中国科学院大学历年习题

习题一 复杂性分析初步

1. 试确定下述程序的执行步数，该函数实现一个 m×n 矩阵与一个 n×p 矩阵之间的乘法：

矩阵乘法运算

```
template<class T>
void Mult(T **a, T **b, int m, int n, int p)
{//m×n 矩阵 a 与 n×p 矩阵 b 相成得到 m×p 矩阵 c
   for(int i=0; i<m; i++)
      for(int j=0; j<p; j++){
        T sum=0;
        for(int k=0; k<n; k++)
           Sum+=a[i][k]*b[k][j];
        C[i][j]=sum;
      }
}
```

语　　句	s/e	频率	总步数
template<class T>			
void Mult(T **a, T **b, int m, int n, int p)	0	0	0
{			
for(int i=0; i<m; i++)	1	**m+1**	m+1
for(int j=0; j<p; j++) {	1	m***(p+1)**	m*p+m
T sum=0;	1	m*p	m*p
for(int k=0; k<n; k++)	1	m*p***(n+1)**	m*p*n+m*p
Sum+=a[i][k]*b[k][j];	**1**	m*p*n	m*p*n
C[i][j]=sum;	**1**	**m*p**	**m*p**
}			
}			
总　　计		**2*m*p*n+4*m*p+2*m+1**	

其中 s/e 表示每次执行该语句所要执行的程序步数。

频率是指该语句总的执行次数。

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？

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，则还回 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;
}
```

最好，最坏，平均比较次数都是 2*（n-1）

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，则还回 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;
}
```

最坏 2*（n-1），　最好 n-1，　平均 $\frac{3(n-1)}{2}$

3. 证明以下不等式不成立：

1）. $10n^2+9=O(n)$;

2）. $n^2\log n=\Theta(n^2)$；

4．证明：当且仅当 $\lim_{n\to\infty} f(n)/g(n)=0$ 时， $f(n)=o(g(n))$ 。

5. 下面那些规则是正确的？为什么？

1). $\{f(n)=O(F(n)), g(n)=O(G(n))\}\Rightarrow f(n)/g(n)=O(F(n)/G(n))$； 错

2). $\{f(n)=O(F(n)), g(n)=O(G(n))\}\Rightarrow f(n)/g(n)=\Omega(F(n)/G(n))$； 错

3). $\{f(n)=O(F(n)), g(n)=O(G(n))\}\Rightarrow f(n)/g(n)=\Theta(F(n)/G(n))$； 错

4). $\{f(n)=\Omega(F(n)), g(n)=\Omega(G(n))\}\Rightarrow f(n)/g(n)=\Omega(F(n)/G(n))$； 错

5). $\{f(n)=\Omega(F(n)), g(n)=\Omega(G(n))\}\Rightarrow f(n)/g(n)=\mathrm{O}(F(n)/G(n))$。 错

6). $\{f(n)=\Theta(F(n)), g(n)=\Theta(G(n))\}\Rightarrow f(n)/g(n)=\Theta(F(n)/G(n))$ 对

6. 按照渐进阶从低到高的顺序排列以下表达式:

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

顺序：$\log n < n^{2/3} < 20n < 4n^2 < 3^n < n!$

7. 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?

规模	时间复杂度(步数)	原运行速度（时间/每步）	总时间
n	$T(n)=3*2^n$	t_0	t

关系式为 时间复杂度（计算步数）*运行速度(时间/每步)=运行所需时间，即

$$T(n)*t_0=t$$

解：设在新机器上 t 秒内能解决规模为 m 的问题，时间复杂度变为 $T(m)=3*2^m$，

由于新机器运行速度提高 64 倍，则运行速度变为 $t_{新}=\frac{t_0}{64}$，

由关系式 $T(n)*t_0=t,\ T(m)*t_{新}=t$，得

$$3*2^n*t_0=t,$$

$$3*2^m*\frac{t_0}{64}=t$$

解得

$$m = n + 6$$

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

设在新机器上用t秒时间能解决输入规模为N的问题，则

由于新复杂度 $T_{新}(N) = N^2$，新机器的运行速度为$t_{新} = \frac{t_0}{64}$，

代入关系式$T_{新}(N) * t_{新} = t$，得

$$N^2 * \frac{t_0}{64} = t = 3 * 2^n * t_0,$$

解得

$$N = 8\sqrt{3 \cdot 2^n}$$

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？

设可解决的最大时间复杂度为$T_{\max}$，则

$$T_{\max} * \frac{t_0}{64} = t = 3 * 2^n * t_0$$

可解决的最大时间复杂度为$T_{\max} = 192 * 2^n$，（n 为原始的输入规模）。

因为$T(n) = 8 < T_{\max}$，且$T(n)$为常数不随输入规模 n 变化，

所以任意规模的 n 都可在 t 秒内解决。

8. 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。

解：方法一：

当$n > 1$时，由递推公式$F(n) = F(n-1) + F(n-2)$得

特征方程为

$$x^2 = x+1$$

解得

$$x_1 = \frac{1+\sqrt{5}}{2}, \quad x_2 = \frac{1-\sqrt{5}}{2}$$

则可设

$$F(n) = c_1 x_1^n + c_2 x_2^n$$

由 $F(2)=2$， $F(3)=3$，解得 $c_1 = \frac{1+\sqrt{5}}{2\sqrt{5}}$， $c_2 = -\frac{1-\sqrt{5}}{2\sqrt{5}}$

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

当 $n = 0,1$ 时，带入验证亦成立。

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

方法二：

也可直接推导

$$F(n) = F(n-1) + F(n-2)$$

可得

$$a_n - \alpha\, a_{n-1} = \beta[a_{n-1} - \alpha\, a_{n-2})]$$

可得

$$\alpha, \beta = \frac{1 \pm \sqrt{5}}{2},$$

设

$$b_n = a_n - \alpha\, a_{n-1},$$

则 b_n 为等比数列，先求出 b_n，然后代入即可求得 a_n。

第二章部分习题参考答案

1.证明下列结论：

1）在一个无向图中，如果每个顶点的度大于等于 2，则该该图一定含有圈；

2）在一个有向图 D 中，如果每个顶点的出度都大于等于 1，则该图一定含有一个有向圈。

1）证明：设无向图最长的迹$P=V_0V_1\cdots V_k$,每个顶点度大于等于 2，故存在与V_1相异的点V'与V_0相邻，若$V'\notin P$,则得到比P更长的迹，与 P 的取法矛盾。因此，$V'\in P$，是闭迹，从而存在圈$V_0V_1\cdots V'V_0$.

证明*：设在无向图 G 中，有 n 个顶点，m 条边。由题意知，m>=(2n)/2=n，而一个含有 n 个顶点的树有 n-1 条边。因 m>=n>n-1，故该图一定含有圈。

（定义：迹是指边不重复的途径，而顶点不重复的途径称为路。起点和终点重合的途径称为闭途径，起点和终点重合的迹称为闭迹，顶点不重复的闭迹称为圈。）

2）证明：设有向图最长的有向迹$P=V_0V_1\cdots V_k$,每个顶点出度大于等于 1，故存在V'为V^k的出度连接点，使得V_kV'成为一条有向边，若$V'\notin P$,则得到比P更长的有向迹，与 P 矛盾，因此必有$V'\in P$，从而该图一定含有有向圈。

2.设 D 是至少有三个顶点的连通有向图。如果 D 中包含有向的 Euler 环游（即是通过 D 中每条有向边恰好一次的闭迹），则 D 中每一顶点的出度和入度相等。反之，如果 D 中每一顶点的出度与入度都相等，则 D 一定包含有向的 Euler 环游。这两个结论是正确的吗？请说明理由。如果 G 是至少有三个顶点的无向图，则 G 包含 Euler 环游的条件是什么？

证明：1）若图 D 中包含有向 Euler 环游，下证明每个顶点的入度和出度相等。

如果该有向图含有 Euler 环游，那么该环游必经过每个顶点至少一次，每经过一次，必为“进”一次接着“出”一次，从而入度等于出度。从而，对于任意顶点，不管该环游经过该顶点多少次，必有入度等于出度。

2）若图 D 中每个顶点的入度和出度相等，则该图 D 包含 Euler 环游。证明如下。

对顶点个数进行归纳。

当顶点数|v(D)|=2 时，因为每个点的入度和出度相等，易得构成有向 Euler 环游。

假设顶点数|v(D)|=k 时结论成立，则

当顶点数|v(D)|=k + 1 时，任取 v∈v(D).设 S={以 v 为终点的边}，K={以 v 为始点的边}，因为 v 的入度和出度相等，故 S 和 K 中边数相等。记 G=D-v.对 G 做如下操作:

任取 S 和 K 中各一条边e_1、e_2，设在 D 中$e_1 = v_1v$，$e_2 = vv_2$，则对 G 和 S 做如下操作 $G = G + v_1v_2$，$S = S - \{e_2\}$，重复此步骤直到 S 为空。这个过程最终得到的 G 有 k 个顶点，且每个顶点的度与在 G 中完全一样。由归纳假设，G 中存在有向 Euler 环游，设为 C。在 G 中从任一点出发沿 C 的对应边前行，每当遇到上述添加边 v1v2 时，都用对应的两条边 e1，e2 代替，这样可以获得有向 Euler 环游。

3）G 是至少有三个顶点的无向图，则 G 包含 Euler 环游等价于 G 中无奇度顶点。（即任意顶点的度为偶数）。

3.设 G 是具有 n 个顶点和 m 条边的无向图,如果 G 是连通的,而且满足 m = n-1,证明 G 是树。

证明：思路一：

只需证明 G 中无圈。

若 G 中有圈，则删去圈上任一条边 G 仍连通。而每个连通图边数 e>=n(顶点数) – 1，但删去一条边后 G 中只有 n-2 条边，此时不连通，从而矛盾，故 G 中无圈，所以 G 为树。

思路二：

当$n = 2$时，$m = 2 - 1 = 1$，两个顶点一条边且连通无环路，显然是树。

设当$n = k - 1(k \in N, k \geq 2)$时，命题成立，则

当$n = k$时，因为 G 连通且无环路，所以至少存在一个顶点V_1，他的度数为 1，设该顶点所关联的边为$e_1 = (V_1, V_2)$.那么去掉顶点V_1和e_1，便得到了一个有 k-1 个顶点的连通无向无环路的子图G'，且G'的边数$m' = m - 1$，顶点数$n' = n - 1$。由

于 m=n-1, 所以 $m^{'}=m-1=(n-1)-1=n^{'}-1$，由归纳假设知，$G^{'}$ 是树。

由于 G 相当于在 $G^{'}$ 中为 V_2 添加了一个子节点，所以 G 也是树。

由（1），（2）原命题得证。

4. 假设用一个 $n\times n$ 的数组来描述一个有向图的 $n\times n$ 邻接矩阵，完成下面工作：

1）编写一个函数以确定顶点的出度，函数的复杂性应为 $\Theta(n)$;：

2）编写一个函数以确定图中边的数目，函数的复杂性应为 $\Theta(n^2)$;

3）编写一个函数删除边 (i,j)，并确定代码的复杂性。

解答：（1）邻接矩阵表示为 $a_{n\times n}$，待确定的顶点为第 m 个顶点 v_m

```
int    CountVout(int *a,int n,int m){
int out = 0;
for(int i=0;i<n;i++)
    if(a[m-1][i]==1) out++;
return    out;
}
```

（2）确定图中边的数目的函数如下：

```
int    EdgeNumber(int*a,int n){
int num =0;
for(int i=0;i<n;i++)
   for(int j=0;j<n;j++)
       if(a[i][j]==1) num++;
return    num;
}
```

（3）删除边（i,j）的函数如下：

```
void    deleteEdge(int *a，int i ,int j){
        if(a[i-1][j-1]==0)    return;
        a[i-1][j-1] = 0;
        return;
```

}

代码的时间复杂性为Θ(1)

5.实现图的 D-搜索算法，要求用 SPARKS 语言写出算法的伪代码，或者用一种计算机高级语言写出程序。

解:D 搜索算法的基本思想是，用栈代替 BFS 中的队列，先将起始顶点存入栈中，搜索时，取出栈顶的元素，遍历搜索其相邻接点，若其邻接点还未搜索，则存入栈中并标记，遍历所有邻接点后，取出此时栈顶的元素转入下一轮遍历搜索，直至栈变为空栈。

```
Proc   DBFS (v)   //从顶点 v 开始，数组 visited 标示顶点被访问的顺序；
    PushS(v , S); //首先访问 v,将 S 初始化为只含有一个元素 v 的栈
    count :=count +1; visited[v] := count;
    While   S 非空 do
     u :=PullHead(S); count :=count +1; visited[w] := count; //区别队列先进先出，此先进后出
        for 邻接于 u 的所有顶点 w   do
            if   s[w] = 0 then
                    PushS(w,S); //将 w 存入栈 S
                    s[w]:= 1;
            end{if}
        end{for}
    end{while}
end{DBFS}
```

注：PushS(w,S)将 w 存入栈 S; PullHead(S)为取出栈最上面的元素，并从栈中删除

```
Proc   DBFT(G，m) //m 为不连通分支数
   count:=0 ;计数器，标示已经被访问的顶点个数
   for i to n do
       s[i]:=0; //数组 s 用来标示各顶点是否曾被搜索，是则标记为 1，否则标记为 0；
   end{for}
   for i to m do   //遍历不连通分支的情况
        if s[i]=0 then
           DBFS (i);
```

end{if}

end{for}

end{DBFT}

6. 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。

邻接链表

A->B->E|0

B->A->C|0

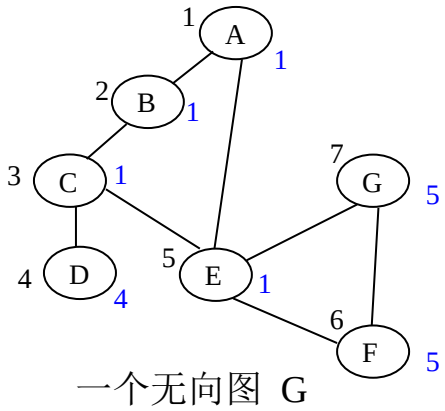
C->B->D->E|0

D->C|0

E->A->C->F->G|0

F->E->G|0

G->E->F|0



一个无向图 G

解：初始化　数组 DFN:=0，　num=1；

A 为树的根节点,对 A 计算 DFNL(A,null)，

DFN(A)：=num=1;　L(A)：=num=1;　num:=1+1=2。

从邻接链表查到 A 的邻接点 B，

因为 DFN(B)=0,对 B 计算 DFNL(B,A)

DFN(B)：= num=2;　　L(B)：=num=2;　　num：=2+1=3。

查邻接链表得到 B 的邻接点 A，因为 DFN(A)=1≠0, 但 A=A,即是 B 的父节点，无操作。

接着查找邻接链表得到 B 的邻接点 C，

因为 DFN(C)=0,对 C 计算 DFNL(C,B)

DFN(C)：= num=3;　　L(C)：=num=3;　　num:=3+1=4。

查找 C 的邻接点 B，因为 DFN(B)=1≠0, 但 B=B,即是 C 的父节点，无操作。

接着查找邻接链表得到 C 的邻接点 D，

因为 DFN(D)=0,对 D **计算 DFNL(D,C)，**

DFN(D)：= num=4;　　L(D)：=num=4;　num:=4+1=5。

查找得 D 邻接点 C，而 DFN(C)=3≠0,但 C=C，为 D 的父节点， L(D)保持不变。

D 的邻接链表结束，DFNL(D,C)的计算结束。

返回到 D 的父节点 C，查找邻接链表得到 C 的邻接点 E，

因为 DFN(E)=0,对 E 计算 DFNL(E,C),

DFN(E)：=num=5; L(E)：=num=5; num:5+1=6;

查找得 E 邻接点 A,因 DFN(A)=1≠0,又 A≠C,变换 L(E)=min(L(E),DFN(A))=1。

查找得 E 邻接点 C，因 DFN(C)=3≠0，但 C=C，无操作。

查找得 E 邻接点 F，因 DFN(F)=0，

对 F 计算 DFNL(F,E)，

DFN(F)：=num=6; L(F)：=num=6; num:=6+1=7;

查找得 F 邻接点 E，因 DFN(E)=5≠0，但 E=E，无操作。

查找得 F 邻接点 G，因 DFN(G)=0，

对 G **计算 DFNL(G,F)，**

DFN(G)：=num=7; L(G)：=num=7; num=7+1=8;

查找 G 邻接点 E,因 DFN(E)=5≠0,又 E≠F,L(G)=min(L(G),DFN(E))=5

查找得 G 邻接点 F,因 DFN(F)=6≠0，但 F=F，无操作。

G 的邻接链表结束，DFNL(G,F)的计算结束。

L(F)：=min(L(F)，L(G))=min(6,5)=5

F 的邻接链表结束，DFNL(F,E)的计算结束。

L(E):=min(L(E),L(F))=min(1,5)=1

E 邻接链表结束， DFNL(E,C)计算结束。

L(C):=min(L(C),L(E))=min(3,1)=1

C 的邻接链表结束，DFNL(C,B)计算结束。

L(B):=min(L(B),L(C))=min(2,1)=1

查找 B 的邻接链表结束，DFNL(B,A)计算结束。

L(A):=min(L(A),L(B))=1

查找得 A 的邻接点 E，因 DFN(E)=0，又 E≠null,则 L(A)=min(L(A),DFN(E))=1

查找 A 的邻接链表结束，DFNL(A,null)计算结束。

最终结果为：深索数 DFN，与最低深索数 L 如下

DFN(A)=1，DFN(B)=2，DFN(C)=3，DFN(D)=4，DFN(E)=5，DFN(F)=6，DFN(G)=7

L(A)=1; L(B)=1; L(C)=1; L(D)=4; L(E)=1; L(F)=5;L(G)=5.

序	节点	DFN	L	栈顶—栈底	2-连通	割点
1	A	1	(1,0,0,0,0,0,0)	(A,B)		
2	B	2	(1,2,0,0,0,0,0)	(B,C),(A,B)		
3	C	3	(1,2,3,0,0,0,0)	(C,D),(B,C),(A,B)		
4	D	4	(1,2,3,4,0,0,0)	(B,C),(A,B)	{(C,D)};	C
5	E	5	(1,1,1,4,1,0,0)	(E,F),(E,A),(B,C),(A,B)		
6	F	6	(1,1,1,4,1,6,0)	(F,G), (E,F),(E,A),(B,C),(A,B)		
7	G	7	(1,1,1,4,1,5,5)	(E,A),(B,C),(A,B)	{(G,E),(F,G), (E,F)}	E
8			(1,1,1,4,1,5,5)		{(E,A),(B,C),(A,B)}	

附课本讲义程序 2-3-1 对图 2-3-5 的执行过程

开始 **DFNL**(A,*)

DFN(A):=1; L(A):=1; num:=2;

∵ DFN(B)=0, ∴ **DFNL**(B,A)

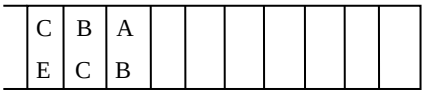
DFN(B):=2; L(B):=2; num:=3;

∵ DFN(A)=1≠0, 但 A=A, ∴不做任何事情

∵ DFN(C)=0, ∴ **DFNL**(C,B)

DFN(C):=3; L(C):=3; num:=4;

∵ DFN(B)=2≠0, 但 B=B, ∴不做任何事情

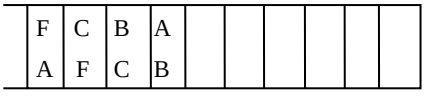


∵ DFN(D)=0, ∴ **DFNL**(D,C)

DFN(D):=4; L(D):=4 **> DFN(C)**; num:=5;

弹出

∵ DFN(C)=3≠0, 但 C=C, ∴不做任何事情



∵ DFN(E)=0, ∴ **DFNL**(E,C)

DFN(E):=5; L(E):=5 **> DFN(C)**; num:=6;

弹出（C,E）边

∵ DFN(C)=3≠0, 但 C=C,

∵ DFN(F)=0, ∴ **DFNL**(F,C)

DFN(F):=6; L(F):=6; num:=7;

∵ DFN(A)=1≠0, A≠C, ∴ L(F):=min{6,1}=1;

∵ DFN(C)=3≠0, 但 C=C,

F	F	C	B	A					
G	A	F	C	B					

∵ DFN(G)=0, ∴ **DFNL**(G,F)

DFN(G):=7; L(G):=7; num:=8;

∵ DFN(F)=6≠0, 但 F=F,

G	F	F	C	B	A				
H	G	A	F	C	B				

∵ DFN(H)=0, ∴ **DFNL**(H,G)

DFN(H):=8; L(H):=8 **> DFN(G)**; num:=9; 弹出

G	F	F	C	B	A				
I	G	A	F	C	B				

∵ DFN(G)=7≠0, 但 G=G,

I	G	F	F	C	B	A			
F	I	G	A	F	C	B			

∵ DFN(I)=0, ∴ **DFNL**(I,G)

DFN(I):=9; L(I):=9; num:=10;

I	I	G	F	F	C	B	A		
J	F	I	G	A	F	C	B		

∵ DFN(F)=6≠0, F≠G, ∴ L(I):=min{9,6}=6;

J	I	I	G	F	F	C	B	A	
F	J	F	I	G	A	F	C	B	

∵ DFN(G)=7≠0, 但 G=G,

J	J	I	I	G	F	F	C	B	A
G	F	J	F	I	G	A	F	C	B

∵ DFN(J)=0, ∴ **DFNL**(J,I)

DFN(J):=10; L(J):=10; num:=11;

∵ DFN(F)=6≠0, F≠I, ∴ L(J):=min{10,6}=6;

∵ DFN(G)=7≠0, G≠I, ∴ L(J):=min{6,7}=6;

∵ DFN(I)=9≠0, 但 I=I,

L(I):=min{6,6}=6;

L(G):=min{7,6}=6 **≥ DFN(F)** 弹出(J,G), (J,F), (I,J), (I,F), (G,I), (F,G) 边

L(F):=min{1,6}=1;

L(C):=min{3,1}=1;

L(B):=min{2,1}=1 **≥ DFN(A)** 弹出(F,A), (C,F),

(B,C), (A,B) 边

7 对图的另一种检索方法是 D-Search。该方法与 BFS 的不同之处在于将队列换成栈，即下一个要检测的结点是最新加到未检测结点表的那个结点。

1）写一个 D-Search 算法；

2）证明由结点 v 开始的 D-Search 能够访问 v 可到达的所有结点；

3）你的算法的时、空复杂度是什么？

解：1）同第 5 题，

```
proc DBFS(v) //宽度优先搜索 G，从顶点 v 开始执行，数组 visited 标示各
//顶点被访问的序数；数组 s 将用来标示各顶点是否曾被放进待检查队
//列，是则过标记为 1，否则标记为 0；计数器 count 计数到目前为止已
//经被访问的顶点个数，其初始化为在 v 之前已经被访问的顶点个数
PushS(v , S);// 将 S 初始化为只含有一个元素 v 的栈
while S 非空 do
u:= PullHead(S); count:=count+1; visited[u]:=count;
for 邻接于 u 的所有顶点 w do
if s[w]=0 then
PushS(w , S); //将 w 压入栈中
s[w]:=1;
end{if}
end{for}
end{while}
end{DBFS}
```

图的 D—搜索算法伪代码：

```
proc DBFT(G, v ) //count、s 同 DBFS 中的说明，branch 是统计图 G 的连通分支数
count:=0; branch:=0;
for i to n do
s[i]:=0; //将所有的顶点标记为未被访问
end{for}
for i to v do
if s[i]=0 then
DBFS(i); branch:=branch+1;
```

end{if}

end{for}

end{DBFT}

2）证明：除结点 v 外，只有当结点 w 满足 s[w]=0 时才被压入栈中，因此每个结点至多有一次被压入栈中，搜索不会出现重叠和死循环现象，对于每一个 v 可到达的节点，要么直接被访问，要么被压入栈中，只有栈内节点全部弹出被访问后，搜索才会结束，所以由结点 v 开始的 D-Search 能够访问 v 可到达的所有结点。

3)：除结点 v 外，只有当结点 w 满足 s[w]=0 时才被压入栈中，因此每个结点至多有一次被压入栈中。需要的栈 空间至多是 v -1；visited 数组变量所需要的空间为 v ；其余变量所用的空间为 O(1)，所以 s(v ,ε)= Θ (v)。

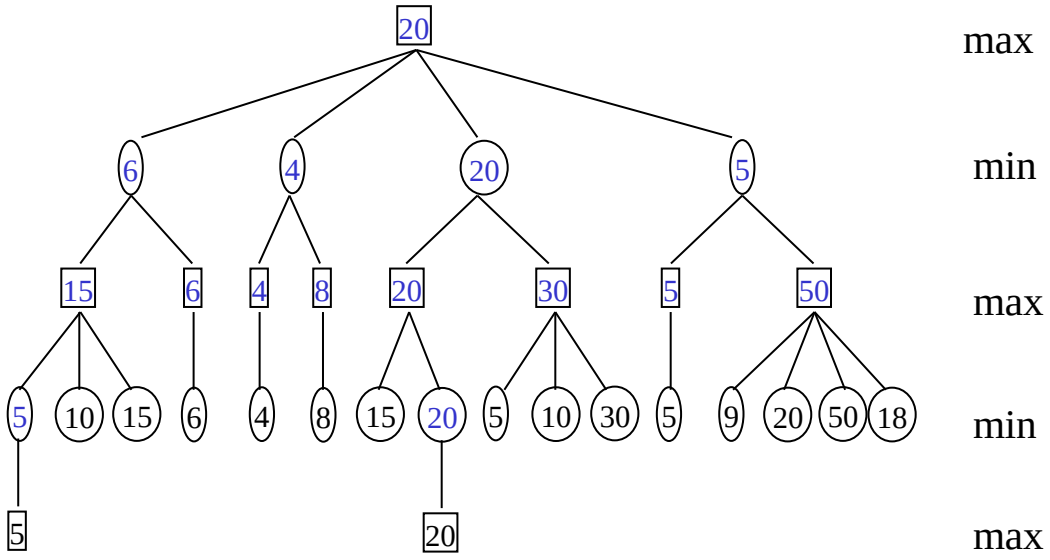
如果使用邻接链表， for 循环要做 d(u)次，而 while 循环需要做 v 次，又 visited、s 和 count 的赋值都需要 v 次操作，因而 t(v ， ε)= Θ (v + ε)。

如果采用邻接矩阵，则 while 循环总共需要做 v^2 次操作，visited、s 和 count 的赋值都需要 v 次操作，因而 t(v ， ε)= Θ (v^2)。

8.考虑下面这棵假想的对策树：

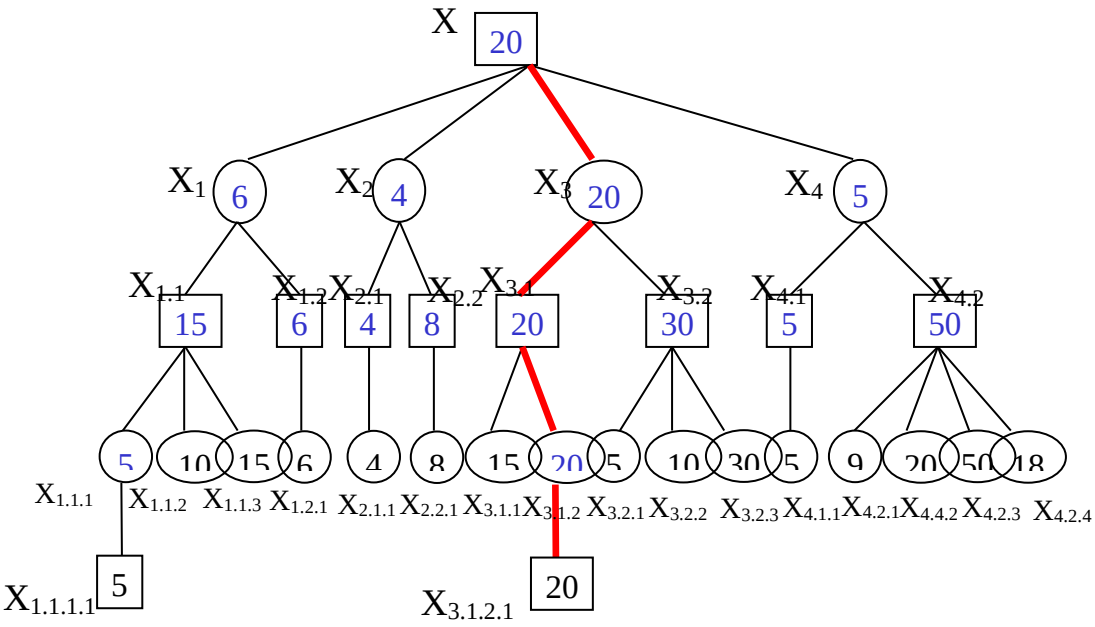
解：

1）使用最大最小方法(2-4-2)式获取各结点的值

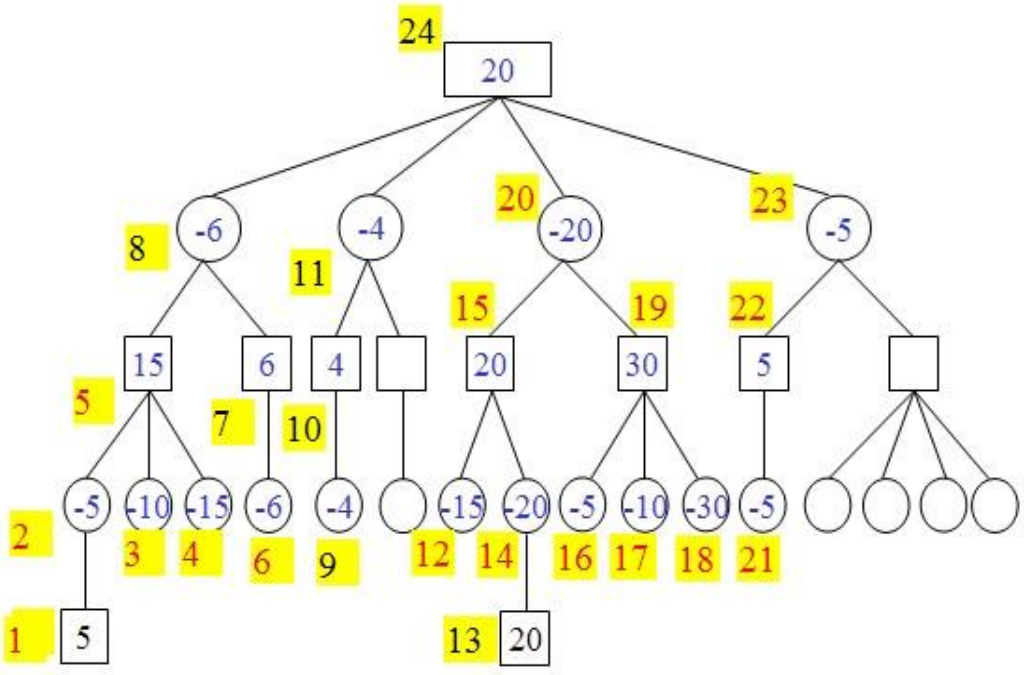


2 ）

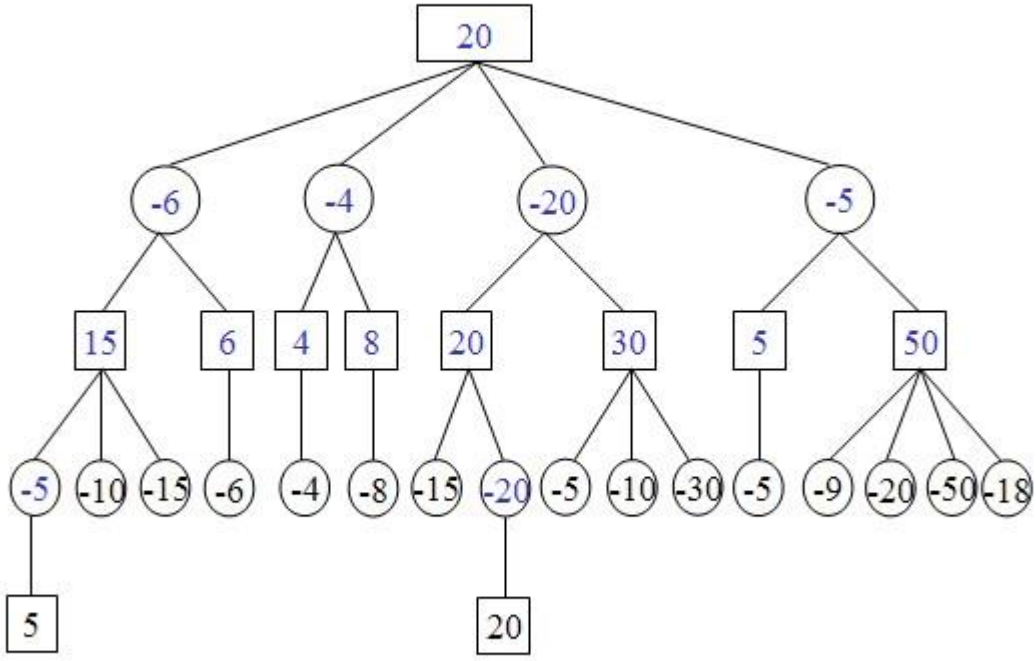
2）弈者A为获胜应该什么棋着？



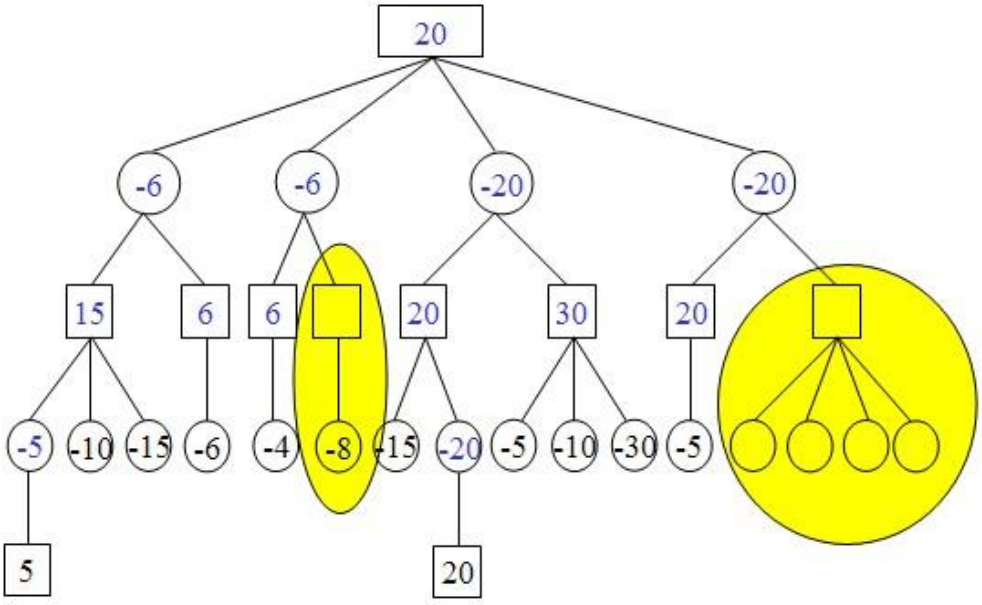
3）列出算法VEB计算这棵对策树结点的值时各结点被计算的顺序



4）对树中每个结点X，用(2-4-3)式计算V(X)；



5）在取X＝根，l=10, LB =-∞, D=∞的情况下，用算法AB计算此树的根的值期间，这棵树的那些结点没有计算？



第三章 分治算法习题

1、编写程序实现归并算法和快速排序算法

参见后附程序

2、用长为 100、200、300、400、500、600、700、800、900、1000 的 10 个数组的排列来统计这两种算法的时间复杂性。

有些同学用的微秒 us

用 s 可能需要把上面的长度改为 10000，……，100000，都可以

大部分的结果是快速排序算法要比归并算法快一些。

3、讨论归并排序算法的空间复杂性。

解析：归并排序由分解与合并两部分组成，如果用 $S(n)$ 表示归并排序所用的空间。

则由

MergeSort(low, mid) //将第一子数组排序 $S(n/2)$

MergeSort(mid+1, high) //将第二子数组排序 $S(n/2)$

Merge(low, mid, high) //归并两个已经排序的子数组 $O(n) \approx 2n$

则

$$S(n) = \max\{S(n/2), O(n)\}$$

$$S(n) \le S(n/2) + O(n)$$

递归推导得

$$\begin{aligned} S(n) &\le S(\frac{n}{2}) + O(n) \le S(\frac{n}{4}) + O(\frac{n}{2}) + O(n) \\ &\le S(1) + O(\frac{n}{2^k}) + O(\frac{n}{2^{k-1}}) + \cdots + O(\frac{n}{2}) + O(n) \\ &\le S(1) + O(2n) \\ &= O(n) \end{aligned}$$

又由存储数组长度为 n ，则有 $S(n) \ge O(n)$

因此，空间复杂度为 $O(n)$ 。

4、说明算法 PartSelect 的时间复杂性为 $O(n)$

证明：提示：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 $1/n$ 。因为 Partition 中的 case 语句所要求的时间都是 $O(n)$ ，所以，存在常数 c ，使得算法 PartSelect 的平均时间复杂度 $C_A^k(n)$ 可以表示为

$$C_A^k(n) \le cn + \frac{1}{n}(\sum_{1\le i<k} C_A^{k-i}(n-i) + \sum_{k<i\le n} C_A^k(i-1))$$

令 $R(n) = \max_k(C_A^k(n)),$取 $C \ge R(1),$试证明 $R(n) \le 4cn$ 。

证明：令 $C_A^k(n)$ 表示 n 个元素的数组 A 中寻找第 k 小元素的平均时间复杂度，因 $\text{Partition}(0, \text{n -1})$的时间复杂度是 $O(n)$，故存在常数 c，使得算法 PartSelect 的平均时间复杂度 $C_A^k(n)$ 可以表示为 $C_A^k(n) \le cn + \frac{1}{n}(\sum_{1\le i<k} C_A^{k-i}(n-i) + \sum_{k<i\le n} C_A^k(i-1))$

令 $R(n) = \max_k(C_A^k(n)),$且不妨设等式在 $k = k_n$ 时成立，则 $R(n)$ 满足

$$R(n) \le cn + \frac{1}{n}(\sum_{1\le i<k} C_A^{k-i}(n-i) + \sum_{k<i\le n} C_A^k(i-1))$$

以下用第二数学归纳法证明 $R(n) \le 4cn$ 。取 $C \ge R(1),$

当 $n = 1$ 时，取 c≥C/4,则 $R(1) \le 4c$

当 $n = 2$ 时， $R(2) \le 2c + \frac{1}{2}R(1) \le 2c + \frac{1}{2}4c = 4c$ 成立。

对于一般的 n，设对所有小于 n 的自然数 $R(n) \le 4cn$ 成立，则

$$\begin{aligned}
R(n) &\le cn + \frac{1}{n}(R(n-1) + \cdots + R(n-k_n+1)) + (R(k_n) + R(k_n+1) + \cdots + R(n-1)) \\
&\le cn + \frac{4c}{n}((n-1) + \cdots + (n-k_n+1)) + (k_n + \cdots + (n-1)) \\
&\le cn + \frac{4c}{n}(\frac{(k_n-1)(2n-k_n)}{2} + \frac{(n-k_n)(k_n+n-1)}{2}) \\
&\le cn - \frac{4c}{n}(k_n^2 - (n+1)k_n - \frac{n^2-3n}{2}) \\
&\le cn - \frac{4c}{n}(-(\frac{n+1}{2})^2 - \frac{n^2-3n}{2}) \\
&\le cn + c\frac{3n^2-4n+1}{n} \\
&\le cn + c(3n-3) \\
&\le 4cn
\end{aligned}$$

得证。

证明：（1）当 $r = 7$ 时，假设数组 A 中元素互不相同。由于每个具有 7 个元素的数组的中间值 u 是该数组中的第 4 小元素，因此数组中至少有 4 个元素不大于 u，$\lfloor n/7 \rfloor$ 个中间值中

至少有$\lceil \lfloor n/7 \rfloor /2 \rceil$个不大于这些中间值的中间值 v。因此，在数组 A 中至少有

$$4*\lceil \lfloor n/7 \rfloor /2 \rceil \geq 2\lfloor n/7 \rfloor$$

个元素不大于 v。换句话说，A 中至多有

$$n-2\lfloor n/7 \rfloor = n-2(n/7-e/7) \leq \frac{5}{7}n+\frac{12}{7}, (0 \leq e \leq 6)$$

个元素大于 v。同理，至多有$\frac{5}{7}n+\frac{12}{7}$个元素小于 v。这样，以 v 为划分元素，产生的新数组至多有$\frac{5}{7}n+\frac{12}{7}$个元素。当$n \geq 24$时，$\frac{5}{7}n+\frac{12}{7} \leq \frac{11}{14}n$。

另一方面，在整个执行过程中，递归调用 Select 函数一次，涉及规模为$n/7$，而下一次循环 Loop 涉及的数组规模为$\frac{11}{14}n$。程序中其他执行步的时间复杂度至多是 n 的倍数cn，用$T(n)$表示算法在数组长度为 n 的时间复杂度，则当$n \geq 24$时，有递归关系

$$T(n) \leq T(\frac{1}{7}n)+T(\frac{11}{14}n)+cn$$

用数学归纳法可以证明，$T(n) \leq 14cn$。所以时间复杂度是$O(n)$。

（2）当$r=3$时，使用上述方法进行分析，可知在进行划分后数组 A 中有至多$\frac{2}{3}n+\frac{2}{3} \leq \frac{5}{6}n$个元素。而递归关系为$T(n) \leq T(\frac{1}{3}n)+T(\frac{5}{6}n)+cn$。若通过归纳法证明出有$T(n) \leq kcn$的形式，用数学归纳法可以证明，$T(n) \leq 28cn$。所以时间复杂度是$O(n)$。

归并排序的 C++语言描述

```
#include<iostream.h>
template<class T>void MergeSort(T a[],int left,int right);
template<class T>void Merge(T c[],T d[], int l,int m,int r);
template<class T>void Copy(T a[],T b[],int l,int r);
void main()
{
        int const n(5);
        int a[n];
        cout<<"Input "<<n<<"numbers please:";
        for(int i=0;i<n;i++)
                cin>>a[i];
        //for(int j=0;j<n;j++)
                //b[j]=a[j];
        MergeSort(a,0,n-1);
        cout<<"The sorted array is"<<endl;
        for(i=0;i<n;i++)
                cout<<a[i];
```

```
	cout<<endl;
}

template<class T>
void MergeSort(T a[],int left,int right) //
{
	if(left<right)
	{
		int i=(left+right)/2;
		  T *b=new T[];
		MergeSort(a,left,i);
		MergeSort(a,i+1,right);
		Merge(a,b,left,i,right);
		Copy(a,b,left,right);
}

template<class T>
void Merge(T c[],T d[],int l,int m,int r)
{
	int i=l;
	int j=m+1;
	int k=l;
	while((i<=m)&&(j<=r))
	{
		if(c[i]<=c[j])d[k++]=c[i++];
		else d[k++]=c[j++];
	}
	if(i>m)
	{
		for(int q=j;q<=r;q++)
			d[k++]=c[q];
	}
	else
		for(int q=i;q<=m;q++)
			d[k++]=c[q];
}

template<class T>
void Copy(T a[],T b[], int l,int r)
{
	for(int i=l;i<=r;i++)
		a[i]=b[i];
}
```

快速排序的 C++语言描述

```
#include<iostream.h>
template<class T>void QuickSort(T a[],int p,int r);
template<class T>int Partition(T a[],int p,int r);
void main()
{
	int const n(5);
	int a[n];
	cout<<"Input "<<n<<"numbers please:";
	for(int i=0;i<n;i++)
		cin>>a[i];
	QuickSort(a,0,n-1);
	cout<<"The sorted array is"<<endl;
	for(i=0;i<n;i++)
		cout<<a[i]<<" ";
	cout<<endl;
}

template<class T>
void QuickSort(T a[],int p,int r)
{
	if(p<r)
	{
		int q=Partition(a,p,r);
		QuickSort(a,p,q-1);
		QuickSort(a,q+1,r);
	}
}

template<class T>
int Partition(T a[],int p,int r)
{
	int i=p,j=r+1;
	T x=a[p];
	while(true)
	{
		while(a[++i]<x);
		while(a[--j]>x);
		if(i>=j)break;
		Swap(a[i],a[j]);
	}
	a[p]=a[j];
	a[j]=x;
	return j;
```

```
}

template<class T>
inline void Swap(T &s,T &t)
{
        T temp=s;
        s=t;
        t=temp;
}
```

第四章作业 部分参考答案

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 $t_i, 1 \le i \le n$。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）。

策略：

对 $t_i \ 1 \le i \le n$ 进行排序，$t_{i_1} \le t_{i_2} \le \cdots \le t_{i_n}$，然后按照递增顺序依次服务 $i_1, i_2, ..., i_n$ 即可。

解析：设得到服务的顾客的顺序为 $j_1, j_2, ..., j_n$，则总等待时间为 $T = (n-1)t_{j_1} + (n-2)t_{j_2} + \cdots + 2t_{j_{n-2}} + t_{j_{n-1}}$，则在总等待时间 T 中 t_{j1} 的权重最大，t_{jn} 的权重最小。故让所需时间少的顾客先得到服务可以减少总等待时间。

证明：设 $t_{i_1} \le t_{i_2} \le \cdots \le t_{i_n}$，，下证明当按照不减顺序依次服务时，为最优策略。

记按照 $i_1 i_2 \cdots i_n$ 次序服务时，等待时间为 T，下证明任意互换两者的次序，T 都不减。即假设互换 $i, j \ (i < j)$ 两位顾客的次序，互换后等待总时间为 $\tilde{T}$，则有 $\tilde{T} \ge T.$

由于

$$T = (n-1)t_{i_1} + (n-2)t_{i_2} + \cdots + 2t_{i_{n-2}} + t_{i_{n-1}},$$

$$T = (n-1)t_{i_1} + (n-2)t_{i_2} + \cdots + (n-i)t_{i_i} + \cdots + (n-j)t_{i_j} + \cdots + 2t_{i_{n-2}} + t_{i_{n-1}},$$

$$\tilde{T} = (n-1)t_{i_1} + (n-2)t_{i_2} + \cdots + (n-i)t_{i_j} + \cdots + (n-j)t_{i_i} + \cdots + 2t_{i_{n-2}} + t_{i_{n-1}},$$

则有

$$\tilde{T} - T = (j-i)(t_{i_j} - t_{i_i}) \ge 0.$$

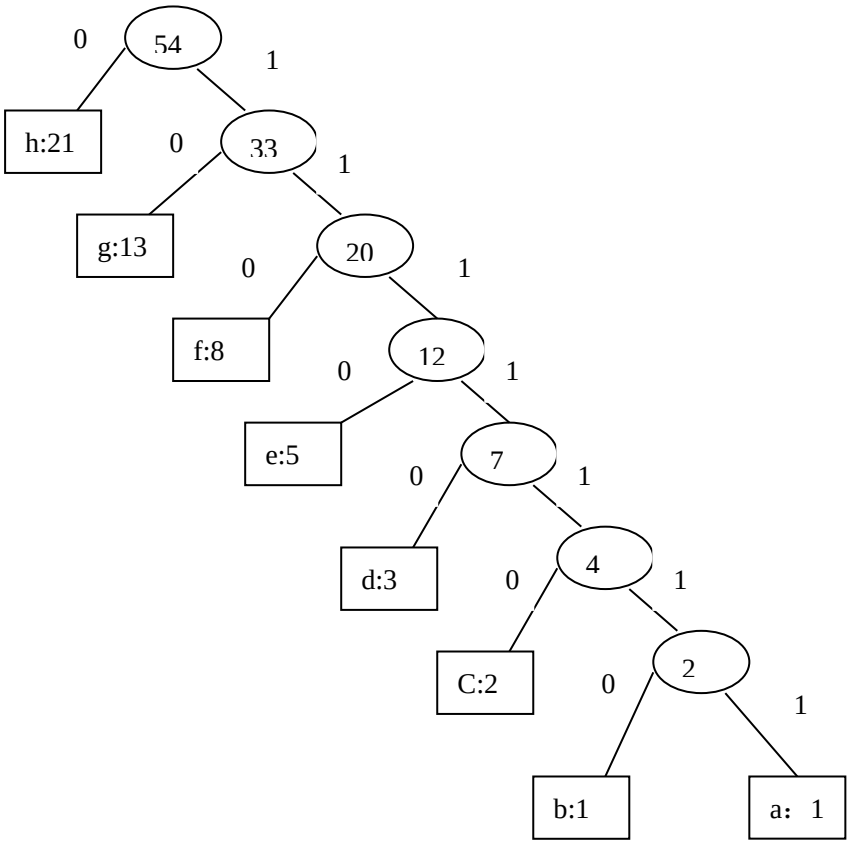
同理可证其它次序，都可以由 $i_1 i_2 \cdots i_n$ 经过有限次两两调换顺序后得到，而每次交换，总时间不减，从而 $i_1 i_2 \cdots i_n$ 为最优策略。

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：$F_0 = 1, F_1 = 1, F_n = F_{n-2} + F_{n-1}$ *if* $n > 1$

解：前 8 个数为 a， b， c， d， e， f， g， h

1， 1， 2， 3， 5， 8， 13， 21

Huffman 哈夫曼编码树为：



所以 a 的编码为：1111111

b 的编码为：1111110

c 的编码为：111110

d 的编码为：11110

e 的编码为：1110
f 的编码为：110
g 的编码为：10
h 的编码为：0

推广到 n 个字符：

第 1 个字符： n-1 个 1， $\underbrace{11\cdots1}_{n-1}$

第 2 个字符： n-2 个 1，1 个 0， $\underbrace{11\cdots10}_{n-2}$

第 3 个字符： n-3 个 1，1 个 0， $\underbrace{11\cdots10}_{n-3}$

……

第 n-1 个字符：1 个 1 ，1 个 0， 10

第 n 个字符：1 个 0 ， 0

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 $\sum_{p_i \in Q} a_i \le L$。

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得到使 $\sum_{p_i \in Q} a_i / L$ 取最大值的子集合。

1）证明：不妨设 $a_1 \le a_2 \le \ldots \le a_n$，若该贪心策略构造的子集合 Q 为 $\{a_1, a_2, \cdots, a_s\}$，

则 s 满足 $\sum_{i=1}^{s} a_i \le L$、$\sum_{i=1}^{s} a_s + a_{s+1} > L$。

要证明能找到最大子集，只需说明 s 为可包含的最多程序段数即可。

即证不存在多于 s 个的程序集合 $\tilde{Q} = \{a_{i_1}, a_{i_2}, \cdots, a_{i_k}\}, (k > s)$，使得 $\sum_{p_i \in \tilde{Q}} a_i \le L$。

反证法，假设存在多于 s 个的程序集 $\tilde{Q} = \{a_{i_1}, a_{i_2}, \cdots, a_{i_k}\}, (k > s)$，满足 $\sum_{j=1}^{k} a_{i_j} \le L$。

因为$a_1 \le a_2 \le ... \le a_n$非降序排列，则$a_1 + a_2 + \cdots a_s + \cdots + a_k \le a_{i_1} + a_{i_2} + \cdots + a_{i_k} \le L$。

因为$k > s$且为整数，则其前 s+1 项满足$a_1 + a_2 + \cdots a_s + a_{s+1} \le L$。

这与贪心策略构造的子集和Q中 s 满足的$\sum_{i=1}^{s} a_s + a_{s+1} > L$矛盾。故假设不成立，得证。

2）磁带的利用率为$\sum_{p_i \in Q} a_i / L$;（甚至最小可为 0，此时任意$a_i > L$或者$\sum_{p_i \in Q} a_i << L$）

3)按照 1）的策略可以使磁带上的程序数量最多，但程序的总长度不一定是最大的，假设$\{a_1, a_2, \cdots, a_i\}$为 Q 的最大子集，但是若用$a_{i+1}$代替$a_i$，仍满足$\sum_{k=1}^{i-1} a_k + a_{i+1} < L$，则$\{a_1, a_2, \cdots, a_{i-1}, a_{i+1}\}$为总长度更优子集。

4.同学们的几种不同答案

构造哈夫曼树思想，将所有的节点放到一个队列中，用一个节点替换两个频率最低的节点，新节点的频率就是这两个节点的频率之和。这样，新节点就是两个被替换节点的父节点了。如此循环，直到队列中只剩一个节点（树根）。

答案 1）

伪代码：

```
typedef struct
{
    unsigned int weight;
    unsigned int parent, lchild, rchild;
}HTNode, * HuffmanTree;

typedef char ** HuffmanCode;

proc HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
  if n<=1 then return
  HuffmanTree p;
   integer s1, s2, i, m, start, c, f;
   char *cd;
   m := 2 * n - 1;
   HT[0].weight := 1000000;
   p := HT+1;
      for i to n do
        (*p).weight := *w;
        (*p).parent := (*p).lchild := (*p).rchild := 0;
        ++p; ++w;
      end{for}
```

```
        for i to m do
          (*p).weight = (*p).parent = (*p).lchild = (*p).rchild =
0;
          ++p;
        end{for}
        for i from n+1 to m do
           Select(HT, i-1, s1, s2);
           HT[s1].parent := i;
           HT[s2].parent := i;
           HT[i].lchild := s1;
           HT[i].rchild := s2;
           HT[i].weight := HT[s1].weight + HT[s2].weight;
       end{for}
       cd[n-1] = '\0';           //编码结束符
        for i to n do
           start := n - 1;         //编码结束符位置
           f := i;
           c := i;
              for f from HT[f].parent to f=0 do
                  if HT[f].lchild = c
                      cd[--start] := '0';
                    else
                        cd[--start] := '1';
                      end{else}
                   end{if}
                end{for}
          end{for}
end{HuffmanCoding}
```

源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
    using namespace std;
#define infinite 50000
    //定义 Huffman 树和 Huffman 编码的存储表示
    typedef struct
    {
        unsigned int weight; //字符的频数
        unsigned int parent, lchild, rchild;   //双亲结点，左孩子，右孩
子
    }HTNode, * HuffmanTree;
    typedef char ** HuffmanCode;
    void Select(HuffmanTree HT, int n, int &s1, int &s2);
```

```
    void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n);

    void main()
    {
        int i, n, *w;
        cout << "enter the size of the code: ";
cin >> n;
        cout << endl;
        w = (int*) malloc( (n+1) * sizeof(int));
        cout << "enter the weight of the code:"<<endl;
        for(i=1; i<=n; i++)              //逐个输入每个字符的出现的频数，并
用空格隔开
            cin>>w[i];

        HuffmanTree HT;
        HuffmanCode HC;

        HuffmanCoding(HT, HC, w+1, n);

        cout<< "the huffmancode is: "<< endl;
        for(i=1; i<=n; i++)
        {
            cout<<w[i]<<": ";
            cout<<HC[i]<<"      ";
        }
        cout<<endl;
        system("pause");
    }

    //在 HuffmanTree 中 HT[1…n]选择 parent 为且 weight 最小的两个结点，其序
号分别为 s1 和 s2
    void Select(HuffmanTree HT, int n, int &s1, int &s2)
    {
        int i;
        s1 = s2 =0;
        for(i=1; i<=n; i++)
        {
            if(HT[i].parent == 0 && HT[s2].weight > HT[i].weight)
                s2 = i;
            else
                if (HT[i].parent == 0 && HT[s1].weight > HT[i].weight)
                    s1 = i;
        }
    }
```

```
//构造 Huffman 树 HT，并求出 n 个字符的 Huffman 编码 HC
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{

    if (n <= 1) return;
    HuffmanTree p;
    int s1, s2, i, m, start, c, f;
    char *cd;
    m = 2 * n - 1;
    HT = (HuffmanTree)malloc((m+1) * sizeof(HTNode));
HT[0].weight = infinite;      //将号单元置一个较大的数
    for(p=HT+1, i=1; i<=n; ++i, ++p, ++w)
    {
        (*p).weight = *w;             //将 n 个字符的频数送到
HT.weight[1…n]
        (*p).parent = (*p).lchild = (*p).rchild = 0;      //双亲孩子
初始化为
    }
    for(; i<=m; ++i, ++p)
        (*p).weight = (*p).parent = (*p).lchild = (*p).rchild = 0; //
将 HuffmanTree 结点权值 HT.weight[n+1…m+1]（频数）及双亲孩子初始化为

    for(i= n+1; i<=m; ++i)      //根据 Huffman 编码原理进行构建 Huffman
树
    {
        Select(HT, i-1, s1, s2);
        HT[s1].parent = i;
        HT[s2].parent = i;
        HT[i].lchild = s1;
        HT[i].rchild = s2;
        HT[i].weight = HT[s1].weight + HT[s2].weight;
    }
    //从叶子到根逆向求每个字符的 Huffman 编码
    HC = (HuffmanCode) malloc ((n+1) * sizeof (char *)); //分配 n 个
字符编码的头指针向量，注号单元未用
    cd = (char *) malloc (n * sizeof (char));                    //分配求编
码的工作空间
    cd[n-1] = '\0';          //编码结束符
    for(i=1; i<=n; ++i)   //逐个字符求 Huffman 编码
    {
        start = n - 1;        //编码结束符位置
        for(c=i, f=HT[i].parent; f!=0; c=f, f=HT[f].parent) //从叶
子到根逆向求编码
        {
```

```
            if(HT[f].lchild == c)
                cd[--start] = '0';
            else
                cd[--start] = '1';
        }
        HC[i] = (char *)malloc((n-start) * sizeof(char)); //为第 i 个
字符编码分配空间
        strcpy_s(HC[i],sizeof(&cd[start]),&cd[start]);
    }
    free(cd);
}
```

答案 2）

c 语言实现:huffman 编码解码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 100
#define M 2*N-1
typedef char * HuffmanCode[2*M];//haffman 编码
typedef struct
{
int weight;//权值
int parent;//父节节点
int LChild;//左子节点
int RChild;//右子节点
}HTNode,Huffman[M+1];//huffman 树
typedef struct Node
{
int weight; //叶子结点的权值
char c; //叶子结点
int num; //叶子结点的二进制码的长度
}WNode,WeightNode[N];
/***产生叶子结点的字符和权值***/
void CreateWeight(char ch[],int *s,WeightNode CW,int *p)
{
int i,j,k;
int tag;
*p=0;//叶子节点个数
//统计字符出现个数,放入 CW
for(i=0;ch[i]!='\0';i++)
{
tag=1;
for(j=0;j<i;j++)
```

```
if(ch[j]==ch[i])
{
tag=0;
break;
}
if(tag)
{
CW[++*p].c=ch[i];
CW[*p].weight=1;
for(k=i+1;ch[k]!='\0';k++)
if(ch[i]==ch[k])
CW[*p].weight++;//权值累加
}
}
*s=i;//字符串长度
}
/********创建 HuffmanTree********/
void CreateHuffmanTree(Huffman ht,WeightNode w,int n)
{
int i,j;
int s1,s2;
//初始化哈夫曼树
for(i=1;i<=n;i++)
{
ht[i].weight =w[i].weight;
ht[i].parent=0;
ht[i].LChild=0;
ht[i].RChild=0;
}
for(i=n+1;i<=2*n-1;i++)
{
ht[i].weight=0;
ht[i].parent=0;
ht[i].LChild=0;
ht[i].RChild=0;
}
for(i=n+1;i<=2*n-1;i++)
{
for(j=1;j<=i-1;j++)
if(!ht[j].parent)
break;
s1=j; //找到第一个双亲不为零的结点
for(;j<=i-1;j++)
if(!ht[j].parent)
```

```
    s1=ht[s1].weight>ht[j].weight?j:s1;
    ht[s1].parent=i;
    ht[i].LChild=s1;
    for(j=1;j<=i-1;j++)
    if(!ht[j].parent)
    break;
    s2=j; //找到第二个双亲不为零的结点
    for(;j<=i-1;j++)
    if(!ht[j].parent)
    s2=ht[s2].weight>ht[j].weight?j:s2;
    ht[s2].parent=i;
    ht[i].RChild=s2;
    ht[i].weight=ht[s1].weight+ht[s2].weight;//权值累加
    }
    }
    /***********叶子结点的编码***********/
    void CrtHuffmanNodeCode(Huffman ht,char ch[],HuffmanCode h,WeightNode weight,int m,int
n)
    {
    int i,c,p,start;
    char *cd;
    cd=(char *)malloc(n*sizeof(char));
    cd[n-1]='\0';//末尾置 0
    for(i=1;i<=n;i++)
    {
    start=n-1; //cd 串每次从末尾开始
    c=i;
    p=ht[i].parent;//p 在 n+1 至 2n-1
    while(p) //沿父亲方向遍历,直到为 0
    {
    start--;//依次向前置值
    if(ht[p].LChild==c)//与左子相同,置 0
    cd[start]='0';
    else //否则置 1
    cd[start]='1';
    c=p;
    p=ht[p].parent;
    }
    weight[i].num=n-start; //二进制码的长度(包含末尾 0)
    h[i]=(char *)malloc((n-start)*sizeof(char));
    strcpy(h[i],&cd[start]);//将二进制字符串拷贝到指针数组 h 中
    }
    free(cd);//释放 cd 内存
    system("pause");
```

```
    }
    /*********所有字符的编码*********/
    void CrtHuffmanCode(char ch[],HuffmanCode h,HuffmanCode hc,WeightNode weight,int n,int
m)
    {
    int i,k;
    for(i=0;i<m;i++)
    {
    for(k=1;k<=n;k++) /*从 weight[k].c 中查找与 ch[i]相等的下标 K*/
    if(ch[i]==weight[k].c)
    break;
    hc[i]=(char *)malloc((weight[k].num)*sizeof(char));
    strcpy(hc[i],h[k]); //拷贝二进制编码
    }
    }
    /*****解码*****/
    void TrsHuffmanTree(Huffman ht,WeightNode w,HuffmanCode hc,int n,int m)
    {
    int i=0,j,p;
    printf("***StringInformation***\n");
    while(i<m)
    {
    p=2*n-1;//从父亲节点向下遍历直到叶子节点
    for(j=0;hc[i][j]!='\0';j++)
    {
    if(hc[i][j]=='0')
    p=ht[p].LChild;
    else
    p=ht[p].RChild;
    }
    printf("%c",w[p].c); /*打印原信息*/
    i++;
    }
    }
    /*****释放 huffman 编码内存*****/
    void FreeHuffmanCode(HuffmanCode h,HuffmanCode hc,int n,int m)
    {
    int i;
    for(i=1;i<=n;i++)//释放叶子结点的编码
    free(h[i]);
    for(i=0;i<m;i++) //释放所有结点的编码
    free(hc[i]);
    }
    void main()
```

```
    {
    int i,n=0; /*n 为叶子结点的个数*/
    int m=0; /*m 为字符串 ch[]的长度*/
    char ch[N]; /*ch[N]存放输入的字符串*/
    Huffman ht; /*Huffman 二叉数 */
    HuffmanCode h,hc; /*h 存放叶子结点的编码，hc 存放所有结点的编码*/
    WeightNode weight; /*存放叶子结点的信息*/
    printf("\t***HuffmanCoding***\n");
    printf("please input information :");
    gets(ch); /*输入字符串*/
    CreateWeight(ch,&m,weight,&n); /*产生叶子结点信息，m 为字符串 ch[]的长度*/
    printf("***WeightInformation***\n Node ");
    for(i=1;i<=n;i++) /*输出叶子结点的字符与权值*/
    printf("%c ",weight[i].c);
    printf("\nWeight ");
    for(i=1;i<=n;i++)
    printf("%d ",weight[i].weight);
    CreateHuffmanTree(ht,weight,n); /*产生 Huffman 树*/
    printf("\n***HuffamnTreeInformation***\n");
    printf("\ti\tweight\tparent\tLChild\tRChild\n");
    for(i=1;i<=2*n-1;i++) /*打印 Huffman 树的信息*/

printf("\t%d\t%d\t%d\t%d\t%d\n",i,ht[i].weight,ht[i].parent,ht[i].LChild,ht[i].RChild);
    CrtHuffmanNodeCode(ht,ch,h,weight,m,n); /*叶子结点的编码*/
    printf(" ***NodeCode***\n"); /*打印叶子结点的编码*/
    for(i=1;i<=n;i++)
    {
    printf("\t%c:",weight[i].c);
    printf("%s\n",h[i]);
    }
    CrtHuffmanCode(ch,h,hc,weight,n,m); /*所有字符的编码*/
    printf("***StringCode***\n"); /*打印字符串的编码*/
    for(i=0;i<m;i++)
    printf("%s",hc[i]);
    system("pause");
    TrsHuffmanTree(ht,weight,hc,n,m); /*解码*/
    FreeHuffmanCode(h,hc,n,m);
    system("pause");
    }
```

答案 3）

(1) 伪代码：

Proc HuffmanCode(A)

//A[1…n]是待编码的数组，n 为数组长度

local h; //最小化堆，内含元素为结点类型，堆初始为空

integer i;
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
for i from 1 to to n do //将数组 A 中的所有元素插入堆
    Insert(h, A(i));
end{for};
while h 元素个数大于 1 then
    p = DeleteMin(h); //移除最小的两个结点
    q = DeleteMin(h);
    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。
                                                      //左儿子为 p 和 q 值较小的，右儿子为较
                                                  //大的
    Insert(h, r); //将 r 插入堆中
end{while};
p = DeleteMin(h); //取出最后一个结点，此节点即为 huffman 树的根节点。
end{HuffmanCode}

(2)java code：

树节点 TreeNode:

```
package graduate;
public class TreeNode{
    private TreeNode left;
    private TreeNode right;
    private int value;
    private int code;

    public TreeNode() {
        setLeft(null);
        setRight(null);
        setValue(0);
        setCode(-1);
    }

    public TreeNode getLeft(){
        return left;
    }

    public void setLeft(TreeNode left){
        this.left = left;
    }

    public TreeNode getRight(){
        return right;
    }

    public void setRight(TreeNode right){
```

```
        this.right = right;
    }

    public int getValue(){
        return value;
    }

    public void setValue(int value){
        this.value = value;
    }

    public int getCode(){
        return code;
    }

    public void setCode(int code){
        this.code = code;
    }
}
```

堆节点 HeapNode:

```
package graduate;
public class HeapNode{
    private int iData;

    public HeapNode(int key) {
        iData = key;
    }

    public void setKey(int id) {
        iData = id;
    }

    public int getKey() {
        return iData;
    }
}
```

堆:

```
package graduate;
public class Heap{
    private HeapNode[] heapArray; // 堆容器
    private int maxSize; // 堆得最大大小
    private int currentSize; // 堆大小
```

```
public Heap(int _maxSize) {
    maxSize = _maxSize;
    heapArray = new HeapNode[maxSize];
    currentSize = 0;
}

public void filterDown(int start, int endOfHeap) {
    int i = start;
    int j = 2 * i + 1; // j是i的左子女位置
    HeapNode temp = heapArray[i];

    while (j <= endOfHeap) { // 检查是否到最后位置
        if (j < endOfHeap // 让j指向两子女中的小者
                && heapArray[j].getKey() > heapArray[j + 1].getKey()) {
            j++;
        }
        if (temp.getKey() <= heapArray[j].getKey()) { // 小则不做调整
            break;
        }
        else { // 否则小者上移，i，j下降
            heapArray[i] = heapArray[j];
            i = j;
            j = 2 * j + 1;
        }
    }
    heapArray[i] = temp;
}

public void filterUp(int start) {
    int j = start;
    int i = (j - 1) / 2;
    HeapNode temp = heapArray[j];

    while (j > 0) { // 沿双亲结点路径向上直达根节点
        if (heapArray[i].getKey() <= temp.getKey()) {// 双亲结点值小，不调整
            break;
        }
        else {// 双亲结点值大，调整
            heapArray[j] = heapArray[i];
            j = i;
            i = (i - 1) / 2;
        }
        heapArray[j] = temp; // 回送
```

```
        }
    }

    public boolean insert(HeapNode newNode) {
        if (isFull()) {
            return false;
        }
        else {
            heapArray[currentSize] = newNode;
            filterUp(currentSize);
            currentSize++;
        }
        return true;
    }

    public HeapNode removeMin() {
        if (isEmpty()) {
            return null;
        }
        HeapNode root = heapArray[0];
        heapArray[0] = heapArray[currentSize - 1];
        currentSize--;
        filterDown(0, currentSize - 1);
        return root;
    }

    public boolean isEmpty() {
        return currentSize == 0;
    }

    public boolean isFull() {
        return currentSize == maxSize;
    }

    public int getCurrentSize(){
        return currentSize;
    }

    public void makeEmpty() {
        currentSize = 0;
    }
}
```

Huffman 编码方法：

```
package graduate;
import java.util.HashMap;
import java.util.Iterator;

public class Huffman{
    public Heap minHeap;
    public HashMap<HeapNode, TreeNode> nodeMap;
    public HashMap<Integer, String> map;

    public Huffman(){
        minHeap = new Heap(20);
        nodeMap = new HashMap<HeapNode, TreeNode>();
        map = new HashMap<Integer, String>();
    }

    public void huffmanCode(int[] array){
        for (int i = 0; i < array.length; i++){
            HeapNode heapNode = new HeapNode(array[i]);
            TreeNode treeNode = new TreeNode();
            treeNode.setValue(array[i]);
            nodeMap.put(heapNode, treeNode);
            minHeap.insert(heapNode);
        }
        while (minHeap.getCurrentSize() > 1) {
            HeapNode n1 = minHeap.removeMin();
            TreeNode tn1 = nodeMap.get(n1);
            HeapNode n2 = minHeap.removeMin();
            TreeNode tn2 = nodeMap.get(n2);

            if (tn1.getValue() < tn2.getValue()) {
                tn1.setCode(0);
                tn2.setCode(1);
            }
            else {
                tn2.setCode(0);
                tn1.setCode(1);
            }

            HeapNode newHeapNode = new HeapNode(n1.getKey()+n2.getKey());
            TreeNode newTreeNode = new TreeNode();
            newTreeNode.setValue(tn1.getValue()+tn2.getValue());
            newTreeNode.setLeft(tn1);
            newTreeNode.setRight(tn2);
            minHeap.insert(newHeapNode);
```

```
            nodeMap.put(newHeapNode, newTreeNode);
        }

        HeapNode root = minHeap.removeMin();
        TreeNode treeRoot = nodeMap.get(root);
        leftFirst(treeRoot, "");
    }

    public void leftFirst(TreeNode node, String code){
        if (node.getCode() != -1){
            code += new Integer(node.getCode()).toString();
        }
        if (node.getLeft() == null){
            map.put(new Integer(node.getValue()), code);
            return;
        }
        leftFirst(node.getLeft(), code);
        leftFirst(node.getRight(), code);
    }

    public static void main(String args[]){
        Huffman h = new Huffman();
        int[] array = {9, 5, 16, 12, 45, 13};
        h.huffmanCode(array);

        Iterator<Integer> i = h.map.keySet().iterator();
        while (i.hasNext()){
            Integer num = i.next();
            System.out.println(num + " : " + h.map.get(num));
        }
    }
}
```

答案 4）伪代码：

```
void CreateHuffmanTree(HuffmanTree T)
  {//构造哈夫曼树，T[m-1]为其根结点
   int i，p1，p2；
   InitHuffmanTree(T)； //将 T 初始化
   InputWeight(T)； //输入叶子权值至 T[0．．n-1]的 weight 域
   for(i=n；i<m；i++){//共进行 n-1 次合并，新结点依次存于 T[i]中
     SelectMin(T，i-1，&p1，&p2)；
     //在 T[0．．i-1]中选择两个权最小的根结点，其序号分别为 p1 和 p2
     T[p1].parent=T[p2].parent=i；
     TIi].1child=p1； //最小权的根结点是新结点的左孩子
     T[j].rchild=p2； //次小权的根结点是新结点的右孩子
```

```
        T[i].weight=T[p1].weight+T[p2].weight；
      } // end for
   }
```

源码：

```
BinaryTree<int> compute::HuffmanTree(UINT a[],short n1,BYTE q[])
{// Generate Huffman tree with weights a[1:n].
    // create an array of single node trees
    Huffman<int> *w = new Huffman<int> [n1+1];
    BinaryTree<int> z, zero;
    for (short i = 1; i <=n1; i++)
    {
          z.MakeTree(i, zero, zero);
          w[i].weight = a[q[i]];
          w[i].tree = z;
    }

    // make array into a min heap
    MinHeap<Huffman<int> > H(1);
    H.Initialize(w,n1,n1);

    // repeatedly combine trees from heap
    Huffman<int> x, y;
    for (int i = 1; i < n1; i++) {
        H.DeleteMin(x);
        H.DeleteMin(y);
        z.MakeTree(0, x.tree, y.tree);
        x.weight += y.weight; x.tree = z;
        H.Insert(x);
        }

    H.DeleteMin(x); // final tree
    H.Deactivate();
    delete [] w;
    return x.tree;
}
template<class T>
class   Huffman {
    //friend BinaryTree<int> HuffmanTree(T [], int);
    public:
        operator T () const {return weight;}
    public:
        BinaryTree<int> tree;
        T weight;
};
```

```
// file binary.h

#ifndef BinaryTree_
#define BinaryTree_


#include<iostream>
#include "lqueue.h"
#include "btnode2.h"
#include "xcept.h"

class compute;
template<class T>
class BinaryTree {
        friend class compute;
    public:
        BinaryTree() {root = 0;w = 0;};
        ~BinaryTree(){};
        bool IsEmpty() const
           {return ((root) ? false : true);}
        bool Root(T& x) const;
        void MakeTree(const T& element,
                BinaryTree<T>& left, BinaryTree<T>& right);
        BinaryTree<T> BreakTree(BYTE& element,
                              char ch,BYTE warry[]);
        void Delete() {PostOrder(Free, root); root = 0;}
        int Height() const {return Height(root);}
         void PreCom(char b[][30],BinaryTreeNode<T> *t,char *w,int i);
         void precompare(char b[][30])
         {
                char *w=new char[30];
                PreCom(b,root,w,0);
                delete []w;
         }
    private:
        BinaryTreeNode<T> *root;    // pointer to root
         int w;
};

template<class T>
bool BinaryTree<T>::Root(T& x) const
{// Return root data in x.
```

```
  // Return false if no root.
     if (root) {x = root->data;
                        return true;}
     else return false;    // no root
}

template<class T>
void BinaryTree<T>::MakeTree(const T& element,
                  BinaryTree<T>& left, BinaryTree<T>& right)
{// Combine left, right, and element to make new tree.
  // left, right, and this must be different trees.
     // create combined tree
     root = new BinaryTreeNode<T>
                        (element, left.root, right.root);

     // deny access from trees left and right
     left.root = right.root = 0;
}

template<class T>
BinaryTree<T> BinaryTree<T>::BreakTree(BYTE& element,
            char ch,BYTE warry[])
{// left, right, and this must be different trees.
     // check if empty
     if (!root) throw BadInput(); // tree empty

     // break the tree
     BinaryTree<T> btree;
     element = 0;
     if(root->LeftChild==NULL)
     {
            element =warry[root->data];
            btree.root=NULL;
            return btree;
     }
     if(ch=='0')
     {
            btree.root = root->LeftChild;
            return btree;
     }
     else
     {
            btree.root = root->RightChild;
            return btree;
```

```
    }

}


template <class T>
void BinaryTree<T>::PreCom(char b[][30],BinaryTreeNode<T> *t,char *w,int i)
{
      if(t){
            if(t->LeftChild==NULL)
            {
                  w[i]='\0';
                  strcpy(b[t->data-1],w);
            }
            else
            {
            w[i]='0';
            PreCom(b,t->LeftChild,w,i+1);
            w[i]='1';
            PreCom(b,t->RightChild,w,i+1);
            }
      }
}

#endif

// file MinHeap.h
#ifndef MinHeap_
#define MinHeap_

#include <stdlib.h>
#include <iostream>
#include "xcept.h"

template<class T>
class MinHeap {
    public:
        MinHeap(int MinHeapSize = 10);
        ~MinHeap() {delete [] heap;}
        int Size() const {return CurrentSize;}
        T Min() {if (CurrentSize == 0)
                        throw OutOfBounds();
                    return heap[1];}
        MinHeap<T>& Insert(const T& x);
```

```
        MinHeap<T>& DeleteMin(T& x);
        void Initialize(T a[], int size, int ArraySize);
        void Deactivate() {heap = 0;}
        void Output() const;
    private:
        int CurrentSize, MaxSize;
        T *heap; // element array
};

template<class T>
MinHeap<T>::MinHeap(int MinHeapSize)
{// Min heap constructor.
    MaxSize = MinHeapSize;
    heap = new T[MaxSize+1];
    CurrentSize = 0;
}

template<class T>
MinHeap<T>& MinHeap<T>::Insert(const T& x)
{// Insert x into the min heap.
    if (CurrentSize == MaxSize)
        throw NoMem(); // no space

    // find place for x
    // i starts at new leaf and moves up tree
    int i = ++CurrentSize;
    while (i != 1 && x < heap[i/2]) {
        // cannot put x in heap[i]
        heap[i] = heap[i/2]; // move element down
        i /= 2; // move to parent
        }
    heap[i] = x;

    return *this;
}

template<class T>
MinHeap<T>& MinHeap<T>::DeleteMin(T& x)
{// Set x to min element and delete
  // min element from heap.
    // check if heap is empty
    if (CurrentSize == 0)
        throw OutOfBounds(); // empty
```

```
    x = heap[1]; // min element

    // restructure heap
    T y = heap[CurrentSize--]; // last element

    // find place for y starting at root
    int i = 1,    // current node of heap
        ci = 2; // child of i
    while (ci <= CurrentSize) {// find place to put y
        // heap[ci] should be smaller child of i
        if (ci < CurrentSize &&
              heap[ci] > heap[ci+1]) ci++;

        // can we put y in heap[i]?
        if (y <= heap[ci]) break;    // yes

        // no
        heap[i] = heap[ci]; // move child up
        i = ci;    // move down a level
        ci *= 2;
        }
    heap[i] = y;

    return *this;
}

template<class T>
void MinHeap<T>::Initialize(T a[], int size, int ArraySize)
{// Initialize min heap to array a.
    delete [] heap;
    heap = a;
    CurrentSize = size;
    MaxSize = ArraySize;

    // make into a min heap
    for (int i = CurrentSize/2; i >= 1; i--) {
        T y = heap[i]; // root of subtree

        // find place to put y
        int c = 2*i; // parent of c is target
                     // location for y
        while (c <= CurrentSize) {
            // make c point to smaller sibling
            if (c < CurrentSize &&
```

```
            heap[c] > heap[c+1]) c++;

        // can we put y in heap[c/2]?
        if (y <= heap[c]) break;    // yes

        // no
        heap[c/2] = heap[c]; // move heap[c] up
        c *= 2;                      // move c down a level
        }
    heap[c/2] = y;
    }
}

template<class T>
void MinHeap<T>::Output() const
{
    cout << "The " << CurrentSize
         << " elements are"<< endl;
    for (int i = 1; i <= CurrentSize; i++)
        cout << heap[i] << ' ';
    cout << endl;
}

#endif
```

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

1）已知一个简单算法如下：

```
int Maxsum(int n,int a,int& best i,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解最大子段和问题。分析算法的时间复杂度。

（提示：令 $b(j)=\max\limits_{1\le i\le j\le n}\sum\limits_{k=i}^{j}a_k, j=1,2,\cdots,n$ ）

解：1）分析按照第一章，列出步数统计表，计算可得 $O(n^2)$

2）分治算法：将所给的序列 a[1:n]分为两段 a [1:n/2]、a[n/2+1:n]，分别求出这两段的最大子段和，则 a[1:n]的最大子段和有三种可能：

①a[1:n]的最大子段和与 a[1:n/2]的最大子段和相同；

②a[1:n]的最大子段和与 a[n/2+1:n]的最大子段和相同；

③a[1:n]的最大子段和为两部分的字段和组成，即

$$\sum_{l=i}^{j}a_j=a_i+\cdots+a_{\left\lfloor\frac{n}{2}\right\rfloor}+a_{\left\lfloor\frac{n}{2}\right\rfloor+1}+\cdots+a_j;$$

```
intMaxSubSum ( int *a, int left , int right){
  int sum =0;
  if( left==right)
      sum = a[left] > 0? a[ left]:0 ;
  else
      {int center = ( left + right) /2;
       int leftsum =MaxSubSum ( a, left , center) ;
       int rightsum =MaxSubSum ( a, center +1, right) ;
       int s_1 =0;
       int left_sum =0;
       for ( int i = center ; i >= left; i--){
           left_sum + = a [ i ];
           if( left_sum > s1)
               s1 = left_sum;
```

```
        }
    int s2 =0;
    int right_sum =0;
    for ( int i = center +1; i <= right ; i++){
          right_sum + = a[ i];
          if( right_sum > s2)
             s2 = right_sum;
        }
    sum = s1 + s2;
    if ( sum < leftsum)
       sum = leftsum;
    if ( sum < rightsum)
      sum = rightsum;
    }
  return sum;
  }
 int MaxSum2 (int n){
   int a;
   returnMaxSubSum ( a, 1, n) ;
```

} 该算法所需的计算时间T(n)满足典型的分治算法递归分式
T(n)=2T(n/2)+0(n)，分治算法的时间复杂度为 O(nlogn)

3）设 $b(j)=\max\limits_{1\le i\le j}\{\sum\limits_{k=i}^{j}a_k\}$,则最大子段和为 $\max\limits_{1\le i\le n}\max\limits_{1\le j\le n}\sum\limits_{k=i}^{j}a_k=\max\limits_{1\le j\le n}\max\limits_{1\le i\le j}\sum\limits_{k=i}^{j}a_k=\max\limits_{1\le j\le n}b(j).$

$$b(j)=\max\{a_1+\cdots a_{j-1}+a_j,a_2+\cdots a_{j-1}+a_j,\cdots,a_{j-1}+a_j,a_j\}$$

最大子段和实际就是 $\max\{b(1),b(2),\cdots,b(n)\}$.

要说明最大子段和具有最优子结构性质，只要找到其前后步骤的迭代关系即可。

$$b(j)=\max_{1\le i\le j}\{\sum_{k=i}^{j}a_k\}=\max\{\underbrace{\{\sum_{k=i}^{j-1}a_k+a_j\}}_{1\le i\le j-1},a_j\}=\max\{\max_{1\le i\le j-1}\{\sum_{k=i}^{j-1}a_k\}+a_j,a_j\}=\max\{b(j-1)+a_j,a_j\}$$

若 $b(j-1)>0$，$b(j)=b(j-1)+a_j$；

若 $b(j-1)\le 0$，$b(j)=a_j$。

因此，计算$b(j)$的动态规划的公式为：$b(j)=\max\{b(j-1)+a_j,a_j\},1\le j\le n.$

```
intMaxSum (int* a，int n )
{
    int sum = 0, b = 0，j=0;
    for( int i=1;i<=n;i++)
    {   if( b >0)
            b = b + a [i];
        else
            b = a [i];
        end{if}
        if( b > sum)
            sum = b;
            j=i;
        end{if}
    }
    return sum;
}
```

自行推导，答案：时间复杂度为 O（n）。

2.动态规划算法的时间复杂度为 O（n）（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的时间）。以下面的例子说明你的算法：

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

解：（思路一）在完成前 k 个作业时，设机器 A 工作了 x 时间，则机器 B 此时最小的工作时间是 x 的一个函数。

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

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

其中$F[k-1](x)+b_k$对应第 k 个作业由机器 B 来处理（完成 k-1 个作业时机器 A

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 工作时间是 x-a[k]，而 B 完成 k 阶段与完成 k-1 阶段用时相同为 $F[k-1](x-a_k)$）。

则完成前 k 个作业所需的时间为 $\max\{x, F[k](x)\}$

1）当处理第一个作业时，a[1]=2,b[1]=3;

机器 A 所花费时间的所有可能值范围：0 ≤x ≤a[0].

x<0 时，设 F[0][x]= ∞，则 max(x, ∞)= ∞；

0≤x<2 时，F[1][x]=3，则 Max(0,3)=3，

x≥2 时， F[1][x]= 0，则 Max(2,0)=2；

2）处理第二个作业时：x 的取值范围是：0 <= x <= (a[0] + a[1])，

当 x<0 时，记 F[2][x] = ∞；以此类推下去

（思路二）假定 n 个作业的集合为 $S_n=\{1,2,\cdots,n\}$。

设 J 为 S_n 的子集，若安排 J 中的作业在机器 A 上处理，其余作业在机器 B 上处理，此时所用时间为 $T(J)=\max\left(\sum_{j\in J}a_j,\sum_{j\in S\setminus J}b_j\right)$，

则双机处理作业问题相当于确定 S_n 的子集 J，使得安排是最省时的。即转化为求 J 使得 $\min_{J\subseteq S_n}\{T(J)\}$。若记 $S_{n-1}=\{1,2,\cdots,n-1\}$，则有如下递推关系：

$$\min_{I\subseteq S_n}\max\left(\sum_{j\in I}a_j,\sum_{j\in S\setminus I}b_j\right)=\min\left(\min_{J\subseteq S_{n-1}}\max\left(a_n+\sum_{j\in J}a_j,\sum_{j\in S\setminus J}b_j\right),\min_{J\subseteq S_{n-1}}\max\left(\sum_{j\in J}a_j,b_n+\sum_{j\in S\setminus J}b_j\right)\right)$$

（思路三）

此问题等价于求（$x_1,\cdots\cdots x_n$），使得它是下面的问题最优解。

$\min\max\{x_1a_1+\cdots\cdots x_na_n,\ (1-x_1)b_1+\cdots\cdots+(1-x_n)b_n\}$　$x_i=0$ 或 1，i=1~n

基于动态规划算法的思想，对每个任务 i，依次计算集合 $S^{(i)}$。其中每个集合中元素都是一个 3 元组（F_1,F_2,x）。这个 3 元组的每个分量定义为

F_1:处理机 A 的完成时间

F_2：处理机 B 的完成时间

x：任务分配变量。当 $x_i=1$ 时表示将任务 i 分配给处理机 A，当 $x_i=0$ 时表示分配给处理机 B。

初始时，$S^{(0)}=\{(0,0,0)\}$

令 F=按处理时间少的原则来分配任务的方案所需的完成时间。

例如，当（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)时，按处理时间少的原则分配任务的方案为（x_1,x_2,x_3,x_4,x_5,x_6）=（1,1,0,1,0,1）

因此，F=max{2+5+10+2,7+5}=19。

然后，依次考虑任务 i，i=1~n。在分配任务 i 时，只有 2 种情形，x_i=1 或 x_i=0。此时，令 $S^{(i)}=\{S^{(i-1)}+(a_i,0,2^i)\}U\{S^{(i-1)}+(0,b_i,0)\}$在做上述集合并集的计算时，遵循下面的原则：

①当（a, b, c），(d, e, f)$\in S^{(i)}$且 a=d,b<=e 时，仅保留（a,b,c）；

②仅当 max{a,b}<=F 时，(a,b,c)$\in S^{(i)}$

最后在 $S^{(n)}$中找出使 max{F_1,F_2}达到最小的元素，相应的 x 即为所求的最优解，其最优值为 max{F_1,F_2}。

当（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)时，按处理时间少的原则分配任务的方案为（x_1,x_2,x_3,x_4,x_5,x_6）=（1,1,0,1,0,1）

因此，F=max{2+5+10+2,7+5}=19。

S(0)={(0,0,0)};

S(1)={(2,0,2),(0,3,0)}

S(2)={(7,0,6),(5,3,4),(2,8,2),(0,11,0)}

S(3)={(14,0,14),(12,3,12),(9,8,10), (7,4,6), (5,7,4),(2,12,2),(0,15,0)}

S(4)={(19,8,26), (17,4,22),(15,7,20),(12,12,18),(14,11,14),(9,19,10),(7,15,6),(5,18,4)}

S(5)={ (19,11,46), (12,15,38), (19,11,26), (17,7,22), (15,10,20),(12,15,18),(14,14,14),(7,18,6)}

S(6)={ (14,15,102),(19,7,86),(17,10,84),(14,15,82), (9,18,70),(12,19,38), (15,14,20),(12,19,18)}

max(F_1,F_2)最小的元组为(14,15,102), (14,15,82), (15,14,20)

所以，完成所有作业最短时间是 15，安排有三种：

（1,1,0,0,1,1），（1,0,0,1,0,1）,(0,1,0,1,0,0)

3．考虑下面特殊的整数线性规划问题

$$\max \sum_{i=1}^{n} c_i x_i$$

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

试设计一个解此问题的动态规划算法，并分析算法的时间复杂度。

解：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：

$$\max \sum_{i=1}^{2n} c_i y_i$$

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

，其中 $c_{i+n} = c_i, a_{i+n} = a_i, 1 \le i \le n$，

把 c_i 看作价值，a_i 看作重量，b 看作背包容量。

转化为 0/1 背包问题，所以可以 0/1 背包问题的动态规划算法来求解。由于 n 件物品的 0/1 背包的时间复杂性是 O（2^n），则此时为 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，背包所容纳的物品的最大价值。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，若可以再判断能否使$p_i = 1$，若可以则就再放入一个c_i，这样就间接满足了$x_i = m_i + p_i = 2$的条件。

递推式：

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

时间复杂度为$O(3^n)$

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，该级设备的可靠性是 $g_i(m_i)$，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。

解：问题描述：$\max \prod_{i=1}^{n} g_i(m_i)$

约束条件：$\sum_{i=1}^{n} m_i c_i \le c, \quad 1 \le m_i \le 1 + \left\lfloor \frac{c - \sum_{i=1}^{n} c_i}{c_n} \right\rfloor$

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

$$G[0](x)=\begin{cases}1, & \sum_{i=1}^{n}c_i\le x\le c\\ -\infty, & else\end{cases}$$

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

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

初始计算$G[0](c)$，依次求解，即可得出策略集。

令$c'=c-\sum_{i=1}^{n}c_i$，$m_i'=m_i-1$，则

转化问题描述：$\max\prod_{i=1}^{n}g_i(m_i')$

约束条件：$\sum_{i=1}^{n}m_i'c_i\le c',\quad 0\le m_i'\le\left\lfloor\dfrac{c-\sum_{i=1}^{n}c_i}{c_n}\right\rfloor$

$$G[0](x)=\begin{cases}\prod_{i=1}^{n}g_i(1), & 0\le x\le c-\sum_{i=1}^{n}c_i\\ -\infty, & else\end{cases}$$

$$G[1](x)=\begin{cases}-\infty, & x<c_1\\ \max\limits_{1\le m_1\le\min(\left\lfloor\frac{x}{c_1}\right\rfloor,\left\lfloor\frac{c'}{c_1}\right\rfloor)}\{g_1(m_1)\cdot G[0](x-c_1m_1)\}, & x\ge c_1\end{cases}$$

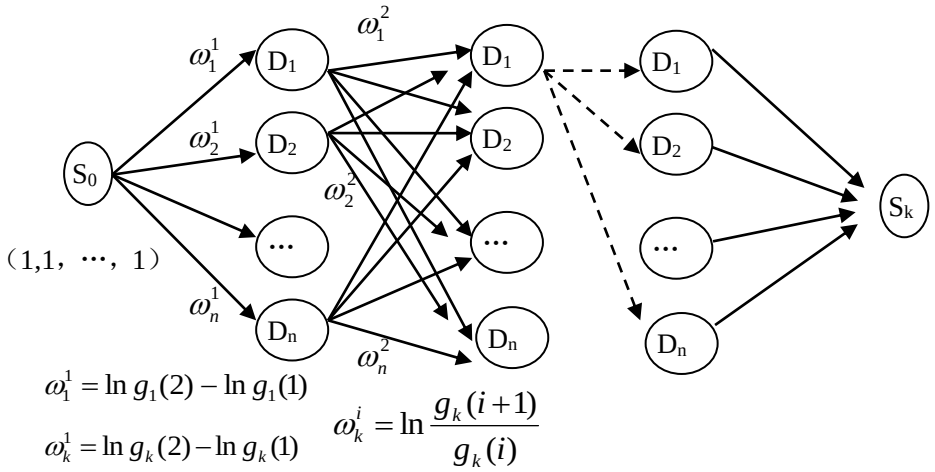
$$G[k](x)=\begin{cases} -\infty, & x<c_k \\ \max\limits_{1\le m_k\le \min\left(\left\lfloor \frac{x}{c_1}\right\rfloor,\left\lfloor \frac{c'}{c_k}\right\rfloor\right)} \{g_k(m_k)\cdot G[k-1](x-c_k m_k)\}, & x\ge c_k \end{cases}$$

思路二：部分同学尝试转化为类似背包问题，但是权重值需要进行一系列的转化，背包中相同的物品，不再具有相同的价值。

问题描述：$\max \sum_{i=1}^{n} \ln g_i(m_i)$

约束条件：$\sum_{i=1}^{n} m_i c_i \le c,\quad 1\le m_i \le 1+\left\lfloor \dfrac{c-\sum_{i=1}^{n} c_i}{c_n} \right\rfloor$

多段图



ω_k^i 表示第i个阶段增加一台第k级设备，而且是该级设备的第i_s台时的权重

当第 i 个阶段第 k 级设备增加第 i_s+1 台时，$\omega_k^i=\begin{cases} \ln \dfrac{g_k(i_s+1)}{g_k(i_s)}, & \text{若满足约束条件} \\ -\infty, & \text{其余情况} \end{cases}$

阶段数上限为 $\left\lfloor \frac{c-\sum_{i=1}^{n} c_i}{\min\{c_k\}} \right\rfloor$，阶段数下限为 $\left\lfloor \frac{c-\sum_{i=1}^{n} c_i}{\max\{c_k\}} \right\rfloor$

（仅供参考，自行整理答案。）

第七章分支限界法习题

1．假设旅行商问题的邻接矩阵如图 1 所示，试用优先队列式分枝限界算法给出最短环游。画出解空间树的搜索图，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 邻接矩阵

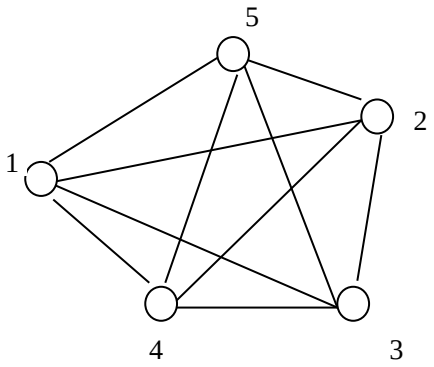
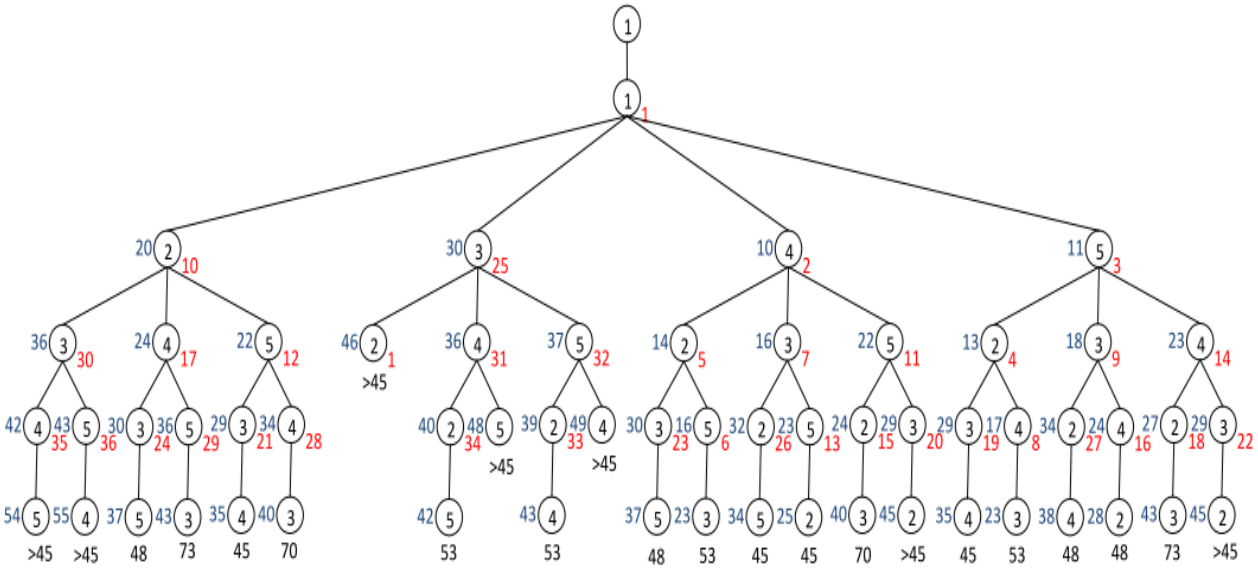


图 2 旅行商问题

解答：

1．假设旅行商问题的邻接矩阵如图 1 所示，试用优先队列式分枝限界算法给出最短环游。画出解空间树的搜索图，并说明搜索过程。

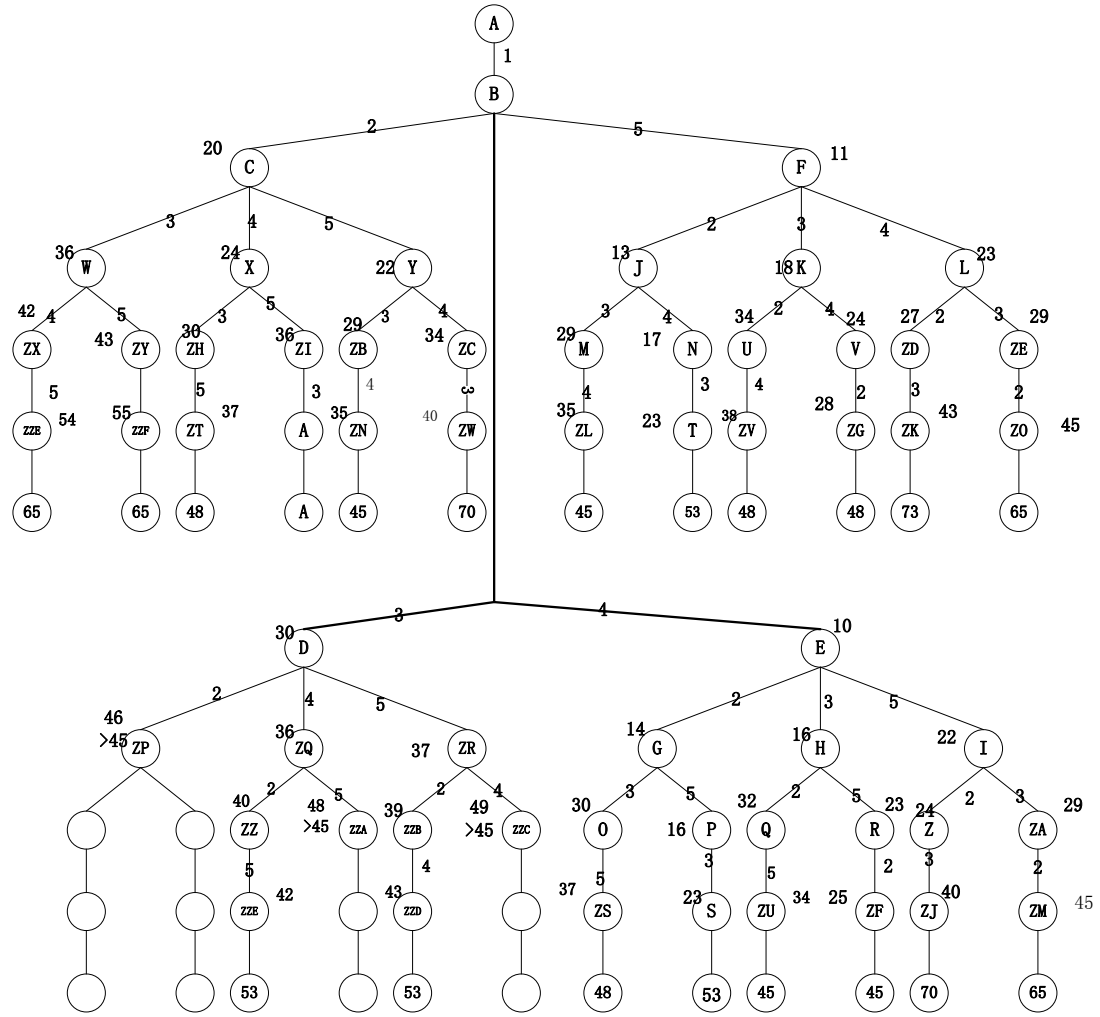
解空间树搜索图如下



为了方便看清路径，圆圈内为顶点序号，未标结点号。左侧蓝色数字为该路径到该结点的总路程，右下红色数字为该结点的搜索序数。从顶点 1 开始搜索，将其四个儿子结点放入队列。然后优先访问队列中当前路程最小的结点，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程最小的结点，将其儿子放入队列。

第一个被访问到的叶子结点为 1-4-2-5-3-1 这条路径，其总路程为 53，记录为当前最短路程。之后若某结点的路程大于最短路程，则不再搜索该结点的子树。若某叶子结点的总路程小于当前最短路程，则更新之。如此搜索，当访问到 1-4-3-5-2-1 这条路径的叶子结点时，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。



搜索的过程如下：

B{E,F,C,D}

E{F,G,H,C,I,D}

F{J,G,H,K,C,I,L,D}

J{G,H,N,K,C,I,L,M,D}

G{H,P,N,K,C,I,L,M,D,O}

H{P,N,K,C,I,L,R,M,D,O,Q}

P{N,K,C,I,L,R,M,D,O,Q}

N{K,C,I,L,R,M,D,O,Q}

K{C,I,L,R,V,M,D,O,Q,U}

C{I,Y,L,R,V,X,M,D,O,Q,U,W}

I{Y,L,R,V,X,Z,M,ZA,D,O,Q,U,W}

Y{L,R,V,X,Z,M,ZA,ZB,D,O,Q,U,ZC,W}

L{R,V,X,Z,ZD,M,ZA,ZB,ZE,D,O,Q,U,ZC,W}

R{ V,X,Z,ZD,M,ZA,ZB,ZE,D,O,Q,U,ZC,W }

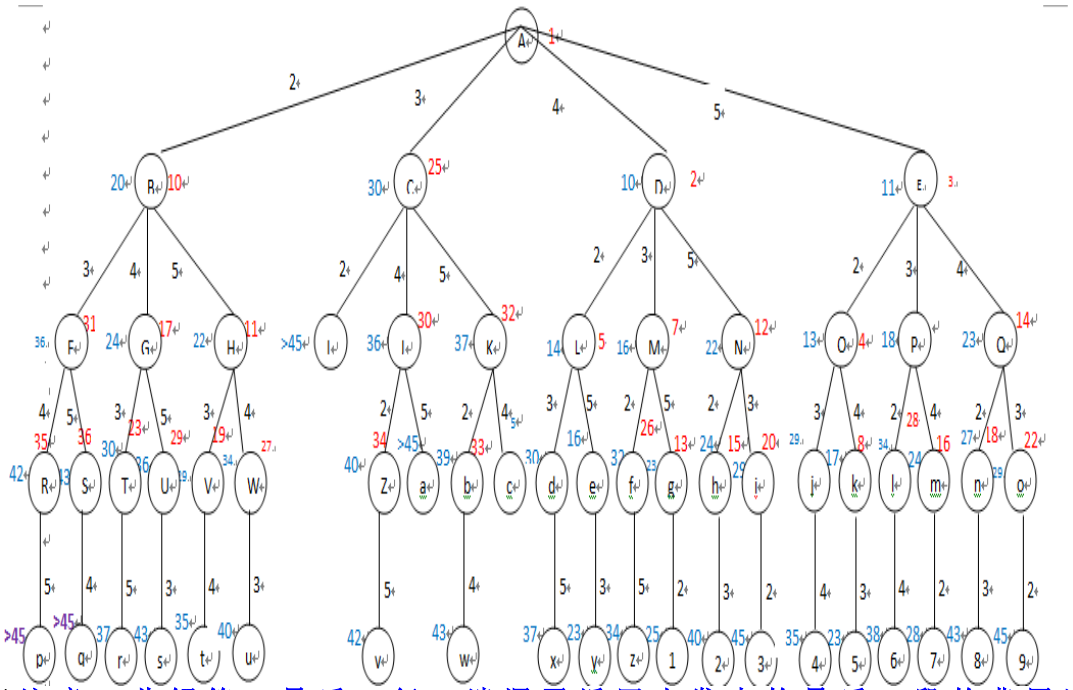
V{ X,Z,ZD,M,ZA,ZB,ZE,D,O,Q,U,ZC,W }

X{Z,ZD, M,ZA,ZB,ZE,D,O,ZH,Q,U,ZC,W }

Z{ ZD, M,ZA,ZB,ZE,D,O,ZH,Q,U,ZC,W }

ZD{ M,ZA,ZB,ZE,D,O,ZH,Q,U,ZC,W }
M{ZA,ZB,ZE,D,O,ZH,Q,U,ZC,W }
ZA{ ZB,ZE,D,O,ZH,Q,U,ZC,W }
ZB{ ZE,D,O,ZH,Q,U,ZC,W }
ZE{ D,O,ZH,Q,U,ZC,W }
D{O,ZH,Q,U,ZC,W,ZQ,ZR,ZP}
O{ ZH,Q,U,ZC,W,ZQ,ZR,ZP }
ZH{ Q,U,ZC,W,ZQ,ZR,ZP }
Q{ U,ZC,W,ZQ,ZR,ZP }
U{ZC,W,ZQ,ZR,ZP}
ZC{W,ZQ,ZR,ZP}
W{ZQ,ZR,ZX,ZY,ZP}
ZQ{ZR,ZZ,ZX,ZY,ZP,ZZA}
ZR{ZZB,ZZ,ZX,ZY,ZP,ZZA,ZZC}
ZZB{ZZ,ZX,ZY,ZP,ZZA,ZZC}
ZZ{ ZX,ZY,ZP,ZZA,ZZC }
ZX{ZY,ZP,ZZA,ZZC}
ZY{ ZP,ZZA,ZZC }
ZP{ZZA,ZZC}
ZZA{ZZC}
ZZC{}

下面的解答为一典型的错误解答。



（注意：此解答，最后一行，遗漏了返回出发点的最后一段的费用）

>45 >45 48 73 45 70 53 53 48 53 45 45 70 >45 45 53 48 48 73 >45

2．试写出 0/1 背包问题的优先队列式分枝限界算法程序，并找一个物品个数是 16 的例子检验程序的运行情况。

（*具体程序见下页附件及 c++源代码，*

关键是要充分理解分支限界的具体实现思路，

自己课下认真推导，真正要理清思路。）

3．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最短。

解：思路一：

将 n 个任务按照所需时间非递减排序，得到任务序列 1，2，3，4…….,n，满足时间关系 t[1]<t[2]<……<t[n].

限界函数：将 n 个任务中的前 k 个任务分配给当前 k 个机器，然后将第 k+1 个任务分配给最早完成已分配任务的机器，依次进行，最后找出这些机器最终分配任务所需时间最长的，此时间作为分支限界函数，如果一个扩展节点所需的时间超过这个已知的最优值，则删掉以此节点为根的子树。否则更新最优值。

优先级：哪台机器完成当前任务的时间越早，也就是所有机器中最终停机时间越早，优先级就越高，即被选作最小堆中的堆顶，作为扩展节点。

设 task[n]用来记录最优的调度顺序

每个节点具有信息：

{Parent：父亲节点，**Level：**节点所在深度加 1，**Ctime：**运行到当前节点所用时间**}**

当 level<=k 时（不能用界限函数来剪枝，也不需要判断优先级），由于机器还未装满（即前面的 k 个任务其实是同时加入的），可以令 Ctime=0，

当 level>k 时（需要界限函数来剪枝，需要判断优先级），Ctime 就是运行到当前状态所用的总时间，Ctime 作为优先级函数，即从最小堆中选取 Ctime 最小的节点优先。当找出第一个解节点时，记录此时的 Ctime 作为目标函数值，以后生成的节点的 Ctime 大于该目标函数值时，就可以剪掉该节点，如此下去一直到最小堆为空为止。

上述就是最佳调度问题的分支限界算法。

解空间树的节点包括以下信息:

```
Node{
    int Path[n];          //节点对应的解空间树的路径，即到该节点为止的策略记录
    int T[k];             //在本策略下的每台机器的运行时间
    int Time;             //本策略的总执行时间，为每台机器运行时间的最大值
    int length;           //本节点的深度,即当前处理的作业
}

Proc BestDispatch(int n,int k,int t[])
    Node Boot,X,P,result; //Boot 为根节点，result 保存最优解
    int f;                //记录当前最优解的执行时间
```

```
    f=n*max(t[]);                    //初始化 f
    Boot.T[n]={0};
    Boot.Time=0;
    Boot.Path[n]={0};
    Boot.length=0;                   //初始化根节点
    AddHeap(Boot);                   //根节点加入堆中，堆中元素按照 Time 值由小到大排序
    While  !Heap.empty()  do
        P=DeleteMinHeap();                  //P 为当前优先级最高的点
        for i=1 to k do                     //扩展 P 的 k 个子节点
            X=Newnode(P.Path[],P.T[],P.length+1);
            X.Path[X.length]=i;
            X.T[i]=X.T[i]+t[X.length];
            X.Time=max(X.T[]);
            if X.length==n then             //X 为叶节点
                if X.Time<f   then          //X 的执行时间小于已知最优解
                    f=X.Time;               //将 X 设为最优解
                    result=X;
                end{if}
                        else        //X 为中间节点
                if X.Time<f then
                    AddHeap(X);
                end{if}//X 的当前执行时间小于已知最优解则加入堆中，否则剪去
            end{if}
        end{for}
    end{while}
end{ BestDispatch }
```

思路二：

算法执行步骤如下：

1．先将任务由大到小排序

2．计算 n 个任务需要的总时间和平均到 k 个机器上的时间

3．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

4．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

5．安排下一台机器直到所有任务安排完，

6. 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。

附件：

习题 2 的 c++代码

```
#include<iostream>
using namespace std;
int *answer;//存储解向量
struct Node{//活节点表中所存储的节点
    int   level;//节点在解空间树中的深度
    int    tag;//用于输出最优解的各个分量
    int    cw;//当前背包装载量
    int    cp;//当前背包所获得的价值
    float ub; //节点的上界值
    Node  *parent;//父节点指针
    Node  *next; //后继节点指针
};
class Knap{
    private:
        Node  *front;//队列队首
        Node  *bestp;//解节点
        Node  *first;//根节点
        int   *p,*w,n,c,*M;//背包价值、重量、物品数、背包容量、记录大小顺序关系
        long  best;//背包容量最优解
    public:
        Knap(int *P,int *W,int C,int N);//构造函数，用于类的初始化
        ~Knap();//析构函数
        void Sort();
        float LUBound(int i,int cw,int cp);//计算背包的上界值
        Node  *NewNode(Node  *pa,int  tagLeftChild,float  uub);// 生 成 一 个 新 的 节
点,tagLeftChild=1 生成左节点,tagLeftChild=0 生成右节点
        void AddNode(Node *node);//将节点添加到队列中
        void DelNode(Node *node);//将节点从队列中删除
        Node *NextNode(); //取下一个节点
        void PrintResult();//输出结果
        void LCKnap();//背包问题求解
};
Knap::Knap(int *P,int *W,int C,int N)
{
    n=N;
    c=C;
    p=new int[n];
    w=new int[n];
    M=new int[n];
    for(int i=0;i<n;i++){
        p[i]=P[i];
```

```
            w[i]=W[i];
            M[i]=i;
        }
        front=new Node[1];
        front->next=NULL;
        best=0;
        bestp=new Node[1];
        bestp=NULL;
        Sort();
    }
    Knap::~Knap()
    {
        delete []front;
        delete []bestp;
        delete []p;
        delete []w;
    }
    float Knap::LUBound(int k,int cw,int cp)
    {
        int remain_c=c-cw;
        float ub=(float)cp;
        while(k<n&&w[k]<=remain_c){
            remain_c-=w[k];
            ub+=p[k];
            k++;
        }
        if(k<n)
            ub+=(float)(1.0*p[k]/w[k]*remain_c);
        return ub;
    }
    Node * Knap::NewNode(Node *pa,int tagLeftChild,float uub)
    {
        Node * node=new(Node);
        node->level=(pa->level)+1;
        node->tag=tagLeftChild;
        node->ub=uub;
        node->parent=pa;
        node->next=NULL;

        if(tagLeftChild==1){
            node->cw=pa->cw+w[pa->level];
            node->cp=pa->cp+p[pa->level] ;
        }
        else{
```

```
        node->cw=pa->cw;
        node->cp=pa->cp;
    }
    return node;
}
void Knap::AddNode(Node *node)
{
    Node *p=front->next,*q=front;
    float ub=node->ub;
    while(p!=NULL){
        if(p->ub<ub){
            node->next=p;
            q->next=node;
            break;
        }
        q=p;
        p=p->next;}
    if(p==NULL){
        q->next=node;
    }
}
Node *Knap::NextNode()
{
    Node *p=front->next;
    front->next=p->next;
    return p;
}
void Knap::Sort()
{
    int i,j,k;
    float temp;
    for(i=1;i<n;i++){
        temp=(float)(1.0*p[i]/w[i]);
        k=0;
        for(j=1;j<=n-i;j++){
            if(temp<1.0*p[j]/w[j]){
                temp=(float)(1.0*p[j]/w[j]);
                swap(p[k],p[j]);
                swap(w[k],w[j]);
                swap(M[k],M[j]);
                k=j;
            }
        }
    }
}
void Knap::PrintResult()
{
    int i=0;
    cout<<"最大价值是:"<<best<<endl;
```

```
    for(i=n;i>=1;i--){
        answer[M[i-1]]=bestp->tag;
        bestp=bestp->parent;
    }
    cout<<"对应的解向量为:"<<" ( ";
    for(i=1;i<=n;i++)
        cout<<answer[i-1]<<" ";
    cout<<")"<<endl;
}
void Knap::LCKnap()
{
    int k;
    float ub;
    Node *nextNode;
    first=new Node[1]; //生成根节点
    first->parent=NULL;
    first->next=NULL;
    first->level=0;
    first->cw=0;
    first->cp=0;
    first->tag=0;

    ub=LUBound(0,0,0);
    first->ub=ub;
    front->next=first;

    while(front->next!=NULL)
    {
        nextNode=NextNode();
        k=nextNode->level;
        if(k==n-1){
            if(nextNode->cw+w[k]<=c&&(long)(nextNode->cp+p[k])>best){
                best=nextNode->cp+p[k];
                bestp=NewNode(nextNode,1,(float)best);
            }
            if((long)(nextNode->cp)>best){
                best=nextNode->cp;
                bestp=NewNode(nextNode,0,(float)best);
            }
        }
        if(k<n-1){

    if(nextNode->cw+w[k]<=c&&LUBound(k+1,nextNode->cw+w[k],nextNode->cp+p[k])>(float)b
est){
                ub=LUBound(k,nextNode->cw+w[k],nextNode->cp+p[k]);
                AddNode(NewNode(nextNode,1,ub));
            }
            ub=ub=LUBound(k,nextNode->cw,nextNode->cp);
            if(ub>best)
                AddNode(NewNode(nextNode,0,ub));
        }
    }
    PrintResult();
}
void main(){
    int c,n,i=0;
   int *p,*w;
    cout<<"请输入背包容量值:";
    cin>>c;
```

```
        cout<<"请输入物品数:";
        cin>>n;
        p=new int[n];
        w=new int[n];
        for(i=0;i<n;i++) {cout<<"请输入第"<<i+1<<"件物品的重量向量"<<endl;cin>>w[i]; }
        cout<<"请输入价值向量"<<endl;
        for(i=0;i<n;i++) {cout<<"请输入第"<<i+1<<"件物品的重量向量"<<endl;cin>>p[i];}
        answer=new int[n];
        Knap kanp(p,w,c,n);
        kanp.LCKnap();
    }
```

1．二元可满足性问题 2SAT

例：给定布尔变量的一个有限集合$U=\{u_1,u_2,\cdots,u_n\}$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？

证明： 2SAT 是P－类问题。为叙述方便，采用数理逻辑中的“合取式”表达逻辑命题，于是

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

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$\bar{u}_i$。

考虑表达式$C=\prod_{k=1}^{m}(x_k+y_k)$，如果有某个$x_k+y_k=u_i+\bar{u}_i$，则在乘积式中可以去掉该子句：$C'=C\setminus(u_i+\bar{u}_i)$，可见$C$与$C'$的可满足性是等价的。所以我们可以假定$C$中不含有形如$u_i+\bar{u}_i$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或它的非$\bar{u}_n$在C的某个子句中出现，我们将C表示成

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

其中G是C的一部分子句，而且不出现逻辑变量u_i或它的非$\bar{u}_i$。令

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

(2)式中不再含有变量u_n和它的非$\bar{u}_n$。记$U'=\{u_1,u_2,\cdots,u_{n-1}\}$。如果存在$U$的真赋值，使得$C$满足，则一定存在$U'$的真赋值使得$C'$满足。

因为如果u_n取真值，则所有的z_j必然取真值；u_n取假值，所有的y_i必然都取真值，不管那中情况，C'的乘积部分必然取真值。反之，假设存在U'的真赋值，使得C'满足。若有某个y_i取假值，则所有的z_j必然取真值，此时令u_n取真值，得到U的真赋值，使得C满足。若有某个z_j取假值，则令u_n取假值，得到U的真赋值，使得C满足。如果所有的y_i和z_j都取真值，u_n取假值得到U的真赋值，使得C满足。

至此我们得到：C与C'的可满足性是等价的。但是后者涉及的变量数比前者少 1，子句数为$m-(k+h)+kh$。但是，我们可以像前面一样简化掉所有形如$u_i+\bar{u}_i$的子句，因而可以假定C'中子句个数不超过$(n-1)(n-2)$。

上述过程可以一直进行到判定只含有一个逻辑变量的逻辑语句的可满足性问题。这需要一个常数时间即可。注意到我们每一步简化都可以在多项式（关于n的）步骤内完成，总共需要至多$n-1$步简化，因而，在多项式时间内可以完成 2SAT 二满足性问题的判定。即 2SAT 是 P－类问题。　　证毕

5. 独立集问题：

例：对于给定的无向图$G=(V,E)$和正整数$k\ (\le|V|)$

问：G是否包含一个于k－独立集V'，即是否存在一个子集$V'\subseteq V,|V'|=k$，使得V'中的任何两个顶点在图中G都不相邻。

证明独立集问题都是 NPC 问题（提示：考虑独立集和团的关系：如果V'是图G的团，则V'是G的补图$\tilde{G}$的独立集；反之亦然）。

证明（一）：思路：证明 $\Pi\in NP$；选取一个已知的 NP 完全问题Π'；构造一个从Π'到Π的变换f；证明f为一个多项式变换。

我们选取团问题作为参照物。给定一个无向图$G(V,E)$，存在$V'\subseteq V$，$|V'|=k$，且对于任意$u,v\in V'$有$(u,v)\in E$。我们构造新的无向图$\tilde{G}(V,\tilde{E})$，使得$\tilde{G}$为G的补图，即对于任意$u,v\in V$，若$(u,v)\in E$，则$(u,v)\notin\tilde{E}$，若$(u,v)\notin E$，则$(u,v)\in\tilde{E}$。

（1）　考虑独立集和团的关系：如果V'是图G的团，则V'是G的补图$\tilde{G}$的独立集；反之亦然。（详见（4）证明）

（2） 首先说明独立集问题是 NP 问题。因为无向图$\tilde{G}(V,\tilde{E})$的团问题是一个 NP 问题，从而由其对应关系可知$G(V,E)$的独立集问题也是一个 NP 问题。（也可仿照无向图的独立团问题的三个阶段的方法，证明独立集问题是 NP 问题）

（3） 已知团问题是 NPC 问题。

（4） 下面证明独立集到团问题的转换是一个变换，只需要证明$\tilde{G}$包含一个 k 独立集V'，当且仅当G包含一个 k 团。

（i）如果$\tilde{G}$包含一个 k 独立集V'，则在无向图$\tilde{G}$中任何两个顶点$u,v\in V'$都不相邻，因为G为$\tilde{G}$的补图，故在无向图G中，对于顶点子集V'，任何两个顶点$u,v\in V'$必然都存在$(u,v)\in E$，即G包含一个 k 团。

（ii）如果G包含一个 k 团，则对于无向图G中任意$u,v\in V'$有$(u,v)\in E$，因为$\tilde{G}$为G的补图，所以在无向图$\tilde{G}$中对于顶点集V'，任何两个顶点$u,v\in V'$必然都不相邻，即$\tilde{G}$包含一个 k 独立集。

上述变换可以在多项式时间内完成，设无向图G中的顶点个数为 n，即$|V|=n$，则对于$|G\cup\tilde{G}|=\frac{n(n-1)}{2}$，所以该变换可以在多项式时间内完成，即该变换为多项式变换。

证明（二）：

思路：由定理 3 如果$L',L\in NP, L'\propto L$，则$L'\in NPC\Rightarrow L\in NPC$.

可以证明独立集问题是一个 NP 问题，通过团问题∝独立集问题，而且团问题是一个 NPC 问题，来证明独立集问题是 NP 难问题，从而说明独立集问题是 NPC 问题。

如果V'是图G的团，则V'是G的补图$\tilde{G}$的独立集；反之亦然。

所以团的问题∝独立集问题，所以只需证明 V ’ 是图 G 的团即可。

由于团问题是一个 NPC 问题，所以独立集问题也是一个 NPC 问题。

（以上各题均来自同学们作业中的思想，仅供参考，自行整理答案。）

中国科学院研究生院 课程编号：

试 题 专 用 纸 课程名称：计算机算

法设计与分析

任课教师：陈玉福

姓名_____ 学号_____ 成 绩_____

一. 回答下列问题：

1. 在对算法进行复杂性分析时，时间复杂性用什么来度量？其间做了什么假定？渐进复杂性指的是什么？精确到什么程度？

2. 在连通图无向图G的宽度优先搜索树和深度优先搜索树中，哪棵树的最长路径可能会更长些？试说明你的理由。

3. 何谓最优化原理？采用动态规划算法必须满足的条件是什么？动态规划算法是通过什么问题的什么特性提高效率的？试比较贪心算法与动态规划算法的异同。

4. 阐述回溯算法与分枝限界算法的区别和联系，各自强调改善那方面以提高效率？

5. 确定性图灵机模型与非确定性图灵机模型的主要区别在那里？确定性图灵机模型下算法的时间复杂度和空间复杂度指的是什么？

二. 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$\sum_{p_i \in Q} a_i \le L$。

2). 设Q是使用上述贪心算法得到的子集合，磁带的利用率$\sum_{p_i \in Q} a_i / L$可以小到何种程度？

3). 试说明 1)中提到的设计策略不一定得到使 $\sum_{p_i \in Q} a_i / L$ 取最大值的子集合

三. 假定已知“无向图的 Hamilton 圈”问题是 NPC 问题，证明“旅行商判定问题”也是 NPC 问题。说明“旅行商最优问题”不是 NPC 问题。

四. 虑无向图上的搜索算法

1．将宽度优先搜索算法中的队列改成栈，其它不变，给出一个搜索算法。

2．下面是一个无向图及其邻接链表，用你给出的算法画出图 G 的搜索生成树，标出各节点被访问的次序号。

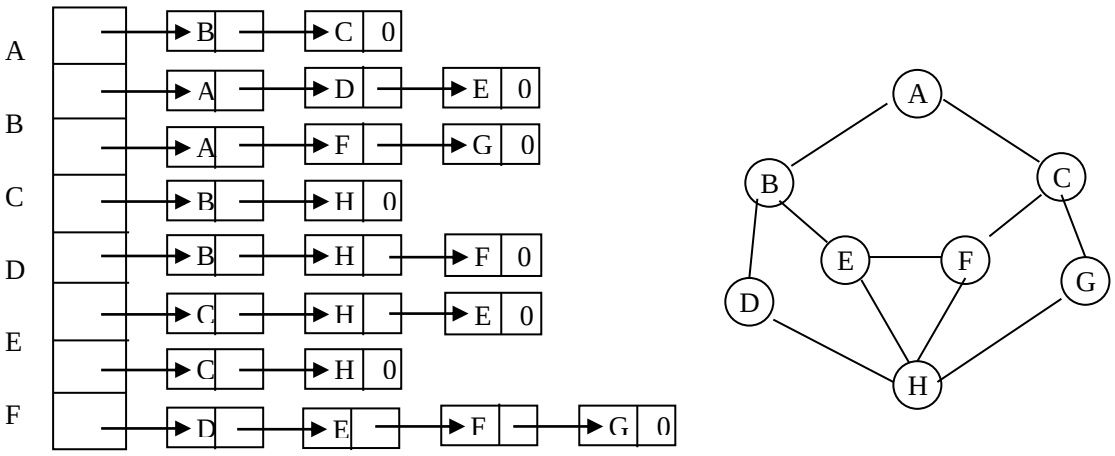


图 1 一个无向图 G 和它的邻接链表

五. 设 $n = 2^m, A$ 是一个 $2n$ 维数组，待求最大值的 n 个数开始存放在 $A[n], A[n+1], \cdots, A[2n-1]$，所求得的最大值存于 $A[1]$。下面是利用平衡树技术设计的求最大值算法，试给出该算法的计算工作量 $W(n)$、运行时间 $t(n)$、处理器数 $p(n)$、成本 $c(n)$、加速比 $s_p(n)$、效率 $E_p(n)$：

SIMD-TC 模型上求最大值算法

输入：$n = 2^m$ 个数，存放于数组 $A[n:2n-1]$ 中

输出：最大数置于 $A[1]$ 中

```
begin
  for k = m-1 to 0 do
    for j = 2^k to 2^(k+1) – 1 par-do
      A[j] ← max{ A[2j], A[2j+1] }
    end for
  end for
end
```

中国科学院研究生院　　课程编号：

试 题 专 用 纸　　课程名称：计算机算法设计与分析

任课教师：陈玉福

姓名＿＿＿＿＿＿＿＿　学号＿＿＿＿＿＿＿＿　成　绩＿＿＿＿＿＿

六．（共 20 分，每小题 5 分）回答下列问题

6．已知求解问题 Π 的两个算法 A_1, A_2 的时间复杂性函数分别为 $T_1(n) = n2^{n/2}$ 和 $T_2(n) = n\log^2 n$。现在有两台计算机 C_1, C_2，它们的速度比为 64。如果采用算法 A_1，计算机 C_1 求解问题 Π 的一个实例 I 所用的时间为 T，那么，采用算法 A_2 时，计算机 C_2 能够在时间 T 内求解问题 Π 的多大输入规模的实例？

7．何谓最优化原理？采用动态规划算法必须满足的条件是什么？动态规划算法是通过什么问题的什么特性提高效率的？

8．阐述回溯算法与分枝限界算法的区别和联系，各自强调改善那方面以提高效率？

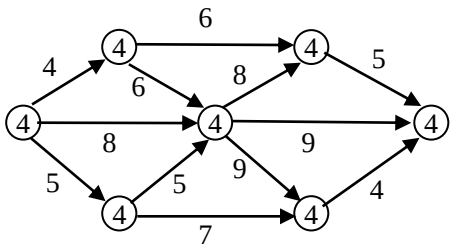
9．多项式时间确定性算法与多项式时间非确定性算法的主要区别是什么？

七. (12 分) 下面是插入排序算法，试分析它在最坏情况下的时间复杂度和平均时间复杂度。

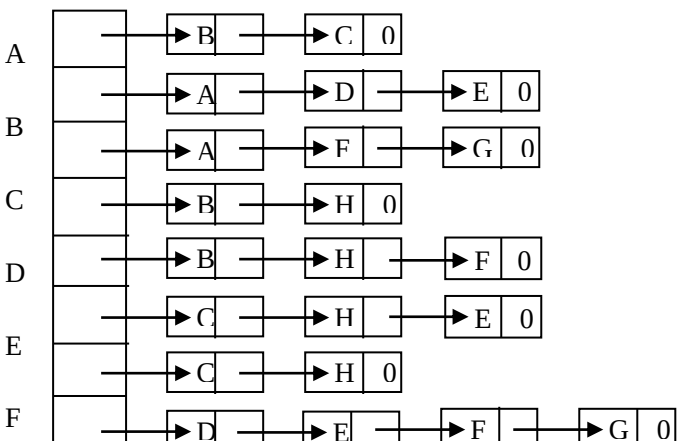
插入排序算法

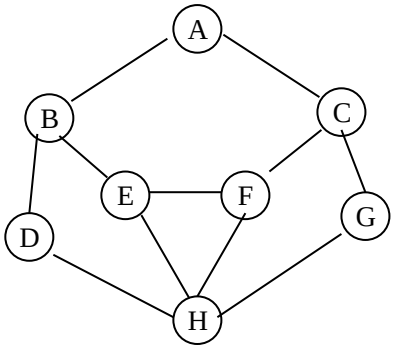
```
proc InSort(a, n)
  for i from 2 to n do
    t:=a[i];
    integer j;
    for j from i-1 to 1 do
      if t<a[j] then a[j+1]:=a[j]; end{if}
    end{for}
    a[j+1]:=t;
  end{for}
end{InSort}
```

八. (12 分) 用动态规划算法求下图中从顶点 0 到顶点 6 的所有最短路径和最长路径。



九. (11 分) 将宽度优先搜索算法中的队列改成栈，其它不变，则得到 D－检索算法。下面是一个无向图及其邻接链表，试画出图 G 的 D－检索生成树，并标出各节点被访问的次序号。





无向图G和它的邻接链表

十. (15 分) 有 5 个物体，其重量分别为 3，5，7，8，9，价值分别为 4，6，7，9，10。有一背包，载重量为 22，物体不可分割地往背包里装。试画出用优先级队列式分枝限界算法 LCKNAP 解此 0/1 背包问题所生成的解空间树，并给出最优解。

十一. （共 10 分,每小题 5 分）假定已知“无向图的 Hamilton 圈”问题是 NP-完全问题.

a）证明旅行商问题判定模式也是 NP-完全问题；

b）证明旅行商问题优化模式不是 NP-完全问题，但是 NP 难问题。

计算机算法设计与分析试题

(2008 年 12 月)

一. （共 16 分，每小题 4 分）简答题

1. 在对算法进行复杂性分析时，时间复杂度用什么量反映的？其间做了什么假定？复杂性函数的渐进上界反映了复杂性函数的什么性质？

2. 叙述最优化原理，并说明动态规划算法是依据问题的那两个性质设计的？

3. 阐述回溯算法与分枝限界算法的区别和联系，各自强调改善那方面以提高

效率？

4．什么是 *NP* 问题？证明一个问题是 *NPC* 问题一般采用哪几个步骤？

二．（共 16 分）下面是归并排序算法（递归程序）

```
MergeSort(low, high) // A[low: high]是一个全程数组，含 high-low+1
   //个待排序的元素
  integer low, high;
  if low < high then
     mid=⌊(low+high)/2⌋  //求当前数组的分割点
     MergeSort(low, mid)   //将第一子组排序
     MergeSort(mid+1, high)   //将第二子组排序
      Merge (low, mid，high)   //合并两个已经排序的子数组
  endif
end MergeSort
```

用 $T(n)$ 表示执行该程序所用的时间，并假定 $T(1)=a$，且合并子程序 Merge 所用时间与 n 成正比，即 cn，c 是正实数。

1． 写出该程序所用时间 $T(n)$ 的递推关系式；（5 分）

2． 当 $n=2^k$ 时，解上述递推关系式得到 $T(n)$ 的表达式；（6 分）

3． 证明：对于一般的 n，有 $T(n)=O(n\log n)$。（5 分）

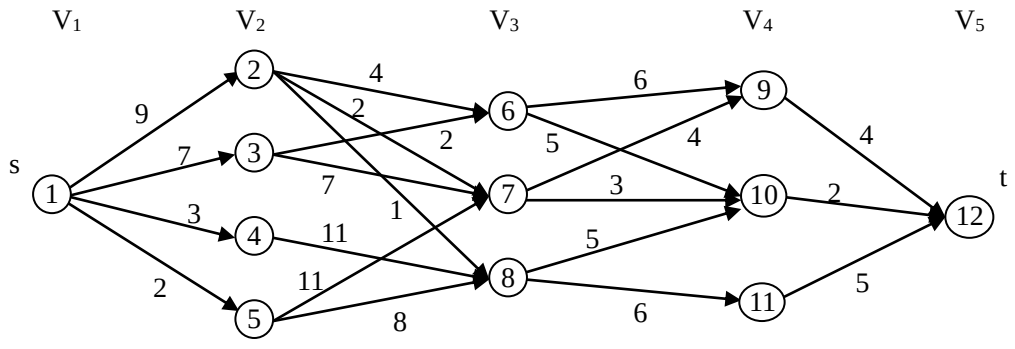
三．（共 16 分）用优先队列式分枝限界算法解如下 0/1 背包问题：

$$n=4,(p_1,p_2,p_3,p_4)=(10,15,6,4),(w_1,w_2,w_3,w_4)=(4,6,3,2),M=12$$

1．画出解空间搜索树，并标注说明；（12 分）

2．给出最优解。（4 分）

四．（供 16 分）用动态规划算法解如下图示的多段图问题，



1．说明多段图问题具有最优子结构性质；（5 分）

2．写出多段图问题最优值的递推公式；（5 分）

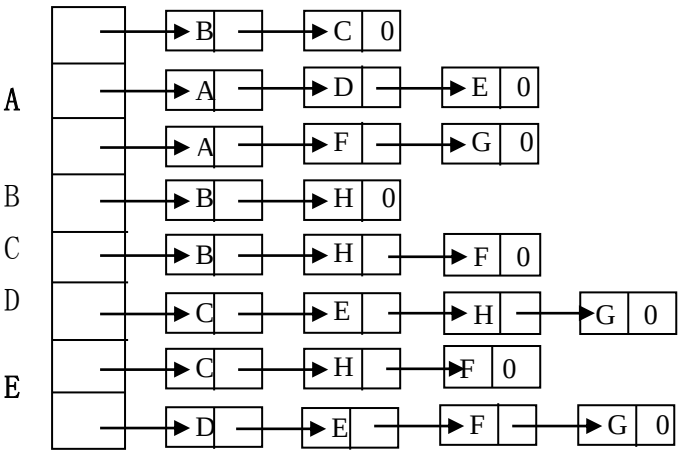
3．给出问题的一个最优解并在图上标注说明。（6 分）

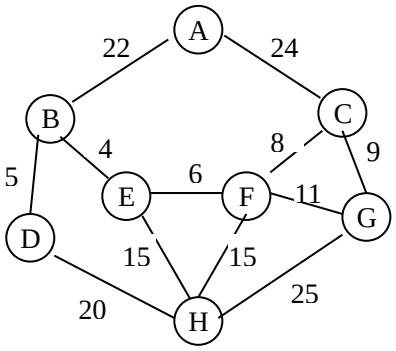
五．（16 分）已知划分问题是 *NPC* 问题，试证明 0/1 背包（判定）问题也是 *NPC* 问题

《计算机算法设计与分析》试题

（2009 年 11 月）

二．　下面是一个无向图及其邻接链表





无向**图 G 及其邻接链表**

1．(8 分) 给出 D-搜索生成树，并在各个顶点旁标出该顶点被搜索的序号；

2．(12 分) 采用 Prim 算法，给出图 G 的最小生成树，并简要描述生成过程

（即边集的增加过程）。

三． 装箱问题：将 n 物品装入（不能分割）容积相等的若干个箱子。假定第 i 件物品装入箱子所占的容积是 $v_i, 0 < v_i \le 1 (i = 1, 2, \cdots, n)$，箱子的容积都是 1。确定装箱方法，使所用的箱子个数尽量少。

1． （5 分）试给出一个贪心算法，并说明算法的时间复杂度；

2． （10 分）已知 $v_i = 1/(i+1)(i = 1, 2, 3, 4, 5, 6, 7, 8)$，按照你给出的算法描述装箱过程。

3． （5 分）你给出的贪心算法能够获得装箱问题的最优解吗？简单说明理由。

四． （20 分） 知连续偶数数组 $D[1:n]$，单调递增函数 $f(x)$，试设计一个分治算法，搜索数组中数 x，使得 $\frac{|f(x) - 0.07|}{0.07} < 2.5\%$。如果这样的 $x \in D[1:n]$ 存在，则返回 x；如果这样的 x 不存在，则返回 false。要求：写出算法的伪代码或程序，并给出最坏情况下的时间复杂性函数(不需证明)。

五． 用优先队列式分子限界法解下面的旅行商问题，假定旅行商开始时处在第一个城市，各个城市（共 5 个城市）间的距离由下面的矩阵给出：

$$A = \begin{pmatrix} \infty & 7 & 3 & 12 & 8 \\ & \infty & 6 & 14 & 9 \\ & & \infty & 5 & 18 \\ & & & \infty & 11 \\ & & & & \infty \end{pmatrix}$$

1．(5 分）说明你所使用的优先级函数和限界函数；

2．(10 分）画出解空间树（即状态空间树），通过活结点表的变化说明搜索过；

3．(5 分）给出一条旅行商最短的环行路线。

中国科学院研究生院 课程编号：

试 题 专 用 纸 课程名称：计算机

算法设计与分析

任课教师：陈玉福

姓名_____ 学号_____ 成 绩_____

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

1. 一个简单算法如下：

```
int Maxsum(int n,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解最大子段和问题。分析算法的时间复杂度（提示：记 $b[j]=\max_{1\le i\le j}\left\{\sum_{k=i}^{j}a[k]\right\}$，$1\le j\le n$，所求的最大子段和为 $\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\max_{1\le i\le j}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}b[j]$）。

二．简答下列问题：

1． 贪心算法和动态规划算法有什么共同点和区别？

2． 试比较回溯法与分枝限界算法，分别谈谈这两个算法比较适合的问题？

3． 什么是多项式时间算法？

4． 什么是 NP 类问题？

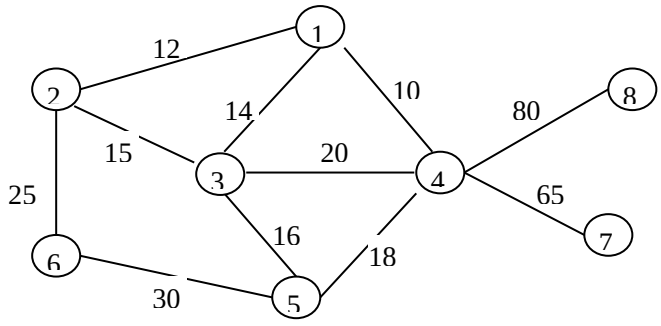
三．下面是二元可满足性问题，它是 P 类问题还是 NPC 问题？

例：给定逻辑语句 $C=C_1\wedge C_2\wedge\cdots\wedge C_l$，其子句定义在布尔变量 $X=\{x_1,x_2,\cdots,x_n\}$ 上，而且每个子句均由两个文字构成 $C_i=y_i\vee z_i,\quad y_i,z_i\in\{x_1,\bar{x}_1,\cdots,x_n,\bar{x}_n\}$，$i=1,2,\cdots,l$。

问：是否存在布尔变量的一个真值分配，使得语句 C 取真值？

四．1.分析 Kruskal 算法的时间复杂度；

2. 试用下面的例子说明用 Kruskal 算法求解最优生成树问题，并总结出算法的基本步骤：



五．采用优先队列式分枝限界算法求解 0/1 背包问题：

$$n=4,P=(15,15,18,27),W=(2,4,6,9),M=15。$$

1. 画出状态空间树，并在各个节点处标出目标值上界估值 Pvu 和下界估值 Pvl；

2. 指出状态空间树中各节点被选作当前扩展节点的顺序（标号）。