



第8章 缩小规模算法

n 本章的主要内容

1、分治与递归

- u 二分搜索技术，归并排序，快速排序（分治）

2、动态规划

- u 矩阵连乘问题

3、贪心算法

- u 背包问题





递归算法设计

- n 一个直接或间接调用自身的算法称为递归算法。
- n 一个函数是用自身给出定义的函数称为递归函数。
- n 使用递归算法分两类：
 - 1) 自身的递归特性（采用递归方式给出定义），特别适用用递归方式描述，如二叉树。
 - 2) 本身没有明显的递归结构，但用递归策略求解会使设计出的算法易懂且易于分析。





分治算法设计思想-1

n 分治法的设计思想是：将一个规模为 n 的问题分解为 k 个规模较小的子问题，这些子问题相互独立且与原问题相同。递归地解这些子问题，然后将子问题的解合并得到原问题的解。

一般算法设计模式如下：

```
DataType divide-and-conquer (P)  
{  
    if ( |P| <= n0 ) Adhoc( P );  
    divide P into smaller subinstances;  
    P1, P2, ... ,Pk;  
    for ( i=1; i<=k; i++)  
        y[i] = divide-and-conquer(Pi);  
    return Merge(y1, y2, ...,yk);  
}
```





分治算法设计思想-2

- n **|P|**表示问题的规模;
- n **n0**是阈值, 表示当问题**P**的规模不超过**n0**时, 问题已容易解出, 不必再继续分解;
- n **Adhoc(P)**是分治法中的基本子算法;
- n 算法**Merge()**是该分支算法中的合并子算法。
- n 用分治法设计算法时, 最好使子问题的规模大致相同, 即将一个问题分割成大小大致相等的**k**个子问题的处理方法行之有效。
- n 分治法设计的程序一般是递归算法, 因此可以用递归方程进行分析。





二分搜索技术

- n 给定一排好序的 n 个元素 $R[0] \sim R[n-1]$ ，现要在这 n 个元素中找出一特定元素 x ？
- n 二分搜索算法的基本思想：
 - 1) 是将 n 个元素分成个数大致相同的两部分，取 $R[n/2]$ 与 x 进行比较；
 - 2) 如果 $R[n/2]$ 与 x 相等，则找到 x ，算法终止；
 - 3) 如果 $x < R[n/2]$ ，则只在数组 R 的左半部分继续搜索 x ；如果 $x > R[n/2]$ ，则只在数组 R 的右半部分继续搜索 x 。





二分搜索算法

```
int BINSEARCH(DataType R[ ], DataType x)
/* 在有序表R中进行折半查找，成功时返回结点的位置，失败时返回-1 */
{
    int low,mid,high; /*low 和high表示当前查找区间的下界和上界*/
    low=0; high=n-1; /* 置查找区间的上、下界初值 */
    while (low<=high) { /* 当前查找区间非空 */
        mid=⌈(low+high)/2⌉;
        if (x==R[mid]) return mid; /* 查找成功返回 */
        if (x<R[mid]) high=mid-1; /* 缩小查找区间为数组的左半部分 */
        else low=mid+1; /* 缩小查找区间为数组的右半部分 */
    }
    return -1; /* 查找失败 */
} /* BINSEARCH */
```





二分搜索举例-1

例如，数组元素的有序序列为：

5,10,19,21,31,37,42,48,50,55，现要查找x为**19**及**66**的元素

查找 k=19 的记录

05	10	19	21	31	37	42	48	50	55
↑					↑				↑
low					mid				high

此时 $mid = \lceil (low + high) / 2 \rceil$ ，由于 $k(19) < 37$ ，则下一步在 $R[1..5]$ 中查找

05	10	19	21	31	37	42	48	50	55
↑		↑		↑					
low		mid		high					

由于 $k=19$ 与 $R[mid].key$ 相等，则查找成功。



二分搜索举例-2

查找 k=66 的记录

05 10 19 21 31 37 42 48 50 55

↑
low

↑
mid

↑
high

 $k > R[mid].key$

low

mid

↑
high


05 10 19 21 31 37 42 48 50 55

$$k > R[mid].key$$

low mid high

 $k > R[mid].key$

↑
high

low 

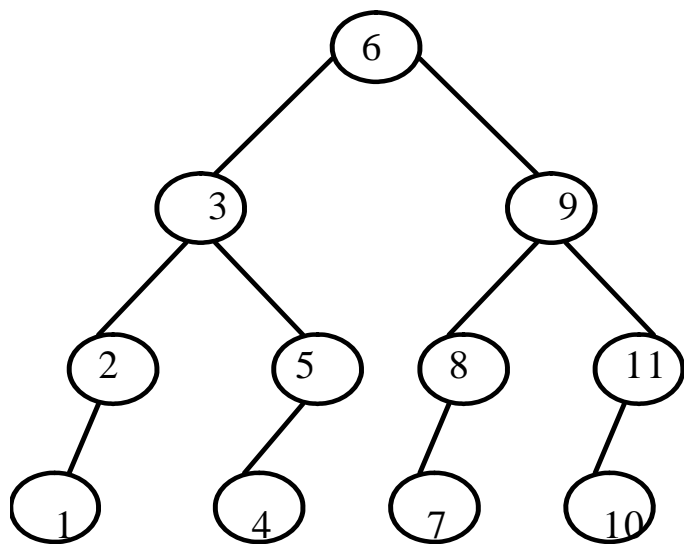
由于 $low > high$, 则说明查找失败。

二分搜索最坏情况下的比较次数是 $O(\log_2 n)$ 。



折半查找判定树

- n 用当前查找区间的中间位置上的记录作为根，左子表和右子表中的记录分别作为根的左子树和右子树，由此得到的二叉树称为**折半查找判定树**，树中结点内的数字表示该结点在有序表中的位置。



折半查找的过程恰好是走了一条从根到被查结点的路径，关键字进行比较的次数即为被查结点在树中的层数。
因此折半查找成功时进行的比较次数最多不超过树的深度。

折半查找的平均查找长度为：

$$ASL_{bin} \approx \log_2(n+1) - 1 \quad (n \text{ 很大})$$



归并排序

- n 归并排序（**Merge Sort**）是将两个或两个以上的有序表合成一个新的有序表。
- n 归并排序基本思想是：
 - 1) 当 $n=1$ 时，终止排序；
 - 2) 否则将待排序的元素分成大致相同的两个子集合，分别对两个子集合进行归并排序；
 - 3) 将排好序的子集合合并成所要求的排序结果。





归并排序算法的递归描述

```
void MergeSort(DataType R[], int left, int right)
{
    if (left < right) { //至少有两个元素
        i = (left + right) / 2; //取中点
        MergeSort ( R, left, i );
        MergeSort ( R, i, right );
        Merge ( a, b, left, i, right ); //归并到b数组
        Copy ( a, b, left, right); //复制回a数组
    }
}
```

上述归并算法的递归过程只是将待排序的顺序表一分为二，直至待排序顺序表中只剩下一个元素为止；然后不断合并两个排好序的数组段。

可以从分治策略入手，消除算法中的递归。





二路归并排序

n 算法思想:

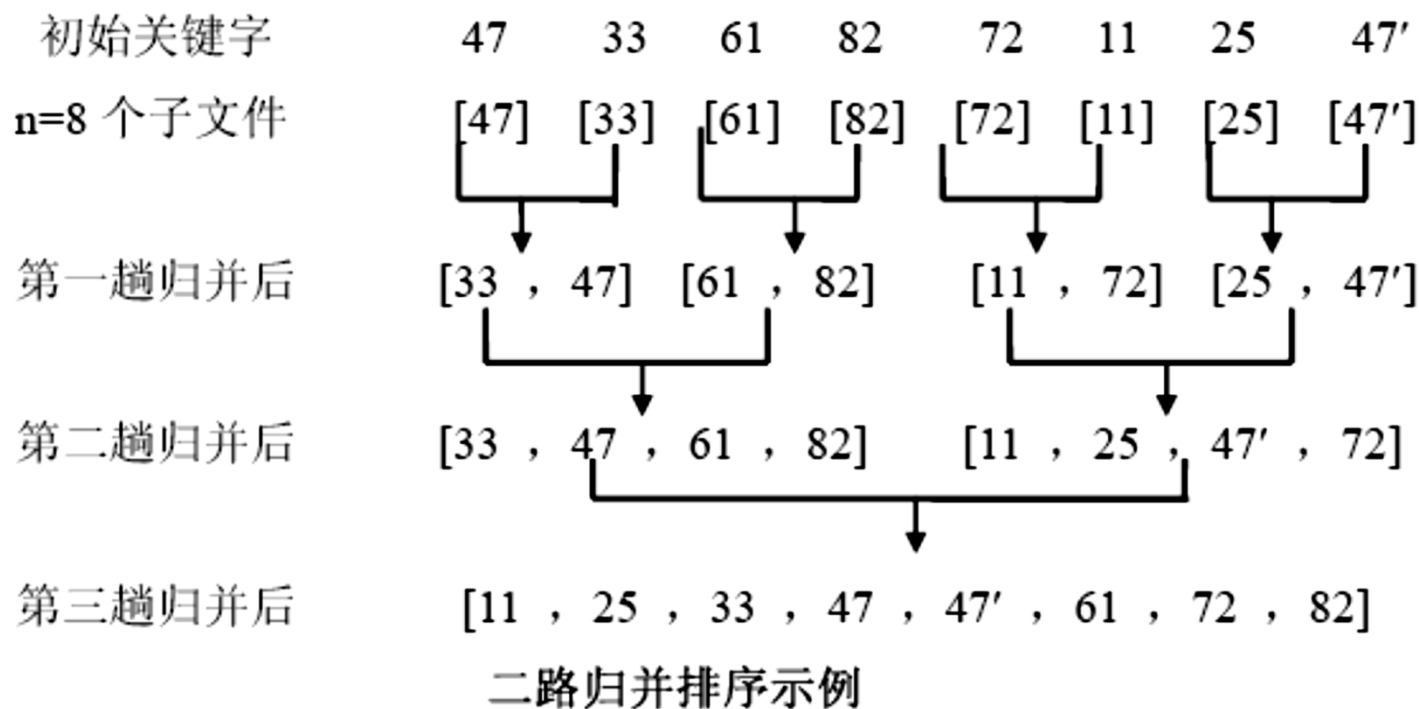
假设初始表含有 n 个记录，则可看成是 n 个有序的子表，每个子表的长度为1，然后两两归并，得到 $\lceil n/2 \rceil$ 个长度为2或1的有序子表，再两两归并，.....如此重复，直至得到一个长度为 n 的有序子表为止。





二路归并排序算法举例

n 对于一组待排序的记录，其关键字分别为：
47,33,61,82,72,11,25,47'





归并算法

```
void Merge ( DataType R[], DataType R1[], int low, int mid, int high)
/* R[low]到R[mid]与R[mid+1]到R[high]是两个有序文件 */
/* 结果为一个有序文件在R1[low]到R1[high]中 */
{
    int i,j,k;
    i=low;j=mid+1;k=low;
    while ((i<=mid) && (j<=high))
        if ( R[i] <= R[j] )    /* 取小者复制 */
            R1[k++]=R[i++];
        else
            R1[k++]=R[j++];
    while (i<=mid) R1[k++]=R[i++];    /* 复制第一个文件的剩余记录 */
    while (j<=high) R1[k++]=R[j++];    /* 复制第二个文件的剩余记录 */
}    /* Merge */
```





一趟归并算法

```
void MergePass ( DataType R[ ], DataType R1[ ], int length )
/*对R做一趟归并，结果放在R1中*/
/* length是本趟归并的有序子件的长度 */
{
    int i,j;
    i=0 ;   /* i向第一对子文件的起始点 */
    while (i+2*length-1<n); { /* 归并长度为length的两个子文件 */
        Merge ( R, R1 , i, i+length-1 , i+2*length-1 );
        i=i+2*length;   /* i指向下一对子文件的起始点 */
    }
    if (i+length-1)<n-1) /* 剩下两个子文件，其中一个长度小于length */
        Merge ( R, R1, i, i+length-1, n-1 );
    else /* 子文件个数为奇数 */
        for (j=i;j<n;j++) R1[j]=R[j]; /* 将最后一个文件复制到R1中 */
} /* MergePass */
```





二路归并排序算法

n “二路归并”排序就是调用“一趟归并”过程，将待排序文件进行若干趟归并，每趟归并后有序子文件的长度**length**扩大一倍。

```
void MergeSort( DataType R[ ] ) /* 对R进行二路归并排序 */
{
    int length;
    length=1;
    while (length<n) {
        MergePass ( R, R1 , length ); /* 一趟归并，结果在R1中 */
        length=2*length;
        MergePass ( R1, R, length ); /* 再次归并，结果在R中 */
        length=2 *length;
    }
} /* MergeSort */
```

二路归并排序算法的时间复杂度为 $O(n\lg n)$





快速排序

- n 快速排序算法是基于分治策略的另一个排序算法。其**基本思想**是：对输入的数组 $R[l:h]$ 按以下三个步骤进行排序：
- n **步骤一：分解**，以 $R[l]$ 为基准元素将 $R[l:h]$ 划分为三段 $R[l:q-1]$ 、 $R[q]$ 和 $R[q+1:h]$ ，使 $R[l:q-1]$ 中任何元素不大于 $R[q]$ ； $R[q+1:h]$ 中任何一个元素大于 $R[q]$ ，下标 q 在划分过程中确定。
- n **步骤二：递归求解**，通过递归调用快速排序算法分别对 $R[l:q-1]$ 和 $R[q+1:h]$ 进行排序。
- n **步骤三：合并**，由于对 $R[l:q-1]$ 和 $R[q+1:h]$ 的排序是就地进行的，因此实际上不需要进一步的合并计算。





快速排序的算法

- n 可设置两个指针*i*和*j*，它们的初值分别为*i=l*和*j=h*。
- n 设基准为无序区中的第一个记录*R[i]*（即*R[l]*），这时*q = i*。
- n 令*j*自*h*起向左扫描，直到找到第一个小于*R[q]*的元素*R[j]*，将*R[j]*移至*q*所指的位置上（这相当于交换了*R[j]*和基准*R[q]*的位置，使小于基准元素的元素移到了基准的左边）；
- n 然后，令*q = j*，且*i*自*i+1*起向右扫描，直至找到第一个大于*R[q]*的元素*R[i]*，将*R[i]*移至*j*指的位置上（这相当于交换了*R[i]*和基准*R[q]*的位置，使大于基准元素的元素移到了基准的右边）；
- n 接着令*q = i*，且*j*自*j-1*起向左扫描，如此交替改变扫描方向，从两端各自往中间靠拢，直至*i=j*时，*q = i = j*便是基准*x (=R[i])*的最终位置，将*x*放在此位置上就完成了第一次划分。



快速排序的算法实现-1

```
int Partition ( DataType R[ ], int l, int h ) /* 返回划分后被定位的基准记录的位置 */
/* 对无序区R[l]到R[h]做划分 */
{
    int i, j, q;
    DataType x;
    i = l; j = h;
    q=i;          /* 初始化, q为基准 */
    x= R[i];
    do {
        while ( ( R[j] >= x ) && ( i<j ) )
            j--; /* 从右向左扫描, 查找第一个小于x的元素 */
        if ( i < j ) R[i++] = R[j]; /* 交换R[i]和R[j] */
        while ( ( R[i] <= x ) && ( i<j ) )
            i++; /* 从左向右扫描, 查找第一个大于x的元素 */
        if ( i < j ) R[j--] = R[i]; /* 交换R[i]和R[j] */
    } while ( i != j );
    R[i] = x; /* 基准已被最后定位在i处 */
    return i;
} /* Partition */
```





快速排序的算法实现-2

```
void QuickSort ( DataType R[ ], int s1, int t1 ); /* 对R[s1]到R[t1]做快速排序 */
{
    int i;
    if (s1<t1) { /* 只有一个元素或无元素时无须排序 */
        i = Partition ( R, s1, t1 ); /* 对R[s1]到R[t1]做划分 */
        QuickSort ( R, s1, i-1 ); /* 递归处理左区间 */
        QuickSort ( R, i+1, t1 ); /* 递归处理右区间 */
    }
} /* QuickSort */
```

注意：对整个文件R[0]到R[n-1]排序，只需调用QUICKSORT(R,0,n-1)即可。

- 快速排序有非常好的时间复杂度，它优于各种排序算法，对n个记录进行快速排序的平均时间复杂度为 $O(n\lg n)$ 。





快速排序的算法举例-1

初始关键字	[(49) 38 65 97 76 13 27 49']
j 向左扫描	[(49) 38 65 97 76 13 27 49']
第一次交换后	[27 38 65 97 76 13 (49) 49']
i 向右扫描	[27 38 65 97 76 13 (49) 49']
第二次交换后	[27 38 (49) 97 76 13 65 49']
j 向左扫描,位置不变,第三次交换后	[27 38 13 97 76 (49) 65 49']
i 向右扫描,位置不变,第四次交换后	[27 38 13 (49) 76 97 65 49']
j 向左扫描	[27 38 13 (49) 76 97 65 49']

(a) 一次划分过程



快速排序的算法举例-2

初始关键字	[49	38	65	97	76	13	27	49']
一趟排序之后	[27	38	13]	49	[76	97	65	49']
二趟排序之后	[13]	27	[38]	49	[49'	65]	76	[97]
三趟排序之后	13	27	38	49	49'	65	76	97

(b) 各趟排序之后的状态

快速排序的最坏时间复杂度应为 $O(n^2)$ ，最好时间复杂度为 $O(n\lg n)$ 。





动态规划

- n 适用于动态规划法求解的问题，经分解得到的子问题往往不是互相独立的；
 - n 基于动态规划法的算法设计通常按以下几个步骤进行：
 - (1) 找出最优解的性质，并描述其结构特征；
 - (2) 递归定义最优值；
 - (3) 以自底向上的方式计算最优值；
 - (4) 根据计算最优值时得到的信息构造一个最优解。
- 通常在步骤(3)计算最优值时，需要记录更多的信息，以便在步骤（4）中快速构造出一个最优解。

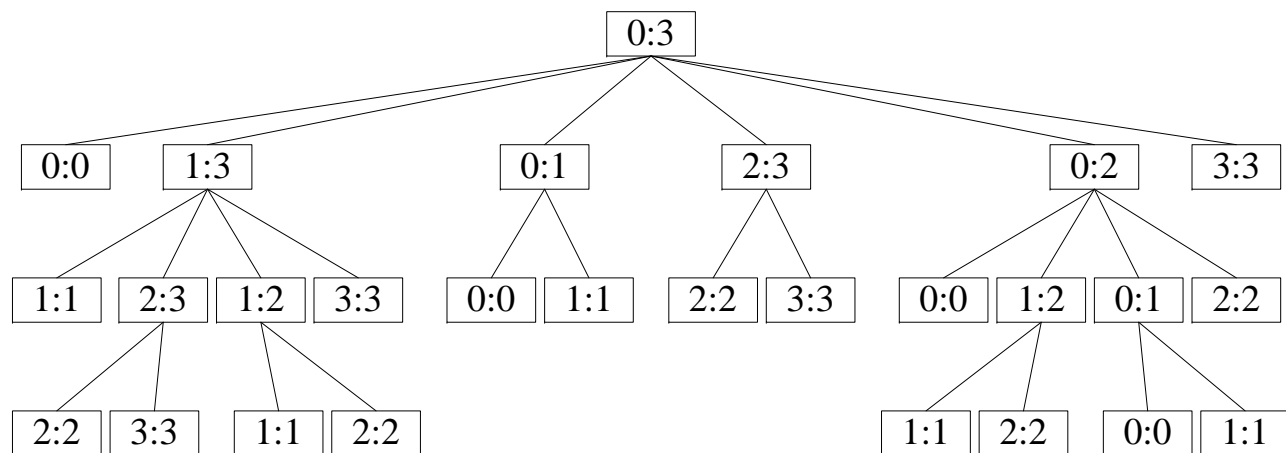




动态规划算法的基本要素

n 动态规划算法的两个基本要素——**最优子结构性质**和**子问题重叠性质**。

- 1、**最优子结构**：当问题的最优解包含了其子问题的最优解时，称该问题具有最优子结构性质。
- 2、**重叠子问题**：对每个子问题只解一次，然后将解保存在一个表格中，当再次需要解此问题时，只是简单地用常数时间查看所保存的结果即可。



计算A[0:3]的
递归树



备忘录方法

- n 是动态规划算法的一个变形。
- n 备忘录方法的递归方式是自顶向下，动态规划算法则是自底向上。具体算法思想：
 - 1) 每一个子问题建立一个记录项，初始化时将其赋值为特定值，表示该子问题尚未求解。
 - 2) 在求解过程中，对每个待求解的子问题首先查看其相应的记录项，若记录项中存储的值是初始化的值，则表示该子问题第一次遇到，就需要计算该子问题的解，并存入相应的记录项，否则表示该子问题已经求解，即不必再计算该子问题。





备忘录算法分析

- n 在调用**LookupChain**时，如果 $m[i][j] > 0$ ，那么 $m[i][j]$ 中存储的值就是待求的值，直接返回此结果即可；否则通过递归算法自顶向下计算，并将计算结果存入 $m[i][j]$ 后返回。
- n 与动态规划算法一样，备忘录算法的复杂性是 $O(n^3)$ 。
- n 当一个问题所有子问题都至少需要求解一次时，则动态规划算法好于备忘录方法；当子问题空间中的部分子问题可以不必求解时，备忘录算法优于动态规划算法。

矩阵连乘积的最优计算次序问题可用自顶向下的备忘录算法或自底向上的动态规划算法在 $O(n^3)$ 计算时间内求解。这两个算法都利用了子问题重叠性质。





贪心算法

- n 贪心算法基本思想是：总是作出在当前看来是最好的选择，即贪心算法并不从整体最优上考虑，而只考虑当前（局部）最优。





贪心算法求解问题的条件

n **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解的选择来实现（贪心选择）。证明的具体步骤是：

- 1) 考察问题的一个全局最优解，并证明可修改该最优解，使其以贪心选择开始；
- 2) 做贪心选择后，原问题简化为一个规模更小的类似子问题；
- 3) 用数学归纳法证明，通过每一步作贪心选择，最终可导致问题的一个全局最优解。

n **最优子结构**是指当一个问题最优解包含其子问题的最优解性质。





贪心算法与动态规划算法的差异

- n 在动态规划算法中，每步所作出的选择往往依赖于相关子问题的解，因此只有在解出相关子问题的解后才能做出选择；
- n 而贪心算法所作的贪心选择仅在当前状态下做出的最好选择，即局部最优选择，然后再求解作出该选择后所产生的相应子问题的解，即贪心算法所作出的贪心选择可以依赖于“过去”所作出的选择，但绝不依赖于将来所作出的选择，也不依赖于子问题的解。
- n 贪心算法和动态规划算法都具有最优子结构性质。





差异在具体算法的体现

- n **0-1背包问题：**给定n种物品和一个背包，物品i的重量为 w_i ，其价值为 v_i ，背包的容量为c。如何选择装入背包中的物品，使得装入背包中的物品的总价值最大？（在选择装入背包中的物品时，对每种物品i只有两种选择：装入或不装入，即不能将物品i多次装入背包，也不能只装物品i的一部分）。

此问题的形式化描述为：给定 $c>0$ ， $w_i>0$ ， $v_i>0$ ， $1\leq i\leq n$ ，要求找出一个n元向量 (x_1, x_2, \dots, x_n) ， $x_i\in\{0, 1\}$ ， $1\leq i\leq n$ ，使得 $\sum_{i=1}^n w_i x_i \leq c$ ，且 $\sum_{i=1}^n v_i x_i$ 达到最大。

- n **背包问题：**给定n种物品和一个背包，物品i的重量为 w_i ，其价值为 v_i ，背包的容量为c。如何选择装入背包中的物品，使得装入背包中的物品的总价值最大？（在选择装入背包中的物品时，对每种物品i只有三种选择：装入、不装入或部分装入）。

此问题的形式化描述为：给定 $c>0$ ， $w_i>0$ ， $v_i>0$ ， $1\leq i\leq n$ ，要求找出一个n元向量 (x_1, x_2, \dots, x_n) ， $0\leq x_i\leq 1$ ， $1\leq i\leq n$ ，使得 $\sum_{i=1}^n w_i x_i \leq c$ ，且 $\sum_{i=1}^n v_i x_i$ 达到最大。



背包问题分析

- n 上述两个问题都具有最优子结构性质。
- n 对于**0-1背包问题**，设**A**是能够装入容量为**c**的背包且具有最大价值的物品集合，则 **$A_j = A - \{j\}$** 是**n-1**个物品可装入容量为 **$c - w_j$** 的背包的具有最大价值的物品集合。
- n 对于**背包问题**，若它的一个最优解包含物品**j**，则从该最优解中拿出所含物品**j**的那部分重量**w**，剩余的重量将是**n-1**个原重物品和重为 **$w_j - w$** 的物品**j**中可装入容量为 **$c - w$** 的背包且具有最大价值的物品。
- n 背包问题可以用贪心算法求解，而**0-1背包问题**却不能使用贪心算法。





贪心算法求解背包问题

具体步骤:

- 1、计算每种物品的单位重量的价值 v_i/w_i ;
- 2、根据贪心选择策略, 将尽可能多的单位重量价值高的物品装入背包;
- 3、若将这种物品全部装入背包后, 被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;
- 4、以此类推, 直到背包装满为止。





求解背包问题算法

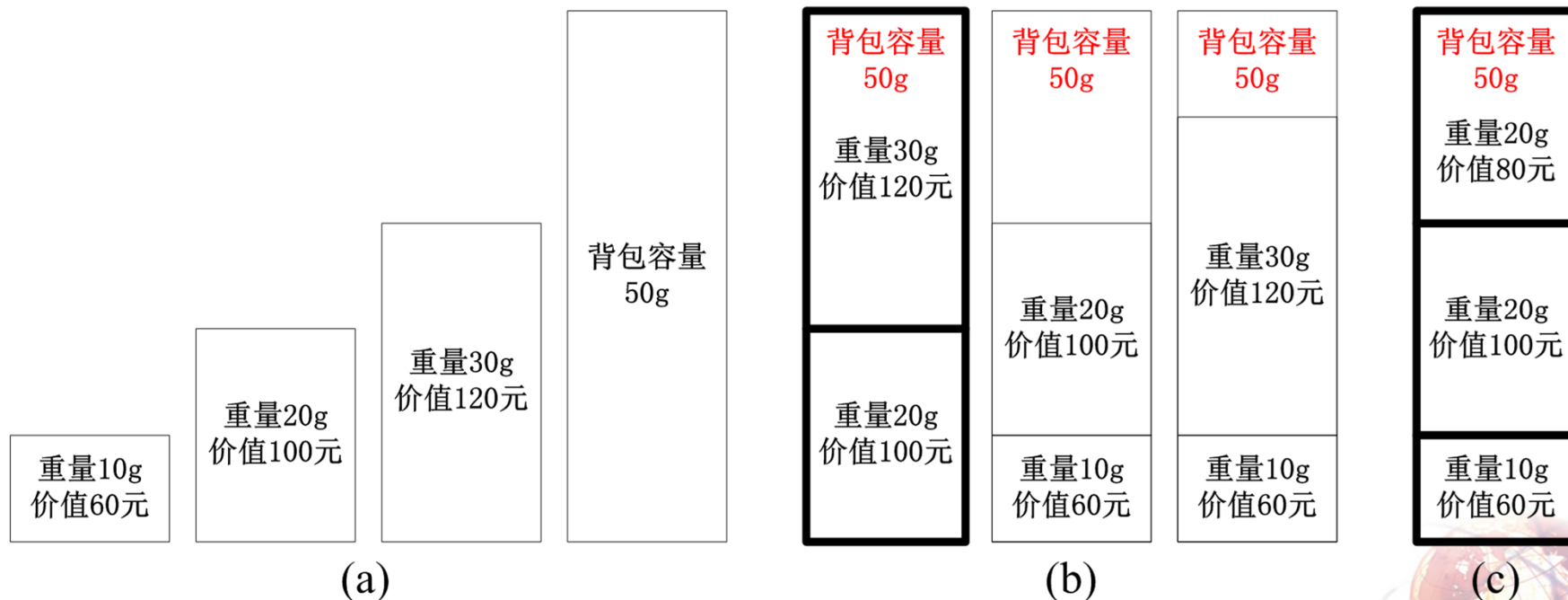
```
void KnapSack (int n, float M, float v[], float w[], float x[])
{
    Sort(n, w, v);          (各物品依其单位重量的价值大小从大到小排列)
    int i;
    for ( i = 1; i <= n; i++)    x[i] = 0;
    float c = M;
    for ( i = 1; i <= n; i++){
        if ( w[i] > c ) break;
        x[i] = 1;
        c -= w[i];
    }
    if ( i <= n )    x[i] = c / w[i];
}
```





0-1背包问题

n 贪心算法不适合**0-1**背包问题，因为它无法保证最终能将背包装满，部分背包空间的闲置使每千克背包空间的价值降低。



图(b)中的粗框为**0-1**背包问题的最优解，图(c)为背包问题的最优解。



大作业提交

n 每组**每个人**均需提交最终的大作业报告。

n 电子版

每个人的作业存在独立文件夹中，包括：**软件的源代码、可执行文件**（如果程序没有全部调试通过，要在大作业报告中写明）、**大作业的电子版**。

文件夹命名格式：学号（学号的所有数字）+姓名。**学号和姓名中间不要有任何符号**，如：**01121117何婷婷**。

n 纸质版

大作业报告按模板要求格式完成，并以纸质版形式提交。

n **6月10日20:00-21:00交大作业（电子版、纸质版）给助教，行政辅楼211。**





期末笔试注意事项

- n 时间地点：学院通知
- n 题型：单项选择题、判断题、填空题、问题求解题、算法设计题

