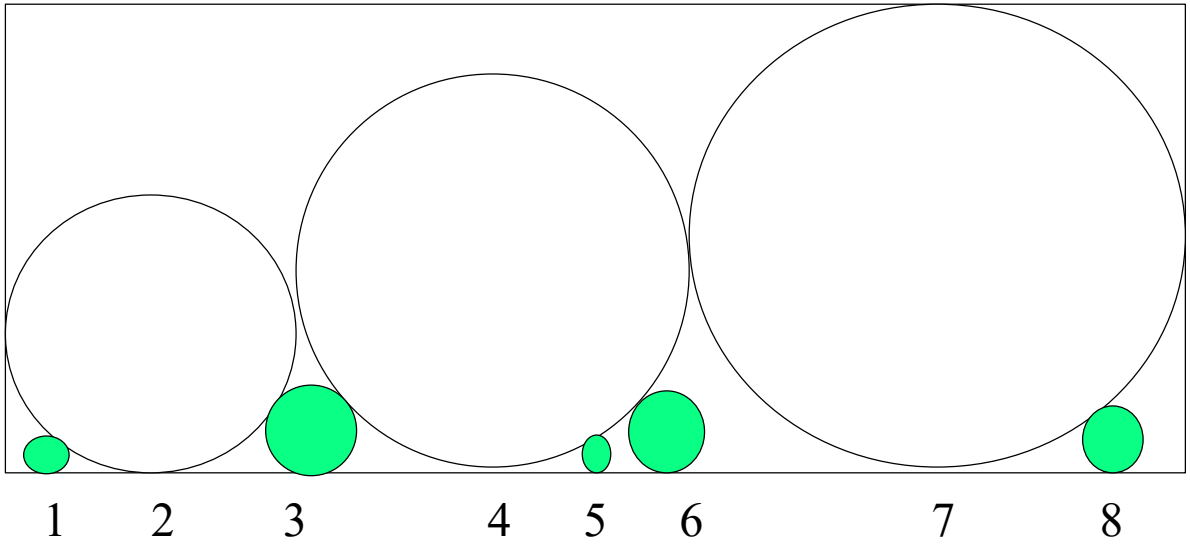
圆排列问题

给定n个大小不等的圆c1,c2,…,cn，现要将这n个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。

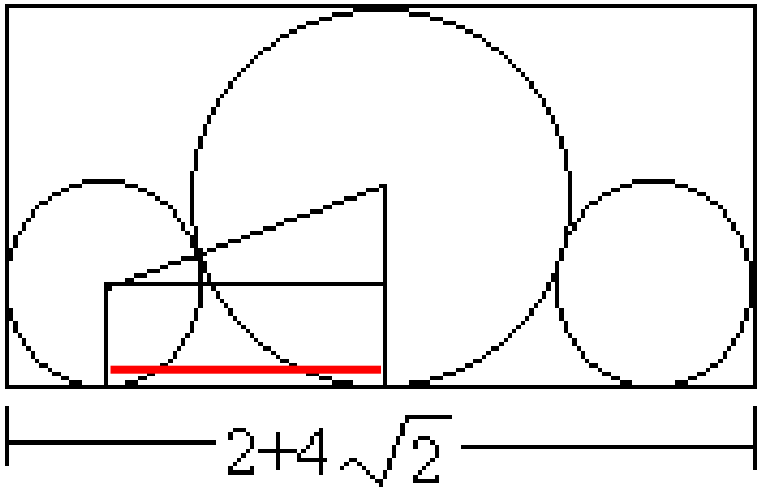
圆排列问题要求从n个圆的所有排列中找出有最小长度的圆排列。

长度最小的圆排列，必有任何一个圆都与前面某一个圆相切。



圆排列问题

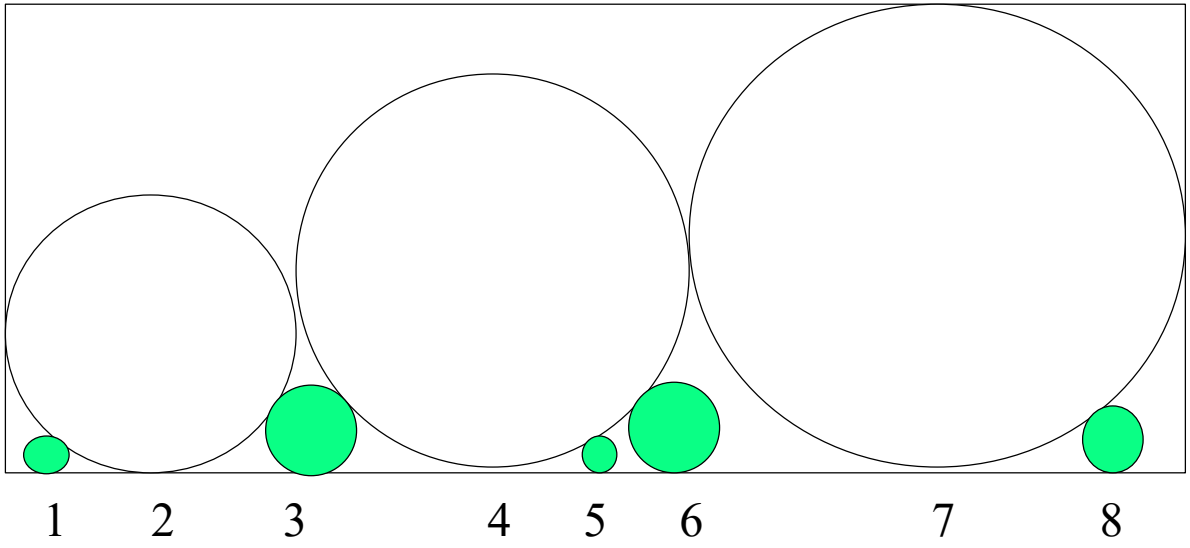
例如，当n=3，且所给的3个圆的半径分别为1，1，2时，这3个圆的最小长度的圆排列如图所示。红色直线的长度为$\sqrt{(2+1)^2-(2-1)^2}=2\sqrt{2}$，所以这三个圆的排列长度为 $2+4\sqrt{2}$ 。



圆排列问题

计算圆心坐标：

设第一个圆的圆心坐标为0，从左至右依次计算各个圆的圆心坐标。假设已计算得到圆1至i-1的圆心坐标，需计算圆i的圆心坐标。

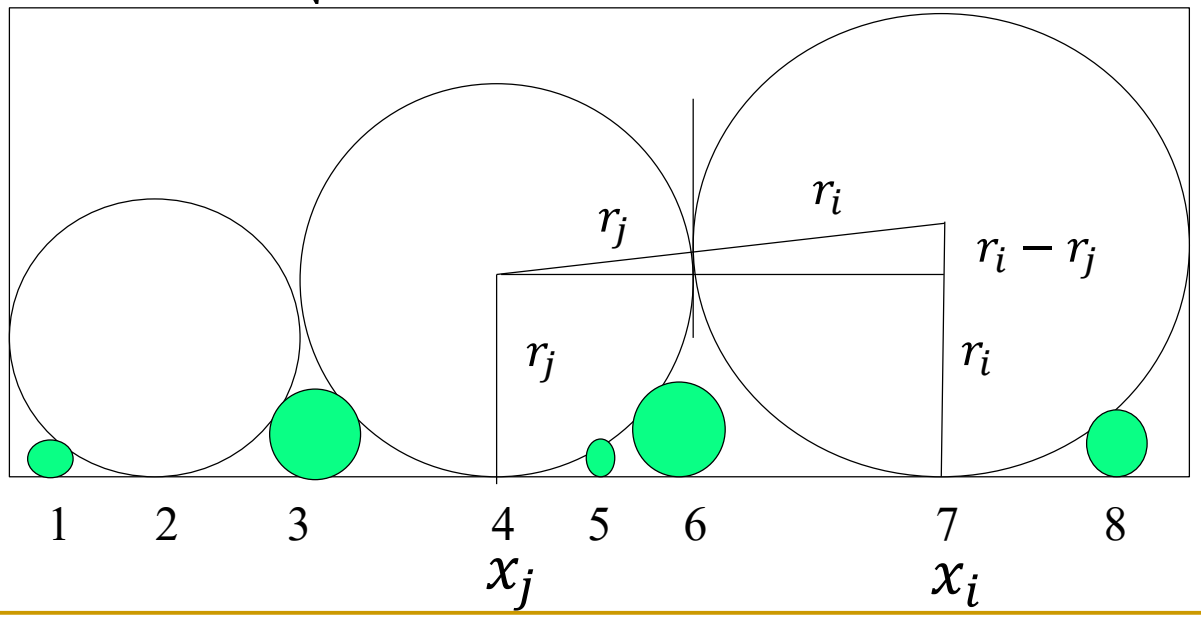


圆排列问题

计算圆心坐标：

假设圆i与圆j相切，圆i与圆j的半径分别为r_i与r_j，圆j的圆心坐标为x_j，则圆j的圆心坐标

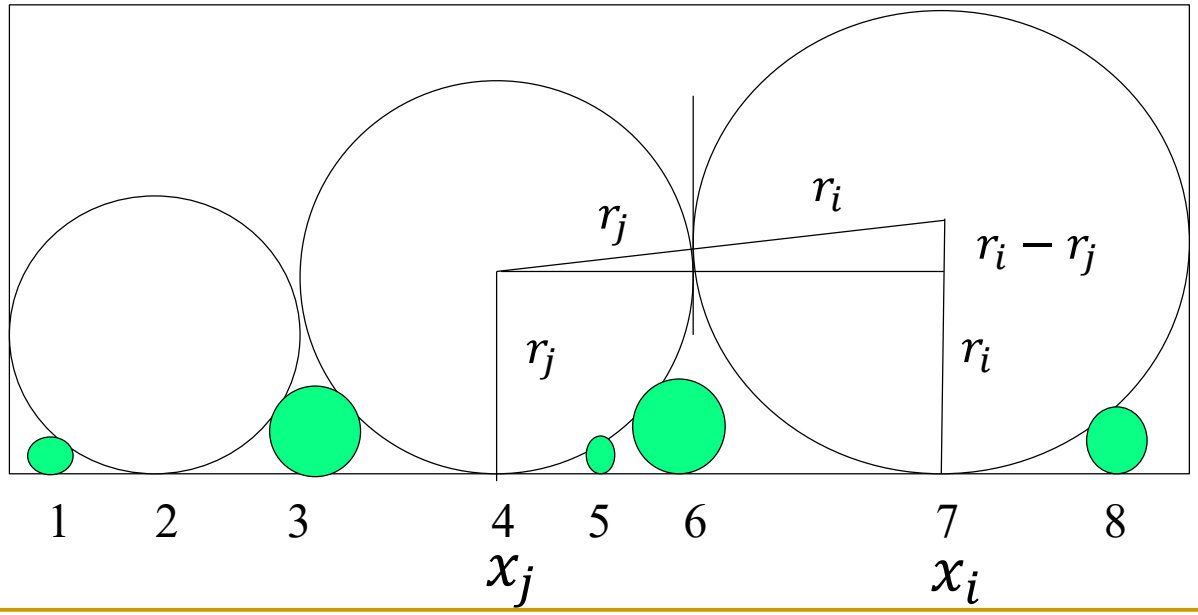
$$x_i = x_j + \sqrt{(r_i + r_j)^2 - (r_i - r_j)^2} = x_j + 2\sqrt{r_i * r_j}$$



圆排列问题

计算圆心坐标：

圆i可能与之前排列1至i-1的的任意一个圆相切（如圆7与圆4相切）。根据与圆i真实相切的圆计算得到的圆心坐标是最大的，该最大值便是圆i的圆心坐标。



圆排列问题

```
private static void backtrack(int t){
    if (t>n) // 计算当前圆排列的长度
        compute();
    else
        for (int j = t; j <= n; j++) {
            swap(r, t, j);
            // 计算圆心横坐标
            float centerx=center(t);
            if (centerx+r[t]+r[1]<min)
            //下界约束
                x[t]=centerx;
                backtrack(t+1);
            swap(r, t, j);
}
```

backtrack(1)

```
private static float center(int t){
    float temp=0;
    for (int j=1;j<t;j++)//圆t可能与之前任一圆相切
        valuex=x[j]+2.0*sqrt(r[t]*r[j]);
        if (valuex>temp)   temp=valuex;
    return temp;
}
```

```
private static void compute(){
    float low=0, high=0;
    for (int i=1;i<=n;i++)
        if (x[i]-r[i]<low)
            low=x[i]-r[i];
        if (x[i]+r[i]>high)
            high=x[i]+r[i];
    if (high-low<min)
        min=high-low;
}
```

求出所有圆的左右两侧坐标，找出最小的左侧坐标和最大的右侧坐标，相减就是圆排列的长度

圆排列问题

复杂度分析

由于算法**backtrack**在最坏情况下可能需要计算O(n!)次当前圆排列长度，每次计算需O(n)计算时间，从而整个算法的计算时间复杂性为O((n+1)!) 。

上述算法尚有许多改进的余地：

- 像1,2,…,n-1,n和n,n-1, …,2,1这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量。
- 如果所给的n个圆中有k个圆有相同的半径，则这k个圆产生的k!个完全相同的圆排列，只计算一个就够了。

连续邮资问题

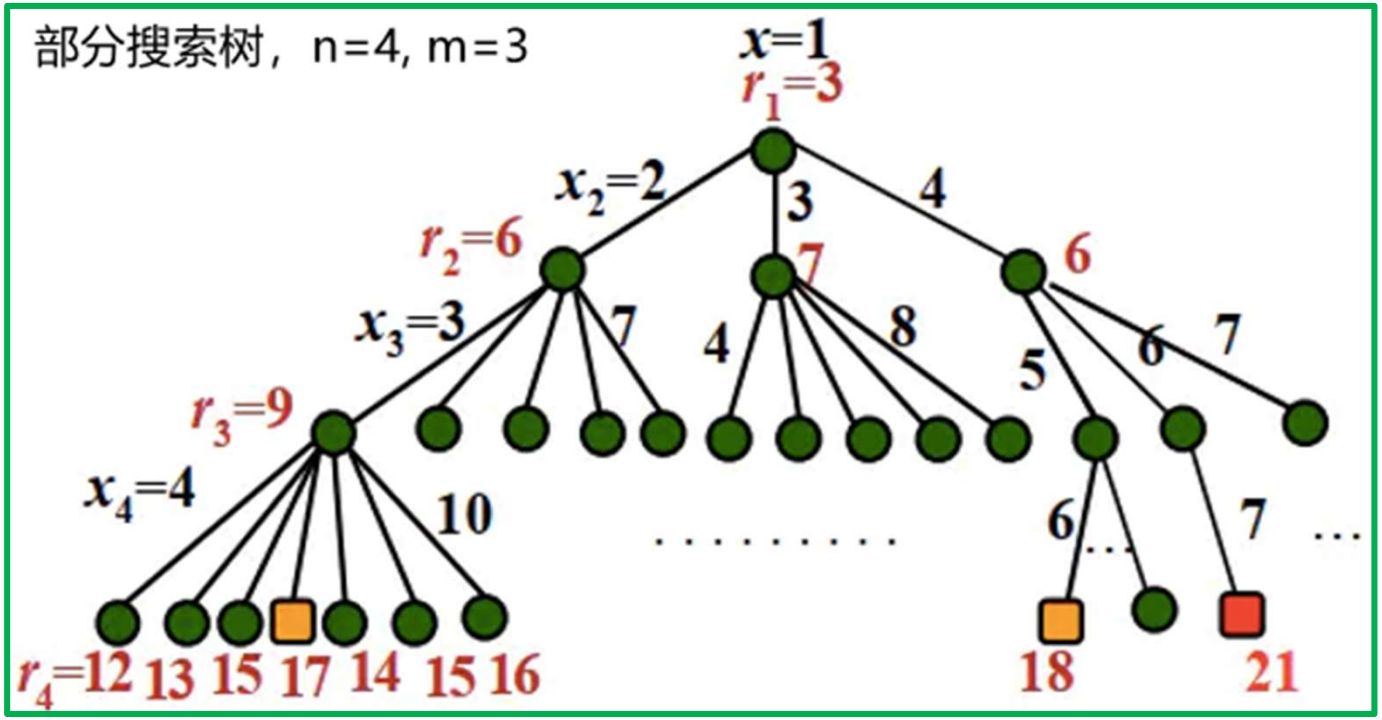
- 假设国家发行了n种不同面值的邮票，并且规定每张信封上最多只允许贴m张邮票。
- 连续邮资问题要求对于给定的n和m的值，给出邮票面值的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

例如1，当n=5和m=4时，面值为 (1,3,11,15,32) 的5种邮票使用最多4张可以贴出邮资的最大连续邮资区间是1到70。

例如2，当n=5和m=4时，面值为 (1,6,10,20,30) 的5种邮票使用最多4张可以贴出邮资的最大连续邮资区间是1到4 。

连续邮资问题

- 解向量：用n元组x[1:n]表示n种不同的邮票面值，并约定它们从小到大排列。x[1]=1是惟一的选择。
- 可行性约束函数：已选定x[1:i-1]，最大连续邮资区间是[1:r]，接下来x[i]的可取值范围是？ [x[i-1]+1:r+1]



连续邮资问题

如何确定r的值?

- 计算x[1:i]的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法，而直接递归的求解复杂度太高。
- 尝试计算用不超过m张面值为x[1:i]的邮票贴出邮资k所需的最少邮票数y[k]。通过y[k]可以很快推出r的值。事实上，y[k]可以通过递推在O(n)时间内解决。

连续邮资问题

如何确定r的值?

$y_i(j)$：用至多m张面值xi的邮票加上x_1，x_2，…，x_{i-1}面值的邮票贴j邮资时的最少邮票数，则

$$y_i(j) = \min_{0 \le t \le m} \{t + y_{i-1}(j - tx_i)\}$$

$$y_1(j) = j$$

$$r_i = \min\{j | yi(j) \le m,\ y_i(j+1) > m\}$$

连续邮资问题

```
void backtrack(int i, int r){
for (int j=0; j<= x[i-2]*(m-1);j++) //通过加入k个x[i-1]对x[1:i-2]的最
                  //大值进行更新，从而获得x[1:i-1]的最大值，以获得x[i]的取值范围上界。
    if (y[j]<m)
       for (int k=1;k<=m-y[j];k++) //k是对表示j剩余的票数进行检查
           if (y[j]+k<y[j+x[i-1]*k])//x[i-1]*k是k张邮票能表示的最大邮资
                              //+j表示增加了i-1邮资后能判断新增加的能表示的邮资需要多少
              y[j+x[i-1]*k]=y[j]+k; //对第i-2层扩展一个x[i-1]后的邮资分布
while (y[r]<maxint) //查看邮资范围扩大多少，然后查询y数组从而找到r
         r++;
```

$$y_i(j) = \min_{0 \le t \le m} \{t + y_{i-1}(j - tx_i)\}$$

$$y_1(j) = j$$

$$r_i = \min\{j | y_i(j) \le m,\ y_i(j+1) > m\}$$

连续邮资问题

```
void backtrack(int i, int r){//续
     if(i>n)
        if(r-1>maxvalue)
           maxvalue = r – 1;
           for(int j=1;j<=n;j++)
                bestx[j] = x[j]
        return;
     for(int j=0;j<=maxvalue;j++)
         z[k] = y[k];
     for(int j=x[i-1]+1;j<=r;j++) //在第i层有这么多的子结点供选择
           x[i] = j;
           backtrack(i+1, r);
           for(int k=1;k<=maxvalue;k++)
                 y[k] = z[k];
```

回溯法效率分析

通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

(1)产生x[k]的时间；

(2)满足显约束的x[k]值的个数；

(3)计算约束函数**constraint**的时间；

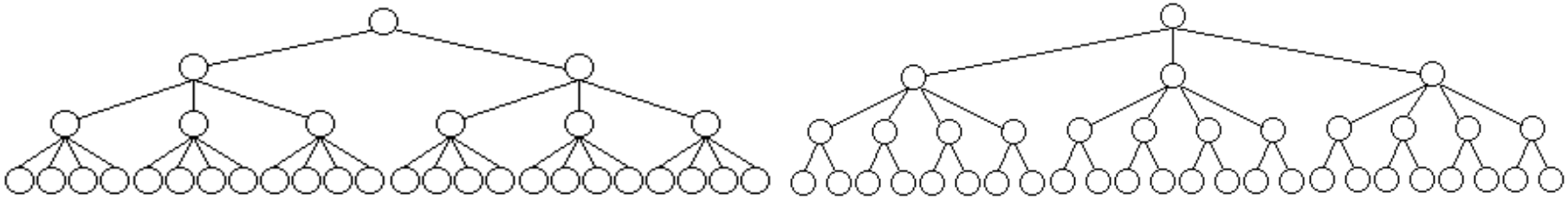
(4)计算上界函数**bound**的时间；

(5)满足约束函数和上界函数约束的所有x[k]的个数。

好的剪枝函数能显著地减少所生成的结点数。但这样的剪枝函数往往计算量较大。因此，在选择剪枝函数时通常存在生成结点数与剪枝函数计算量之间的折衷。(3)(4)&(5)

重排原理

对于许多问题而言，在搜索试探时选取x[i]值的顺序是任意的。在其他条件相当的前提下，让可取值最少的x[i]优先。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



(a)　　(b)

图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

小结

- 适合求解组合搜索问题及优化问题
- 解的表示：
  - 回溯法解决的问题，其解一定可以表示成n元组解向量的形式
  - 求解是不断扩充解向量的过程
- 确定易于求解的解空间树。
  - 常用:子集树、排列树、n叉树等
  - 一个问题有可能对应二种或多种解空间树，评价的标准:最后一层叶子结点的数量越少越好。

小结（续）

- 剪枝函数的设计：在保证剪枝效率的情况下计算尽量简单。
  - 对于子集树，通常左分支使用约束函数、右分支使用限界函数剪枝。
  - 对于排列树或n叉树，这两个函数同时应用于每个分支。
- 回溯法的设计：

  一般在递归调用backtrack函数前、后的代码动作是相反的。

小结（续）

- 算法时间复杂度:

$$W(n) = \left(p(n)f(n)\right)$$

$p(n)$每个结点的工作量

$f(n)$为结点个数

最坏情况下时间通常为指数级或者阶乘级

平均情况下比穷举算法好

空间代价小