算法知识点

1） 什么是算法?
算法是求解某一特定问题的一组有穷规则的集合，它是由若干条指令组成的有穷符号串。

2） 算法的五个重要特性
确定性、可实现性、输入、输出、有穷性

3） 算法设计的质量指标
正确性，可读性，健壮性，效率与存储量

- 算法和程序的区别
  - 程序：一个计算机程序是对一个算法使用某种程序设计语言的具体实现
  - 任何一种程序设计语言都可以实现任何一个算法
  - 算法的有穷性意味着不是所有的计算机程序都是算法

定义 **1.2**

设算法的执行时间$T(n)$，如果存在$T^*(n)$，使得 $\lim_{n\to\infty}\frac{T(n)-T^*(n)}{T(n)}=0$

就称$T^*(n)$为算法的渐近时间复杂性。

- 定义1 ：如果存在两个正常数c和n0，对于所有的n≥n0，有|f(n)| ≤ c|g(n)|，则记作f(n) = O(g(n))
- 含义：如果算法用n值不变的同一类数据在某台机器上运行时，所用的时间总是小于|g(n)|的一个常数倍。所以g(n)是计算时间f(n)的一个上界函数。 f(n)的数量级就是g(n)。f(n)的增长最多像g(n)的增长那样快，试图求出最小的g(n)，使得f(n) = O(g(n))。
- 常见的多项式限界函数有：
  O(1) < O(logn) < O(n) < O(nlogn) < O(n2) < O(n3)
- 指数时间算法：计算时间用指数函数限界的算法。常见的指数时间限界函数：
  O(2n) < O(n!) < O(n^n)

- 定义1.2 如果存在两个正常数c和n0，对于所有的n≥n0，有|f(n)| ≥ c|g(n)|,则记作f(n) = Ω(g(n))
- 含义：如果算法用n值不变的同一类数据在某台机器上运行时，所用的时间总是不小于|g(n)|的一个常数倍。所以g(n)是计算时间f(n)的一个下界函数。f(n)的增长至少像

g(n)的增长那样快，试图求出“最大”的g(n)，使得f(n) = Ω(g(n))。

- 定义1.3 如果存在正常数c1，c2和n0，对于所有的n≥n0，有c1|g(n)| ≤|f(n)| ≤ c2|g(n)|，则记作$f(n)=\Theta(g(n))$
- 含义：算法在最好和最坏情况下的计算时间就一个常数因子范围内而言是相同的。可看作：既有f(n) = Ω(g(n))，又有f(n) = O(g(n))，记号表明算法的运行时间有一个较准确的界

- 分治算法整体思想：分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治法所能解决的问题一般具有以下几个特征：

- **该问题的规模缩小到一定的程度就可以容易地解决；**
- **该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质**
- **利用该问题分解出的子问题的解可以合并为该问题的解；**
- **该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。**

分治法的基本步骤

（1）分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

（2）解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；

（3）合并：将各个子问题的解合并为原问题的解。

Master定理（递归复杂度判定定理）

设常数a >= 1，b > 1，f(n)为函数，

T(n)为非负整数，$T(n)=aT(n/b)+f(n)$ ，则有：

1.若 $f(n)=O(n^{\log_b a-\varepsilon}),\varepsilon>0$，则 $T(n)=\Theta(n^{\log_b a})$ ；

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

3.若 $f(n)=\Omega(n^{\log_b a+\varepsilon}),\varepsilon>0$ ，并且对于某个常数c<1和充分大的n有 $af(n/b)\le cf(n)$ ，那么 $T(n)=\Theta(f(n))$ 。

- 一个分治法将规模为n的问题分成k个规模为n／k的子问题去解。设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用f(n)个单位时间。用T(n)表示该分治法解规模为|P|=n的问题所需的计算时间，则有：

$$T(n)=\begin{cases}O(1) & n=1\\ kT(n/k)+f(n) & n>1\end{cases}$$

---

1. 二分搜索技术：给定已按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x，因此整个算法在最坏情况下的计算时间复杂性为O(logn)
2. 合并排序：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

```
public static void mergeSort(Comparable a[], int left, int right)
   {
      if (left<right) {//至少有2个元素
      int i=(left+right)/2;  //取中点
      mergeSort(a, left, i);
      mergeSort(a, i+1, right);
      merge(a, b, left, i, right);  //合并到数组b
      copy(a, b, left, right);    //复制回数组a
      }
   }
```

$$T(n)=\begin{cases}O(1) & n\le 1\\ 2T(n/2)+O(n) & n>1\end{cases}$$

- **因此可以得出$T(n)=O(nlogn)$**

3. 快速排序

```
private static void qSort(int p, int r)
   {
      if (p<r) {
        //以a[p]为基准元素将a[p:r]划分成3段a[p:q-1],a[q]和a[q+1:r]，
        // 使得a[p:q-1]中任何元素小于等于a[q]，
        // a[q+1:r]中任何元素大于等于a[q]。下标q在划分过程中确定。
        int q=partition(p,r);
        qSort (p,q-1); //对左半段排序
        qSort (q+1,r); //对右半段排序
        }
   }
template<class Type>
int Partition (Type a[], int p, int r)
{
        int i = p, j = r + 1;
        Type x=a[p];
        // 将< x的元素交换到左边区域
        // 将> x的元素交换到右边区域
        while (true) {
           while (a[- -j] >=x) ;
           Swap(a[i], a[j]); //交换
           while (a[++i] <=x && i<r );
           if (i >= j) break;
           Swap(a[i], a[j]); //交换
           }
          return j;
}
```

- **因此可以得出$T(n)=O(nlogn)$**

4. 线性时间选择问题

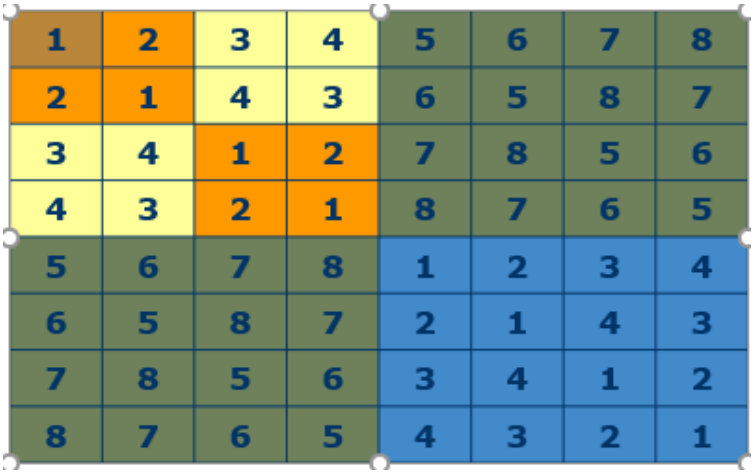
```
int RandSelect(int A[], int start, int end, int k) {
      if (start ==end) return A[start];
      int i= RandomizedPartition(A, start, end); //划分元位置i
      int n=i-start+1;  // 左子数组A[start :i]的元素个数
      if (k<=n)
            return RandSelect (A, start, i, k);
      else
            return RandSelect(A, i+1, end, k-n);
}
```

- 若k<j,则第k小元素在$A(1:j-1)$中，再对之进一步划分。
- 若k=j,则$A(j)$就是第k小元素
- 若k>j,则第k小元素在$A(j+1:n)$中，再对之进一步划分，此时需要把第k小元素改成找到第$k-j+1$小的元素

$$T(n)=\begin{cases} C_1 & n<75 \\ C_2n+T(n/5)+T(3n/4) & n\geq 75 \end{cases}$$

- n个元素的数组调用select( )需要T(n)
- 找中位数的中位数x需要T(n/5)
- 使用基准n进行划分所得到的两个子数组分别至多有3n/4个元素
- **因此可以得出$T(n)=O(n)$**

5. 循环赛日程表问题
   - 将所有的选手分为两半，n个选手的比赛日程表可以通过n/2个选手设计的比赛日程表来决定
   - 递归地用对选手进行分割，直到只剩下2个选手时，只要让这2个选手进行比赛就可以了



1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

动态规划算法的两个基本要素：**1. 最优子结构性质：原问题的最优解包含了其子问题的最优解。**2. 子问题重叠性质：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。

1. 矩阵连乘问题

```
void MatrixChain(int *p, int n, int **m, int **s)
{
        for (int i = 1; i <= n; i++) m[i][i] = 0;
        for (int r = 2; r <= n; r++)       //r表示链长，取值2~n
           for (int i = 1; i <= n - r+1; i++) {
              int j=i+r-1;                 //j依次取值i+1,i+2,……,n
              m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
```

```
                    //即m[i][j] = m[i][i]+m[i+1][j]+ p[i-1]*p[i]*p[j]
                s[i][j] = i;                        //i为初始断开位置
                //依次设断开位置为i+1,i+2,……
                for (int k = i+1; k < j; k++) {
                   int 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;}
             }
          }
}
```

最优子结构性质

$$m[i][j] = \min_{i \le k < j} m[i][k] + m[k+1][j] + p_{i-1}p_k p_j$$

- **因此可以得出$T(n) = O(n^3)$，占用的空间为$O(n^2)$**

m	A1	A2	A3	A4	A5	A6
A1	0	15750				
A2		0	2625			
A3			0	750		
A4				0	1000	
A5					0	5000
A6						0

$m[1,2] = m[1,1] + m[2,2] + P_0P_1P_2\ (k=1) = 0+0+30\times35\times15 = 15750$

$m[2,3] = m[2,2] + m[3,3] + P_1P_2P_3\ (k=2) = 0+0+30\times15\times5 = 2625$

$m[3,4] = m[3,3] + m[4,4] + P_2P_3P_4\ (k=3) = 0+0+15\times5\times10 = 750$

$m[4,5] = m[4,4] + m[5,5] + P_3P_4P_5\ (k=4) = 0+0+5\times10\times20 = 1000$

$m[5,6] = m[5,5] + m[6,6] + P_4P_5P_6\ (k=5) = 0+0+10\times20\times25 = 5000$

2. 最长公共子序列问题

最优子结构性质

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

已知：

x	1	2	3	4	5	6	7
	A	B	C	B	D	A	B

y	1	2	3	4	5	6
	B	D	C	A	B	A

b	y_i	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
A	0	2	2	2	1	3	1
B	0	1	3	3	2	1	3
C	0	2	2	1	3	2	2
B	0	1	2	2	2	1	3
D	0	2	1	2	2	2	2
A	0	2	2	2	1	2	1
B	0	1	2	2	2	1	2

---

3. 0-1背包问题

- 令V(i, j)表示在前i(1≤i≤n)个物品中能够装入容量为j（1≤j≤C）的背包中的物品的最大值，则可以得到如下动态规划函数：其中，$V(i,0)=V(0,j)=0$

最优子结构性质

$$V(i,j)=\begin{cases} V(i-1,j) & j<w_i \\ max\{V(i-1,j),\ V(i-1,j-w_i)+v_i\} & j\geq w_i \end{cases}$$

贪心算法总是作出在当前看来最好的选择，即所作的选择只是局部最优选择。
可用贪心算法求解的问题的基本要素： 最优子结构性质/贪心选择性质。

- 具有最优子结构性质是一个问题可用动态规划算法或贪心算法求解的一个关键特征。
- 贪心选择性质:所求问题的整体最优解可以通过一系列局部最优的选择来完成即贪心选择来达到。
- 贪心算法和动态规划算法的差异
  - 相同点：都具有最优子结构性质
  - 区别：动态规划算法每步所作的选择往往依赖于相关子问题的解。只有解出相关子问题才能作出选择。
  - 贪心算法仅在当前状态下作出最好选择，即局部最优选择，但不依赖于子问题的解
  - 贪心：自顶向下，动态规划：自底向上

---

1. 活动安排问题

   - 在安排时应该将结束时间早的活动尽量往前安排，好给后面的活动安排留出更多的空间，从而达到安排最多活动的目标。
   - 贪心准则应当是：在未安排的活动中挑选结束时间最早的活动安排。

```
Public static void greedySelector(int s[ ] , int f[ ],  bool a [ ])
{     int n=s.length-1;
       a[0]=true;
       int j=0;   int count=1;
       for (int i=1;i<=n;i++) {
            if (s[i]>=f[j]) { a[i]=true; j=i; count++; }
            else a[i]=false;
       }
    return count;
}
```

2. 背包问题：**背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。**

   - 首先计算每种物品单位重量的价值vi/wi，然后，依贪心选择策略，将尽可能多的单位重量价值最高(即vi/wi尽可大的)的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。若最后一个物品不能全部装入时，仅装其一部分使背包装满即可。

```
void Knapsack(int n, float M, float v[ ], float w[ ], float x[ ] )
{
      Sort (n, v, w);//将各种物品按单位重量价值排序
      int i;
      for ( i = 1; i <= n; i ++ ) x[i] = 0;//将解向量初始化为零
      float c = M; // 是背包剩余容量初始化为M
      for ( i = 1;i <= n; i ++ ) {
           if ( w[i]>c) break;
           x[ i ] = 1;
           c -= w[i];
       }
       if ( i <= n ) x[i]=c/w[i];
}
```

   - **因此可以得出**$T(n) = O(nlogn)$

- 回溯法从根结点出发，按照深度优先策略搜索（遍历）解空间树，搜索满足约束条件的解。
- 初始时，根结点成为一个活结点，同时也称为当前的扩展结点。在当前扩展结点处，搜索向纵深方向移至一个新结点。这个新结点成为一个新的活结点，并成为当

前的扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前的扩展结点就成为一个死结点（即不再是一个活节点）。此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。

- 回溯法求解过程
  1. 针对所给问题，定义问题的解空间；
  2. 确定易于搜索的解空间结构；
  3. 深度优先搜索解空间，并在搜索中用剪枝函数避免无效搜索。
- 如果解空间树中从根结点到叶结点的最长路径的长度为$h(n)$，则回溯法所需的计算空间通常为$O(h(n))$。而显式地存储整个解空间则需要$O(2h(n))$或$O(h(n)!)$内存空间。
- 在搜索过程中，通常采用两种策略避免无效搜索：
  - 用约束条件剪去得不到可行解的子树；
  - 用限界函数剪去得不到最优解的子树。
- 当所给问题是从n个元素的集合S中找出满足某种性质的子集时，解空间为子集树。（装载问题）
- 当所给问题是从n个元素的集合S中找出满足某种性质的排列时，解空间为排列树。（N-皇后问题）

---

1. 装载问题
   - 解空间：子集树
   - 可行性约束函数(选择当前元素)：$\sum_{i=1}^{n} w_i x_i \leq c_1$
   - 限界函数(不选择当前元素)：当前载重量cw+剩余集装箱的重量r>当前最优载重量bestw

```
void backtrack (int i) {
    if (i > n){
        if(wc > wm) wm = wc; return;
    }
    wr -= w[i];
    if (wc + w[i] <= c){  // x[i] = 1; 搜索左子树
        wc += w[i];
        backtrack(i+1);
        wc -= w[i];
    }
    if (wc + wr > wm){  // x[i] = 0; 搜索右子树
        backtrack(i+1);
    }
    wr += w[i];
}
```

2. N-皇后问题
    - 隐约束：
        - 不同列：$x_i \neq x_j (i \neq j)$
        - 不处于同一正、反对角线：$|i-j| \neq |x_i - x_j|$

```
bool Bound(int k){
    for (int i = 1; i < k; i++){
        if ((abs(k-i)==abs(x[i]-x[k]))||(x[i]==x[k]))
            return false;
    }
    return true;
}

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

分支限界法的基本思想

1. 以广度优先或以最小耗费（最大效益）优先的方式搜索；
2. 每一个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。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。
3. 从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

- 分支限界法与回溯法
    - 求解目标：回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解。
    - 搜索方式的不同：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树。

- 常见的两种分支限界法
  - 队列式(FIFO)分支限界法：按照队列先进先出（FIFO）原则选取下一个节点为扩展节点。
  - 优先队列式分支限界法：按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

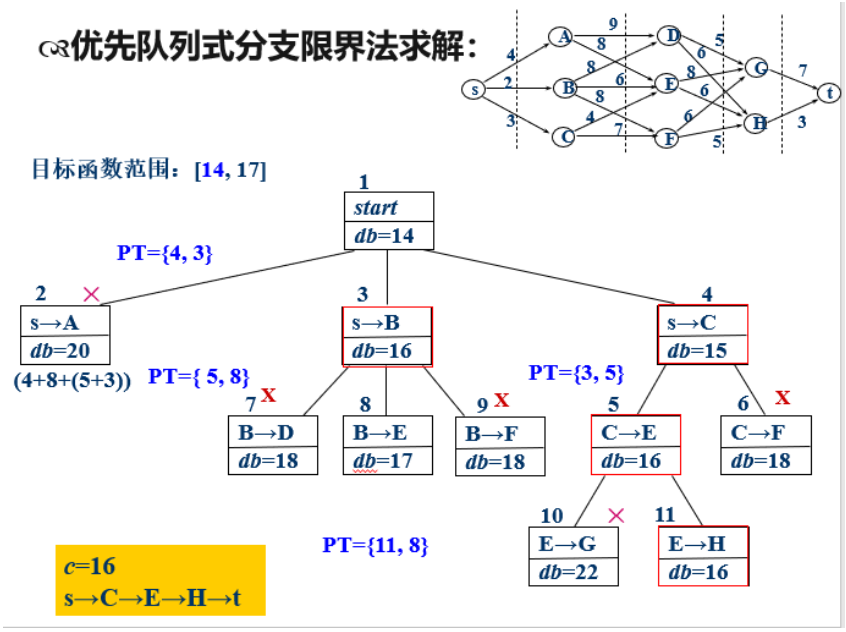
---

1. 0-1背包问题：物品已经按照单位重量价值从大到小排列
   - 约束条件：$\sum_{i=1}^{n} w_i x_i \leq W$
   - 目标函数：$V = max \sum_{i=1}^{n} v_i x_i$
   - 下界一般是使用贪心算法
   - 上界是把所有空间分配给单位价值最高的物品
   - 限界函数：前i个物品获得的价值 + 剩余容量全部装入物品i+1，最多能够获得的价值 $ub = v + (W - w) \times (v_{i+1} / w_{i+1})$
   - **0/1背包问题，采用最大优先队列式分支限界法—PT队列是采用的最大堆，每次取排在队首的结点作为下一步的扩展结点**

---

2. 单元路径最短问题：**采用优先队列式分支限界法，并用一极小堆来存储活结点表。其优先级是结点所对应的当前路长**
   - 下界：把每一段最小的代价相加
   - 上界：找到一个可行解

$$db = \sum_{j=1}^{i} c[r_j][r_{j+1}] + \min_{\langle r_i+1, v_p \rangle \in E} \{c[r_{i+1}][v_p]\} + \sum_{j=i+2}^{k} \text{第 } j \text{ 段的最短边}$$



3. 装载问题

- 优化方案：
  - 算法MaxLoading初始时bestw=0，直到搜索到第一个叶结点才更新bestw。在搜索到第一个叶结点前，总有Ew+r>bestw, 此时右子树测试不起作用。
  - 为确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新bestw的值。不过基本上也就是在找到第一个叶子节点之前起作用

```
while(true)    {
        //检查左儿子
        Type wt=Ew+w[i];    //wt为左儿子节点的重量
        if(wt<=c)    //若装载之后不超过船体可承受范围
            if(wt>bestw) { //更新最优装载重量，提前更新最优值
                bestw=wt;
                if(i<n) Q.Add(wt);     //将左儿子添加到队列
            }
        //将右儿子添加到队列
        if(Ew+r>bestw &&  i<n)
            Q.Add(Ew);    //可能含有最优解
        Q.Delete(Ew);     //取下一个节点为扩展节点并将重量保存在Ew
        if(Ew==-1) {  //检查是否到了同层结束
            if(Q.IsEmpty()) return bestw;     //遍历完毕，返回最优值
            Q.Add(-1);     //添加分层标志
            Q.Delete(Ew);     //取下一扩展结点
            i++;
            r-=w[i];     //剩余集装箱重量
        }
    }
```