算法设计与分析

2013-秋季学期

课程介绍

- 教学参考书
- **[1] J. Kleinberg, E. Tardos： Algorithms Design, 清华大学出版社影印本, 2006.**
- **[2] Thomas H. Cormen， Charles E. Leiserson， Ronald L. Rivest， Clifford Stein：算法导论, MIT第3版, 2009.**
- 教学内容
- **1. 搜索　2. 分治法　3. 贪心法　4. 动态规划**
- **5. 数据结构　6. 随机算法　7. NP完全理论**
- 作业要求
- 每周一次，独立完成，有困难可选做一部分，
- 格式为：问题分析+解决方案(算法描述)+正确性证明+复杂度分析
- 　　+程序代码+运行情况总结(部分习题不需要实现)

第一章 搜索

- 1.1 通用蛮力算法
- 1.2 隐式图的搜索算法
- 1.3 回溯算法
- 1.4 分枝限界法

求解组合问题的通用蛮力算法--Brute-find

```
public interface ItrObj { boolean next(); }
public interface ComProb {
   boolean target(Object obj); void output(Object obj);}
public class BruteForce {
   public static int find(ItrObj itr, ComProb prob, int bound){
       int num=0;
      do if (prob.target(itr)){ prob.output(itr); ++num; }
      while (num<bound && itr.next());
      return num;
   }
}
```

求解优化问题的通用蛮力算法--Brute-findleast

```
public interface Copiable { Copiable cloneSelf(); void copy(Object obj); }
public interface ComObj extends ItrObj, Copiable {}
public interface OptProb extends ComProb { int val(Object obj); }
public class BruteForce {
   public static Copiable findleast(ComObj itr, OptProb prob){
      Copiable ans=null;  boolean found=false; int lval=0;
      do if (prob.target(itr)){
         ans=itr.cloneSelf(); lval=prob.val(itr); found=true; }
      while (!found && itr.next());
      if (found){
         while (itr.next())
         if (prob.target(itr)){ int v=prob.val(itr);
            if (v<lval){ ans.copy(itr); lval=v; } }
      prob.output(ans); }
      return ans; }
}
```

组合对象的枚举

- 递增序列的枚举：
- 递增序列可以表示子集，比子集的2进制数表示时空效率更高。
- 例如{a1, a2, a3, a4}的全体子集的递增序列表示以字典序排列如下：
- ∅，1， 12， 123，1234，124，13，134，14，2， 23，234，24，3，34，4 。

- 固定长度递增序列的枚举：
- 例如{a1, a2, a3, a4, a5}的全体3-子集递增序列表示以字典序排列如下：
- 123，124，125，134，135，145，234，235，245，345.

- 全排列的枚举：
- 例如4个元素的全排列(或置换)以字典序排列如下：
- 1234，1243，1324，1342，1423，1432，2134，2143，2314，2341，
- 2413，2431，3124，3142，3214，3241，3412，3421，4123，4132
- 4213，4231，4312，4321.

集合划分的枚举

- restricted growth function:
- fm: {1, ..., m}->N-{0}
- fm(1) = 1
- fm(i) <= max{fm(1),..., fm(i - 1)} + 1, (2<=i<=m)

- [1111] {{1,2,3,4}} [1112] {{1,2,3}, {4}} [1121] {{1,2,4}, {3}}
- [1122] {{1,2}, {3,4}} [1123] {{1,2}, {3}, {4}} [1211] {{1,3,4},{2}}
- [1212] {{1,3}, {2,4}} [1213] {{1,3}, {2}, {4}} [1221] {{1,4}, {2,3}}
- [1222] {{1}, {2,3,4}} [1223] {{1}, {2,3}, {4}} [1231] {{1,4},{2},{3}}
- [1232] {{1},{2,4}, {3}} [1233] {{1,2}, {3}, {4}} [1234] {{1}, {2}, {3}, {4}}

例1 子集和数问题：设A为正整数的集合，求A的子集, 使得B的全体元素的和sum满足某个性质(广义)。例如在一个特定的区间[a, b]中，
当区间收缩成一个点[a, a]时，即sum=a(狭义)。

```
public class BinSet implements Copiable { int n; boolean[] set;
   public BinSet(int k){ n=k; set=new boolean[k]; }
   public Copiable cloneSelf(){
        BinSet ans=new  BinSet(n); ans.copy(this);   return ans; }
   public void copy(Object obj){ BinSet ob=(BinSet)obj;
        n=ob.n; for (int i=0; i<n; ++i) set[i]=ob.set[i]; }
        public boolean contains(int i){ return set[i]; }
}
public class LexBinSet extends BinSet implements ComObj {
        public LexBinSet(int k){ super(k); }
        public boolean next(){  boolean hasn=true; int i=0;
                while (i<n && set[i]) set[i++]=false;
                if (i>=n) hasn=false; else set[i]=true;
                return hasn;  }
}
```

子集和数问题的蛮力求解(续)

```
public class BinSumSet implements ComProb {
  int n, sum; int[] data;
  public BinSumSet(int k, int s, int[] d){
       n=k; sum=s; data=d; }
  public boolean target(Object obj){
      int s=0; BinSet ob=(BinSet)obj;
      for (int i=0; i<n; ++i)
        if (ob.contains(i)) s+=data[i];
      return s==sum;
  }
```

子集和数问题的另一种蛮力求解

```
public class ComSet implements Copiable { int n, m; int[] set;
    public ComSet(int k){ n=k; m=0; set=new int[k]; }
    public Copiable cloneSelf(){
        ComSet ans=new ComSet(n); ans.copy(this); return ans;
    }
    public void copy(Object obj){
        ComSet ob=(ComSet)obj;
        m=ob.m; for (int i=0; i<m; ++i) set[i]=ob.set[i];
    }
    public int subcardinal(){ return m; }
    public int content(int i){ return set[i]; }
}
```

子集和数问题的另一种蛮力求解(续)

```
public class LexComSet extends ComSet implements ComObj {
        public LexComSet(int k){ super(k); }
        public boolean next(){  boolean hasn=true;
                if (m==0) { set[0]=1; m=1; }
                else if (set[m-1]!=n){ set[m]=set[m-1]+1; ++m; }
                else if (m==1) hasn=false;
                else {  --m; ++set[m-1]; }
                return hasn;
        }
}
```

子集和数问题的另一种蛮力求解(续)

```
public class ComSumSet implements ComProb {
    int n, sum; int[] data; ComSet itr;
    public ComSumSet(int k, int s, int[] d){
        n=k; sum=s; data=d;
    }
    public boolean target(Object obj){
    int s=0; ComSet ob=(ComSet)obj; int m=ob.subcardinal();
        for (int i=0; i<m; ++i) s+=data[ob.content(i)-1];
        return s==sum;
    }
```

- 例 2: 如何将+,-, *, /分别代入A-D,将1，2，3，4，5，6，7，12分别代入a-h, 使得下面的算式 成立？ (减法和除法的顺序是从上倒下，从左到右)
- a A b = c
- = B
- d e
- C =
- f = g D h

全排列的枚举

```
public class Permutation implements Copiable {
    int n; int[] perm;
    public Permutation(int k){ n=k; perm=new int[k];
        for (int i=0; i<n; ++i) perm[i]=i; }
    public Copiable cloneSelf(){
        Permutation ans=new Permutation(n);
        ans.copy(this); return ans;
    }
    public void copy(Object obj){
        Permutation ob=(Permutation)obj;
        for (int i=0; i<n; ++i) perm[i]=ob.perm[i];
    }
    public int content(int i){ return perm[i]; }
}
```

全排列的枚举(续)

```
public class LexPerm extends Permutation implements ComObj {
   public LexPerm(int k){ super(k); }
   public boolean next(){
      boolean hasn=true; int i=n-1, j=n-1;
      while (i>0 && perm[i-1]>perm[i]) --i;
      if (i==0) hasn=false;
      else { --i;
          while (perm[j]<perm[i]) --j;
         int t=perm[i]; perm[i]=perm[j];
          perm[j]=t;  ++i; j=n-1;
         while (i<j){
             t=perm[i]; perm[i]=perm[j]; perm[j]=t; ++i; --j; }
      }
      return hasn;
   }
}
```

遍历器的嵌套

```
public class ComObjCombine implements ComObj {
    public ComObj x1, x2; Copiable initx1;
    public ComObjCombine(ComObj y1, ComObj y2){
        x1=y1; initx1=y1.cloneSelf(); x2=y2; }
    public Copiable cloneSelf(){
        return new CopiCombine(x1.cloneSelf(), x2.cloneSelf()); }
    public void copy(Object obj){
        ComObjCombine ob=(ComObjCombine)obj;
        x1.copy(ob.x1); x2.copy(ob.x2);  }
    public boolean next(){
        boolean hasn=x1.next();
        if (!hasn){ x1.copy(initx1); hasn=x2.next(); }
        return hasn;    }
}
```

隐式图的通用搜索算法

```
public interface Node {
   boolean target(); int subnum();
   Node down(int i); void output(); }
class Link { public Node prob; public Link next;
    public Link(Node p, Link n){ prob=p; next=n; }
    public void output(){
        if (next!=null) next.output(); prob.output(); }
}
public class GraphSearch {
    private Link head; private int num, bound;
    private Set nodeset;
    public GraphSearch(Node p, int b, Set s) {
        head=new Link(p, null); bound=b;
        nodeset=s; nodeset.add(p); }
```

狭义深度优先搜索

```
public void depthfirst(){
    Node p=head.prob, sub=null;
    int n=p.subnum();
    if (p.target()){ head.output(); ++num; }
    for (int i=0; num<bound && i<n; ++i)
      if ((sub=p.down(i))!=null &&
            !nodeset.contains(sub)){
        head=new Link(sub, head);
        nodeset.add(sub);  depthfirst();
        head=head.next; }
}
```

广义深度优先搜索

```
public boolean depthfirst1(){ LinkedList queue=new LinkedList();
    Link cur=null; queue.addFirst(head);
    while (num<bound && !queue.isEmpty()){
      cur=(Link)queue.removeFirst();
      Node curnode=cur.prob; int m=curnode.subnum();
     for (int i=0; i<m; ++i){ Node subnode=curnode.down(i);
       if (subnode!=null && !nodeset.contains(subnode)){
          Link sub=new Link(subnode, cur);
          queue.addFirst(sub); nodeset.add(subnode);
          if (subnode.target()){
              while (sub!=null){  sub.prob.output(); sub=sub.next;  }
              ++num; }
        } } }
    return num!=0;
 }
```

宽度优先搜索

```
public boolean widefirst(){
   LinkedList queue=new LinkedList(); Link cur=null;
   queue.addLast(head);
   while (num<bound && !queue.isEmpty()){
      cur=(Link)queue.removeFirst();
      Node curnode=cur.prob; int m=curnode.subnum();
      for (int i=0; i<m; ++i){ Node subnode=curnode.down(i);
       if (subnode!=null && !nodeset.contains(subnode)){
          Link sub=new Link(subnode, cur);
          queue.addLast(sub); nodeset.add(subnode);
          if (subnode.target()){ sub.output(); ++num;  }
        } } }
    return num!=0;
  }
```

例3：一个16升的容器中装满了牛奶，另外有一个6升和一个11升的空容器，如何用它们来平分牛奶？

```
public class Cup1  implements Node, Comparable{
    static int a, b, c; int i, j, k;
    public Cup1(int x, int y, int z){ i=x; j=y; k=z; }
    public Cup1(){ k=c; }
    public int compareTo(Object o){ Cup1 obj=(Cup1)o;
        return i<obj.i ? -1 : i>obj.i ? 1 : j<obj.j ? -1 : j>obj.j ? 1 :
                k<obj.k ? -1 : k>obj.k ? 1 : 0;
    }
    public boolean target(){ return i==0 && j==c/2 && k==c/2; }
    public int subnum(){ return 6; }
```

```
public Node down(int x){
if (x==0)
   return i==0 || j==b ? null : i<b-j ? new Cup1(0, j+i, k) : new Cup1(i-b+j, b, k);
if (x==1)
   return i==0 || k==c ? null : i<c-k ? new Cup1(0, j, k+i) : new Cup1(i-c+k, j, c);
if (x==2)
   return j==0 || i==a ? null : j<a-i ? new Cup1(i+j, 0, k) : new Cup1(a, j-a+i, k);
if (x==3)
   return j==0 || k==c ? null : j<c-k ? new Cup1(i, 0, k+j) : new Cup1(i, j-c+k, c);
if (x==4)
   return k==0 || i==a ? null : k<a-i ? new Cup1(i+k, j, 0) : new Cup1(a, j, k-a+i);
if (x==5)
   return k==0 || j==b ? null : k<b-j ? new Cup1(i, j+k, 0) : new Cup1(i, b, k-b+j);
return null;
}

GraphSearch s=new GraphSearch(new Cup1(), 1, new TreeSet());
 s.depthfirst();    s.depthfirst1();   s.widefirst();
```

树的狭义深度优先搜索

```
public void depthfirst(){
    Node p=head.prob, sub=null;
    int n=p.subnum();
    if (p.target()){ head.output(); ++num; }
    for (int i=0; num<bound && i<n; ++i)
      if ((sub=p.down(i))!=null){
        head=new Link(sub, head);
        depthfirst();   head=head.next; }
}
```

树的广义深度优先搜索

```
public boolean depthfirst1(){ LinkedList queue=new LinkedList();
    Link cur=null; queue.addFirst(head);
    while (num<bound && !queue.isEmpty()){
       cur=(Link)queue.removeFirst();
       Node curnode=cur.prob; int m=curnode.subnum();
      for (int i=0; i<m; ++i){ Node subnode=curnode.down(i);
         if (subnode!=null ){
            Link sub=new Link(subnode, cur);
            queue.addFirst(sub);
            if (subnode.target()){
                while (sub!=null){  sub.prob.output(); sub=sub.next;  }
                ++num; }
          } } }
     return num!=0;
 }
```

树的宽度优先搜索

```
public boolean widefirst(){
   LinkedList queue=new LinkedList(); Link cur=null;
   queue.addLast(head);
   while (num<bound && !queue.isEmpty()){
      cur=(Link)queue.removeFirst();
      Node curnode=cur.prob; int m=curnode.subnum();
      for (int i=0; i<m; ++i){ Node subnode=curnode.down(i);
        if (subnode!=null){
           Link sub=new Link(subnode, cur);
           queue.addLast(sub);
           if (subnode.target()){ sub.output(); ++num;  }
        } } }
    return num!=0;
  }
```

子集和数问题的搜索求解

```
public class BinNode implements Node {
   static int n, sum; static int[] data, rsum;
   int level, cursum; boolean isleft;
   public BinNode(int lev, int cs, boolean il){
      level=lev; cursum=cs;  isleft=il; }
   public boolean target(){ return level==n; }
   public int subnum(){ return level<n ? 2 : 0; }
   public Node down(int i){ BinNode ans=null;
        if (i==0 && cursum+rsum[level+1]>=sum)
          ans=new BinNode(level+1, cursum, true);
        if (i==1 && cursum+data[level]<=sum)
          ans=new BinNode(level+1, cursum+data[level], false);
        return ans;
}
```

子集和数问题的另一种搜索求解

```
public class ComNode implements Node {
    static int n, sum; static int[] data, rsum;
    public int curnode; int cursum;
    public ComNode(int cn, int cs){ curnode=cn; cursum=cs; }
    public boolean target(){ return cursum==sum; }
    public int subnum(){ return n-curnode-1; }
    public Node down(int i){ ComNode ans=null;
        int cn=curnode+i+1, cs=cursum+data[cn];
        if (cs<=sum && cs+rsum[cn+1]>=sum)
            ans=new ComNode(cn, cs);
        return ans;
    }
```

基于展开树的搜索

```
public void depthfirst(){
   Node p=head.prob, sub=null; int n=p.subnum();
   if (p.target()){ head.output(true); ++num; }
    for (int i=0; num<bound && i<n; ++i)
      if ((sub=p.down(i))!=null &&
                !nodeset.contains(sub)){
        head=new Link(sub, head);
        nodeset.add(sub); depthfirst();
        head=head.next; nodeset.remove(sub); }
  }
```

例4：跳马问题，在m×n的国际象棋棋盘上，求一个马按移动规则从某一个位置遍历整个棋盘的一个或全体方案。

```
public class Knight1 implements Node, Pair {
    static int m, n;  static int[][]move=
        {{2,1},{1,2},{-1,2},{-2,1},{-2,-1},{-1,-2},{1,-2},{2,-1}};
    static int[][] board;  int level, x, y;
    public Knight1(int lev, int xx, int yy){
        level=lev; x=xx; y=yy; }
     public boolean target(){ return level==m*n; }
     public int subnum(){ return level<m*n ? 8 : 0; }
     public Node down(int i){
         int x1=x+move[i][0], y1=y+move[i][1];
         return x1>=0 && x1<m && y1>=0 && y1<n ?
                new Knight1(level+1, x1, y1) : null;
}
```

通用回溯算法

```
public interface Mono{ boolean target(); int subnum();
    boolean down(int i); void up(); void output(); }
public class BackSearch { Mono prob; int num, bound;
  public BackSearch(Mono p, int b) { prob=p; bound=b; }
  public void depthfirst(){ int n=prob.subnum();
    if (prob.target()){ prob.output(); ++num; }
    for (int i=0; num<bound && i<n; ++i)
    if (prob.down(i)){ depthfirst(); prob.up(); }
  }
}
```

子集和数问题的回溯求解

```
public class BinMono implements Mono {
   int n, sum, level, cursum; int[] data, rsum; boolean[] set;
   public BinMono(int nn, int ss, int[] d, int[] rs){
      n=nn; sum=ss; data=d; rsum=rs; set=new boolean[nn]; }
   public boolean target(){ return level==n; }
   public int subnum(){ return level<n ? 2 : 0; }
   public boolean down(int i){
      boolean ans = i==0 && cursum+rsum[level+1]>=sum ||
                    i==1 && cursum+data[level]<=sum;
      if (ans){ set[level] = i!=0;
         if (i!=0) cursum+=data[level];  ++level; }
      return ans;    }
   public void up(){ --level;
      if (set[level]) cursum-=data[level]; }
```

子集和数问题的另一种回溯求解

```
public class ComMono implements Mono {
    int n, sum, level, cursum; int[] data, rsum, set;
    public ComMono(int nn, int ss, int[] d, int[] rs){
        n=nn; sum=ss; data=d; rsum=rs;set=new int[nn+1]; set[0]=-1; }
    public boolean target(){ return cursum==sum; }
    public int subnum(){ return n-set[level]-1; }
    public boolean down(int i){ int t=set[level]+i+1;
        boolean ans = cursum+data[t]<=sum && cursum+rsum[t]>=sum;
        if (ans){ set[++level]=t; cursum+=data[t]; }
        return ans;
    }
    public void up(){ cursum-=data[set[level--]]; }
```

例5：N皇后问题的回溯求解

```
public class Queen1 implements Mono {
    int n, level; int[] board;
    public Queen1(int k){ n=k; board=new int[n]; }
    public boolean target(){ return level==n; }
    public int subnum(){ return level<n ? n : 0; }
    public boolean down(int i){ boolean q=true;
        for (int j=0; q && j<level; ++j)
        q=board[j]!=i && Math.abs(board[j]-i)!=Math.abs(j-level);
        if (q) board[level++]=i;  return q;
    }
    public void up(){ --level; }
```

N皇后问题的另一种回溯求解

```
public class Queen2 extends Queen1 {
    boolean[] column, d1, d2;
    public Queen2(int k){ super(k); column=new boolean[n];
        d1=new boolean[2*n-1]; d2=new boolean[2*n-1]; }
    public boolean down(int i){
        if (!column[i]&&!d1[level+i] && !d2[level-i+n-1]){
            column[i]=d1[level+i]=d2[level-i+n-1]=true;
            board[level++]=i; return true; }
        return false; }
     public void up(){ int i=board[--level];
        column[i]=d1[level+i]=d2[level-i+n-1]=false;  }
```

排列树的回溯求解

```
public class PerBackSearch {
    Mono prob; int dep, level; int num, bound; int[] perm;
    public PerBackSearch(Mono p, int b, int k) {
        prob=p; bound=b; dep=k;
        perm=new int[dep]; for (int i=0; i<dep; ++i) perm[i]=i; }
    public void depthfirst(){
        if (prob.target()){ prob.output(); ++num; }
        else for (int i=level; num<bound && i<dep; ++i)
            if (prob.down(perm[i])){
                int t=perm[level]; perm[level]=perm[i]; perm[i]=t;
                ++level; depthfirst(); prob.up(); --level;
                t=perm[level]; perm[level]=perm[i]; perm[i]=t; }
        }
```

基于遍历器的回溯算法

```
public interface ItrMono1 {
      boolean target(); ItrObj subnodes();
      boolean down(ItrObj itr);
      void up(); void output();
}
public class ItrBackSearch1 {
    ItrMono1 prob; int num, bound;
public ItrBackSearch1(ItrMono1 p, int b) {
    prob=p; bound=b; }
```

基于遍历器的回溯算法(续)

```
public void depthfirst(){
    ItrObj nodes=prob.subnodes();
    boolean hasnext=true;
    if (prob.target()){ prob.output(); ++num; }
    while (num<bound && nodes!=null &&
           hasnext){
        if (prob.down(nodes)){
            depthfirst(); prob.up(); }
        hasnext=nodes.next(); }
  }
```

例6 0-1矩阵问题：求一个0-1矩阵，使每行的和与每列的和都等于给定的数值。

更准确地说，令$M = \{0, \ldots, m-1\}$，$N = \{0, \ldots, n-1\}$，

$row: M \to \omega$, $col: N \to \omega$, 求函数$f: M \times N \to \{0, 1\}$, 使得对每个$i \in M$,

$\Sigma\{f(i, k) \mid k \in N\} = row(i)$, 对每个$j \in N$, $\Sigma\{f(k, j) \mid k \in M\} = col(j)$。

```
public class LexIncSeq  extends ComSet implements ComObj {
    public LexIncSeq(int mm, int nn){ super(nn); m=mm;
        for (int i=0; i<n; ++i) set[i]=i;   }
    public boolean next(){ int i=n-1;
        while (i>=0 && set[i]==m-n+i) --i;
        if (i>=0){ ++set[i];
           for (int j=i+1; j<n; ++j) set[j]=set[j-1]+1;
        }
        return i>=0;
     }
}
```

```
public class BitMatrix4 implements ItrMono1 {
    int m, n; int[] row, column; int[][] data;
    int[]  count; int level;
    public BitMatrix4(int i, int j, int[] r, int[] c){
        m=i; n=j; row=r; column=c; fout=f;
        data=new int[m][n]; count=new int[n]; }
    public boolean target(){ return level==m; }
    public ItrObj subnodes(){
        return
        level<m ? new LexIncSeq(n, row[level]) : null;
    }
```

```
public boolean down(ItrObj itr){
    boolean res=true;  LexIncSeq com=(LexIncSeq)itr;
    Arrays.fill(data[level], 0, n, 0);
    for (int j=0; j<row[level]; ++j) data[level][com.content(j)]=1;
    for (int i=0; res && i<n; ++i)
       res=data[level][i]==1?count[i]<column[i]:
                  level-count[i]<m-column[i];
       if (res){
         for (int i=0; i<n; ++i) count[i]+=data[level][i]; ++level; }
    return res;
}
public void up(){ --level;
       for (int i=0; i<n; ++i) count[i]-=data[level][i]; }
```

赋值树的深度优先搜索

```
public interface Valued { int val(); void setval(int k); }
public interface Check { boolean target();  void output();
public interface WNode extends Check, Valued {
   int subnum(); WNode down(int i); }
public class WLink implements Comparable {
   public WNode prob; public WLink next;
   public WLink(WNode p, WLink n){ prob=p; next=n; }
   public void output(){ if (next!=null) next.output(); prob.output(); }
   public Object cloneself(){
        return next==null ? new WLink(prob, null) :
             new WLink(prob, (WLink)next.cloneself()); }
   public int compareTo(Object obj){
      return prob.val()-((WLink)obj).prob.val(); }
}
```

赋值树的深度优先搜索(续)

```
public class TreeOptBruteSearch {
   WLink head, ans; WNode node, lnode;
   public TreeOptBruteSearch(WNode p){
        head=new WLink(p, null); node=p; }
   public void depthfirst(){
        WNode p=head.prob, sub=null;
        if (p.target())
           if (lnode==null || p.val()<lnode.val()){
              ans=(WLink)head.cloneself(); lnode=p; }
        int n=p.subnum();
        for (int i=0; i<n; ++i)
           if ((sub=p.down(i))!=null){
              head=new WLink(sub, head);
              depthfirst(); head=head.next; }
        }
```

优化问题的回溯算法

```
public interface WMono extends Mono, Valued {
   Object cloneself(); }
public class BackOptBruteSearch  {
   private WMono prob, ans;
   public BackOptBruteSearch(WMono p) { prob=p; }
   public void depthfirst(){
        if (prob.target())
           if (ans==null || prob.val()<ans.val())
              ans=(WMono)prob.cloneself();
        int n=prob.subnum();
        for (int i=0; i<n; ++i)
           if (prob.down(i)){
              depthfirst(); prob.up(); }
    }
```

- 例7 连续邮资问题：假设国家发行了n种不同面值的邮票，并且规定每张信封上最多只允许贴m张邮票。连续邮资问题要求对于给定的n和m的值，给出邮票面值的最佳设计，在一张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。例如，n=5,m=4时，面值为(1, 3, 11, 15, 32)的5种邮票可以贴出邮资的最大连续区间是1-70.

```
public class Stamp1  implements  WNode {
      static byte m, n; static int maxv=2000;
      byte[] least=new byte[maxv+1];
      int name, level, curv;
      public Stamp1(int n, int lev){
          name=n; level=lev; }
      public boolean target(){ return level==n; }
      public int subnum(){
          return level<n ? curv-name+1 : 0; }
```

```
public  WNode down(int i){
   Stamp1 ans=new Stamp1(name+i+1, level+1);
   if (level==0){ Arrays.fill(ans.least, (byte)(m+1)); ans.least[0]=0; }
   else System.arraycopy(least, 0, ans.least, 0, maxv);
   for (int j=0; j<=Math.min(maxv-1, (m-1)*name); ++j)
      if (ans.least[j]<m)
        for (byte k=1; k<=m-ans.least[j]; ++k){
          int loc=Math.min(maxv-1, j+k*ans.name);
          ans.least[loc]=ans.least[loc]<(byte)(ans.least[j]+k) ?
                                  ans.least[loc] : (byte)(ans.least[j]+k); }
        int j=curv+1;
        while(j<=maxv && ans.least[j]<=m) ++j;
        ans.curv=j-1; return ans;
  }
   public  int val(){  return -curv; }
```

例8：0-1背包问题，给定n种物品和一个背包，每个物品具有特定重量和价值，背包具有最大负荷重量，如何向背包中放入物品使得总重量不超过负荷且它们的总价值达到最大。

```
public class KnapBestMonoBin implements WMono {
    int level, cval, cw; boolean[] set;
    public KnapBestMonoBin(){
        set=new boolean[Knap.n]; }
    public KnapBestMonoBin
            (boolean[] d, int w, int v){
        set=d; cw=w; cval=v; }
    public boolean target(){
        return level==Knap.n && cw<=Knap.wlimit; }
    public int subnum(){ return level<Knap.n ? 2 : 0; }
```

```
public boolean down(int i){
    boolean ans = i==0 || i==1 &&
                              cw+Knap.weight[level]<=Knap.wlimit;
    if (ans){ set[level] = i!=0;
        if (i!=0){
            cw+=Knap.weight[level];
            cval+=Knap.value[level]; }
        ++level; }
        return ans;
    }
    public void up(){ --level;
        if (set[level]){ cw-=Knap.weight[level];
        cval-=Knap.value[level]; }
    }
public  int val(){  return -cval; }
```

单调赋值树的深度优先搜索

```
public class TreeOptSearch {
   WLink head, ans; WNode node, lnode;
   public TreeOptSearch(WNode p){
        head=new WLink(p, null); node=p; }
   public void depthfirst(){
        WNode p=head.prob, sub=null;
          if (lnode==null || p.val()<lnode.val())
            if (p.target()){
              ans=(WLink)head.cloneself(); lnode=p; }
            else { int n=p.subnum();
              for (int i=0; i<n; ++i)
                if ((sub=p.down(i))!=null){
                head=new WLink(sub, head);
                              depthfirst(); head=head.next; } } }
```

单调赋值树的宽度优先搜索

```
public class TreeOptSearch {
   WLink head, ans; WNode node, lnode;
   public TreeOptSearch(WNode p){
      head=new WLink(p, null); node=p; }
   public void optfirst1(){
      PriorityQueue<WLink> queue=new PriorityQueue<WLink>();
      queue.add(new WLink(node, null));
      while (queue.size()>0){
        WLink cur=queue.poll(); WNode prob=cur.prob;
        if (prob.target()) { ans=cur; return; }
        else { int n=prob.subnum(); WNode sub=null;
        for (int i=0; i<n; ++i)
           if ((sub=prob.down(i))!=null)
              queue.add(new WLink(sub, cur)); } } }
```

单调优化问题的回溯算法

```
public class BackOptSearch { private WMono prob, ans;
    public BackOptSearch(WMono p) { prob=p;  }
    public void depthfirst(){
      if (ans==null || prob.val()<ans.val())
        if (prob.target()) ans=(WMono)prob.cloneself();
        else { int n=prob.subnum();
          for (int i=0; i<n; ++i)
            if (prob.down(i)){
              depthfirst(); prob.up(); }  }  }
```

0-1背包问题的单调搜索

```
public class KnapLeastNodeCom implements WNode {
    public int curnode; int cval, cw;
    public KnapLeastNodeCom(int cn, int cv, int w){
        curnode=cn; cval=cv; cw=w; }
    public boolean target(){
        return cw>=Knap.wall-Knap.wlimit; }
    public int subnum(){ return Knap.n-curnode-1; }
    public WNode down(int i){
        int cn=curnode+i+1, w=cw+Knap.weight[cn],
        v=cval+Knap.value[cn];
        return new KnapLeastNodeCom(cn, v, w); }
    public  int val(){  return cval; }
```

0-1背包问题的单调回溯

```
public class KnapLeastMonoBin implements WMono {
        int level, cval, cw; boolean[] set;
        public KnapLeastMonoBin(){
                set=new boolean[Knap.n]; }
        public KnapLeastMonoBin(boolean[] d, int w, int v){
                set=d; cw=w; cval=v; }
        public boolean target(){
                return level==Knap.n && cw>=Knap.wall-
Knap.wlimit; }
        public int subnum(){
                return level<Knap.n ? 2 : 0; }
```

0-1背包问题的单调回溯(续)

```
public boolean down(int i){ set[level] = i!=0;
    if (i!=0){
        cw+=Knap.weight[level];
        cval+=Knap.value[level]; }
        ++level; return true;   }
public void up(){ --level;
    if (set[level]){
        cw-=Knap.weight[level];
        cval-=Knap.value[level]; }
}
public  int val(){  return cval; }
```