, **FPTAS** 得到最优解. 不妨设 $b > 1, b\left(t_i' - 1\right) < t_i \leq b t_i'$. 对任意的 $S \subseteq \{1, 2, \cdots, n\}$, 有

$$
\begin{aligned}
& b \sum_{i \in S} t_i' - b|S| < \sum_{i \in S} t_i \leq b \sum_{i \in S} t_i', \\
& 0 \leq b \sum_{i \in S} t_i' - \sum_{i \in S} t_i < b|S| \leq bn
\end{aligned} \tag{$*$}
$$

记最优解为 J^*, **FPTAS** 得到的近似解为 J 和 $J' = \{1, 2, \cdots, n\} \setminus J$, $\mathrm{OPT}(I) = \sum_{i \in J^*} t_i$, $\mathbf{FPTAS}(I) = \sum_{i \in J'} t_i$, 于是有

$$
\begin{aligned}
\mathbf{FTPAS}(I) - \mathrm{OPT}(I) &= \sum_{i \in J'} t_i - \sum_{i \in J^*} t_i \\
&= \left(\sum_{i \in J'} t_i - b \sum_{i \in J'} t_i'\right) + \left(b \sum_{i \in J'} t_i' - b \sum_{i \in J^*} t_i'\right) + \left(b \sum_{i \in J^*} t_i' - \sum_{i \in J^*} t_i\right)
\end{aligned}
$$

根据($*$)式可知上述第一项 ≤ 0. J' 是关于 $\left\{t_i'\right\}$ 的最优解, 第二项也 ≤ 0, 于是得到

$$
\mathbf{FTPAS}(I) - \mathrm{OPT}(I) \leq b \sum_{i \in J^*} t_i' - \sum_{i \in J^*} t_i < bn \leq t_{\max} \Big/ \left(1 + \frac{1}{\epsilon}\right) \leq \mathbf{FTPAS}(I) \Big/ \left(1 + \frac{1}{\epsilon}\right)
$$

化简得到 $\mathbf{FTPAS}(I) \leq (1 + \epsilon)\mathrm{OPT}(I)$, 因此 **FTPAS** 是双机调度问题的完全多项式时间近似算法,□.

Problem 11

顶点覆盖问题: 任给一个图 $G = \langle V, E\rangle$, 求 G 的顶点数最少的顶点覆盖. 复习顶点覆盖问题的近似算法及其证明.

Solution: MVC算法如下所示:

Algorithm 3 算法 **MVC**(G)

Input: 图 $G = \langle V, E\rangle$
Output: 最小顶点覆盖 V'

```
1: V' ← ∅, e_1 ← E;
2: while e_1 ≠ ∅ do
3:     从 e_1 中任选一条边 (u, v);
4:     V' ← V' ∪ {u, v};
5:     从 e_1 中删去与 u 和 v 相关联的所有边;
6: end while
7: return V';
8: end {MVC};
```

显然算法 MVC 的时间复杂度为 $O(m), m = |E|$. 记 $|V'| = 2k$, V' 中的顶点是 k 条边的端点, 这 k 条边互不关联. 为了覆盖这 k 条边则需要 k 个顶点, 从而 $\mathrm{OPT}(I) \geq k$. 于是有

$$\frac{\mathrm{MVC}\,(I)}{\mathrm{OPT}\,(I)} \leq \frac{2k}{k} = 2$$

故 MVC 是最小顶点覆盖问题的 2-近似算法,□.

Problem 12

判断正误:

- 旅行商问题存在多项式时间近似方案. (　)
- 0/1 背包问题存在多项式时间近似方案. (　)
- 0/1 背包问题的贪心算法 (单位价值高优先装入) 是绝对近似算法. (　)
- 多机调度问题的贪心近似算法 (按输入顺序将作业分配给当前最小负载机器) 是 ϵ-近似算法. (　)

Solution:

- 错误. 根据教材可知, 旅行商问题不存在多项式时间近似算法, 除非 $\mathcal{P} = \mathcal{NP}$. 如果存在的话, 那么就可以证得 $\mathcal{P} = \mathcal{NP}$, 即可以拿图灵奖了.
- 正确, **PTAS 算法**就是 0/1 背包问题的多项式时间近似方案.
- 错误, 0/1 背包问题的贪心算法**不是**绝对近似算法.
- 正确, 多机调度问题的贪心近似算法有 GMPS 和 DGMPS 分别是 2-近似和 3/2-近似算法.

Problem 13

设旅行商问题的解表示为 $D = F = \{S|S = (i_1, i_2, \cdots, i_n), i_1, i_2, \cdots, i_n$是$1, 2, \cdots, n$的一个排列$\}$, 邻域定义为 2-OPT(即 S 中的两个元素对换), 求 $S = (3, 1, 2, 4)$ 的邻域 $N(S)$.

Solution: 将 S 中的两个元素对换即可得到 $N(S)$:

$$N(S) = \{(1, 3, 2, 4), (2, 1, 3, 4), (4, 1, 2, 3), (3, 2, 1, 4), (3, 4, 2, 1), (3, 1, 4, 2)\}$$

Problem 14

0/1 背包问题的解记作 $X = (x_1, x_2, \cdots, x_n), x_i \in \{0, 1\}, i = 1, 2, \cdots, n$. 邻域定义为

$$N(X) = \left\{ Y \left| \sum_{i=1}^{n} |y_i - x_i| \leq 1 \right. \right\}, X = (1, 1, 0, 0, 1)$$

求邻域 $N(X)$.

Solution: 每次只允许一个分量变化即可求出邻域 $N(X)$:

$$N(X) = \{(0, 1, 0, 0, 1), (1, 0, 0, 0, 1), (1, 1, 1, 0, 1), (1, 1, 0, 1, 1), (1, 1, 0, 0, 0)\}$$

Problem 15

写出禁忌搜索算法的主要步骤.

Solution: 禁忌搜索算法的主要步骤如下算法4中所示:

Algorithm 4 禁忌搜索算法步骤

1: 选定一个初始可行解 $\boldsymbol{x}^{\text{cb}}$ 并初始化禁忌表 $H \leftarrow \{\}$;
2: **while** 不满足停止规则 **do**
3: 在 $\boldsymbol{x}^{\text{cb}}$ 的邻域中选出满足禁忌要求的候选集 Can-$N\left(\boldsymbol{x}^{\text{cb}}\right)$;
4: 从该候选集中选出一个评价值最佳的解 $\boldsymbol{x}^{\text{lb}}$;
5: 令 $\boldsymbol{x}^{\text{cb}} \leftarrow \boldsymbol{x}^{\text{lb}}$ 并更新记录 H;
6: **end while**

Problem 16

禁忌对象特赦可以基于影响力规则: 即特赦影响力大的禁忌对象. 影响力大什么含义? 举例说明该规则的好处.

Solution: 影响力大意味着有些对象变化对目标值影响很大. 如 0/1 背包问题, 当包中无法装入新物品时, 特赦体积大的分量来避开局部最优解.

Problem 17

判断正误:

- 禁忌搜索中, 禁忌某些对象是为了避免领域中的不可行解. ( )
- 禁忌长度越大越好. ( )
- 禁忌长度越小越好. ( )

Solution:

- 错误, 选取禁忌对象是为了引起解的变化, 根本目的在于避开邻域内的局部最优解而不是不可行解.
- 错误, 禁忌长度短了则可能陷入局部最优解.
- 错误, 禁忌长度长了则导致计算时间长.

Problem 18

写出模拟退火算法的主要步骤.

Solution: 模拟退火算法的主要步骤如下算法5中所示:

Algorithm 5 模拟退火算法步骤

任选初始解 x_0 并初始化 $x_i \leftarrow x_0, k \leftarrow 0, t_0 \leftarrow t_{\max}$(初始温度);
while $k \leq k_{\max}$ && $t_k \geq T_f$ **do**
    从邻域 $N(x_i)$ 中随机选择 x_j, 即 $x_j \leftarrow_R N(x_i)$;
    计算 $\Delta f_{ij} = f(x_j) - f(x_i)$;
    if $\Delta f_{ij} \leq 0 \parallel \exp\left(-\Delta f_{ij}/t_k\right) >$ RANDOM(0, 1) **then**
        $x_i \leftarrow x_j$;
    end if
    $t_{k+1} \leftarrow d(t_k)$;
    $k \leftarrow k + 1$;
end while

Problem 19

为避免陷入局部最优 (小), 模拟退火算法以概率 $\exp\left(-\Delta f_{ij}/t_k\right)$ 接受一个退步 (比当前最优解差) 的解, 以跳出局部最优. 试说明参数 $t_k, \Delta f_{ij}$ 对是否接受退步解的影响.

Solution: 很明显, 当 t_k 较大时, 接受退步解的概率越大; 当 Δf_{ij} 较大时, 接受退步解的概率越小.

Problem 20

下面属于模拟退火算法实现的关键技术问题的有 _____.

(A) .初始温度　　(B) .温度下降控制　　(C) .邻域定义　　(D) .目标函数

Solution: 模拟退火算法实现的关键技术问题有**邻域的定义 (构造)、起始温度的选择、温度下降方法、每一温度的迭代长度以及算法终止规则**. 因此选择 (A), (B), (C).

Problem 21

用遗传算法解某些问题, fitnees $= f(x)$ 可能导致适应函数难以区分这些染色体. 请给出一种解决办法.

Solution: 采用非线性加速适应函数如下

$$\text{fitness}\,(x) = \begin{cases} \dfrac{1}{f_{\max} - f\,(x)}, & f\,(x) < f_{\max} \\ M > 0, & f\,(x) = f_{\max} \end{cases}$$

其中 M 是一个充分大的值, $f_{\max}$ 是当前最优值.

Problem 22

用非常规编码染色体实现的遗传算法, 如 TSP 问题使用 $1, 2, \cdots, n$ 的排列编码, 简单交配会产生什么问题? 如何解决?

Solution: 后代可能会出现非可行解, 因此需要通过罚值和交叉新规则来解决.

Problem 23

下面属于遗传算法实现的关键技术问题的有 _____.

(A) .解的编码　　(B) .初始种群的选择　　(C) .邻域定义　　(D) .适应函数

Solution: 遗传算法实现的关键技术问题有**解的编码、适应函数、初始种群的选取、交叉规则以及终止规则**. 因此选择 (A), (B), (D).

至此, 9-10-11 章的所有练习都已做完.

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

中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

考试后作业

2022 年 12 月 19 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

设计一个 Las Vegas 随机算法, 求解电路板布线问题. 将该算法与分支限界算法结合, 观察求解效率.

Solution: 该算法的设计核心是: **采用随机放置位置策略并结合分支限界算法**. 算法的 C++ 代码如下所示:

```cpp
#include <bits/stdc++.h> //万能头文件, 刷题时可以用, 大型项目千万不能用
using namespace std;

//表示方格位置上的结构体
struct position{
    int row;
    int col;
};

//分支限界算法
bool FindPath(
    position start, position finish, int &PathLen,
    position *&path, int **grid, int m, int n
) { //找到最短布线路径, 若找得到则返回 true, 否则返回 false
    //起点终点重合则不用布线
    if((start.row == finish.row) && (start.col == finish.col)) {
        PathLen = 0;
        return true;
    }

    //设置方向移动坐标值: 东南西北
    position offset[4];
    offset[0].row = 0;
    offset[0].col = 1; //右移
    offset[1].row = 1;
    offset[1].col = 0; //下移
    offset[2].row = 0;
    offset[2].col = -1; //左移
    offset[3].row = -1;
    offset[3].col = 0; //上移

    int NumNeighBlo = 4; //相邻的方格数
    position here, nbr;
    here.row = start.row; //设置当前方格, 即搜索单位
    here.col = start.col;
    //由于 0 和 1 用于表示方格的开放和封闭, 故距离: 2-0 3-1
    grid[start.row][start.col] = 0; //-2 表示强, -1 表示可行, -3 表示不能当作路线
    //队列式搜索, 标记可达相邻方格
    queue<position> q_FindPath;
```

```
    do {
        int num = 0; //方格未标记个数
        position selectPosition[5]; //保存选择位置
        for(int i = 0; i < NumNeighBlo; i++) {
            //到达四个方向
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == -1) { //该方格未标记
                grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
                if((nbr.row == finish.row) && (nbr.col == finish.col)) {
                    break;
                }
                selectPosition[num].row = nbr.row;
                selectPosition[num].col = nbr.col;
                num++;
            }
        }
        if(num > 0) { //如果标记，则在这么多个未标记个数中随机选择一个位置（本算法核心）
            q_FindPath.push(selectPosition[rand()%(num)]); //随机选一个入队
        }
        if((nbr.row == finish.row) && (nbr.col == finish.col)) {
            break; //是否到达目标位置 finish
        }
        //判断活结点队列是否为空
        if(q_FindPath.empty() == true) return false; // 无解
        //访问队首元素出队
        here = q_FindPath.front();
        q_FindPath.pop();
    } while (true);

    //构造最短布线路径
    PathLen = grid[finish.row][finish.col];
    path = new position[PathLen]; //路径
    //从目标位置 finish 开始向起始位置回溯
    here = finish;
    for(int j = PathLen - 1; j >= 0; j--) {
        path[j] = here;
        //找前驱位置
        for(int i = 0; i <= NumNeighBlo; i++) {
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == j) {//距离 +2 正好是前驱位置
                break;
            }
        }
        here = nbr;
    }
    return true;
}
```

```
int main() {
    cout << "---------分支限界算法之布线问题---------" << endl;
    int path_len, path_len1, m, n;
    position *path, *path1, start, finish, start1, finish1;
    cout << " 在一个 m*n 的棋盘上，请分别输入 m 和 n，代表行数和列数，然后输入回车" << endl;
    cin >> m >> n;
    //创建棋盘格
    int **grid = new int * [m + 2], **grid1 = new int * [m + 2];
    for(int i = 0; i < m + 2; i++) {
        grid[i] = new int[n + 2];
        grid1[i] = new int[n + 2];
    }
    //初始化棋盘
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            grid[i][j] = -1;
        }
    }
    //设置方格阵列的围墙
    for(int i = 0; i <= n + 1; i++) {
        grid[0][i] = grid[m + 1][i] = -2; //上下的围墙
    }
    for(int i = 0; i <= m + 1; i++) {
        grid[i][0] = grid[i][n + 1] = -2; //左右的围墙
    }
    cout << " 初始化棋盘格和加围墙" << endl;
    cout << "------------------" << endl;
    for(int i = 0; i < m + 2; i++) {
        for(int j = 0; j < n + 2; j++) {
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }
    cout << "------------------" << endl;
    cout << " 请输入已经占据的位置（行坐标，列坐标），代表此位置不能布线" << endl;
    cout << " 例如输入 2 2(然后输入回车)，表示坐标（2,2）不能布线;" <<
    " 当输入坐标为 0 0(然后输入回车) 表示结束输入" << endl;
    //添加已经布线的棋盘格
    while(true) {
        int ci, cj;
        cin >> ci >> cj;
        if(ci > m || cj > n) {
            cout << " 输入非法!";
            cout << " 行坐标 <" << m << "，列坐标 <" << n << " 当输入的坐标为 0,0 时结束输入"
↪   << endl;
            continue;
        } else if (ci == 0 || cj == 0) {
            break;
        } else {
            grid[ci][cj] = -3;
        }
    }
```

```
//布线前的棋盘格
cout << " 布线前的棋盘格" << endl;
cout << "------------------" << endl;
for(int i = 0; i < m + 2; i++) {
    for(int j = 0; j < n + 2; j++) {
        cout << grid[i][j] << " ";
    }
    cout << endl;
}
cout << "------------------" << endl;
cout << " 请输入起点位置坐标" << endl;
cin >> start.row >> start.col;
cout << " 请输入终点位置坐标" << endl;
cin >> finish.row >> finish.col;
clock_t starttime, endtime;
starttime = clock(); //程序开始时间
srand((unsigned) time (NULL)); //初始化时间种子，是随机选择的关键
int time = 0; //为假设运行次数
start1 = start, finish1 = finish, path_len1 = path_len, path1 = NULL; //初始值拷贝
for(int i = 0; i < m + 2; i++) {
    for(int j = 0; j < n + 2; j++) {
        grid1[i][j] = grid[i][j];
    }
}
bool result = FindPath(start1, finish1, path_len1, path1, grid1, m, n);
while(result == 0 && time < 1000) { //尝试次数最多为 1000 次
    //初始值拷贝
    start1 = start, finish1 = finish, path_len1 = path_len, path1 = NULL;
    for(int i = 0; i < m + 2; i++) {
        for(int j = 0; j < n + 2; j++) {
            grid1[i][j] = grid[i][j];
        }
    }
    time++;
    cout << endl;
    cout << " 没有找到路线，第" << time << " 次尝试" << endl;
    result = FindPath(start1, finish1, path_len1, path1, grid1, m, n);
}
endtime = clock(); //程序结束时间

if(result == 1) {
    cout << "------------------" << endl;
    cout << "$ 代表围墙" << endl;
    cout << "# 代表已经占据的点" << endl;
    cout << "* 代表布线路线" << endl;
    cout << "= 代表还没有布线的点" << endl;
    cout << "------------------" << endl;
    for(int i = 0; i <= m + 1; i++) {
        for(int j = 0; j <= n + 1; j++) {
            if(grid1[i][j] == -2) cout << "$ ";
            else if(grid1[i][j] == -3) cout << "# ";
```

```
                else {
                    int r;
                    for(r = 0; r < path_len1; r++) {
                        if(i == path1[r].row && j == path1[r].col) {
                            cout << "* ";
                            break;
                        }
                        if(i == start1.row && j == start1.col) {
                            cout << "* ";
                            break;
                        }
                    }
                    if(r == path_len1) cout << "= ";
                }
            }
            cout << endl;
        }
        cout << "-------------------" << endl;
        cout << " 路径坐标和长度" << endl;
        cout << endl;
        cout << "(" << start1.row << "," << start1.col << ")" << " ";
        for(int i = 0; i < path_len1; i++) {
            cout << "(" << path1[i].row << "," << path1[i].col << ")" << " ";
        }
        cout << endl;
        cout << endl;
        cout << " 路径长度: " << path_len1 + 1 << endl;
        cout << endl;
        time++;
        cout << " 布线完毕, 共查找" << time << " 次" << endl;
        cout << " 算法运行时间为: " << (endtime - starttime) << "ms" << endl;
    } else {
        cout << endl;
        cout << " 经过多次尝试, 依然没有找到路线" << endl;
    }
    return 0;
}
```

上述代码的关键之处在于:

- P3 页的第 13 行代码, 这里是当前点相邻四个点是否可以放置, 如果可以放置用 selectPostion 保存下来, 并用 num 记录有多少个位置可以放置;
- P3 页的第 19 行代码, 这里利用上面保存的可以放置的点, 然后**随机选取其中一个加入队列**, 这就是 Las Vegas 算法的精髓;
- P5 页的第 17 行代码, 作用是初始化时间种子, 是伪随机生成器的关键, 即是随机选择的关键.

结果分析:

- 测试样例 1: 3×3 棋盘, 代码的交互输出过程如下:

```
开始运行...
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上，请分别输入 m 和 n，代表行数和列数，然后输入回车
3 3
初始化棋盘格和加围墙
------------------
-2 -2 -2 -2 -2
-2 -1 -1 -1 -2
-2 -1 -1 -1 -2
-2 -1 -1 -1 -2
-2 -2 -2 -2 -2
------------------
请输入已经占据的位置(行坐标，列坐标)，代表此位置不能布线
例如输入 2 2(然后输入回车)，表示坐标(2,2)不能布线;当输入坐标为 0 0(然后输入回车)表示结束输
↪  入
2 1
2 3
3 3
0 0
布线前的棋盘格
------------------
-2 -2 -2 -2 -2
-2 -1 -1 -1 -2
-2 -3 -1 -3 -2
-2 -1 -1 -3 -2
-2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
3 1

没有找到路线，第 1 次尝试

没有找到路线，第 2 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $
$ * * = $
$ # * # $
$ * * # $
$ $ $ $ $
------------------
路径坐标和长度
(1,1) (1,2) (2,2) (3,2) (3,1)
路径长度: 5
布线完毕，共查找 3 次
算法运行时间为: 39ms
运行结束.
```

- 测试样例 2: 5×5 棋盘, 代码的交互输出过程如下:

```
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上, 请分别输入 m 和 n, 代表行数和列数, 然后输入回车
5 5
请输入已经占据的位置(行坐标, 列坐标), 代表此位置不能布线
例如输入 2 2(然后输入回车), 表示坐标(2,2)不能布线;当输入坐标为 0 0(然后输入回车)表示结束输入
3 1
3 2
3 4
3 5
4 5
0 0
布线前的棋盘格
------------------
-2 -2 -2 -2 -2 -2 -2
-2 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -2
-2 -3 -3 -1 -3 -3 -2
-2 -1 -1 -1 -1 -3 -2
-2 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
5 2
没有找到路线, 第 1 次尝试
没有找到路线, 第 2 次尝试
没有找到路线, 第 3 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $ $ $
$ * = = = = $
$ * * * = = $
$ # # * # # $
$ = = * = # $
$ = * * = = $
$ $ $ $ $ $ $
------------------
路径坐标和长度
(1,1) (2,1) (2,2) (2,3) (3,3) (4,3) (5,3) (5,2)
路径长度: 8
布线完毕, 共查找 4 次
算法运行时间为: 61ms
```

- 测试样例 2: 10×10 棋盘, 代码的交互输出过程如下:

```
---------分支限界算法之布线问题---------
在一个 m*n 的棋盘上，请分别输入 m 和 n，代表行数和列数，然后输入回车
10 10
布线前的棋盘格
------------------
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -3 -3 -3 -3 -1 -1 -3 -3 -3 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
------------------
请输入起点位置坐标
1 1
请输入终点位置坐标
9 9

没有找到路线，第 1 次尝试
没有找到路线，第 2 次尝试
没有找到路线，第 3 次尝试
没有找到路线，第 4 次尝试
------------------
$ 代表围墙
# 代表已经占据的点
* 代表布线路线
= 代表还没有布线的点
------------------
$ $ $ $ $ $ $ $ $ $ $ $
$ * = = = = = = = = = $
$ * * * = = = = = = = $
$ = = * * = = = = = = $
$ = = = * * = = = = = $
$ # # # # * = # # # = $
$ = = = = * = = = = = $
$ = = = = * * * = = = $
$ = = = = = = * = = = $
$ = = = = = = * * * = $
$ = = = = = = = = = = $
$ $ $ $ $ $ $ $ $ $ $ $
------------------
路径坐标和长度
(1,1) (2,1) (2,2) (2,3) (3,3) (3,4) (4,4) (4,5) (5,5) (6,5) (7,5) (7,6) (7,7) (8,7)
↪  (9,7) (9,8) (9,9)
路径长度: 17
布线完毕，共查找 5 次
算法运行时间为: 73ms
```

由此可见, 结合随机化和分支限界的 Las Vegas 算法的求解效率还是相当不错的.

Problem 2

上机实现 0/1 背包问题的遗传算法, 分析算法的性能.
Solution:python 代码如下:

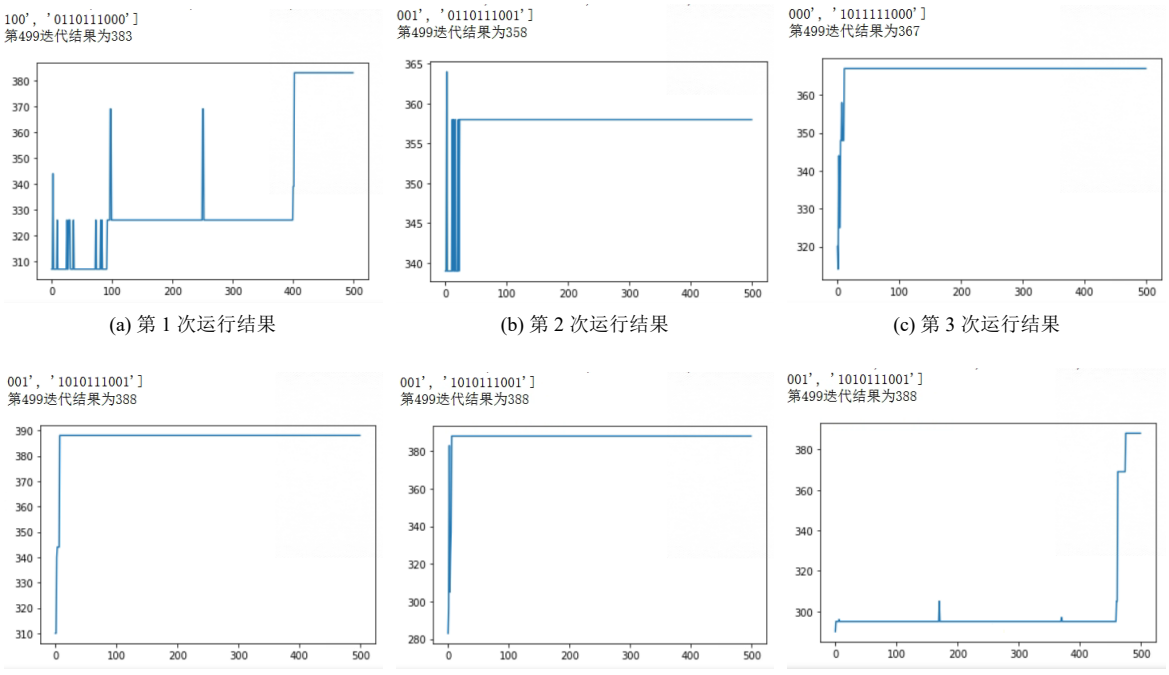
```python
import numpy as np
import matplotlib.pyplot as plt
# 初始化种群,popsize 代表种群个数, n 代表基因长度,
def init(popsize,n):
    population=[]
    for i in range(popsize):
        pop=''
        for j in range(n):
            pop=pop+str(np.random.randint(0,2))
        population.append(pop)
    return population

# 计算种群中每个个体此时所代表的解的重量和效益
def computeFitness(population,weight,profit):
    total_weight = []
    total_profit = []
    for pop in population:
        weight_temp = 0
        profit_temp = 0
        for index in range(len(pop)):
            if pop[index] == '1':
                weight_temp += int(weight[index])
                profit_temp += int(profit[index])
        total_weight.append(weight_temp)
        total_profit.append(profit_temp)
    return  total_weight,total_profit

def computesingle(single,profit):
    profit_temp = 0
    for index in range(len(single)):
        if single[index] == '1':
            profit_temp += int(profit[index])
    return profit_temp
# 筛选符合条件的
def select(population,weight_limit,total_weight,total_profit):
    w_temp = []
    p_temp = []
    pop_temp = []
    for weight in total_weight:
        out = total_weight.index(weight)
        if weight <= weight_limit:
            w_temp.append(total_weight[out])
            p_temp.append(total_profit[out])
            pop_temp.append(population[out])
    return pop_temp,w_temp,p_temp
```

```python
def roulettewheel(s_pop,total_profit):
    p =[0]
    temp = 0
    sum_profit = sum(total_profit)
    for i in range(len(total_profit)):
        unit = total_profit[i]/sum_profit
        p.append(temp+unit)
        temp += unit
    new_population = []
    i0 = 0
    while i0 < popsize:
        select_p = np.random.uniform()
        for i in range(len(s_pop)):
            if select_p > p[i] and select_p <= p[i+1]:
                new_population.append(s_pop[i])
        i0 += 1
    return new_population

def ga_cross(new_population,total_profit,pcross):# 随机交配
    new = []
    while len(new) < popsize:
        mother_index = np.random.randint(0, len(new_population))
        father_index = np.random.randint(0, len(new_population))
        threshold = np.random.randint(0, n)
        if (np.random.uniform() < pcross):
            temp11 = new_population[father_index][:threshold]
            temp12 = new_population[father_index][threshold:]
            temp21 = new_population[mother_index][threshold:]
            temp22 = new_population[mother_index][:threshold]
            child1 = temp11 + temp21
            child2 = temp12 + temp22
            pro1 = computesingle(child1, profit)
            pro2 = computesingle(child2, profit)
            if pro1 > total_profit[mother_index] and pro1 > total_profit[father_index]:
                new.append(child1)
            else:
                if total_profit[mother_index] > total_profit[father_index]:
                    new.append(new_population[mother_index])
                else:
                    new.append(new_population[father_index])
            if pro2 > total_profit[mother_index] and pro1 > total_profit[mother_index]:
                new.append(child2)
            else:
                if total_profit[mother_index] > total_profit[father_index]:
                    new.append(new_population[mother_index])
                else:
                    new.append(new_population[father_index])
    return new
```

```python
def mutation(new,pm):
    temp =[]
    for pop in new:
        p = np.random.uniform()
        if p < pm:
            point = np.random.randint(0, len(new[0]))
            pop = list(pop)
            if pop[point] == '0':
                pop[point] = '1'
            elif pop[point] == '1':
                pop[point] = '0'
            pop = ''.join(pop)
            temp.append(pop)
        else:
            temp.append(pop)
    return temp

def plot():
    x= range(iters)

    plt.plot(x,ylable)
    plt.show()

if __name__ == "__main__":
    weight = [95,75,23,73,50,22,6,57,89,98]
    profit = [89,59,19,43,100,72,44,16,7,64]
    n = len(profit)
    weight_limit = 300
    pm = 0.05
    pc = 0.8
    popsize = 30
    iters = 500
    population = init(popsize, n)
    ylable = []
    iter = 0
    best_pop = []
    best_p = []
    best_w = []
    while iter < iters:
        print(f'第{iter}代')
        print(" 初始为",population)
        w, p = computeFitness(population, weight, profit)
        print('weight:',w,'profit:',p)
        print(w)
        print(p)
        s_pop, s_w, s_p = select(population, weight_limit, w, p)

        best_index = s_p.index(max(s_p))
        ylable.append(max(s_p))
        best_pop.append(s_pop[best_index])
        best_p.append(s_p[best_index])
        best_w.append(s_w[best_index])
```

```
    print(s_pop[best_index])
    print(s_p[best_index])
    print(s_w[best_index])
    print(f'筛选后的种群{s_pop}, 长度{len(s_pop)}, 筛选后的 weight{s_w}, 筛选后的
    ↪  profit{s_p}')
    new_pop = roulettewheel(s_pop, s_p)
    w,p1 = computeFitness(new_pop, weight, profit)
    print(f'轮盘赌选择后{new_pop},{len(new_pop)}')
    new_pop1 = ga_cross(new_pop, p1, pc)
    print(f'交叉后{len(new_pop1)}')
    population = mutation(new_pop1, pm)
    print(population)
    print(f'第{iter}迭代结果为{max(s_p)}')
    iter += 1
best_i = best_p.index(max(best_p))
plot()
```

上述算法的多次运行结果为:



(a) 第 1 次运行结果 (b) 第 2 次运行结果 (c) 第 3 次运行结果

(d) 第 4 次运行结果 (e) 第 5 次运行结果 (f) 第 6 次运行结果

图 1: 各次的运行结果

主函数中的实例的最优解可以用分支限界算法计算出解向量为 $X = (1, 0, 1, 0, 1, 1, 1, 0, 0, 1)$, 最优值为 388. 上述遗传算法在后 3 次的运行结果是同正确解相吻合的, 并且上述遗传算法相较于分支限界算法是很快的, 60ms 就可以得出结果, 只不过需要多次运行并观察以得出最优解.

Problem 3

上机实现 TSP 的模拟退火算法, 随机生成一定规模的数据或用通用数据集比较其它人的结果, 分析算法的性能, 摸索实现中技术问题的解决.

Solution: 算法对应的 Matlab 代码如下所示:

```
function D = Distance(citys)
%% 计算两两城市之间的距离
% 输入 citys  各城市的位置坐标
% 输出 D  两两城市之间的距离
n = size(citys,1);
D = zeros(n,n);
for i = 1:n
   for j = i+1:n
       D(i,j) = sqrt(sum((citys(i,:) - citys(j,:)).^2));
       D(j,i) = D(i,j);
   end
end
```

```
function S2 = NewAnswer(S1)
%% 输入
% S1: 当前解
%% 输出
% S2: 新解
N = length(S1);
S2 = S1;
a = round(rand(1,2)*(N-1)+1); % 产生两个随机位置 用来交换
W = S2(a(1));
S2(a(1)) = S2(a(2));
S2(a(2)) = W;          % 得到一个新路线
```

```
function DrawPath(Route,citys)
%% 画路径函数
% 输入
% Route  待画路径
% citys  各城市坐标位置

figure
plot([citys(Route,1);citys(Route(1),1)],...
     [citys(Route,2);citys(Route(1),2)],'o-');
grid on

for i = 1:size(citys,1)
    text(citys(i,1),citys(i,2),['    ' num2str(i)]);
end

text(citys(Route(1),1),citys(Route(1),2),'       起点');
text(citys(Route(end),1),citys(Route(end),2),'       终点');
```

```
function [S,R] = Metropolis(S1,S2,D,T)
%% 输入
% S1:    当前解
% S2:    新解
% D:     距离矩阵（两两城市的之间的距离）
% T:     当前温度
%% 输出
% S:     下一个当前解
% R:     下一个当前解的路线距离
R1 = PathLength(D,S1);  % 计算路线长度
N = length(S1);         % 得到城市的个数
R2 = PathLength(D,S2);  % 计算路线长度
dC = R2 - R1;   % 计算能力之差
if dC < 0       % 如果能力降低 接受新路线
    S = S2;
    R = R2;
elseif exp(-dC/T) >= rand   % 以 exp(-dC/T) 概率接受新路线
    S = S2;
    R = R2;
else        % 不接受新路线
    S = S1;
    R = R1;
end
```

```
function p = OutputPath(R)
%% 输出路径函数
% 输入：R 路径
R = [R,R(1)];
N = length(R);
p = num2str(R(1));
for i = 2:N
    p = [p,'$\to $',num2str(R(i))];
end
disp(p)
```

```
function Length = PathLength(D,Route)
%% 计算各个体的路径长度
% 输入：
% D      两两城市之间的距离
% Route 个体的轨迹

Length = 0;
n = size(Route,2);
for i = 1:(n - 1)
    Length = Length + D(Route(i),Route(i + 1));
end
Length = Length + D(Route(n),Route(1));
```

主函数编码如下:

```m
%% I. 清空环境变量
clear all
clc

%% II. 导入城市位置数据
X = [16.4700   96.1000
     16.4700   94.4400
     20.0900   92.5400
     22.3900   93.3700
     25.2300   97.2400
     22.0000   96.0500
     20.4700   97.0200
     17.2000   96.2900
     16.3000   97.3800
     14.0500   98.1200
     16.5300   97.3800
     21.5200   95.5900
     19.4100   97.1300
     20.0900   92.5500];

%% III. 计算距离矩阵
D = Distance(X);  % 计算距离矩阵
N = size(D,1);    % 城市的个数

%% IV. 初始化参数
T0 = 1e10;   % 初始温度
Tend = 1e-30;  % 终止温度
L = 2;    % 各温度下的迭代次数
q = 0.9;    % 降温速率
syms x;
Time = ceil(double(solve(T0*(0.9)^x == Tend)));    % 计算迭代的次数
% Time = 132;
count = 0;        % 迭代计数
Obj = zeros(Time,1);          % 目标值矩阵初始化
track = zeros(Time,N);        % 每代的最优路线矩阵初始化

%% V. 随机产生一个初始路线
S1 = randperm(N);
DrawPath(S1,X)
disp('初始种群中的一个随机值:')
OutputPath(S1);
Rlength = PathLength(D,S1);
disp(['总距离: ',num2str(Rlength)]);

%% VI. 迭代优化
while T0 > Tend
    count = count + 1;     % 更新迭代次数
    temp = zeros(L,N+1);
    %%
    for k = 1:L
        % 1. 产生新解
        S2 = NewAnswer(S1);

```

```
        % 2. Metropolis 法则判断是否接受新解
        [S1 R] = Metropolis(S1, S2, D, T0);   % Metropolis 抽样算法
        temp(k, :) = [S1 R];              % 记录下一路线及其长度
    end
    %% 3. 记录每次迭代过程的最优路线
    [d0, index] = min(temp(:, end));     % 找出当前温度下最优路线
    if count == 1 || d0 <= Obj(count-1)
        Obj(count) = d0;                      % 如果当前温度下最优路程小于上一路程则记录当前路程
    else
        Obj(count) = Obj(count-1);     % 如果当前温度下最优路程大于上一路程则记录上一路程
    end
    track(count, :) = temp(index, 1:end-1);  % 记录当前温度的最优路线
    % 降温
    T0 = q * T0;
end

%% VII. 优化过程迭代图
figure
plot(1:count,Obj)
xlabel('迭代次数')
ylabel('距离')
title('优化过程')

%% VIII. 绘制最优路径图
DrawPath(track(end,:),X)

%% IX. 输出最优解的路线和总距离
disp('最优解:')
S = track(end,:);
p = OutputPath(S);
disp(['总距离: ',num2str(PathLength(D,S))]);
```

优化前的一个随机路线轨迹图如图2所示:

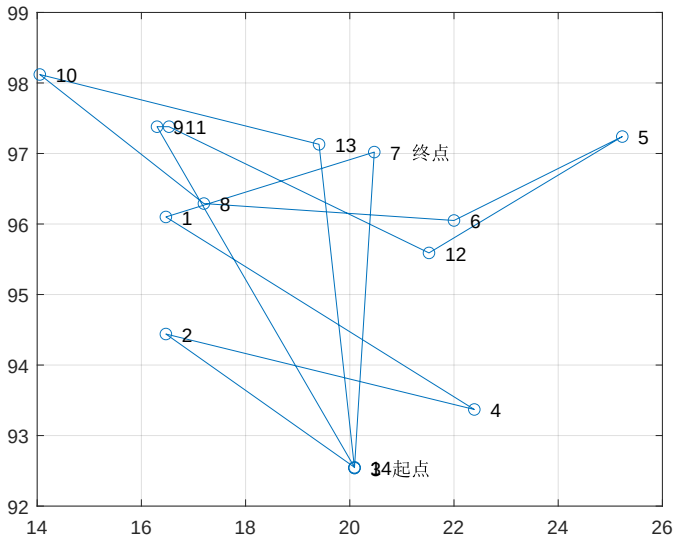


图 2: 随机路线图

初始种群中的一个随机值: 5→12→6→11→7→13→10→4→8→9→3→2→14→1→5,
总距离为 66.0171. 优化后的路线如图3所示:

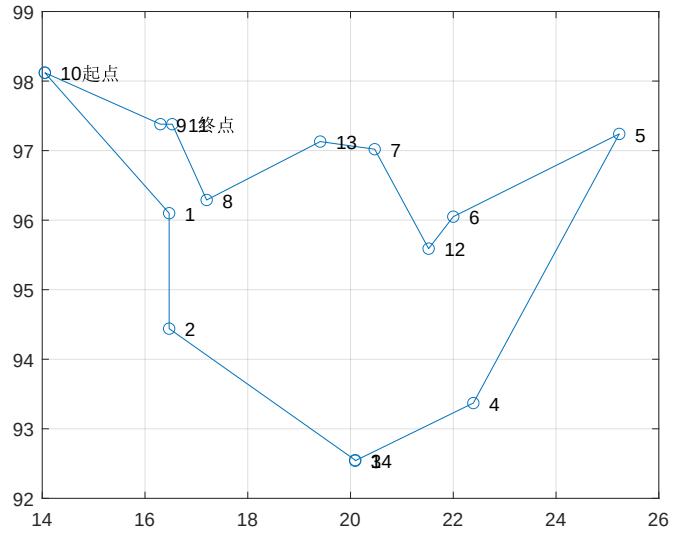


图 3: 最优解路线图

最优解: 6→12→7→13→8→11→9→10→1→2→14→3→4→5→6, 总距离为 29.3405. 优化迭代过程如下图4所示:

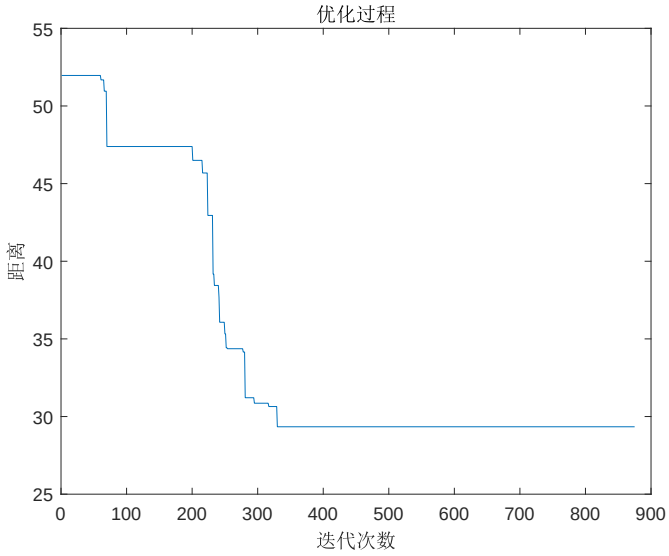


图 4: 模拟退火算法优化过程图

由上图可以看出: 优化前后路径长度得到很大改进, 由优化前的 66.0171 变为 29.3405, 变为原来的 44.4%, 400 多代以后路径长度已经保持不变了, 可以认为已经是最优解了.