

《算法设计》课程报告

课题名称: 解题报告

课题负责人名（学号）：宋明清 2018141461347

同组成员名单（角色）：宋明清：撰写人

指导教师_____

评阅成绩

评阅意见

提交报告时间: 2020 年 6 月 23 日

解题报告

计算机科学与技术 专业

学生 宋明清 指导老师 左劼

[摘要] 学习算法设计课程后，对各类算法有了更深入的了解，这里用五类算法解决了相关问题，写为报告。

关键词: 分治 动态规划 贪心 回溯 分支限界

分治-快速傅立叶变换

问题描述

[题目链接](#)

用尽量短的时间

- (1) 求 $g(x) = f_1(x)f_2(x)$,
- (2) 求 $a \times b$

分析

阅读书籍¹后，自己完全理解的基础上写出这篇报告。

两个n次多项式


- 相加的最直接的方法所需时间为 $\Theta(n)$ (Θ 与 $f(n)$ 的增长速率相同)

- 相乘的最直接的方法所需时间为 $\Theta(n^2)$
- 多项式的乘积也就是两系数向量的卷积

$$\begin{aligned} \text{设 } A(x) &= \sum_{j=0}^{n-1} a_j x^j \\ \text{和 } B(x) &= \sum_{j=0}^{n-1} b_j x^j \\ \text{有 } C(x) &= \sum_{j=0}^{2n-2} c_j x^j, \text{ 其中, } c_j = \sum_{k=0}^j a_k b_{j-k} \end{aligned}$$

两个数

- 把每一位看作是对应多项式的系数，即转化为问题(1)，最后进位。

对于一个多项式，可以有两种表达方式。正如对于一个信号，可以在时域上表示它，也可以在频域上表示它。一首钢琴曲，我们享受它时，是在感受它在时域上的变化，但是我们要演奏它时，却是要通过上的音符来演奏，也就是从频域上来观察它，这样更简洁。

多项式-系数表达

对于一个 $\text{degree}(A) = n$ 的多项式 $\sum_{j=0}^{n-1} a_j x^j$ 来说，我们只用得到它的系数的向量即可以表示它， $a = (a_n, a_1, \dots, a_{n-1})$

- 优点：可以快速求出在给定点 x_0 处的 $A(x_0)$
 - 若直接求的话，要计算 $\frac{n^2}{2}$ 次乘法运算和 n 次加法运算，即使用上快速幂，也是 $O(n \log n)$ 的复杂度^{[t1](#)}
 - 有一种线性(即 $O(n)$)求出系数表达多项式值的算法，叫做霍纳法则，也叫秦九韶算法^{[2](#)}
- 缺点：对于两多项式的乘积，即 $c = a \otimes b$ ，时间复杂度达到 n^2

多项式-点值表达

对于一个 $\text{degree} = n$ 的多项式来说，知道曲线上的 n 个点，就可以确定这一条曲线，例如一次曲线用两个点可以确定，二次曲线用三个点可以确定。 $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ 这 n 个点就可以确定一条曲线^{[p1](#)}，即一个多项式。但是这种表示方法并不是唯一的，在曲线上任取 n 个不同的点对都可以表示这条曲线。这 n 个点对的集合就是多项式的点值表达方法。

- 优点：对于 $C(x) = A(x) \times B(x)$ 来说，只用对A、B在相同的 n 处取值，那么就可以求出C的点值表达 $\{(x_i, y_i) | y_i = y_{i_a} \times y_{i_b}, 0 \leq i < n\}$ ，这样可以 $O(n)$ 地求出 $C(x)$
- 缺点：使用点值表达计算多项式在 x_0 处的值的代价太高，为 $\Theta(n^2)$ ^{[p2](#)}

- 从一个多项式的点值表达确定系数表达，即求值计算的逆被称为插值。

我们可以看到，系数表达和点值表达各有优劣，且可以相互转化。

系数表达求值方便，点值表达求多项式方便。我们求 $C(x) = A(x) \times B(x)$ 时，为了利用点值表达的 $O(n)$ 的优点，我们可以先把系数表达转化为点值表达，计算出 $C(x)$ ，再把点值表达转化为系数表达。正如我们可以对一个信号傅立叶变换到频域上，分析出它有哪些频率和振幅，消除某些成分，再变回去。但是两个表达的转换代价都为 $O(n^2)$ ，我们何必多此一举呢？

快速傅立叶变换

既然这个过程和傅立叶变换如此相像，那么我们考察离散傅立叶变换(Discret Fourier Transform,DFT)的形式和我们求多项式值的形式。快速傅立叶变换和傅立叶变换无关，与离散傅立叶变换有关。应该叫FDFT。记住我们下面是在求多项式在n个点处的取值！！

$$DFT : \sum_{j=0}^{n-1} f(j)e^{i\omega j}, \omega = \frac{\pi}{N}$$

$$Pol : \sum_{j=0}^{n-1} a_j x^j$$

我们可以看见这个形式是非常相似的，由于x可以取任意n个不同的值，那么我们就可以巧妙地选取一些具有特殊性质的值以期望减少我们的计算量。

- x可以是复数吗？当然是可以的，复数只是扩充了数域，实数域上的性质在复数域上得到了很好的满足，使得我们更好地进行旋转和缠绕等操作，这甚至比矩阵更有优势。比如逆时针旋转90度,复数形式明显更简洁。

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, e^{-i\theta}$$

- 既然复数的优势是旋转，那么就要选择那些旋转之后能回到原点的复数，这样的周期性经常会带来意想不到的惊喜，这些复数的模长肯定为1。记 $\omega_n = e^{\frac{2\pi}{n}i}$ 为单位根，显然 $\omega_n^n = 1$ ，对此我们有个引理^[2]。
- 我们把复数代入多项式后有 $y_i = A(\omega_n^i) = \sum_{j=0}^{n-1} a_j \omega_n^{ij}$ ，记为 $y = DFT_n(a)$
- 由折半引理，我们可以看到 ω 的重数变为了一半，因此这个问题或许可以用分治来解决。

分治

a_{2k} 比 a_{2k+1} 的次数少一，因此奇数项提出一个 x 后，两者的次数都是 $n/2-1$ 。

$$a = [a_0, a_1, a_2, a_3, a_4, \dots, a_{n-1}]$$

$$a^{[0]} = [a_0, a_2, a_4, \dots, a_{n-2}],$$

$$a^{[1]} = [a_1, a_3, a_5, \dots, a_{n-1}],$$

我们可以看到对于奇偶次项，由于只是原来的一半，所以要次数要翻倍后，才是对应的次数，因此

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2),$$

$$A(\omega_n^k) = A^{[0]}((\omega_n^k)^2) + \omega_n^k A^{[1]}((\omega_n^k)^2), \quad (1)$$

$$A(\omega_n^{k+\frac{n}{2}}) = A^{[0]}((\omega_n^{k+\frac{n}{2}})^2) + \omega_n^{k+\frac{n}{2}} A^{[1]}((\omega_n^{k+\frac{n}{2}})^2) \quad (2)$$

为什么会有(2)式呢？从下面我们可以看到 $A^{[0]}$ 和 $A^{[1]}$ 的规模只是 $n/2$ ；因此 k 只能取到 $n/2-1$ ，所以还要求剩下的一半值。

由消去引理和折半引理可以得到(1)与(2)的化简

$$A(\omega_n^k) = A^{[0]}(\omega_{\frac{n}{2}}^k) + \omega_n^k A^{[1]}(\omega_{\frac{n}{2}}^k), \quad (1)$$

$$A(\omega_n^{k+\frac{n}{2}}) = A^{[0]}(\omega_{\frac{n}{2}}^k) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_{\frac{n}{2}}^k) \quad (2)$$

$\omega_n^{\frac{n}{2}}$ 是旋转180度，因此(2)式变为

$$A(\omega_n^{k+\frac{n}{2}}) = A^{[0]}(\omega_{\frac{n}{2}}^k) - \omega_n^k A^{[1]}(\omega_{\frac{n}{2}}^k) \quad (3)$$

于是，求 $A(x)$ 在 $w_n^0, w_n^1, w_n^2 \dots w_n^{n-1}$ 处的值，就被转化为了两个规模为 $\frac{n}{2}$ 的子问题：

求 $A^{[0]}$ 和 $A^{[1]}$ 在 $\omega_{\frac{n}{2}}^0, \omega_{\frac{n}{2}}^1, \omega_{\frac{n}{2}}^2 \dots \omega_{\frac{n}{2}}^{\frac{n}{2}-1}$ 的取值，时间复杂度为 $O(n \lg n)$ ¹²

这里 ω_n^k 被称为旋转因子

快速傅立叶逆变换

这里是要把我们得到的点值表达转回系数表达。

我们把DFT写成矩阵乘积 $y = V_n a$, $(V_n)_{ij} = \omega_n^{ij}$, (i, j从0开始)

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

那么 $a = V_n^{-1} y$, $(V_n^{-1})_{ij} = \frac{\omega_n^{-ij}}{n}$, 这可以由 $V V_n^{-1}$ 验证得到，简略地说，只有对应行列相乘才能避免求和引理的出现。

现在就有了 $a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$ ，这与快速傅立叶变换的式子十分相像，因此逆变换的时间也是 $\Theta(n \lg n)$ 的。另外注意这里的 $\frac{1}{n}$ 是不像旋转因子一样递归的，只在最后出现。

最终我们就得到了解决问题的两个工具DFT与IDFT，问题的解决办法：

$a \otimes b = IDFT_{2n}(DFT_{2n}(a) \circ DFT_{2n}(b))$ 这里 \circ 为笛卡尔积，即对应项相乘。

证明

插值多项式的唯一性

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

该矩阵为范德蒙德矩阵，行列式为 $\prod_{0 \leq j < k \leq n-1} (x_k - x_j)$ ，此矩阵可逆，因此有唯一 a^T 。

拉格朗日公式

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

三个引理

- 消去引理 $\omega_{dn}^{dk} = \omega_n^k$
 $\omega_{dn}^{dk} = (e^{\frac{2\pi i}{dn}})^{dk} = (e^{\frac{2\pi i}{n}})^k = \omega_n^k$
- 折半引理 $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2 = \omega_{\frac{n}{2}}^k$
 $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2 \times \omega_n^n = (\omega_n^k)^2$
- 求和引理 $\sum_{j=0}^{n-1} (\omega_n^k)^j$ ，IFFT中用到
等比数列求和，分母为0

代码

FFT实现

```
1  #include <bits/stdc++.h>
2
3  const double PI = acos(-1);
4  const int maxn = 3e6+10;
5
6  inline int read(){
7      int x = 0, f = 1;
8      char ch = getchar();
```

```

9     while (ch<'0' || ch>'9')
10    {
11        if(ch=='-') f= -1;
12        ch = getchar();
13    }
14    while (ch>='0' && ch<='9')
15    {
16        x = (x<<1)+(x<<3)+(ch^48);
17        ch = getchar();
18    }
19    return x*f;
20 }
21
22 struct Complex{
23     double r,i;
24     Complex(double _r=0, double _i=0): r(_r), i(_i){}
25 };
26 Complex operator+(Complex a, Complex b){
27     return Complex(a.r+b.r, a.i+b.i);
28 }
29 Complex operator-(Complex a, Complex b){
30     return Complex(a.r-b.r,a.i-b.i);
31 }
32 Complex operator*(Complex a, Complex b){
33     return Complex(a.r*b.r - a.i*b.i , a.r*b.i + b.r*a.i);
34 }
35
36
37 void FFT(Complex *a, int powLim,int op){//FFT == fast-fast-tle
38     if(powLim==1) return;
39     Complex a0[powLim>>1],a1[powLim>>1];
40     for(int i=0; i<powLim;i+=2){
41         a0[i>>1] = a[i];
42         a1[i>>1] = a[i+1];
43     }
44     FFT(a0,powLim>>1,op);
45     FFT(a1,powLim>>1,op);
46     Complex wn = Complex(cos(2*PI/powLim),op*sin(2*PI/powLim));
47     Complex w = Complex(1,0);
48     for(int k=0;k<(powLim>>1);k++){//次数界个点的值, 暂存在a中
49         Complex t = w*a1[k];//蝴蝶操作
50         a[k] = a0[k] + t;
51         a[k+(powLim>>1)] = a0[k] - t;
52         w = w*wn;
53     }
54 }
55 Complex A[maxn],B[maxn];
56 int main(){
57     int n,m;

```

```

58     n=read();m=read();
59     for(int i=0;i<=n;i++) A[i].r = read();
60     for(int i=0;i<=m;i++) B[i].r = read();
61     int limit = 1;
62     while(limit<=n+m) {
63         limit<<=1;
64     }
65     FFT(A,limit,1);
66     FFT(B,limit,1);
67     for(int i=0;i<=limit;i++) A[i] = A[i]*B[i];
68     FFT(A,limit,-1);
69     for(int i=0;i<=n+m;i++) printf("%d ",(int)(A[i].r/limit+0.5));
70     return 0;
71 }

```

编程语言
C++

代码长度
1.67KB

用时
3.50s

内存
128.86MB

高效FFT实现

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const double PI = acos(-1);
6  const int maxn = (1 << 20) + 10 << 1;
7
8  int rev[maxn], len,lim = 1;
9
10 inline int read(){
11     int x = 0,f = 1;
12     char ch = getchar();
13     while (ch<'0' || ch>'9')
14     {
15         if(ch=='-') f= -1;
16         ch = getchar();
17     }
18     while (ch>='0'&&ch<='9')
19     {
20         x = (x<<1)+(x<<3)+(ch^48);
21         ch = getchar();
22     }
23     return x*f;
24 }
25
26 struct Complex{
27     double r,i;
28     Complex(double _r=0, double _i=0): r(_r), i(_i){}

```



```

29 };
30 Complex operator+(Complex a, Complex b){
31     return Complex(a.r+b.r, a.i+b.i);
32 }
33 Complex operator-(Complex a, Complex b){
34     return Complex(a.r-b.r,a.i-b.i);
35 }
36 Complex operator*(Complex a, Complex b){
37     return Complex(a.r*b.r - a.i*b.i , a.r*b.i + b.r*a.i);
38 }
39
40
41 void FFT(Complex *a, int opt){
42     for(int i=0;i<lim;i++) if(i<rev[i]) swap(a[i],a[rev[i]]);
43     for(int dep =1;dep<=log2(lim);dep++){ //合并到第dep层
44         int m = 1<<dep;
45         Complex wn = Complex(cos(2*PI/m),opt*sin(2*PI/m));
46         for(int k=0;k<lim;k+=m){ //到次数界, 每组求出2^dep个点值
47             Complex w = Complex(1,0);
48             for(int j=0;j<m/2;j++){//当前次数界个值的的合并,合并DFT
49                 Complex t = w*a[k+j+m/2];
50                 Complex u = a[k+j];
51                 a[k+j] = u+t;
52                 a[k+j+m/2] = u-t;
53                 w = w*wn;
54             }
55         }
56     }
57     if(opt== -1) for(int i=0;i<lim;i++) a[i].r/=lim;
58 }
59
60
61 Complex A[maxn],B[maxn];
62 int main(){
63     int n,m;
64     n=read();m=read();
65     for(int i=0;i<=n;i++) A[i].r=read();
66     for(int i=0;i<=m;i++) B[i].r=read();
67     while(lim<=n+m) lim<<=1,len++;
68     for(int i=0;i<lim;i++) rev[i] = (rev[i>>1]>>1) | ((i&1)<<(len-1));
69
70     FFT(A,1);
71     FFT(B,1);
72     for(int i=0;i<=lim;i++) A[i] = A[i]*B[i];
73     FFT(A,-1);
74     for(int i=0;i<=n+m;i++) printf("%d ",(int)(A[i].r+0.5));
75     return 0;
76 }

```

编程语言	代码长度	用时	内存
C++	1.74KB	2.42s	72.82MB

这时间快了有1s!

解释

对于FFT的迭代实现有如下解释：

我们每次会把奇数位和偶数位的系数提出来，然后递归。如果我们知道递归时，系数的最终排列顺序，那么直接就可以合并。深入调用时产生的输入向量树，我们可以发现规律。

$$\begin{array}{l}
 3 \ w_8 \qquad (a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7) \\
 2 \ w_4 \qquad (a_0 \ a_2 \ a_4 \ a_6) \qquad (a_1 \ a_3 \ a_5 \ a_7) \\
 1 \ w_2 \qquad (a_0 \ a_4) \quad (a_2 \ a_6) \quad (a_1 \ a_5) \quad (a_3 \ a_7) \\
 0 \qquad (a_0) \quad (a_4) \quad (a_2) \quad (a_6) \quad (a_1) \quad (a_5) \quad (a_3) \quad (a_7)
 \end{array}$$

因为这是一个自底向上的过程，所以每一层每一组的元素个数就是 2^{dep} ，对于元素a的排列，有人发现了规律，这里称之为位逆序置换。考察最后一行的a0-a7的位置的二进制数

000 001 010 011 100 101 110 111

000 100 010 110 001 101 011 111

因为每个数的位置都是原来位置的逆序，因此被称为位逆序置换。朴素模拟复杂度为 $n \log n$ ，因为有n个数，每个数位为 $\log n$ 。但是我们有复杂度为 $O(n)$ 的算法：

我们可以发现几个规律

- 偶数位上的二进制数首位为0，奇为1。
- 第2n和2n+1的数的二进制数除了首位之外都相同。
- 第2n和第2n+2的数的二进制数除了首位外是上一级子问题的解。
- 前一半的数的二进制数除了末尾外是上一级子问题的解。

00 01 10 11

00 10 01 11

其实这个为位逆序置换规律也可以由0、1这两个数推出来的，观察下表可以知道除了首位外，都可以看成1、2、4、8这样的组，每一组对应的数有：有后一组的数的后lim-1位=前一组的数的前lim-1位。

```
00000
01000
00100
01100
00010
01010
00110
01110
00001
01001
00101
01101
00011
01011
00111
01111
```

```
1  for(int i=0;i<lim;i++)//这里次数界较高，把高次系数看作零也可以
2      rev[i] = (rev[i>>1]>>1) | ((i&1)<<(len-1));    //A[rev[k]] = ak rev[k]是第k个系数
           在A数组中的下标/位置
3
4  (rev[i>>1]>>1) //这里是该数在上一级子问题对应集合中的位置，即上一级子问题的解，规律4
5  ((i&1)<<(len-1)) //这里是该数所属的上一级子问题的集合，即性质1
6
7  for(int i=0;i<lim;i++) if(i<rev[i]) swap(a[i],a[rev[i]]); //进行排序，如果该数在自
           己的位置之前，就交换
```

复杂度分析

$0, 1, \dots, n-1$ 求和为 $\frac{n^2}{2}$ ，快速幂可以用 $\log n$ 求出每一项，总的是 $n \log n$

$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ ，有 $\log n$ 层，每层都是 n 的代价，总的就是 $\Theta(n \log n)$

动态规划-电脑

问题描述

[题目链接](#)

给定一棵树，树边有制定的长度，求其他所有点到 i 点的距离的最大值， i : for i in tree。

tree的大小: $1e4$ 。

分析

其他所有点到*i*点的距离的最大值，也是*i*点到其他所有点距离的最大值。

一个很朴素的想法是对每个点dfs一下，统计出*i*能到达的最大距离，但是*n*个点都要访问*n*次，时间复杂度到达 $O(n^2)$ ，肯定会超时。用bfs也是同理。

由于我们在求出一个点到其他点的最大距离的时候，可能对另外的点的求解有帮助，所以尝试使用动态规划的方法。

由于树是有“方向”的，为了讨论的有序性，把树看作是有根的。假设节点 `fa[i]` 到不经过 *i* 的节点和 *i* 到 *i* 所有子树的节点的最大距离已经知道，那么很显然 $maxDis[i] = \max(upMax[i], downMax[i])$ 。在树上的转移方向有向上和向下，因此要用两个dfs。

首先第一个dfs求出节点 *i* 在其子树中能到达的最大距离，这很好求，直接累加就行了。

第二个dfs求出节点 *i* 往上走能到达的最大距离，到父亲后有可能一直向上走，也可能向上走几步之后向下走反而经过了节点 *i* (因为事先并不知道不经过 *i*)。那么如何求这个向上走能到达的最大距离呢？

设 *i* 的父亲为 *j*，`upMax[i]` 可以由 `downMax[j]` 和 `upMax[j]` 转移而来。

$upMax[i] = dis[i][j] + \max(upMax[j], downMax[j])$ 。

但是呢，如果 `downMax[j]` 经过了节点 *i*，那么以上式子就不成立。显然就要走第二大的边。因此需要一个 `maxSon[j]` 来标示 $fa[i] = j$ 的子树最大值要经过哪个儿子。

考虑以上两点，有状态转移方程：

$$upMax[i] = \begin{cases} dis[i][j] + \max(upMax[j], downMax[j]) & , maxSon[j] \neq i \\ dis[i][j] + \max(upMax[j], downSec[j]) & , maxSon[j] = i \end{cases}$$

证明

最优子结构性质

$maxDis[i] = \max(upMax[i], downMax[i])$,

反证法：

若 $downMax[i] \neq \max\{Dis[i][j] | j \in sonTree(i)\}$ ，那么存在 $Dis[i][k] > downMax[i]$ ，使得 $maxDis[i]' = Dis[i][k] > maxDis[i] = downMax[i]$ 。与 $maxDis[i]$ 为最远距离矛盾，因此最优子结构性质得证。

对于其他情况同理。

代码

```
1  #include <iostream>
2  #include <algorithm>
3  #include <cstring>
4  #include <cstdio>
5
6  using namespace std;
7  const int maxn = 1e4+10;
```

```

8  struct edge{
9      int to;
10     int next;
11     int w;
12     edge(int t=-1,int n=-1,int _w=0){
13         to = t;
14         next = n;
15         w = _w;
16     }
17 }e[maxn<<1];
18 int cnt,head[maxn];
19
20 void addEdge(int u,int v,int w){
21     e[cnt] = edge(v,head[u],w);
22     head[u] = cnt++;
23 }
24 int maxSon[maxn],upMax[maxn],downMax[maxn],downSec[maxn];
25 int n;
26
27 void init(){
28     cnt = 0;
29     memset(maxSon,-1,sizeof maxSon);
30     memset(upMax,-1,sizeof upMax);
31     memset(downMax,-1, sizeof downMax);
32     memset(downSec,-1 ,sizeof downSec);
33     memset(head,-1,sizeof head);
34 }
35
36 int dfs1(int u,int fa){
37     if(downMax[u]>=0) return downMax[u];
38     downMax[u] = upMax[u] = downSec[u] = maxSon[u] = 0;
39
40     for(int i = head[u];~i;i = e[i].next){
41         int v = e[i].to;
42         if(v==fa) continue;
43         if(downMax[u]<dfs1(v,u)+e[i].w){
44             maxSon[u] = v;
45             downSec[u] = max(downSec[u],downMax[u]);
46             downMax[u] = dfs1(v,u) + e[i].w;
47         }
48         else if(downSec[u]<dfs1(v,u)+e[i].w)
49             downSec[u] = max(downSec[u],dfs1(v,u)+e[i].w);
50     }
51     return downMax[u];
52 }
53 void dfs2(int u,int fa){
54     for(int i=head[u];~i;i=e[i].next){
55         int v = e[i].to;
56         if(v==fa) continue;

```

```

57         if(v==maxSon[u]) upMax[v] = e[i].w+max(upMax[u],downSec[u]);
58         else upMax[v] = e[i].w+max(upMax[u],downMax[u]);
59         dfs2(v,u);
60     }
61 }
62
63 int main(){
64     while (~scanf("%d",&n))
65     {
66         init();
67         for(int i=2;i<=n;i++){
68             int v,w;
69             scanf("%d%d",&v,&w);
70             addEdge(i,v,w);
71             addEdge(v,i,w);
72         }
73         dfs1(1,-1);
74         dfs2(1,-1);
75         for(int i = 1;i<=n;i++)
76             printf("%d\n",max(upMax[i],downMax[i]));
77     }
78     return 0;
79 }

```

解释

采用链式前向星的数据结构存图，可以节省空间，加快速度。

对于`downSec`，即向下走的第二远距离，可以在求向下走的最远距离的时候求出来，当`i`到`i`的子树的距离小于之前的最大值的时候，与次大值比较一下，如果大于次大值的时候更新一下就好了。

对于`dfs2`则是实现了上面的状态转移方程。

复杂度分析

由于每个节点都只访问了一遍，因此时间复杂度是 $O(n)$ 的。

贪心-过河

问题描述

[题目链接](#)

N个人用一条船过河，船只能载两个人，每个人有一个划船速度，船的速度取决于船上人最慢的划船速度。计算出需要多少时间可以让所有人过河。

分析

人多的时候，船划过去还要划过来，因此朴素的想法就是要使回来的时间最短，因此总是选时间最短和最长的人搭配。

但是当人数少到4人的时候，就需要考虑最后一趟是不需要计算把船划回来的时间了，因此上面的贪心策略就不成立了，就得是使去回去回去的时间最短。在这里就有两种策略，一种依然采用之前的贪心策略，一种是时间短的两人过去后，回来一人，再让时间长的两人共同渡河，采取两种策略耗时短的那种。

为什么不是3人的时候讨论呢？因为此时的情况肯定是唯一的。至于5人的时候，可以转到只剩4人的情况讨论。

有了上面的结论之后，我们再审视人多的时候的贪心策略，发现这是不合理的，因为人多的时候也可以考虑耗时最长和最短的两人，区别是最后一趟不划过去而是直接考虑下一个子问题，这时把耗时最长的两人送了过去。

证明

对于人数=4时：

设耗费时间为 $t_1 < t_2 < t_3 < t_4$ 。

对于策略1，有耗时： $t' = (t_4 + t_1) + (t_3 + t_1) + t_2$ 。

对于策略2，有耗时： $t'' = (t_2 + t_1) + (t_4 + t_2) + t_2$ 。

$t' - t'' = t_3 + t_1 - 2 * t_2$, 当 $t_3 + t_1 > 2 * t_2$ 的时候选择策略1，否则选择2。

对于人数大于4时，

对于策略1，有耗时： $t' = (t_4 + t_1) + (t_3 + t_1)$ 。

对于策略2，有耗时： $t'' = (t_2 + t_1) + (t_4 + t_2)$ 。

$t' - t'' = t_3 + t_1 - 2 * t_2$, 当 $t_3 + t_1 > 2 * t_2$ 的时候选择策略1，否则选择2。

代码

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int maxn=1010;
8  int t,n,ans;
```

```

9  int arr[maxn];
10
11  int Partition(int a[],int l,int r){
12      int x = a[l];
13      int i=l,j=r;
14      while (true)
15      {
16          while(x<=a[j]&& i<j) j--;
17          while(a[i]<=x&& i<j) i++;
18          if(i==j) break;
19          swap(a[i],a[j]);
20      }
21      a[l] = a[i];
22      a[i] = x;
23      return i;
24  }
25
26
27  int quickSort(int a[],int l,int r,int k){
28      if(l>=r) return a[l];
29      int p = Partition(a,l,r);
30      int curK = r-p+1;
31      if(curK>k) return quickSort(a,p+1,r,k);
32      else return quickSort(a,l,p,k-curK+1);
33  }
34
35
36  int main()
37  {
38      scanf("%d",&t);
39      while (t--)
40      {
41          ans=0;
42          scanf("%d",&n);
43          for(int i=1;i<=n;i++)
44              scanf("%d",&arr[i]);
45          quickSort(arr,1,n,n);
46          while(n>3)
47          {
48              ans+=min(arr[1]*2+arr[n]+arr[n-1],2*arr[2]+arr[1]+arr[n]);
49              n-=2;
50          }
51          if(n==3)
52          {
53              ans+=arr[1]+arr[2]+arr[3];
54          }
55          else ans+=arr[n];
56
57          printf("%d\n",ans);

```



```
58     }
59
60     return 0;
61 }
62
```

解释

对于普通贪心想法： $t'=(t_4+t_1)+(t_3+t_1)+t_2$ ，观察这个式子，我们可以看到若 t_3 、 t_4 很大的话，其实耗时主要是取决于分别运送 t_3 、 t_4 的时间。如果能够将 t_3 、 t_4 一起运过去的话，就可以节约很多时间。同时还要使船能够回来，就要让人先在对岸等着，因此很容易想到先让耗时最短的两人去对面，然后一人运回来，等 t_3 、 t_4 过去后再由另一人运回来，这样就节约了近一半的时间。这大概就是策略2的来源。

具体选择的时候，选择两种策略较短的就可以了。

对于贪心算法，我们可以看到其实并不是无脑的选择一种方案就可以，有时也需要在一些方案里比较以达到最优解。

复杂度分析

时间复杂度主要由快速排序和线性部分组成，因此时间复杂度为 $O(n\lg n)$ 。

回溯-带宽

问题描述

[题目链接](#)

输入一个图，节点数不超过8，有一个由所有节点组成的序列。定义带宽为一个节点与其在图中相邻的节点在序列中的最远距离，整个序列的带宽为所有节点的带宽的最大值。求出带宽最小的序列。

分析

这个问题的解状态空间树是一颗排列树。因此考虑用dfs的方法来求解问题。

显约束：每个字符只出现一次。

隐约束：由于是求的所有的最大值的最小值，因此在dfs过程中可以用启发函数进行剪枝，即：考虑当前获得的最终最小值和当前最大值比较，看是否能够有得到更优解的可能。

有两种方法剪枝：

- 如果当前的最大值 \geq 之前的最小值，剪枝。
- 如果未来的带宽最小值 \geq 之前的最小值，剪枝。

同时这道题需要处理输入，需要把离散的字母映射为连续的数字，这样在搜索时可以提升速度。

证明

剪枝1：如果当前的最大值小 \geq 之前的最小值，那么无论之后的最大值有多么小，都不可能得到更优的解，因此可以剪枝。

剪枝2：考虑未来的带宽，与当前节点相邻的但是还未出现的节点挨个排列在当前节点之后，即未来的最小带宽，如果此带宽已经 \geq 之前的最小值，也就是说未来的最最优解都不比之前的最小值更优，那么就没有搜索的必要，因此可以剪枝。

代码

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int INF = 0x3f3f3f3f;
6
7  vector<int> G[30];
8  int vis[30],res;
9  char prt[30];
10 char permu[30];
11 map<char,int> mp;
12 char num2ch[30];
13 int cnt = 0;
14
15 void PRT(int k){
16     for(int i=0;i<cnt;i++){
17         printf("%c ",prt[i]);
18     }
19     printf("-> %d\n",k);
20 }
21
22 void dfs(int dep,int maxWid){
23     if(dep == cnt && res!=maxWid ){
24         for(int i=0;i<cnt;i++) prt[i] = permu[i];
25         res = maxWid;
26         // PRT(res);
27         return ;
28     }
29     for(int i=1;i<=cnt;i++){
30         if(!vis[i]) vis[i] = 1;
31         else continue;
32         permu[dep] = num2ch[i];
33         int tmpWid = 0;
34         int unvisCnt = G[i].size();
35         for(int j=0;j<G[i].size();j++){
36             for(int k=0;k<dep;k++) if(G[i][j]==mp[permu[k]]) tmpWid =
max(tmpWid,dep-k),unvisCnt--;
```

```

37     }
38     maxWid = max(tmpWid,maxWid);
39     if(maxWid<res && unvisCnt<res ) dfs(dep+1,maxWid);
40     vis[i] = 0;
41 }
42 }
43 void init(){
44     for(int i=0;i<30;i++) G[i].clear();
45     mp.clear();
46     cnt = 0;
47     res = INF;
48 }
49 int main(){
50     char s[1000];
51     while (cin>>s&&s[0]!='#'){
52         init();
53         int len = strlen(s);
54         for (char ch = 'A'; ch <= 'Z'; ch++)
55             if (strchr(s, ch) != NULL) {
56                 mp[ch] = ++cnt;
57                 num2ch[cnt] = ch;
58             }
59         char u = s[0];
60         for(int i=1;i<len;i++){
61             char v = s[i];
62             if(v==':') continue;
63             if(v==';'){
64                 u = s[++i];
65                 continue;
66             }
67             auto it = find(G[mp[u]].begin(),G[mp[u]].end(),mp[v]);
68             if(it!=G[mp[u]].end()) continue;
69             G[mp[u]].push_back(mp[v]);
70             G[mp[v]].push_back(mp[u]);
71         }
72         for(int i=0;i<30;i++) sort(G[i].begin(),G[i].end());
73         dfs(0,0);
74         PRT(res);
75     }
76     return 0;
77 }

```

解释

先将出现的字母从小到大映射为1-cnt。从小到大的目的和对邻接表的排序是为了保证搜索时是按照字典序来排列的。

另外输入时，是会有重复的边出现，因此还要判重。

*vis*来表示显约束。

*tmpWid*表示遍历当前节点的邻点获得的最大值，*unvisCnt*是还未出现的邻点个数，表示了未来可能的最优带宽。

23行 `res!=maxWid` 是为了保证储存的是先访问到排列，使字典序最小。

复杂度分析

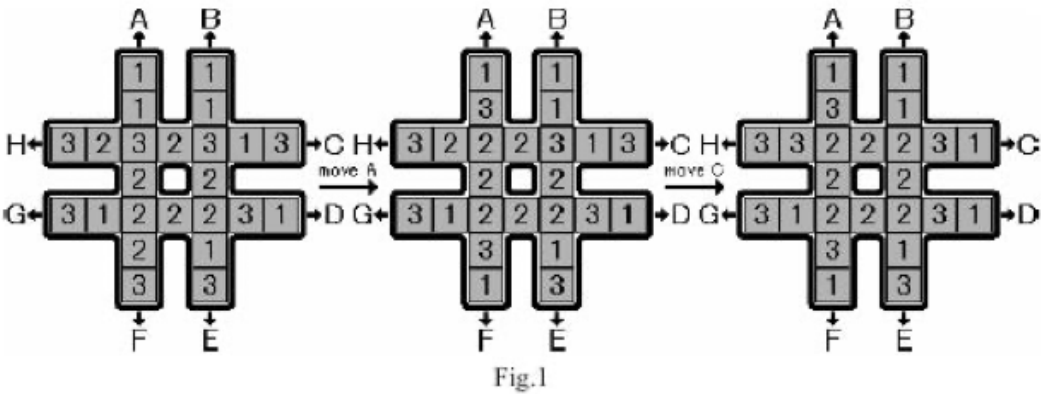
因为解空间树是一颗排序树，所以时间复杂度是 $O(n)$ 。

分支限界-旋转游戏

问题描述

题目链接

有如图的棋盘，分别有8个1、2、3。可以沿A-H方向循环滑动一格里面的数字（即滑出的数字会补到空位上），例如从第一步到第二步就是沿F方向滑动。现在用尽量少的次数使中间8个数字相同，输出最小的字典序，若不需要移动则输出“No moves needed”，都要输出那个相同的数字。



输入顺序为

		0		1		
		2		3		
4	5	6	7	8	9	10
		11		12		
13	14	15	16	17	18	19
		20		21		
		22		23		

分析

这道题有 $\frac{24!}{8! \times 8! \times 8!} \approx 9e9$ 种状态。因为找不到状态转移的思路和贪心以及划分为子问题求解，因此这道题应该是对状态空间树的搜索来求解。

确定了大概思路，就要想用什么方法来搜索了。

- 用dfs吗？好像节点有点多，盲目地搜索会在前面的子树中浪费过多时间而导致超时；可以用回溯法来剪枝吗？这里的状态直接好像没有显约束和隐约束，就是位置变换了一下而已，因此好像不可行。
- 用bfs吗？可以最早搜索到解，但是如果解的深度够深的话， $9e9$ 的空间明显不够，因此也不可行。
- 那么我们就需要采用一种兼具两者优点的方法——迭代加深搜索(IDA*)，即限定搜索下界的dfs。每次搜索失败后，就加深下界，这样就保证了搜索深度不会超过可行解的深度，同时保留了dfs的优点：线性存储的要求，避免了bfs导致的空间爆炸。虽然可能在某些情况下比dfs慢一些，但是通用性却更好。
- 乐观估计函数 $H(x)$ =当前中心8个数字最少能走几步到达全部相同的状态。

确定了方法后，就要考虑怎么进行建图、旋转操作和判断、输出了。

- 建图：用G[24]数组来表示输入的位置。
- 旋转操作就是把对应的位置上的数重新赋值。
- G[24]数组中对应中心的数字相同了就找到了解。
- 从A-H顺序操作，把操作储存到输出字符串中，自然是最小的字典序。

证明

在状态空间树上bfs遍历的每个节点，迭代加深搜索都能访问到，由于bfs的完备性，所以迭代加深搜索肯定能找到最优解。

同时每次只沿着一条路径搜索，所以空间复杂度和dfs一样是线性的。

代码

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int maxn = 1e4+10;
5  int cent8[] = {6,7,8,11,12,15,16,17},G[24],findAns,maxDep;
6  char sol[maxn];
7  int op[][7] = {
8      {0,2,6,11,15,20,22 }, //A
9      {1,3,8,12,17,21,23 }, //B
10     {10,9,8,7,6,5,4 }, //C
11     {19,18,17,16,15,14,13}, //D
12     {23,21,17,12,8,3,1 }, //E
13     {22,20,15,11,6,2,0 }, //F
14     {13,14,15,16,17,18,19}, //G
15     {4,5,6,7,8,9,10 } //H
16 };
```

```

17
18 int invOp[8] = {5,4,7,6,1,0,3,2};
19
20 void Grotate(int opNum){
21     int tmp = G[op[opNum][0]];
22     for(int i=0;i<=5;i++){
23         G[op[opNum][i]] = G[op[opNum][i+1]];
24     }
25     G[op[opNum][6]] = tmp;
26 }
27
28 int H(){
29     int cnt = 0;
30     int nCnt[4]={0};
31     for(int i=0;i<8;i++){
32         cnt = max(cnt,++nCnt[G[cent8[i]]]);
33     }
34     return 8-cnt;
35 }
36
37 void dfs(int dep,int preOp){
38     if(findAns||dep>maxDep||dep+H(>maxDep) return;
39     if(!H()){
40         sol[dep] = '\0';
41         printf("%s\n%d\n",sol,G[cent8[0]]);
42         findAns = 1;
43         return ;
44     }
45     for(int i=0;i<8;i++){
46         if(dep&&i==invOp[preOp]) continue;
47         Grotate(i);
48         sol[dep] = i+'A';
49         dfs(dep+1,i);
50         Grotate(invOp[i]);
51     }
52 }
53
54 int main(){
55     while(scanf("%d",&G[0])&&G[0]){
56         for(int i =1;i<24;i++) scanf("%d",&G[i]);
57         if(!H()){
58             printf("No moves needed\n%d\n",G[cent8[0]]);
59             continue;
60         }
61         findAns = 0;
62         maxDep = 1;
63         while (true){
64             dfs(0,100);
65             maxDep++;

```

```
66         if(findAns) break;
67     }
68 }
69 return 0;
70 }
```

解释

`cent8[8]` 记录的是图中间的8个数字在 $G[24]$ 中的下标。

`op[8][7]`记录的是要进行操作的数在 $G[24]$ 中的下标。

`invOp[8]`记录的是 $A - H$ 操作的逆操作序号，防止dfs时逆向搜索。

$H()$ 是乐观估计函数，统计了当前1、2、3中个数最多的数，然后求与8相补的数，就得到了最少的步数。

`maxDep`是搜索深度的下界，每次搜索失败后递增。

复杂度分析

$\lceil \log_7(9e9) \rceil = 12$ ，所以树的深度为12。

时间复杂度：每次多搜索一层的时候，迭代加深搜索的节点数都是呈指数增长，由等比数列可知，最后一层搜索的节点数约等于前面搜索的所有节点数。又因为迭代加深搜索最后一层访问的节点数和bfs的节点数相同，所以其复杂度约为bfs两倍，即 $O(2n)$ 。

空间复杂度为 $O(\log_7 n)$ 。

参考文献

[1] T. H. Cormen, et al. (著), 殷建平等 (译), 算法导论 (原书第三版) [M], 北京, 机械工业出版社, 2013: 527-542

[2] 霍纳法则 [OL], <https://baike.baidu.com/item/霍纳法则/4822190?fr=aladdin>, 2020-06-05