> 考试重点

- 并查集union-find算法（按秩合并）
- 递推关系（会进行递推表达式到算法复杂性计算的证明）
- 二叉树（堆）相关的证明
- P，NP，NP完全问题的概念和比较
- 会求图的邻接矩阵，熟练掌握Floyd算法求图任意两点之间的最短路径（动态规划）
- 单源最短路径算法
- 动态规划算法（分治法）求子段和最大值
- 回溯法求子集和

- 掌握分治法和动态规划算法的区别
- 最长公共子序列问题
- 排序问题（特别是快排算法）
- 0/1背包问题
- 算法复杂度的概念，几种符号的区别
- 用贪心算法和动态规划算法解决硬币找零问题

最大堆和最小堆

分治法设计算法

贪心法设计算法

动态规划法设计算法

回溯法设计算法

最大子段和问题。给定n个整数的序列A=<$a_1$,$a_2$,…,$a_n$>，求$\max\{0, \max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k\}$

硬币找零问题。给出一些不同面值的硬币<a0,a1,…,an>,其中a0=1,以及数值M，要计算找M所需要的的最少硬币数。

比如，我们有硬币<1,5,10,20>，那么如果要找33块钱的话，最少的零钱数是20+10+1+1+1，一共5个硬币。

> 运动员最佳匹配问题。羽毛球队有男女运动员各n人。给定两个n*n矩阵P和Q。P[i][j]是男运动员i和女运动员j配对组成混合双打的男运动员竞赛优势；Q[i][j]是女运动员i和男运动员j配合时女运动员的竞赛优势。由于技术配合和心理状态等各种因素影响，P[i][j]不一定等于Q[i][j]。男运动员i和女运动员j配对组成混合双打的男女双方竞赛优势为P[i][j] *Q[j][i]。设计一个算法，对于给定的男女运动员竞赛优势，计算男女运动员最佳配对法，使得各组男女双方竞赛优势的总和达到最大。

**搜索**

1.线性搜索，对数组进行扫描

> **Algorithm 1.1 LINEARSEARCH**
>
> **Input**: An array A [*1…n*] of n elements and an element *x*.
> **Output**: *j* if *x* = A [*j*], 1 ≤ *j* ≤ *n*, and 0 otherwise.
> 1. *j* ←1
> 2.**while** (*j* < n) **and** (*x* ≠A [*j*])
> 3. *j*← *j* + 1
> 4. **end while**
> 5. **if** *x* = A [j] **then return** *j* **else return** *0*.

2.二分搜索

> **Algorithm 1.2 BINARYSEARCH**
>
> **Input**: 数组A [*1…n*]包含n个非递减次序的元，查找元素*x*.
> **Output**: *j* if *x* = A [*j*], 1 ≤*j* ≤*n*, and 0 otherwise.
> 1.*low*←1; *high* ←*n*; *j* ←0
> 2.**while** (*low* ≤*high*) **and** (*j* = 0)
> 3. *mid*←⌊*low*+*high*/2⌋
> 4. **if** *x* = A [*mid*] **then** *j* ←*mid*
> 5. **else if** *x*< A [*mid*] **then** *high* ← *mid* – 1
> 6. **else** *low* ← *mid* + 1
> 7. **end while**
> 8.**return** *j*

> 规模为n的二分搜索算法种元素比较次数最多为$\lfloor logn \rfloor + 1$
>
> > 在while循环的第j次循环时，剩余元素的数目是$\lfloor n/2^{j-1} \rfloor$
> > 循环最大次数是满足条件$\lfloor n/2^{j-1} \rfloor = 1$时j的值
> > 由$1 <= \lfloor n/2^{j-1} \rfloor < 2$ --> $j = \lfloor logn \rfloor + 1$

3.选择排序

> **Algorithm 1.4 SELECTIONSORT**
>
> **Input**: 包含n个元素的数组 A [*1..n* ] .
> **Output**: A [*1..n* ] 中的元素非递减排列
> 1. **for** *i*←*1* **to** *n-1*
> 2. *k*← *i*
> 3. **for** *j*←*i*+*1* **to** *n* *{找到第 it 个最小元素}*
> 4. **if** A [*j*]<A [*k*] **then** *k* ←*j*
> 5. **end for**
> 6. **if** *k* ≠ *i* **then** 交换 A [*i*] **and** A [*k*]
> 7.**end for**

> 选择排序算法的元素**比较次数为**：$n*(n-1)/2$
> 元素**赋值的次数**介于0和3(n-1)之间

4.插入排序

> **Algorithm 1.5 INSERTIONSORT**
>
> **Input**: An array A [*1..n*] of *n* elements.
> **Output**: A [*1..n*] sorted in nondecreasing order.
> 1.**for** *i*←*2* **to** *n*
> 2. *x*←A[*i*]
> 3. *j*←*i-1*
> 4. **while** (*j>0*) **and** (A[*j*]>*x*)
> 5. A[*j+1*]←A[*j*]
> 6. *j*←*j-1*
> 7. **end while**
> 8. A[*j+1*]=*x*
> 9.**end for**

> 插入排序的元素**比较次数介于** $n-1$和$n*(n-1)/2$之间
> 元素**赋值次数**等于元素比较次数加上n-1

5.合并算法

> **Algorithm 1.3 MERGE**
>
> **Input**:数组 A[1…*m*] 和三个下标*p*, *q* 和 *r*, 其中 *1*≤*p*≤*q*<*r*≤*m*. 该数组的两个子部分A[*p*…*q*] 和 A[*q+1*…*r*] 都是非递减排序
> **Output**: A[*p*…*r*] 中的元素非递减排序.
> 1.**comment**: B [*p*…*r*] 是一个辅助数组
> 2.*s*←*p*; *t* ←*q* + *1*; *k* ←*p*
> 3.**while** *s* ≤*q* **and** *t* ≤*r*
> 4. **if** A [*s*] ≤A [*t*] **then**
> 5. B [*k*] ← A [*s*]; *s*←*s+1*
> 6. **else** *B [k]* ←*A [t]*; *t*←*t+1*
> 7. **end if**
> 8. *k*←*k+1*
> 9. **end while**
> 10. **if** s = q + 1 **then** B [k…r] ←A [t…r]
> 11.**else** B [*k…r]* ←*A [s…q]*
> 12. **end if**
> 13. A *[p…r]* ←*B [p…r]*

> 对于合并长度为n1和n2的两个数组(n1 <= n2, n = n1 + n2)
> merge算法的元素比较次数介于n1和n-1之间
> 特别的是，如果两个数组的大小为$\lfloor n/2 \rfloor$
> 则**元素的比较次数**介于$\lfloor n/2 \rfloor$和$n-1$之间
> **元素的赋值次数**为$2*n$

6.自底向上合并排序

> **Algorithm 1.6 BOTTOMUPSORT**
>
> **Input**: 包含n个元素的数组 A [*1..n*]
> **Output**: A [*1..n*] 按照非递减排序.
> *1. t*←1
> **2.while** *t<n*
> *3. s* ←*t*; *t*←*2s*; *i*←*0*
> **4. while** *i* + *t* ≤ *n*
> 5. MERGE(A, *i+1*, *i* + *s*, *i+t*)
> 6. *i*←*i+t*
> **7. end while**
> 8. **if** *i* + *s* < *n* **then** MERGE(A, *i+1*, *i* + *s*, *n*)
> **9.end while**

> 当n是2的幂，在这种情况下外层循环次数k = logn,在第j次循环时，对于两个规模为$2^{j-1}$子序列，合并操作进行的次数为$n/2^j$

> 自底向上合并排序对规模为n的数组进行操作时(n是2的幂)，
> **元素比较次数**介于$nlogn/2$和$logn-n+1$之间
> **元素赋值次数**为$2*n*logn$

**时间复杂度的分析**

> 参考文献：https://blog.csdn.net/qq_41856733/article/details/89034055

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

**O(n)表示渐近上界**

> 设函数f(n)和g(n)是定义在非负整数集合上的正函数，如果存在两个正常数c和n0，使得当n≥n0时，有f(n)≤cg(n)，则记做f(n) = O(g(n))，称为大O记号（big Oh notation） 称g(n)是f(n)的一个上界 注： f(n)的阶不高于g(n)。

**Ω(n)表示渐近下界**

> 设有函数f(n)和g(n)是定义在非负整数集合上的正函数，如果存在两个正常数 c和n0，使得当n≥n0时，有f(n)≥c g(n)，则记做f(n) = Ω (g(n))，称为Ω记号（omega notation）。 注： f(n)的阶不低于g(n)。

**θ(n)表示紧渐近界**

> 设有函数f(n)和g(n)是定义在非负整数集合上的正函数，如果存在正常数c1，c2和n0，使得当n≥n0时，有c1 g(n)≤f(n)≤c2 g(n)，则记做f(n) = θ(g(n))，称为θ记号（Theta notation）。 注：此时f(n)和g(n)同阶

**如何估算时间复杂度**

> $1<loglogn<logn<\sqrt{n}<n<nlogn<n^2<2^n<n!$

- 有限项等比级数的求和

$$a_1(1+q+q^2+...+q^k)=\frac{a_1(1-q^{k+1})}{1-q}$$

- 无限等比序列的收敛值

$$a_1(1+q+q^2+...)=\frac{a_1}{1-q},\quad q<1$$

- 调和级数的估计值

$$1+\frac{1}{2}+...+\frac{1}{n}+...=O(\log n)$$

- 对数函数和的估计值

$$\log 1+\log 2+...+\log n=O(n\log n)$$

1. 计算迭代次数

例子:
**Input**:n=$2^k$, for some positive integer k.
**Output**: count=number of times Step 4 is executed.

```
1.count←0
2. while n≥1
3.        for j←1 to n
4.            count←count+1
5.        end for
6.        n←n/2
7. end while
8. return count
```

- COUNT1包含两个嵌套循环和一个变量count,这个变量用来对算法执行的迭代次数计数，这里$n=2^k$，其中 k为正整数.
  while循环执行 k+1次,这里 k=logn. For循环执行n次,之后是 n/2，n/4,...,1.

$$\sum_{j=0}^{k}\frac{n}{2^j}=n\sum_{j=0}^{k}\frac{1}{2^j}=n(2-\frac{1}{2^k})=2n-1=\Theta(n)$$

2. 计算基本操作的频度

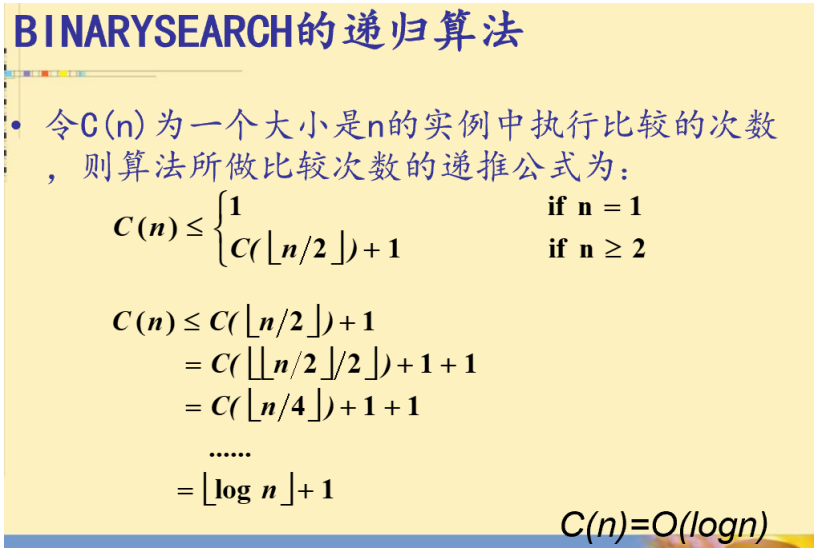
元运算:

如果算法中的一个元运算具有最高频率, 所以其他元运算频率均在他的频率的常数倍内,则在这个元运算称为基本运算。

· 在搜索和排序算法中，元运算一般是元素比较
· 在矩阵乘法运算中，数量乘法运算是元运算
· 在遍历一个链表时，元运算是设置或更新指针
· 在图遍历中，元运算是访问节点的"动作"和被访问节点的计数
MERGE算法中的元素赋值是基本运算.注意元素比较一般意义下不是基本运算, 极端情况下可能只有一次

- 确定算法BOTTOMUPSORT的确界
- 首先，这种算法的基本运算从MERGE算法继承而来，在while循环中每次循环都调用算法MERGE，所以，我们可以选择元素比较作为元运算
- 当n是2的幂时，BOTTOMUPSORT的元素比较次数在$(nlogn)/2$和$nlogn-n+1$之间。这意味着，元素比较次数是$\Theta(nlogn)$
- 由于算法用到的元素比较运算，在相差一个常数因子的意义下具有最大频度，我们可以得出结论，算法运行时间和比较次数成正比。所以算法运行时间是$\Theta(nlogn)$

3. 使用递推关系




**空间复杂度**

> •在线性搜索算法中,只需要一个内存单元用来存储搜索结果.我们可以说需要的空间为Q(1). 二分搜索,选择排序和插入排序的情况一样.

**离散数学基础**

设R是在集合A上的关系.

- 自反/反自反
  - 如果对所有的a ∈A，有(a,a)∈R，则称R是自反的；
  - 如果对所有的a ∈A，有(a,a)∉R，则称R是反自反的；
- 对称/非对称/反对称
  - 如果 (a,b) ∈R 蕴含着(b,a) ∈R，则称R是为对称的；
  - 如果 (a,b) ∈R 蕴含着(b,a) ∉R，则称R是为非对称的；
  - 如果 (a,b) ∈R，(b,a) ∈R，蕴含着a=b，则称R是为反对称的；
- 传递
  - 如果 (a,b) ∈R 并且 (b,c) ∈R 得出 (a,c) ∈R，则称R是传递的.
- 偏序
  - 一个关系是自反的、反对称的 和传递的，则称为是一个偏序。

**等价关系**

- 如果集合A上的关系是自反的、对称的和传递的，则称它是等价关系。

**等价类**

- R是A上的等价关系，此情况下，R划分A成为等价类 $C_1,C_2,\ldots,C_k$，这样等价类中的任意两个元素有R关系。也就是对于任意$C_i$,1≤i≤k，如果x ∈$C_i$ 且y ∈$C_i$，则(x,y)∈R。另一方面，如果 x ∈$C_i$ 且y ∈$C_j$，i≠j，那么(x,y)∉R。

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$$

> 如何求解递推式?

· 线性齐次递推式
1.特征方程 （$c_1 = s + r; c_2 = s * r$)
2.特征值
3.带入$f(n) = c_1 r_1^n + c_2 r_2^n$
4.带入特殊值求出c1 c2即可

> 递推式的展开

**数据结构**

**堆和不相交集**

> 堆

什么是堆？
一个（二叉）堆是一个几乎完全的二叉树，它的每个节点都满足堆的特性：如果v和p(v)分别是节点和它的父节点，那么存储在p(v)中的数据项键值不小于存储在v中数据项的键值。

堆使用二叉树存储方式

> 堆上的运算

```
· Sift-Up
· Sift-Down
· Insert
· Delete
· Delete-Max
```

> **Sift-Up**

```
过程 SIFT-UP
输入:数组H[1…n]和位于1和n之间的索引i
输出:上移H[i](如果需要),以使它不大于父节点。
1. done←false
2. if i=1 then exit                {节点i为根}
3. repeat
4.     if key (H[i])>key(H [⌊i/2⌋]) then 互换 H[i] and H [⌊i/2⌋]
5.     else done←true
6.     i←⌊i/2⌋
7. until i=1 or done
```

**Sift-Down**

```
过程 SIFT-DOWN
输入:数组H[1…n]和位于1和n之间的索引i
输出:下移H[i](如果需要),以使它不小于子节点。
1. done←false
2. if 2i>n then exit               {节点 i 是叶子}
3. repeat
4.     i←2i
5.     if i+1 ≤n and key(H[i+1])>key(H[i]) then i←i+1
6.     if key(H [⌊i/2⌋])< key (H[i]) then 互换H[i] and H [⌊i/2⌋]
7.     else done←true
8.     end if
9. until 2i>n or done
```

**堆的插入操作**：时间复杂度$O(logn)$

•基本思想 为了把元素x插入到堆H中，先将堆大小加1，然后将x添加到H的末尾，再根据需要，把x上移，直到满足堆的特性。

```
算法 4.1 INSERT
    输入: 堆H[1…n] 和元素x。
    输出: 新的堆 H[1…n+1] ， x为其元素之一。
        1. n←n+1          {增加H的大小}
        2. H[n]←x
        3. SIFT-UP(H,n)
```

**堆的删除操作**：时间复杂度$O(logn)$

• 要从大小为n的堆H中删除元素H[i]，可先用H[n]替换H[i]，然后将堆栈大小减1，如果需要的话，根据H[i]的键值与存储在它父节点和子节点中元素键值的关系，对H[i]做Sift-up或Sift-down运算，直到满足堆特性为止。

**算法 4.2 DELETE**

**输入**:非空堆H[1…n]和位于1和n之间的索引i。

**输出**:删除H[i]之后的新堆H[1…n-1] 。

```
1. x←H[i];              y←H[n]
2. n←n-1            {decrease the size of H}
3. if i=n+1  then exit     {done}
4. H[i]←y
5. if key(y) ≥ key(x) then SIFT-UP(H,i)
6. else SIFT-DOWN(H,i)
7. end if
```

**删除最大值操作**：时间复杂度$O（logn)$

这项运算在一个非空堆H中删除并返回最大键值的数据项。直接完成这项运算要用到删除运算:只要返回根节点中的元素并将其从堆中删除。

**算法 4.3 DELETEMAX**

**输入**: 堆H[1…n].

**输出**:返回最大键值元素x并将其从堆中删除。

```
1. x←H[1]
2. DELETE(H,1)
3. return x
```

创建堆

**法一**：给出一个有n个元素的数组A[1..n]，很容易创建一个包含这些元素的堆，可以这样进行:从空的堆开始，不断插人每一个元素，直到A完全被转移到堆中为止。

将数组A[1..10]={4,3,8,10,11,13,7,30,17,26} 转换成一个堆.

时间复杂度:

因为插入第j个键值用时O(logj)，因此用这种方法创建堆栈的时间复杂性是O(nlogn)。

**法二**： 把一棵n个节点的几乎完全的二叉树转换成堆H[1..n]。从最后一个节点开始(编码为n的那一个)到根节点(编码为1的节点)，逐个扫描所有的节点，根据需要，每一次将以当前节点为根节点的子树转换成堆。这些元素A[n/2+1],...,A [n]它们对应于T的叶子，这样可以从A [n/2]开始调整数组，并且继续A[n/2-1],...,A[1]进行调整。

**算法 4.4 MAKEHEAP**

**输入**: n个元素的数组A[1…n] 。

**输出**: A[1…n] 转换成堆

```
1. for i← ⌊n/2⌋ downto 1
2.       SIFT-DOWN(A, i)
3.  end for
```

建立堆：时间复杂度: θ(n), 空间复杂度: θ(1) 自下而上与树的深度无关，所以这里的建立堆的时间复杂度是$O(n)$的

堆排序首先将A变换成堆，并使其具有这样的性质，每个元素的键值是该元素本身，即key(A[i])=A[i]。

> 由于A中各项的最大值存储在A[1]中，可以将A[1]和A[n]交换，使得A[n]是数组中最大元素。这时，A[1]中的元素可能小于存放在它的一个子节点中的元素，于是用过程SIFT-DOWN将A[1...n-1]转换成堆。
> 接下来将A[1]和A[n-1]交换，并调整数组A[1...n-2]成为堆。
> 交换元素和调整堆的过程一直重复，直到堆的大小变成i为止，这时，A[1]是最小的。


堆排序:算法HEAPSORT对n个元素排序要用**θ(nlogn)**时间

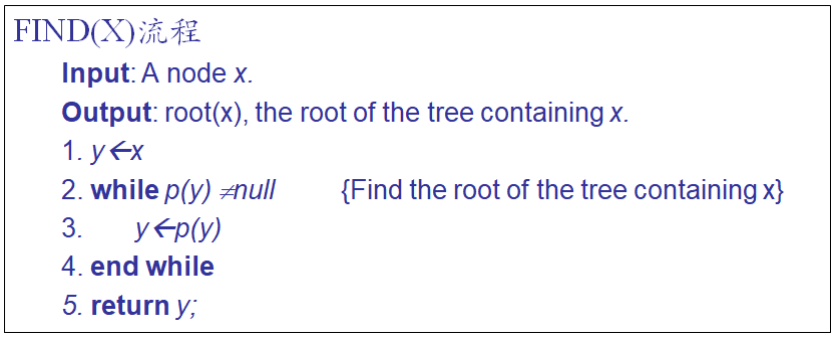
**应用：** Prime算法和堆优化的Djisktra算法

> 并查集（不相交集）

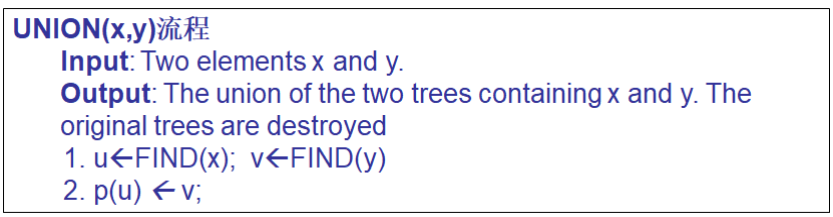
含义： 设给出一个有n个不同元素的集合S，这些元素被分成不相交集合。在每个子集中，用一个特殊的元素作为集合的名字或代表。

> 主要操作：
>
> •FIND(x): 寻找并返回包含元素x的集合的名字。
>
> •UNION(x,y):包含元素x和y的两个集合用它们的并集替换。并集的名字或者是原来包含元素x的那个集合的名字，或者是原来包含元素y的那个集合的名字，这将会在以后确定。

> Find操作：




> Union操作：




按秩合并措施：

> 为限制每棵树的高度，采用按秩合并措施；

是使包含较少结点的树的根指向包含较多结点的树的根，而这个树的大小可以抽象为树的高度，即高度小的树合并到高度大的树，这样资源利用更加合理。

基本思想
给每个节点存储一个非负数作为该节点的秩，记为rank，节点的秩基本上就是它的高度。
设x和y是当前森林中两棵不同的树的根
初始状态时，每个节点的秩是0
在执行运算UNION(x, y),时，比较rank(x) 和 rank(y)。

如果rank(x)<rank(y)， 就使得y 为x 的父节点
如果rank(x)>rank(y)，就使得x 为y 的父节点
否则，如果rank(x)=rank(y)，就使得y 为x 的父节点，并将 rank(y) 加1。

为了实现按秩合并不相交集合森林，要记录下秩的变化。
对于每个结点x，有一个整数rank[x]，它是x的高度（从x到其某一个后代叶结点的最长路径上边的数目）的一个上界。（即树高）。
当由MAKE-SET创建了一个单元集时，对应的树中结点的初始秩为0，每个FIND-SET操作不改变任何秩。
当对两棵树应用UNION时，有两种情况，具体取决于根是否有相等的秩。当两个秩不相等时，我们使具有高秩的根成为具有较低秩的根的父结点，但秩本身保持不变。当两个秩相同时，任选一个根作为父结点，并增加其秩的值。

> 按秩合并的一些性质：
>
> •对于任意的整数: r≥0，秩r的节点数至多是$n/2^r$.
> •rank(p(x)) ≥ rank(x)+1.
> •包括根节点x在内的树中节点的个数至少是$2^{rank(x)}$.
> •推论 **4.1**任何节点的秩最大是$\lfloor logn \rfloor$.

**Find-Union算法优化**

> Find路径压缩：
>
> **算法 4.6 FIND**
> 输入: 节点$x$.
> 输出: root(x),包含x的树的根。
> 1. $y \leftarrow x$
> 2. **while** $p(y) \neq null$ {寻找包含x的树的根}
> 3. $y \leftarrow p(y)$
> 4. **end while**
> 5. $root \leftarrow y$; $y \leftarrow x$
> 6. **while** $p(y) \neq$null {执行路径压缩}
> 7. $w \leftarrow p(y)$; $p(y) \leftarrow root$; $y \leftarrow w$
> 8. **end while**
> 9. **return** *root*
>
> Union按秩合并：

**算法 4.7 UNION**

**Input**:两个元素x和y

**Output**:包含x和y的两个树的合并，原来的树被破坏。

```
1. u←FIND(x);  v←FIND(y)
2. if rank(u) ≤ rank(v) then
3.    p(u)←v
4.    if rank(u) =rank(v) then rank(v)←rank(v)+1
5. else p(v)←u
6. end if
```

优化算法分析：

•设T(m)表示用按秩合并和路径压缩处理m个合并和寻找运算的交替序列s所需的运行时间，那么在最坏情况下 T(m)=O(m*log*n)

•注意对于几乎所有的实际用途，*log*n≤5.这说明事实上对于所有的实际应用，运行时间是O(m)。

**递归**

基数排序

基数排序的过程演示

算法 5.3 RADIXSORT

输入：一张有n个数的表L={$a_1$…$a_n$}和k位数字。

输出：按非降序排列的L。

```
1. for j←1 to k
2.    准备10 个空表 L0,L1,…,L9.
3.    while L 非空
4.       a←L中的下一元素；删除L中的a
5.       i←a中的第j位数字；将a加入表Li中
6.    end while
7.    L←L0
8.    for i←1 to 9
9.       L←L,Li        {将表Li 加入表 L中}
10.   end for
11. end for
12. return L
```

算法分析：时间复杂度$O(n)$, 空间复杂度$O(n)$

整数幂的求解

```
EXPREC
输入：实数x和非负整数n。
输出：x^n
过程：power(x ,m) {计算x^m}
1. if m = 0 then y <- 1
2. else
3.    y <- power(x, m/2)
4.    y <- y ^ 2
5.    if m 为奇数 then y <- x*y
6.    end if
7.  return y
```

```
EXP
输入：实数x和非负整数n。
输出：x^n
--二进制考虑
1. y <- 1
2. 将n用二进制数dk,dk-1,...d0进行表示
3.for j <- k downto 0
4.   y <- y^2
5.   if dj = 1 then y <- x*y
6.end for
7.return y
```

时间复杂度$O(logn)$

多项式求值

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$
$$= ((((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})\ldots)x + a_1)x + a_0$$

**算法5.6 HORNER**

Input: $n+2$ 个实数$a_0, a_1, \ldots, a_n$ $x$的序列

Output:

1. $p \leftarrow a_n$
2. for $j \leftarrow 1$ to $n$
3. $p \leftarrow xp + a_{n-j}$ n(n+1)/2次乘法
4. end for
5. return $p$

转化为了 n次乘法与n次加法

生成排列算法

```
算法5.6 PERMUTATIONS1
    Input: 正整数n.
    Output: 数1,2,…,n.的所有可能排列
        1. for j←1 to n      //生成数组
        2.   P[j]←j
        3. end for
        4. perm1(1)
    过程 perm1(m)
        1. if m=n then output P[1…n]
        2. else
        3.  for j←m to n
        4.    互换  P[j] and P[m]
        5.    perm1(m+1)
        6.    互换  P[j] and P[m]
        7.    comment: at this point P[m..n]=m, m+1, …, n
        8.  end for
        9. end if
```

> 本质就是两两互换
>
> 时间复杂度$O(n*n!)$

**分治**

**将待求解的原问题划分成k个较小规模的子问题，对这k个子问题分别求解。**

> **MinMax()算法**
>
> ```
> 算法6.1 MINMAX
>     输入: n个整数元素的数组A[1..n]，n为2的幂。
>     输出: (x,y): A中的最大元素和最小元素。
>       1. minmax(1,n)
>     过程  minmax(low,high)
>       1. if high-low=1 then    {数组只有2个值的情况}
>       2.    if A[low]<A[high] then return (A[low],A[high])
>       3.    else return (A[high], A[low])
>       4.    end if
>       5. else
>       6.    mid←⌊(low+high)/2⌋
>       7.    (x1,y1)←minmax(low,mid)
>       8.    (x2,y2)←minmax(mid+1,high)
>       9.    x←min(x1,x2)
>       10.   y←max(y1,y2)
>       11. return(x,y)
>       12. end if
> ```
>
> 算法分析:
>
> $$C(n)=\begin{cases}1 & \text{if } n=2\\ 2C(n/2)+2 & \text{if } n>2\end{cases}$$
>
> 设数组A[1..n]包含n个元素，其中n是2的幂，则仅用$3n/2-2$次元素比较就能够在数组A中找出最大和最小元素。

**归并排序算法分析**

a)划分步骤 b)治理步骤 c)组合步骤 [分治思想]

## 算法6.3 Mergesort

**输入:** n个元素的数组A[1..n]。
**输出:** 按非降序排列的数组A[1..n]。

```
1. mergesort(A,1,n)
过程 mergesort(A,low,high)
1. if low<high then
2.                 mid←⌊(low+high)/2⌋
3.                 mergesort(A,low,mid)
4.                 mergesort(A,mid+1,high)
5.                 MERGE(A,low,mid,high)
6. end if
```

基本运算: 元素比较。

为了简单，假定n是2的幂。

- 如果n=1，则算法不执行任何元素的比较，比较次数是0。
- 如果 n≥2，则将问题分解为(whose size is n)2个子问题(whose size are n/2)。
- 合并两个子数组所需的元素比较次数在n/2与n-1之间。这样算法所需的最小比较次数:

$$c(n)=\begin{cases}0 & \text{if n=1}\\ 2c(n/2)+n/2 & \text{if n}\geq 2\end{cases} \qquad c(n)=\frac{n\log n}{2}$$

算法分析 时间复杂度 $O(logn)$空间$O(n)$

基本运算是：元素比较，其中元素的比较次数 $\frac{n*logn}{2}$ ~ $nlogn-n+1$

**寻找中项和第k小元素**

**算法6.4 SELECT**

输入:n个元素的数组A[1..n]和整数k,1≤k ≤n 。
输出:A中的第k小元素。

```
1.select(A,1,n,k)
过程 select(A,low,high,k)
1. p←high-low+1
2. if p<44 then 将A排序 return (A[k])
3. Let q=⌊p/5⌋.将A分成9组，每组5个元素。如果5不整除P，则排除剩余的元素
4.将q组中的每一组单独排序，找出中项。所有中项的集合为M
5. mm←select(M,1,q ,⌈q/2⌉)          {mm 为中项集合的中项}
6. 将A[low..high] 分成三组:
   A1={a|a<mm}   A2={a|a=mm}     A3={a|a>mm}
7. case
  |A1| ≥k: return select(A1,1,|A1|,k)
  |A1|+|A2| ≥k: return mm
   |A1|+|A2|<k: return select(A3,1,|A3|,k-|A1|-|A2|)
8. end case
```

算法分析 (注意常数很大)

$\Theta(n)$

可以使用类似快速排序的Split划分算法来替换掉A1,A2,A3数组的使用

**快速排序算法分析**

•原位上排序，即对于被排序的元素，不需要辅助的存储空间。

基本思想

•设A[low..high]是一个包含n个数的数组，并且x=A[low]

•我们考虑重新安排数组A中的元素的问题，使得小于或等于x的元素在x的前面，随后x又在所有大于它的元素的前面

•经过数组中元素改变排列后，对于某个w，low≤w≤high，x在A[w] 中。

1. 划分算法SPLIT:

**算法6.5 SPLIT**

输入:数组A[low..high].
输出: (1)如有必要，输出按上述描述的重新排列的数组A 。
(2)划分元素A[low]的新位置w。

```
1. i←low
2. x←A[low]
3. for j←low+1 to high
4.     if A[j] ≤x then
5.         i←i+1
6.                 if i ≠j then 互换A[i] and A[j]
7.         end if
8. end for
9.    互换A[low] and A[i]
10. w←i
11. return A and w
```

2. 快速排序QuickSort：

算法6.6 QIUCKSORT

输入：n个元素的数组A[1..n]

输出：按非降序排列的数组A中的元素

```
1. quicksort(A,1,n)
过程 quicksort(A,low,high)
1. if low<high then
2.                 w =SPLIT(A[low..high])    {w为A[low]的新位置}
3.                 quicksort(A,low,w-1)
4.                 quicksort(A,w+1,high)
5. end if
```

•在算法MERGESORT中，合并排序序列属于组合步骤，而在算法QUICKSORT中的划分过程则属于划分步骤。事实上，在算法QUICKSORT中组合步骤是不存在的。

算法分析：

时间复杂度$\Theta(n^2) \rightarrow \Theta(n)$

空间复杂度$\Theta(logn) \rightarrow \Theta(n)$

STRASSEN算法

-->矩阵乘法求解

-->增加加减法的运算来减少乘法的运算次数

Strassen算法的高效之处，就在于，它成功的减少了乘法次数，将8次乘法，减少到7次。

不要小看这减少的一次，和传统的的递归算法比较起来，每递归计算一次，效率就可以提高1 / 8，比如一个大的矩阵递归5次后，（7 / 8）5 = 0.5129，效率提升一倍。不过，这只是理论值，实际情况中，有隐含开销，并不是最常用算法。

矩阵是稀疏矩阵时，为稀疏矩阵设计的方法更快。所以，稠密矩阵上的快速矩阵乘法实现一般采用Strassen算法。

**最近点对Closepair问题：**

**算法6.7 CLOSESTPAIR**

输入: 平面上n个点的集合S。

输出: S中两点的最小距离。

1.以x坐标增序对S中的点排序。

2.序Y←以y坐标增序对S中的点排序。

3. δ←cp(1,n).

过程*cp(low,high)*

1. **if** *high-low+1* ≤3 **then** 用直接方法计算δ
2. **else**
3. *mid*←⌊(low+high)/2⌋
4. $x_0$←x(S[mid])
5. $\delta_l$←cp(low,mid)
6. $\delta_r$←cp(mid+1,high)
7. δ←mid{$\delta_l$, δr}
8. *k*←*0*

过程 *cp(low,high)*

9. **for** *i*←*1* **to** n {从Y中抽取T}
10. **if** |x(Y[i])-$x_0$| ≤δ **then**
11. *k*←*k+1*
12. *T[k]*←*Y[i]*
13. **end if**
14. **end for**
15. δ'←2δ
16. **for** i←1 **to** k-1
17. **for** j←i+1 **to** min{i+7,k}
18. **if** d(T[i],T[j]) <δ' **then** δ'← d(T[i],T[j])
19. **end for**
20. **end for**
21. δ←*min{δ,δ'}*
22. **end if**
23. **return** δ

时间复杂度$\Theta(n)$

> 获取了右边的最小点对距离a1和左边的最小点对距离a2后，获取中间的最小点对距离a3时，可以证明获取的中间T矩形框中左边最多只需要与右边的7个点进行比较。

```
    算法思想：
    1.对所有点以X轴坐标排序为X[]，对所有点以Y坐标自下而上排序为Y[]；
    2.以X[mid]为界，找出左半子集σ_1=closepair(X,1,mid)和右半子集最小
σ_2=closepair(X,mid+1,n)，σ_=min(σ_1,σ_2);
    3.For Y[i] in Y[1…n]:
        If(|Y[i]对应的x坐标-X[mid]|< σ_：
                  T[k]=Y[i];
                  k++;
    4. σ^~=2σ_
        for i<-1 to k:
          for j<-1 to min(i+7,k):
              if T[i]对应的点到T[j]对应的点的距离<σ^~:
                   σ^~=……;
    5.最终结果=min(σ^~, σ);
```

> 分治算法心得：
> 分治问题的明显特征就是再求解时，问题的答案与**问题的内在结构**无关。例如，答案可能在序列的左半部分，也可能在右半部分，此时便可放弃另一半部分。再比如，解决问题时，我们可以先解决左半部分，在解决右半部分，两部分的解组合起来是最终解。

**动态规划**

**动态规划，当前子问题的解将由上一次子问题的解推算出。**

> 分治法与动态规划的区别：
>
> 区别：
> 1.分治法是把大问题分解成一些相互独立的子问题，递归的求解这些子问题然后将他们合并来得到整个问题的解
> 2．动态规划是通过组合子问题的解来解决整个大问题。各个子问题不是独立的，也就是各个子问题包含公共子问题。它可以避免遇到的子问题的重复求解。
>
> 最优化问题：
> 通常采用动态规划对问题进行优化。对于一个问题，可以有很多可能的解决方案。每个解决方案有一个值，我们希望找到一个最佳的（最小或最大）值对应的解决方案，我们称这样的解决办法为最优的解决方案。
>
> 动态规划的基本步骤：
> 1．找出最优解的性质，并刻划其结构特征。
> 2．递归地定义最优值。
> 3．以自底向上的方式计算出最优值。
> 4．根据计算最优值时得到的信息，构造最优解。

**动态规划算法的基本要素：**

1. **最优子结构**
   •矩阵连乘计算次序**问题的最优解包含着其子问题的最优解。**这种性质称为**最优子结构性质。**

> 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

2.**重叠子问题**
•递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为**子问题的重叠性质。**

**动态规划算法的基本步骤：**

1.划分阶段

> 按照问题的时间或空间特征，把问题划分为若干个阶段。在划分阶段时，注意划分后的阶段一定是有序的或者是可排序的，否则问题就无法求解。

2.确定状态和状态变量

> 将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要**满足无后效性。**

3.确定决策并写出状态转移方程

> 因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可以写出。但事实上常常是反过来做，根据相邻两段的各个状态之间的关系来确定决策。

4.寻找边界条件

> 给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

**背包问题**

> V[i,j]是下面两个量的最大值:
>
> 1. V[i-1,j]:仅用最优的方法取自$\{u_1, u_2, \ldots, u_{i-1}\}$的物品去装入体积为j的背包所得到的价值最大值。
>
> 2. V[i-1,j-$s_i$]+$v_i$:用最优的方法取自$\{u_1, u_2, \ldots, u_{i-1}\}$的物品去装入体积为j-$s_i$的背包所得到的价值最大值加上物品$u_i$的价值。这仅应用于如果j≥$s_i$以及它等于把物品$u_i$加到背包上的情况。
>
> $$V[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0\\ V[i-1,j] & \text{if } j<s_i\\ \max\{V[i-1,j],V[i-1,j-s_i]+v_i\} & \text{if } i>0 \text{ and } j\geq s_i\end{cases}$$

> **算法7.4 KNAPSACK**
>
> 输入:物品集合U={$u_1,u_2,\ldots,u_n$}，体积分别为$s_1,s_2,\ldots,s_n$，价值分别为$v_1,v_2,\ldots,v_n$，容量为C的背包。
>
> 输出:$\sum_{ui\in S} v_i$的最大总价值，且满足$\sum_{ui\in S} s_i \leq C$，其中S⊆U.
>
> ```
> 1. for i←0 to n
> 2.         V[i,0]←0
> 3.end for
> 4. for j←0 to n
> 5.         V[0,j]←0
> 6. end for
> 7. for i←1 to n
> 8.         for j←1 to C
> 9.                 V[i,j]←V[i-1,j]
> 1.                 if s_i ≤j then V[i,j]←max{V[i,j],V[i-1,j-s_i]+v_i}
> 11.        end for
> 12. end for
> 13. return V[n,C]
> ```

算法分析： 时间复杂度$\Theta(Cn)$空间复杂度$\Theta(C)$

### 完全背包问题

```
for(int i=1;i<=n;i++)
{
    for(int j=0;j<=m;j++)
    {
        dp[i][j] = dp[i-1][j];
        if(j >= v[i])
            dp[i][j]=max(dp[i][j],dp[i][j-v[i]]+w[i]);
    }
}
return dp[n][m];
```

### 最长公共子序列LCS

$$c[i,j]=\begin{cases}0 & if\ \ i=0\ or\ j=0,\\ c[i-1,j-1]+1 & if\ \ i,j>0\ and\ x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]) & if\ \ i,j>0\ and\ x_i\neq y_j.\end{cases}$$

**算法7.1 LCS**

输入:字母表∑上的两个字符串A和B，长度分别为n和m
输出: A和B最长公共子序列的长度。

```
1. for i←0 to n  //j=0时，  L[i,j]=0
2.    L[i,0] ←0
3. end for
4. for j←0 to m //i=0时，  L[i,j]=0
5.    L[0,j] ←0
6. end for
7. for i←0 to n
8.      for j←0 to m
9.          if a_i = b_j then L[i,j] ←L[i-1,j-1] + 1
10.         else L[i,j] ←max{L[i,j-1], L[i-1,j]}
11.         end if
12.     end for
13.end for
14. return L[n,m]
```

算法分析：时间复杂度$\Theta(n*m)$空间复杂度$\Theta(min\{m,n\})$

### 最长上升子序列LIS

| | | |
|---|---|---|
| LIS问题 [最长上升子序列] | 问题描述 | 最长上升子序。给定一个长度为N的数列A，求数值单调递增的子序列的长度最长是多少。A的任意子序列B可以表示为B = $\{A_{k1}, A_{k2}, \ldots, A_{kp}\}$，其中$k_1 < k_2 < \ldots < k_p$ |
| | 状态表示 | F[i]表示以A[i]为结尾的“最长上升子序列”的长度 |
| | 阶段划分 | 子序列的结尾位置（数列A种的位置，从前到后） |
| | 转移方程 | $F[i] = max_{0 \le j \le i, A[j] < A[i]}\{F[j] + 1\}$ |
| | 边界 | · F[0] = 0 |
| | 目标 | $max_{1 \le i \le N}\{F[i]\}$ |
| | | |

**数字三角形问题**

| | | |
|---|---|---|
| 数字三角形问题 | 问题描述 | 给定一个共有N行的三角矩阵A，其中第i行有i列。从左上角除法，每次可以向下方或右下方走一步，最终到达底部。球吧经过的所有位置上的数加起来的和最大是多少 |
| | 状态表示 | F[i,j]表示从左上角走到第i行第j列，和的最大值是多少 |
| | 阶段划分 | 路径的结尾位置（矩阵中的行、列位置，也就是一个二维坐标） |
| | 转移方程 | $F[i,j] = A[i,j] + max\begin{cases} F[i-1,j] \\ F[i-1,j-1] \end{cases} \quad if \quad j > 1$ |
| | 边界 | F[1,1] =A[1,1] |
| | 目标 | max{F[N,j]} |

**矩阵链相乘**

$$m[i,j] = \begin{cases} 0 & if \ i = j \\ \min\{m[i,k] + m[k+1,j] + p[i-1]p[k]p[j]\} & if \ i < j \end{cases}$$
$$i \le k < j$$

```
Algorithm MATRIX-CHAIN-ORDER(p)
 n ← length[p] – 1  //n个矩阵
 for i ← 1 to n
     m[i, i] ← 0
 end for
 for l ← 2 to n       ▷l is the chain length.
     for i ← 1 to n – (l – 1)
          j ← i + (l – 1)
          m[i, j] ← ∞
          for k ← i to j - 1
               q ← m[i, k] + m[k + 1, j] + p[i-1] p[k]p[j] //矩阵乘法次数
               if q < m[i, j]  //最小数乘次数
               then m[i, j] ← q, s[i, j] ← k
               end if
          end for
     end for
 end for
return m and s
```

算法分析

时间复杂度$\Theta(n^3)$

空间复杂度$\Theta(n^2) - >$ 用于存储$m, s$表

### 流水调度问题

> 如果在调度方案的作业排列中，作业i和j满足$\min\{b_i,a_j\}\geq\min\{b_j,a_i\}$，则称作业i和j满足Johnson不等式。
>
> 可以设计下列作业排列方法。这样做能得到最优调度方案
>
> (1)如果$\min\{a_0,a_1,\ldots,a_{n-1},b_0,b_1,\ldots,b_{n-1}\}$是$a_i$，则$a_i$应是最优排列的第一个作业；
>
> (2)如果$\min\{a_0,a_1,\ldots,a_{n-1},b_0,b_1,\ldots,b_{n-1}\}$是$b_j$，则$b_j$应是最优排列的最后一个作业；
>
> (3) 继续（1）和（2）的做法，直到完成所有作业的排列。

### Floyd算法

> Floyd算法基本思想：
>
> - 下面是Floyd设计出的算法，用自底向上地解上面的递推式的方法来处理。它用n+1个n x n维矩阵$D_0$, $D_1$,. . . , $D_n$来计算最短约束路径的长度。
> - 开始时，如果$i \neq j$并且(i,j)是G中的边，则置$D_0[i,i] = 0$，$D_0[i,j]=l[i,j]$；否则置$D_0[i,j] = \infty$。然后做n次迭代，使在第k次迭代后，$D_k[i,j]$含有从顶点i到顶点j，且不经过编号大于k的任何顶点的最短路径的长度。这样在第k次迭代中，可以用公式计算$D_k[i,j]$:

$$D_k[i,j] = \min\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

> 算法7.3 FLOYD
>
> 输入：$n \times n$维矩阵$l[1..n,1..n]$，对于有向图$G=(\{1,2,\ldots,n\},E)$中的边$(i,j)$的长度是$l[i,j]$。
> 输出：矩阵D 使得D[i,j]=i 到 j的距离。
>
> ```
> 1. D←l        {将输入矩阵l复制到D}
> 2. for k←1 to n
> 3.         for i←1 to n
> 4.                 for j←1 to n
> 5.                   D[i,j]=min{D[i,j],D[i,k]+D[k,j]}
> 6.                 end for
> 7.         end for
> 8.end for
> ```
>
> - 显然，算法的运行时间是$\Theta(n^3)$，它的空间复杂性是$\Theta(n^2)$。

时间复杂度$\Theta(n^3)$空间复杂度$\Theta(n^2)$

```
Floyd与Dijikstra算法的对比
1.Dijkstra算法,图所有边权值都为非负的; Floyd算法,不允许所有权值为负的回路,
2.Dijkstra只可以求出任意点到达源点的最短距离; Floyd可以求出任意两点间的最短距离,
3.Dijkstra算法的思想是贪心, Floyd算法的思想是动态规划
4.Dijkstra时间复杂度O(n2), Floyd算法O(n3)
5.Dijkstra 算法 在网络中用得多，一个一个节点添加，加一个点刷一次路由表。Floyd 算法把所有已经连接的路径都标出来，再通过不等式比较来更改路径。
```

**贪心**

**贪心法并不是从整体最优考虑，它**所做出的选择只是在某种意义上的局部最优。

> 贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。

**最短路问题Dijikstra**

## 算法8.1 DIJKSTRA

```
输入:含权有向图G=(V,E), where V={1,2,…,n}.
输出: G中顶点1到其他顶点的距离。
1. X={1};  Y←V-{1}; λ[1]←0
2. for y←2 to n
3.          if y 相邻于 1 then λ[y]←length[1,y]
4.          else λ[y]← ∞
5.          end if
6.end for
7.for j←1 to n-1
8.          令y ∈Y 使得λ[y] 最小
9.                    X←X ∪{y}          {将顶点y 加入X}
10.         Y←Y-{y}              {将顶点y 从Y中删除}
11.         for 每条边(y,w)
12.                   if w ∈Y and λ[y]+length[y,w]< λ[w] then
13.                             λ[w]← λ[y]+length[y,w]
14.         end for
15. end for
```

朴素Djkstra算法分析：时间复杂度$\Theta(n^2)$

**堆优化的Dijikstra (SHORTESTPATH)**

## 算法8.2 SHORTESTPATH

```
输入:含权有向图G=(V,E), where V={1,2,..,n}.
输出: G中顶点1到其他顶点的距离。假设已有一个空堆H。
1.Y←V-{1}; λ[1]←0;  key(1)← λ[1]
2.for y←2 to n
3.          if y 邻接于1 then
4.               λ[y]=length[1,y]
5.               key(y)← λ[y]
6.               INSERT(H,y)
7.          else
8.               λ[y]← ∞
9.               key[y]← λ[y]
10.         end if
11. end for
```

NEXT PAGE

**算法8.2 SHORTESTPATH**

```
12. for j←1 to n-1
13.         y←DELETEMIN(H)
14.         Y←Y-{y}   {将顶点y从Y删除}
15.         for每个邻接于y的顶点w ∈Y
16.            if λ[y]+length[y,w]< λ[w] then
17.                  λ[w]← λ[y]+length[y,w]
18.                  key(w)← λ[w]
19.            end if
20.            if w    H then INSERT(H,w)
21.            else SIFTUP(H,H-1(w))  {H-1(w)返回w在H中的位置}
22.            end if
23.         end for
24. end for
```

堆优化Djikstra算法分析 时间复杂度$\Theta(mlogn)$

注意这里图的输入必须是邻接表的形式，如果是邻接矩阵->$\Theta(n^2)$

**最小生成树(Kruscal) 边角度**

**算法8.3 KRUSKAL**

输入:包含n个顶点的含权连通无向图G=(V,E)。
输出:
由G生成的最小耗费生成树T组成的边的集合。

```
1. 按非降序权重将E中的边排序
2.for 每条边v∈V
3.          MAKESET{v}
4.end for
5.T={}
6.while |T|<n-1
7.          令(x,y)为E中的下一条边.
8.          if FIND(x) ≠FIND(y) then {检查x，y是否在同一颗树中}
9.                    将(x,y)加入T
10                    UNION(x,y)
11          end if
12. end while
```

需要用到并查集 时间复杂度$\Theta(mlogn)$

**最小生成树(Prim) 顶点角度**

此算法概况如下：

```
1. T←{}; X←{1}; Y←V-{1}
2. while Y ≠{}
3.     设(x,y)是最小权重的边，其中x∈X and y∈Y
4.     X←X∪{y}
5.     Y←Y-{y}
6.     T←T∪{(x,y)}
7. end while
```

详细算法:

**算法 8.4 PRIM　　p154**

输入:含权连通无向图G=(V,E), 其中 V={1,2,..,n}.
输出:由G生成的最小耗费生成树T组成的边的集合。

```
1.T←{};  X←{1};   Y←V-{1}
2.for y←2 to n
3.        if y 邻接于1 then
4.                N[y]←1
5.                C[y]←c[1,y]
6.        else C[y]←∞
7.        end if
8.end for
```

NEXT PAGE

**算法 8.4 PRIM**

```
9.  for j←1 to n-1  {寻找n-1 条边}
10.       令y∈Y  使得 C[y] 最小
11.       T←T∪{y,N[y]}          {将边(y,N[y]) 加入 T}
12.       X←X∪{y}               {将顶点y 加入 X}
13.       Y←Y-{y}               {从Y删除顶点}
14.       for 每个邻接于y的顶点w ∈Y
15.               if  c[y,w]<C[w]   then
16.                       N[w]←y
17.                       C[w]←c[y,w]
18.               end if
19.       end for
20. end for
```

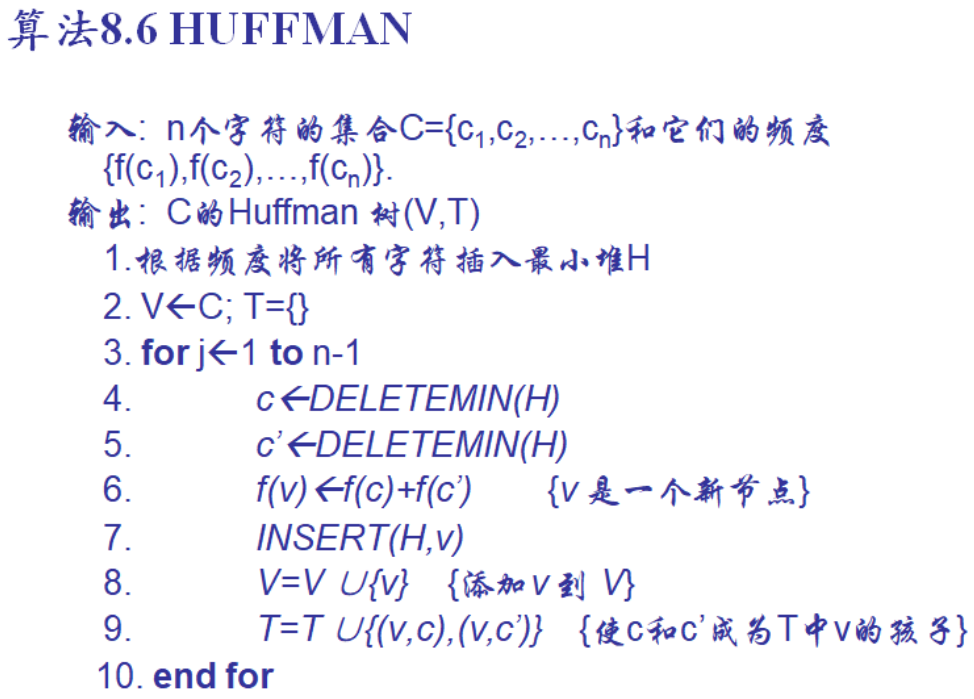
时间复杂度 $\Theta(m+n^2)$

**堆优化的Prim算法（MST）**

> 算法 8.5 MST
>
> 输入:含权连通无向图G=(V,E), where V={1,2,..,n}.
> 输出:由G生成的最小耗费生成树T组成的边的集合。假设我们已有一个空堆H.
>
> ```
> 1.          T←{};   Y←V-{1}
> 2.          for y←2 to n
> 3.    if y 邻接于1 then
> 4.                    N[y]←1
> 5.                    key(y)←c[1,y]
> 6.                    INSERT(H,y)
> 7.     else key(y)←∞
> 8.              end if
> 9. end for                                          NEXT PAGE
> ```
>
> 算法 8.5 MST
>
> ```
> 10. for j←1 to n-1                {查找n-1 条边}
> 11.     y←DELETEMIN(H)
> 12.     T←T∪{(y,N[y])}            {添加边(y,N[y])到T}
> 13.     Y←Y-{y}                   {从Y删除顶点y}
> 14.     for 每个邻接于y的顶点w∈ Y
> 15.         if c[y,w]<key(w)  then
> 16.            N[w]←y
> 17.            key(w)←c[y,w]
> 18.         end if
> 19.         if w     H then INSERT(H,w)
> 20.         else SIFTUP(H,H^-1(w))
> 21.       end for
> 22. end for
> ```

优化后时间复杂度 $\Theta(mlogn)$

> 哈夫曼树/编码




时间复杂度$\Theta(nlogn)$

**回溯**

**回溯法采用深度优先方法搜索遍历问题的解空间**，可以看作是蛮力法穷举搜索的改进。

算法思路

1.写出问题的解向量；
2.画出问题的状态树；
3.写出回溯的伪代码（记得代码里回溯时要有还原操作）；
4.回溯算法基本上都是由循环加递归组成。

---

（1）定义一个解空间，它包含问题的解
（2）用易于搜索的方式组织解空间
（3）深度优先搜索解空间，获得问题的解。

（1）解空间

设问题的解向量为X=(x1,x2,...,xn)，xi的取值范围为有穷集Si 。把xi的所有可能取值组合，称为问题的解空间。每一个组合是问题的一个可能解。

（2）状态空间树：问题解空间的树形式表示
状态空间树分为排列树n!和子集树2^n

所谓回溯法：

•回溯法（backtracking）是一种系统地搜索问题解的方法。为实现回溯，首先需要定义一个解空间（solution space），然后以易于搜索的方式组织解空间，最后用深度优先的方法搜索解空间，获得问题的解。

着色问题

## 算法 13.1 3-COLORREC（递归）

输入: 无向图G=(V,E).
输出: G的顶点的3着色c[1..n]，其中每个c[j]为1,2或3。

```
1. for k←1 to n
2.   c[k]←0
3. end for
4. flag←false
5. graphcolor(1)
6. if flag then output c
7. else output "no solution"
```

过程 graphcolor(k)

```
1. for color=1 to 3
2.    c[k]←color
3.    if c 为合法着色then set flag←true and exit
4.    else if c 是部分的 then graphcolor(k+1)
5. end for
```

4皇后问题

**算法13.3 4-QUEENS(迭代)**
输入: 空。
输出:对应于4皇后问题的解的向量x[1..4]。

```
1.   for k←1 to 4
2.        x[k]←0 {没有皇后放置在棋盘上}
3.   end for
4.   flag←false
5.   k←1
6.   while k ≥1
7.        while x[k] ≤3
8.                x[k]←x[k]+1
9.                if x为合法解then set flag←true且从两个while循环退出
10.               else if x是部分解  then k←k+1 {前进}
11.       end while
12.       x[k]←0
13.       k←k-1 {回溯}
14.  end while
15.  if flag then output x
16.  else output "no solution"
```

**回溯算法一般形式**

递归

## 算法13.4 BACKTRACKREC

输入:集合$X_1,X_2,\ldots,X_n$的清楚的或隐含的描述。
输出:解向量$v=(x_1,x_2,\ldots,x_i)$, $0 \le i \le n$.

```
1.v←()
2.flag←false
3.advance(1)
4.if flag then output v
5.else output "no solution"
```

过程 *advance(k)*

```
1.for each x ∈Xk
2.        xk←x; 将 xk 加入 v
3.        if v为最终解then set flag←true and exit
4.        else if v是部分解then advance(k+1)
5. end for
```

迭代

## 算法13.5 BACKTRACKITER

输入:集合$X_1,X_2,\ldots,X_n$的清楚的或隐含的描述。
输出:解向量$v=(x_1,x_2,\ldots,x_i)$, $0 \le i \le n$.

```
1.v←()
2.flag←false
3.k←1
4.while k ≥1
5.        while Xk没有被穷举
6.                xk←Xk中的下一个元素； 将 xk 加入 v
7.                if v 为最终解then set flag←true
                          退两个while循环退出
8                 else  if v 是部分解 then k←k+1 {前进}
9.        end while
10.       重置Xk, 使得下一个元素排在第一位
11.       k←k-1
12. end while
13. if flag then output v
14. else output "no solution"
```

**子集和问题（回溯）**

```
算法SUMOFSUBREC

    输入:w[1,2,...n] and M, w[i]>0, M >0.
    输出: A solution vector x[1,2,...,n], where each x[i] is 0 or 1.
1. for k←1 to n
2.     x[k]←0
3. end for
4. flag←false
5. s ← 0; r ←w[1]+w[2]+...+w[n]
6. If r ≥ M then SUBOFSUB(s, 1,r) endif
7. if flag then output x
8. else output "no solution"
9. end if

Procedure SUMOFSUB (s,k,r)
1. x[k] ← 1
2. if s+w[k]=M     //x 是合法解
3. then set flag←true and exit
4. else
5.    if s+w[k]+w[k+1]≤M     //x 是部分的      Bk=true, 第k个数选入
6.  then SUMOFSUB(s+w[k],k+1,r-w[k])  //递归调用 第k+1个数
7.    end if
8. end if
9.  if s+r-w[k]≥M and s+w[k+1]≤M        //Bk=false 第k个数不选入
10. then x[k] ← 0, SUMOFSUB(s,k+1,r-w[k])
11.end if
```

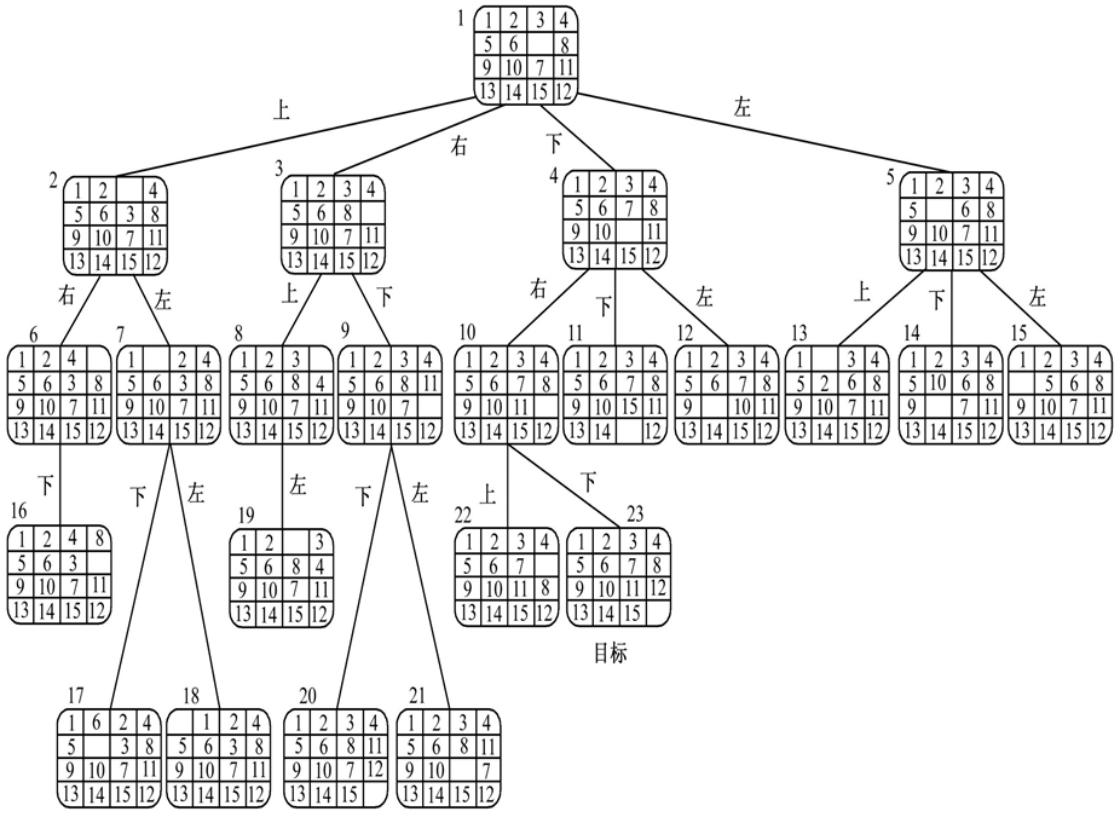
### 分支限界

**分支限界法按广度优先策略遍历问题的解空间**，在遍历过程中，对已经处理的每一个结点根据限界函数估算目标函数的可能值，从中选取使目标函数取得极值（极大或极小）的结点优先进行广度优先搜索，从而不断调整搜索方向，尽快找到问题的解。

**区别与联系**

| | **回溯法** | **分治限界法** |
|---|---|---|
| **对解空间的搜索方式** | 深度优先搜索 | **广度优先**或**最小消耗优先搜索** |
| **存储结点的常用数据结构** | 堆栈 | 队列、优先队列 |
| **结点存储特征** | 活结点的所有可行子结点被遍历后才被从栈中弹出 | 每个结点只有一次成为活结点的机会 |
| **常用应用** | 找出满足约束条件的所有解 | 找出满足约束条件的一个解 |

> 谜问题

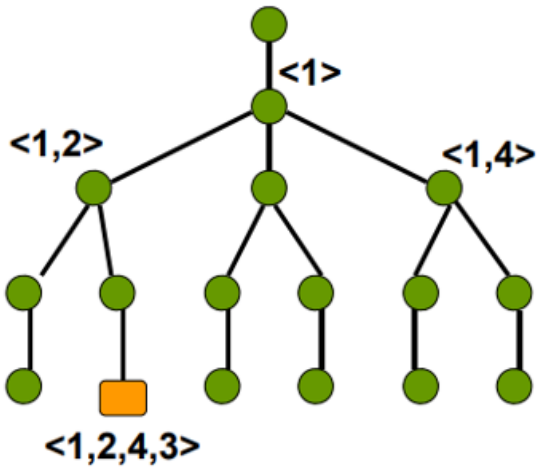


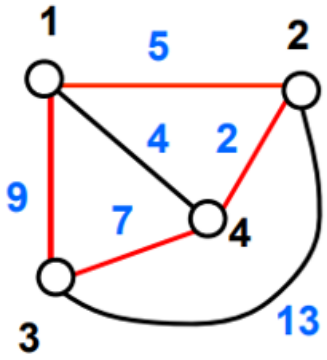

**Algorithm LC-SEARCH**

**Input**: 状态空间树T,T中结点的估计成本函数ĉ。
**Output**: 问题的解（从根到答案结点的路径）

```
1.if T 是答案节点 then 输出T and return; end if
2.E ←T  //E-结点
3.将活结点表初始为空
4.repeat
5.    for E 的每个儿子X
6.          if X 是答案结点 then
7.                    输出从X到T的路径, return
8           end if
9.          call ADD(X)  //将新的活结点X加到活结点表中
10.         PARENT(X) ←E  //E结点标记为X的父结点
11.   end for
12.   if 活结点表为空 then
13.         输出 “无解”, return
14.   end if
15.   call LEAST(E)  //找一个具有最小ĉ值的活结点放到E中，从活结点表中删除
16.end repeat
```

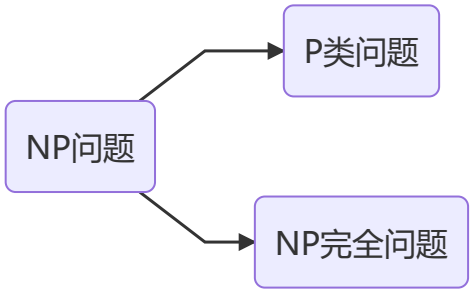
旅行商问题TSP（排列树）




**概念题汇总（NP/P问题）**

> •**P问题**：如果一个问题可以找到一个能在多项式的时间里解决它的算法，那么这个问题就属于P问题
>
> •**NP问题**： NP问题是指可以在多项式的时间里验证一个解的问题，先猜出了一个解，然后在多项式运算复杂度步骤内可以验证这个解正确与否。
>
> •**NP完全问题**：同时满足下面两个条件的问题就是NPC问题。首先，它得是一个NP问题；然后，所有的NP完全问题都可以约化到它。




> Karp的伟大贡献: 利用一个已知的NP-Complete
>
> 问题，通过规约，证明另一个问题也是NP-Complete.

**P类问题定义：**

1.问题解决的每一步都是确定的，只有1种选择（前面的所有算法都是P类问题）。

2.P类问题可在多项式时间(算法时间复杂度不是$n!$和$2^n$)内求解.

**NP类问题定义：**

1.问题的答案可在多项式时间内验证的问题为NP类问题

2.NP类问题或许可以用多项式时间求解，或许不可以。

3．P类问题、NP完全问题是NP类问题的子集。

1、有穷性，算法必须能在执行有限个步骤之后终止；
2、确切性，算法的每一步骤必须有确切的定义；
3、可行性，每个计算步骤都可以在有限时间内完成。
4、输入项，一个算法有0个或多个输入；
5、输出项，一个算法有一个或多个输出；

可用动态规划的显著特征：当前的最优解取决于上一步的最优解

回溯法的基本思想是在一棵含有问题全部可能解的状态空间树上进行深度优先搜索，解为叶子结点。搜索过程中，每到达一个结点时，则判断该结点为根的子树是否含有问题的解，如果可以确定该子树中不含有问题的解，则放弃对该子树的搜索，退回到上层父结点，继续下一步深度优先搜索过程。在回溯法中，并不是先构造出整棵状态空间树，再进行搜索，而是在搜索过程，逐步构造出状态空间树，即边搜索，边构造

---

（一）分治法和动态规划法的区别

共同点：二者都要求原问题具有最优子结构性质，都将原问题分成若干个子问题，然后将子问题的解合并，形成原问题的解。

不同点：动态规划法是将待求解问题分解成若干个相互重叠的子问题，即不同的子问题具有公共的子子问题，而分治法是分解成若干个互不相交的子问题。利用分治法求解，这些子问题的重叠部分被重复计算多次。而动态规划法将每个子问题只求解一次并将其保存在一个数组中，当需要再次求解此子问题时，从数组中获得该子问题的解，从而避免了大量的重复计算。

（二）动态规划法和贪心法的区别

共同点：贪心算法和动态规划算法都要求问题具有最优子结构性质。

不同点：动态规划法用到之前的最优解，贪心则不是，贪心无法解决动态规划的问题，但是动态规划能解决贪心的问题。虽然能够应用贪心算法一定能够应用动态规划法，但是一般来说，贪心算法的效率高于动态规划法，因而还是应用贪心算法。动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每做一次贪心选择就将所求问题简化为规模更小的子问题。

（三）回溯法和分支限界法的区别

共同点：一种在问题的解空间树上搜索问题解的算法。

不同点：求解目标不同，回溯法的目标是找出解空间树满足约束条件的所有解，而分支限界法的求解目标是尽快地找出满足约束条件的一个解；搜索方法不同，回溯法采用深度优先方法搜索解空间，而分支限界法一般采用广度优先或以最小消耗优先的方式搜索解空间树；对扩展结点的扩展方式不同，回溯法中，如果当前的扩展结点不能够再向纵深方向移动，则当前扩展结点就成为死结点，此时应回溯到最近一个活结点处，并使此活结点成为扩展结点。分支限界法中，每一次活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点；存储空间的要求不同，**分支限界法的存储空间比回溯法大得多**，当内存容量有限时，回溯法成功的可能性更大。