最优子结构的定义：如果一个问题的最优解包含其子问题的最优解，则该问题具备最优子结构性质。

具备最优子结构的条件：分解性：可以将问题分解为多个子问题。无后效性：子问题的解不会受到其他决策的影响，即子问题的最优解不会因为其他部分的决策而改变。

相同点：递归子结构，将待求解的问题分解成若干个规模较小的相同类型的子问题，先求解子问题，然后从子问题中的出原问题的解。

不同点：重叠子问题，适用于动态规划求解的问题，经分解得到的子问题往往不是互相独立的。若用分治法来解决这类问题，则分解得到的子问题数目太多，有些子问题被重复计算。

不同点：求解问题顺序。分治自顶向下，动态规划自底向上。

分治法问题的特征

1. 规模缩小到的一定程度可以解决
2. 具有最优子结构性质
3. 分解出的子问题可以合并为问题的解
4. 各个子问题相互独立

主定理法

$$T(n)=\begin{cases} c & n=1 \\ aT\left(\frac{n}{b}\right)+cn & n>1 \end{cases}$$

$$T(n)=\begin{cases} \Theta(n) & a<b \\ \Theta(n\log_b n) & a=b \\ \Theta\left(n^{\log_b a}\right) & a>b \end{cases}$$

设 $a \ge 1$ 和 $b > 1$ 为常数，设 $f(n)$为一函数，$T(n)$由递归式

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

其中$\frac{n}{b}$指$\left\lfloor\frac{n}{b}\right\rfloor$和$\left\lceil\frac{n}{b}\right\rceil$，可以证明，略去上下去整不会对结果造成影响。那么 $T(n)$可能有如下的渐进界

(1)若 $f(n) < n^{\log_b^a}$，且是多项式的小于。即

$\exists\, \varepsilon > 0$，有 $f(n) = O\left(n^{\log_b^a - \varepsilon}\right)$，则 $T(n) = \Theta\left(n^{\log_b^a}\right)$

(2)若 $f(n) = n^{\log_b^a}$，则 $T(n) = \Theta\left(n^{\log_b^a}\log n\right)$

(3)若 $f(n) > n^{\log_b^a}$，且是多项式的大于。即

$\exists\, \varepsilon > 0$，有 $f(n) = \Omega\left(n^{\log_b^a + \varepsilon}\right)$，且对$\forall\, c < 1$与所有足够大的 n，有 $af\left(\frac{n}{b}\right) \le cf(n)$，则 $T(n) = \Theta(f(n))$

大整数乘法

- $XY = (a*c)\cdot 10^n + (a*d+b*c)\cdot 10^{n/2} + b*d$
- 变换：$a*d+b*c = (a+b)*(c+d) - (a*c) - (b*d)$
- $XY = (a*c)\cdot(10^n - 10^{n/2}) + (a+b)*(c+d)\cdot 10^{n/2} + (b*d)\cdot(1 - 10^{n/2})$
- 4次乘法减为3次乘法

- $XY = a*c\cdot 10^n + ((a-c)*(b-d)+a*c+b*d)\cdot 10^{n/2} + b*d$
- $XY = a*c\cdot 10^n + ((a+c)*(b+d)-a*c-b*d)\cdot 10^{n/2} + b*d$

细节问题：两个XY的复杂度都是O(nlog3)，但考虑到a+c,b+d可能得到m+1位的结果，使问题的规模变大，故不选择第2种方案。

棋盘覆盖

将这 3 个无特殊方格的子棋盘转化为特殊棋盘，可以用一个 L 型骨牌覆盖这 3 个较小棋盘的会合处。从而将原问题转化为 4 个较小规模的棋盘覆盖问题。

时间复杂度分析

$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

设M(k)为chessBoard算法在计算覆盖一个$2^k \times 2^k$棋盘所需时间:

当k>1时，M(k)=4M(k-1)，　M(0)=1

M(k)=4M(k-1)　　　替换M(k-1)=4M(k-2)

$=4[4M(k-2)]=4^2 M(k-2)$

$=\ldots\ldots$

$=4^k M(k-k) = 4^k$

$T(K)=O(4^K)$

30

归并排序

```
void bin_sort(int a[], int l, int r) {
    if (r - l <= 1) { // 如果区间长度小于等于 1，则直接返回
        return;
    }
    int mid = (l + r) / 2;
    bin_sort(a, l, mid);   // 排序左半部分
    bin_sort(a, mid, r);   // 排序右半部分
    int i = l, j = mid, k = 0;
    while (i < mid && j < r) { // 合并两个有序数组
        if (a[i] <= a[j]) {
            temp[k++] = a[i++];
        } else {
            temp[k++] = a[j++];
        }
    }
    while (i < mid) temp[k++] = a[i++]; // 将剩余的左半部分加入
    while (j < r) temp[k++] = a[j++];  // 将剩余的右半部分加入
    for (int i = 0; i < k; ++i) { // 将排序后的结果复制回原数组
        a[l + i] = temp[i];
    }
}
```

快速排序

```
void quickSort(int a[], int l, int r){
    //如果数组中就一个数，就已经排好了，直接返回。
    if(l >= r) return;
    //选取分界线。这里选数组中间那个数
    int x = a[l + r >> 1];
    int i = l - 1, j = r + 1;
    //划分成左右两个部分
    while(i < j){
        while(a[++i] < x);
        while(a[--j] > x);
        if(i < j){
            swap(a[i], a[j]);
        }
    }
    //对左部分排序
    quickSort(a, l, j);
    //对右部分排序
    quickSort(a, j + 1, r);
}
```

线性时间元素选择

```
int findK(int a[], int l, int r, int k){
    if(l >= r) return a[l];
    int x = a[l + r >> 1];
    int i = l - 1, j = r + 1;
    while(i < j){
        while(a[++i] < x);
        while(a[--j] > x);
        if(i < j){
            swap(a[i], a[j]);
        }
    }
    if(j + 1 >= k)
        return findK(a, l, j, k);
    else
        return findK(a, j + 1, r, k);
}
```

循环赛日程表

最长公共子序列

$$c[i][j]=\begin{cases} 0 & i=0, j=0 \\ c[i-1][j-1]+1 & i,j>0; x_i = y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i,j>0; x_i \neq y_j \end{cases}$$

如果当前的$A1[i]$和$A2[j]$相同（即是有新的公共元素） 那么

$$dp[i][j] = max(dp[i][j], dp[i-1][j-1]+1);$$

如果不相同，即无法更新公共元素，考虑继承：

$$dp[i][j] = max(dp[i-1][j], dp[i][j-1]$$

```
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
     {
     dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
     if(a1[i]==a2[j])
     dp[i][j]=max(dp[i][j],dp[i-1][j-1]+1);
     //因为更新，所以++；
     }
```

最长上升子序列

```
for(int i=1;i<=n;i++)
     {
          dp[i]=1;//初始化
          for(int j=1;j<i;j++)//枚举 i 之前的每一个 j
          if(data[j]<data[i] && dp[i]<dp[j]+1)
          //用 if 判断是否可以拼凑成上升子序列，
          //并且判断当前状态是否优于之前枚举
          //过的所有状态,如果是，则↓
          dp[i]=dp[j]+1;//更新最优状态


     }
```

矩阵连乘

$$m[i,j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\} & i<j \end{cases}$$

```
void  MatricChain(int *p, int n, int **m, int **s)
{   for(i=1;i<=n; ++i)  m[i][i]=0; //单个矩阵无计算
    for(r=2; r<=n; ++r)   //连乘矩阵的个数
      for(i=1; i<n-r; ++i)
      {   j=i+r-1;
          m[i][j]=m[i][i]+m[i+1][j]+p[i-1]*p[i]*p[j];
          s[i][j]=i;
          for(k=i+1; k<j; ++k)
          {  t=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
             if(t<m[i][j]) { m[i][j]=t;  s[i][j]=k;
}    }  }
```

算法复杂度分析：

标价函数 s，记录从哪个开始分割的

01 背包

$$m(i,j)=\begin{cases} \max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i \end{cases}$$

```
for (int i = 1; i <= n; i++)
  for (int j = W; j >= w[i]; j--)
    f[j] = max(f[j], f[j - w[i]] + v[i]);
```

逆序对

```
void msort(int b,int e)//归并排序
{
    if(b==e)
         return;
    int mid=(b+e)/2,i=b,j=mid+1,k=b;
    msort(b,mid),msort(mid+1,e);
    while(i<=mid&&j<=e)
      if(a[i]<=a[j])
         c[k++]=a[i++];
      else
         c[k++]=a[j++],ans+=mid-i+1;//统计答案
    while(i<=mid)
      c[k++]=a[i++];
    while(j<=e)
      c[k++]=a[j++];
    for(int l=b;l<=e;l++)
      a[l]=c[l];
}
```

Prim 算法

```
void prim()
{
    memset(dt,0x3f, sizeof(dt));//初始化距离数组为一个很大的数（10亿左右）
    int res= 0;
    dt[1] = 0;//从 1 号节点开始生成
    for(int i = 0; i < n; i++)//每次循环选出一个点加入到生成树
    {
        int t = -1;
        for(int j = 1; j <= n; j++)//每个节点一次判断
        {
            if(!st[j] && (t == -1 || dt[j] < dt[t]))//如果没有在树中，且到树的距离
                t = j;
        }
        //如果孤立点，直返输出不能，然后退出
        if(dt[t] == 0x3f3f3f3f) {
            cout << "impossible";
            return;
        }
        st[t] = 1;// 选择该点
        res += dt[t];
        for(int i = 1; i <= n; i++)//更新生成树外的点到生成树的距离
        {
            if(dt[i] > g[t][i] && !st[i])//从 t 到节点 i 的距离小于原来距离，则更新
            {
                dt[i] = g[t][i];//更新距离
                pre[i] = t;//从 t 到 i 的距离更短，i 的前驱变为 t.
            }
        }
    }
    cout << res;
}
```

Kruskal 算法

1. 所有边按权值排序
2. 若不形成回路，则取最小的边加入图中

```
int find(int a){//并查集找祖宗
    if(p[a] != a) p[a] = find(p[a]);
    return p[a];
}
void klskr(){
    for(int i = 1; i <= m; i++)//依次尝试加入每条边
    {
        int pa = find(edg[i].a);// a 点所在的集合
        int pb = find(edg[i].b);// b 点所在的集合
        if(pa != pb){//如果 a b 不在一个集合中
            res += edg[i].w;//a b 之间这条边要
            p[pa] = pb;// 合并a b
            cnt ++; // 保留的边数量+1
        }
    }
}
```

活动安排问题

1. 将各个活动按照活动结束时间 fi 排序
2. 选择结束时间最早的为第一个活动
3. 遍历剩下的 n 个活动寻找相容的活动

最优前缀编码问题

循环地选择具有最低频率的两个结点，生成一棵子树，直至形成树

最优装载问题回溯

```
void Loading<Type>:: void backtrack (int i)
{// 搜索第i层结点
  if (i > n)  {// 到达叶结点
     bestw=cw;return;}
 //搜索子树
  r -= w[i];
  if (cw + w[i] <= c) {// 搜索左子树
    x[i] = 1;
    cw += w[i];
    backtrack(i + 1);
    cw - = w[i];     }
  if (cw + r > bestw)    {
    x[i] = 0;  // 搜索右子树
    backtrack(i + 1);      }
  r += w[i];  }
```

Dijkstra 算法

```
void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);//距离初始化为无穷大
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;//小根堆
    heap.push({0, 1});//插入距离和节点编号
    while (heap.size())
    {
        auto t = heap.top();//取距离源点最近的点
        heap.pop();
        int ver = t.second, distance = t.first;//ver:节点编号，di
        if (st[ver]) continue;//如果距离已经确定，则跳过该点
        st[ver] = true;
        for (int i = h[ver]; i != -1; i = ne[i])//更新ver所指向的
        {
            int j = e[i];
            if (dist[j] > dist[ver] + w[i])
            {
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j], j});//距离变小，则入堆
            }
        }
    }
    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

背包问题回溯法

1. 当前正在处理的物品索引 i；
2. 当前已放入背包的总重量 curWeight 和 3. 总价值 curVal；
3. 记录最优解（最大价值）及对应的选法。

剪枝策略

1. 当前的价值+剩余的所有价值<最优解，回退
2. 加上第 i 个物品后超重，回退。

```
backtrack(t)
  if (t>=n):
    if bestp < cp:
      bestp = cp
  else:
    if cw + w[t] <= c:
      x[t] = 1
      cw = cw + w[t]
      cp = cp + v[t]
      backtrack(t+1)
      cw = cw – w[t]
      cp = cp – v[t]
      x[t] = 0
      backtrack(t+1)
```

符号三角形问题

第一行在排列数搜索，然后生成三角形，统计对应的结果

剪枝函数：

可行性约束函数：当前符号三角形所包含的“+”个数与“-”个数均不超过 n*(n+1)/4

无解的判断：n*(n+1)/2 为奇数

```
void Triangle::Backtrack(int t)
{
  if ((count>half)||(t*(t-1)/2-count>half)) return; if
  (t>n) sum++;
   else
    for (int i=0;i<2;i++) {
     p[1][t]=i;
     count+=i;
     for (int j=2;j<=t;j++) {
       p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
       count+=p[j][t-j+1];
     }
    Backtrack(t+1);
    for (int j=2;j<=t;j++)
    count-=p[j][t-j+1];
    count-=i;
   }
}
```

装载问题分支限界

策略：

1. 尽可能将第一艘船装满，再将剩余的放到第二艘船
2. 如何装满第一艘船是一个子集树（01 背包问题）

剪枝函数：

1. 如果加上当前货物超重，就剪枝
2. 如果当前重量+剩余的重量<最优解，就剪枝

```
// 检查左子结点
Type wt = Ew + w[i];      // 左子结点的重量
  if (wt <= c) {          // 可行结点
     if (wt > bestw) bestw = wt;
     // 加入活结点队列
     if (i < n) Q.Add(wt);
}
```

提前更新bestw

```
// 检查右子结点
if (Ew + r > bestw && i < n)
  Q.Add(Ew);              // 可能含最优解
  Q.Delete(Ew);           // 取下一扩展结点
```

右子节点剪枝

N 后问题

约束条件：

皇后不能在同一列 xi≠xj

皇后不能在同一主对角线$xi - i \neq xj - j$

皇后不能在同一副对角线$xi + i \neq xj + j$

```
bool Queen::Place(int k)
{//判定两个皇后是否在同一斜线或在同一列上
 for (int j=1;j<k;j++)
   if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
 return true;
}
void Queen::Backtrack(int t)
{
  if (t>n) sum++;
    else
      for (int i=1;i<=n;i++) {
        x[t]=i;
        if (Place(t)) Backtrack(t+1);
      }
}
int nQueen(int n)
{
 QueenX;
 //初始化X
 X. n=n; //皇后个数
 X. sum=0;
 int*p=new int [n+1];
 for(int i=0; i<=n; i++) p[i]=0;
 X.x=p;
 X.Backtrack(1);
 delete [] p;
 returnX. sum;
}
```

图的 M 着色问题

用排列数求解，每个子节点有 m 个孩子节点

约束函数：

顶点 i 与已着色的相邻顶点颜色不重复

```
GraphColor(int n,int m,int color[],bool c[][5])
{
    int i,k;
    for (i=0; i<n; i++ )                //将解向量color[n]初始化为0
        color[i]=0;
    k=0;
    while (k>=0)
    {   color[k]=color[k]+1;            //使当前颜色数加1
        while ((color[k]<=m) &&(!ok(color,k,c,n))) //当前颜色是否有效
                color[k]=color[k]+1;    //无效，搜索下一个颜色
        if (color[k]<=m)                //求解完毕，输出解
        {
            if (k==n-1)break;           //是最后的顶点，完成搜索
            else k=k+1;                 //否，处理下一个顶点
        }
        else                            //搜索失败，回溯到前一个顶点
        {
            color[k]=0;
            k=k-1;
        }
    }
}
```

最大团问题

首先设最大团为一个空团，往其中加入一个顶点，然后依次考虑每个顶点，查看该顶点加入团之后仍然构成一个团，如果可以，考虑将该顶点加入团或者舍弃两种情况，如果不行，直接舍弃，然后递归判断下一顶点。

剪枝策略：

剩余未考虑的顶点数加上团中顶点数不大于当前解的顶点数

```
void Clique::Backtrack(int i)
{// 计算最大团
if (i > n) {// 到达叶结点
   for (int j = 1; j <= n; j++) bestx[j] = x[j];
   bestn = cn; return;}
// 检查顶点 i 与当前团的连接
int OK = 1;
for (int j = 1; j < i; j++)
   if (x[j] && a[i][j] == 0) {
      // i与j不相连
      OK = 0; break;}
if (OK) {// 进入左子树
   x[i] = 1; cn++;
   Backtrack(i+1);
   x[i] = 0; cn--;}
if (cn + n - i > bestn) {// 进入右子树
   x[i] = 0;
   Backtrack(i+1);}
}
```

回溯法效率分析

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；

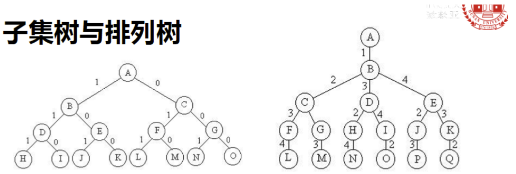
（4）计算上界函数 bound 的时间；

（5）满足约束函数和上界函数约束的所有 x[k]的个数。

好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折中。

回溯法框架

子集树与排列树



遍历子集树需$O(2^n)$计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (legal(t)) backtrack(t+1); }
}
```

遍历排列树需要$O(n!)$计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++) {
        swap(x[t], x[i]);
        if (legal(t)) backtrack(t+1);
        swap(x[t], x[i]); }
}
```

TSP 问题回溯

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
  if (i == n) { //当前扩展结点是排列树的叶结点的父结点
    if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
      (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge)) { for
      (int j = 1; j <= n; j++) bestx[j] = x[j];
      bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];} }
  else { //当前扩展结点位于排列树的第i-1层
  for (int j = i; j <= n; j++) // 是否可进入x[j]子树?
      if (a[x[i-1]][x[j]] != NoEdge &&
        (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) { // 搜索子树
        Swap(x[i], x[j]);
        cc += a[x[i-1]][x[i]];
        Backtrack(i+1);
        cc -= a[x[i-1]][x[i]];
        Swap(x[i], x[j]);} }
}
```

随机数值算法

1. 主要用于数值问题求解
2. 算法的输出往往是近似解
3. 近似解的精确度与算法执行时间成正比

蒙特卡罗算法

1. 主要用于求解需要准确解的问题
2. 算法可能给出错误解
3. 获得精确解概率与算法执行时间成正比

对于一个解所给问题的蒙特卡罗算法 MC(x)，如果存在问题实例的子集 X 使得：

1.当 x 属于 X 时，MC(x)返回的解是正确的；

2.当 x 属于 X 时，正确解是 y0，但 MC(x)返回的解未必是 y0。称上述算法 MC(x)是偏 y0 的算法。

```
template<class Type>
bool Majority(Type *T, int n)
{// 判定主元素的蒙特卡罗算法
   int i=rnd.Random(n)+1;
   Type x=T[i];// 随机选择数组元素
   int k=0;
   for (int j=1;j<=n;j++)
   if (T[j]==x) k++;
   return (k>n/2);
  // k>n/2 时T含有主元素
}
```

```
template<class Type>
bool MajorityMC(Type *T, int n, double e)
{// 重复调用k次Majority算法
   int k=ceil(log(1/e)/log(2));
   for (int i=1;i<=k;i++)
    if (Majority(T,n)) return true;
   return false;
}
```

舍伍德算法

1. 一定能够求得一个正确解
2. 确定算法的最坏与平均复杂性差别大时，加入随机性，即得到 Sherwood 算法
3. 消除最坏行为与特定实例的联系

基本思想：获得一个随机化算法 B，使得对问题的输入规模为 n 的每一个实例均有$t_B(x) = \bar{t_A}(n) + s(n)$

舍伍德算法可获得很好的平均性

```
static RandomNumber rnd;
   int i = l,
   j = l + rnd.Random(r - l + 1);//随
   Swap(a[i], a[j]);
   j = r + 1;
   Type pivot = a[l];
```

拉斯维加斯算法

求得的解总是正确的，但有时拉斯维加斯算法可能 始终找不到解.

1. 一旦找到一个解，该解一定是正确的
2. 找到解的概率与算法执行时间成正比
3. 增加对问题反复求解次数，可使求解无效的概率任意小