```yaml
title: 大学课程 | 《算法分析与设计》笔记
tags:
  - 算法
  - 大学课程
categories:
  - 学习笔记
abbrlink: 16050
reward: true
copyright: true
date: 2019-11-19 21:19:57
cover: https://npm.elemecdn.com/justlovesmile-img/H21b5f6b8496141a1979a33666e1074d9x.jpg
top_img: https://npm.elemecdn.com/justlovesmile-img/H21b5f6b8496141a1979a33666e1074d9x.jpg
```

作者博客：Justlovesmile's BLOG

大三算法设计与分析笔记总结与知识点整理

笔记总结

第一章 算法引论

1.1 算法与程序

- 算法定义：解决问题的方法或过程
- 算法的性质：
  - （1）输入：有零个或多个外部量作为算法的输入
  - （2）输出：算法产生至少一个量作为输出
  - （3）确定性：组成算法的每条指令是清晰的，无歧义的
  - （4）有限性：算法中每条指令的执行次数有限，执行每条指令的时间也有限
  - 有时还会加入通用性或可行性
- 程序的定义：是算法用某种程序设计语言的具体实现。
- 程序与算法的区别：程序可以不满足算法的第四点性质即有限性。例如操作系统，是在无限循环中执行的程序。

1.2 表达算法的抽象机制

- 为了将顶层算法与底层算法隔开，使二者在设计时不互相牵制，互相影响，必须对二者的接口进行抽象。让底层只通过接口为顶层服务，顶层也只通过接口调用底层运算。这个接口就是**抽象数据类型**(ADT)。

1.3 描述算法

- 有多种方式，如：自然语言方式，表格方式，高级程序语言方式等...

1.4 算法复杂性分析

- 算法分析的目的：分析算法占用计算机资源的情况，对算法做出比较和评价，设计出更好的算法
- 算法的复杂性是算法运行时所需的计算机资源的量，需要时间资源的量称为**时间复杂性**，需要空间资源的量称为**空间复杂性。**
- C=F(N,I,A)，用N，I，A分别表示算法要解的问题的规模，算法的输入和算法本身，F表示是上诉N，I，A的确定的三元函数，C表示复杂性
- 一般只考虑3种情况下的时间复杂性，即最坏情况，最好情况，平均情况
- 实践表明，可操作性最好且最有实际价值的是**最坏情况下的时间复杂性。**
- 复杂性渐进性态：

> 对于T(N)，如果存在~T(N)，使得当N→∞时有(T(N)-~T(N))/T(N)→0，那么就说~T(N)是T(N)当N→∞时的渐进性态。

- 如果存在正的常数C和自然数N0，使得当N≥N0时有f(N)≤Cg(N)，则称函数f(N)当N充分大时上有界，且g(N)是它的一个上界，记为f(N)=O(g(N))。这时还说f(N)的阶不高于g(N)的阶。
- 对于符号O，有如下运算规则：
  - O(f)+O(g)=O(max(f,g))
  - O(f)+O(g)=O(f+g)
  - O(f)O(g)=O(fg)
  - 如果g(N)=O(f(N)),则O(f)+O(g)=O(f)
  - O(Cf(N))=O(f(N)),其中C是一个正的常数
  - f=O(f)

第二章 递归与分治策略

2.1 递归的概念

- 直接或间接地调用自身的算法称为**递归算法。**用函数自身给出定义的函数称为**递归函数**
- 递归函数的两个要素：**边界条件**和**递归方程**
- 阶乘函数：

```
#include<iostream>
using namespace std;
```

```
int factorial(int n){
    if(n==0) return 1;
    else{
        return n*factorial(n-1);
    }
}
```

- Fibonacci数列：

```
#include<iostream>
using namespace std;

int fibonacci(int n){
    if(n<=1) return 1;
    else return fibonacci(n-1)+fibonacci(n-2);
}
```

- hanoi塔：

```
PYTHON
def hanoi(n,a,b,c):
    #将a上的n个圆盘经过c移动到b
    if(n>0):
        hanoi(n-1,a,c,b)
        move(a,b)
        hanoi(n-1,c,b,a)
```

- 递归算法的优点：结构清晰，可读性强，容易用数学归纳法来证明算法的正确性
- 递归算法的缺点：运行效率低，无论是耗费的计算时间还是占用的存储空间都比非递归算法多。
- 消除递归的方法：①采用一个用户定义的栈来模拟系统的递归调用工作栈，从而达到将递归算法改为非递归算法的目的②用递推来实现递归函数

2.2 分治法的基本思想

- 分治法的基本思想：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，然后将各子问题的解 **合并** 得到原问题的解。
- 分治法的适用条件：
  - ①该问题的规模缩小到一定程度容易解决。
  - ②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。
  - ③该问题分解出的子问题的解可以合并为该问题的解。
  - ④子问题间不包含公共的子问题（各子问题相互独立）
- 分治法的步骤：
  - 划分
  - 解决
  - 合并

2.3 二分搜索技术

```
PYTHON
def binarySearch(a,x):
    #a是数组,x是要搜索的数
    a=sorted(a)
    #a要求有序（从小到大）
    n=len(a)
    left,right=0,n-1
    while(left<=right):
        middle=(left+right)//2
        if(x==a[middle]):
            return middle
        elif(x>a[middle]):
            left=middle+1
        else:
            right=middle-1
    #未找到
    return -1
```

- 最坏情况下，时间复杂度是O(logn)

2.4 大整数乘法

- 设x和y都是n位的二进制整数，现在要计算他们的乘积xy。如果直接相乘，需要O(n^2)步，而其分治法是：将n位二进制整数X和Y都分为2段，每段的长为n/2位：

```
X=[A][B],Y=[C][D],其中X，Y有n位；A，B，C，D均有n/2位
由此可以得到：
```

```
X=A*2^(n/2)+B ， Y=C*2^(n/2)+D

XY=(A*2^(n/2)+B)(C*2^(n/2)+D)
  =A*C*2^n+(A*D+C*B)*2^(n/2)+B*D
  =A*C*2^n+((A-B)(D-C)+A*C+B*D)*2^(n/2)+B*D

最后一个式子看起来似乎复杂了，但是它仅需做3次n/2位整数的乘法，6次加减法和2次移位
```

2.5 Strassen矩阵乘法

- 对于方阵（n*n）A,B,C，有C=A*B,将它们都分块成4个大小相等的子矩阵，每个子矩阵都是`(n/2)*(n/2)`的方阵
- $$\begin{aligned} M_1 &= A_{11}(B_{12}-B_{22}) \\ M_2 &= (A_{11}+A_{12})B_{22} \\ M_3 &= (A_{21}+A_{22})B_{11} \\ M_4 &= A_{22}(B_{21}-B_{11}) \\ M_5 &= (A_{11}+A_{22})(B_{11}+B_{22}) \\ M_6 &= (A_{12}-A_{22})(B_{21}+B_{22}) \\ M_7 &= (A_{11}-A_{21})(B_{11}+B_{12}) \end{aligned} \Rightarrow \begin{aligned} C_{11} &= M_5+M_4-M_2+M_6 \\ C_{12} &= M_1+M_2 \\ C_{21} &= M_3+M_4 \\ C_{22} &= M_5+M_1-M_3-M_7 \end{aligned}$$

2.7 合并排序

PYTHON

```python
def merge(arr,left,mid,right):
    #left，right为需要合并的数组范围
    #mid为中间下标，左边比中值小，右边比中值大
    i=left
    j=mid+1
    #复制一个临时数组
    aux=arr[:]
    for k in range(left,right+1):
        #如果左指针超过mid，即右边还有剩余
        if(i>mid):
            arr[k]=aux[j]
            j=j+1
        #如果右指针超过right，即左边还有剩余
        elif(j>right):
            arr[k]=aux[i]
            i=i+1
        #如果左边小，则左边合并
        elif(aux[i]<aux[j]):
            arr[k]=aux[i]
            i=i+1
        #如果右边小
        else:
            arr[k]=aux[j]
            j=j+1


def mergeSort(arr,left,right):
    #如果已经遍历完
    if(left>=right):
        return ;
    #取中值，拆成左右两边
    mid=(left+right)//2
    #对左半边进行归并排序
    mergeSort(arr,left,mid)
    #对右半边进行归并排序
    mergeSort(arr,mid+1,right)
    #合并算法
    merge(arr,left,mid,right)
```

- 最坏情况下的时间复杂度为O(nlogn)

2.8 快速排序

- 步骤：分解，递归求解，合并

PYTHON

```python
def quicksort(arr,low,high):
    if low<high :
        index=getindex(arr,low,high)
        quicksort(arr,low,index-1)
        quicksort(arr,index+1,high)

#快速排序算法核心
#作用：将小于基准值的数放在其左边，大于在右边
def getindex(arr,low,high):
    #默认第一个数字为标准值
```

```
    temp=arr[low]
    #当未遍历完，即左右指针未相遇
    while(low<high):
        #如果右边大于标准值，右指针左移
        while((low<high)and(arr[high]>=temp)):
            high=high-1
        #此时右指针对应值小于标准值，将其复制给左指针位置
        arr[low]=arr[high]
        #当左边小于标准值，左指针右移
        while((low<high)and(arr[low]<=temp)):
            low=low+1
        #此时左指针对应值大于标准值，将其复制给右指针位置
        arr[high]=arr[low]
    #将标准值赋值给左右指针相遇的位置
    arr[low]=temp
    #此时low左边全部小于等于arr[low],low右边全部大于等于arr[low]
    return low
```

- 快排平均情况下的时间复杂度是O(nlogn)，最坏情况下的时间复杂度是O(n^2)

2.9 线性时间选择

- 找出一组数中，第X大（小）的数
- 采用了随机划分算法

2.10 最近点对问题

- 时间复杂度分析O(nlogn)

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 最近点对问题
"""

#按x坐标排序的点
class Point1:
    #x,y为坐标，id为序号
    def __init__(self,xx,yy,index):
        self.x=xx
        self.y=yy
        self.id=index


#按y坐标排序的点
class Point2(Point1):
    #x，y为坐标，id为该点按x排序时的序号
    def __init__(self,xx,yy,index):
        self.x=xx
        self.y=yy
        self.id=index

#表示输出的平面点对
class Pair:
    #a，b为点，dist为距离
    def __init__(self, aa, bb,dd):
        self.a=aa
        self.b=bb
        self.dist=dd

#求平面上任意两点u,v的距离
def dist(u,v):
    dx=u.x-v.x
    dy=u.y-v.y
    return dx*dx+dy*dy

#归并排序
def merge(S,order,left,mid,right):
    i=left
    j=mid+1
    aux=S[:]
    #按x排序
    if(order=='x'):
        for k in range(left,right+1):
            if(i>mid):
                S[k]=aux[j]
                j=j+1
```

```
            elif(j>right):
                S[k]=aux[i]
                i=i+1
            elif(S[i].x<aux[j].x):
                S[k]=aux[i]
                i=i+1
            else:
                S[k]=aux[j]
                j=j+1
    #按y排序
    elif(order=='y'):
        for k in range(left,right+1):
            if(i>mid):
                S[k]=aux[j]
                j=j+1
            elif(j>right):
                S[k]=aux[i]
                i=i+1
            elif(S[i].y<aux[j].y):
                S[k]=aux[i]
                i=i+1
            else:
                S[k]=aux[j]
                j=j+1

#归并排序
def mergeSort(S,x,left,right):
    if(left>=right):
        return ;
    mid=(left+right)//2
    mergeSort(S,x,left,mid)
    mergeSort(S,x,mid+1,right)
    merge(S,x,left,mid,right)

#计算最接近点对
def closePair(S,Y,Z,l,r):
    #两个点
    if(r-l==1):
        return Pair(S[l],S[r],dist(S[l],S[r]))
    #三个点
    if(r-l==2):
        d1=dist(S[l],S[l+1])
        d2=dist(S[l+1],S[r])
        d3=dist(S[l],S[r])
        if((d1<=d2)and(d1<=d3)):
            return Pair(S[l],S[l+1],d1)
        if(d2<=d3):
            return Pair(S[l+1],S[r],d2)
        else:
            return Pair(S[l],S[r],d3)
    #多于三个点
    m=(l+r)//2
    f=l
    g=m+1
    for i in range(l,r+1):
        if(Y[i].id>m):
            Z[g]=Y[i]
            g=g+1
        else:
            Z[f]=Y[i]
            f=f+1
    #递归求解
    best = closePair(S,Z,Y,l,m)
    right = closePair(S,Z,Y,m+1,r)
    #选最近的点对
    if(right.dist<best.dist):
        best=right
    merge(Y,"y",l,m,r)

    k=l
    #距离中线最近的
    for i in range(l,r+1):
        if(abs(S[m].x-Y[i].x)<best.dist):
            Z[k]=Y[i]
            k=k+1
    for i in range(l,k):
        for j in range(i+1,k):
```

```
            if(Z[j].y-Z[i].y<best.dist):
                dp=dist(Z[i],Z[j])
                if(dp<best.dist):
                    best=Pair(S[Z[i].id],S[Z[j].id],dp)
    #返回最近点对
    return best


#一维点集
def cpair1(S):
    #先设为正无穷
    min_d=float("inf")
    S=sorted(S)
    for i in range(1,len(S)):
        dist=abs(S[i]-S[i-1])
        if(dist<min_d):
            pair=[]
            min_d=dist
            pair.append([S[i-1],S[i]])
        elif(dist==min_d):
            pair.append([S[i-1],S[i]])
    print("Closest point:")
    for i in pair:
        print(i,end=" ")
    print("\nMin_dist:",min_d)

#二维点集
def cpair2(S):
    Y=[]
    n=len(S)
    if(n<2):
        return ;
    #按X坐标排序
    mergeSort(S,"x",0,n-1)
    #以Point2类型赋值
    for i in range(n):
        p=Point2(S[i].x,S[i].y,i)
        Y.append(p)
    #按y坐标排序
    mergeSort(Y,"y",0,n-1)
    Z=Y[:]
    return closePair(S,Y,Z,0,n-1)


def main():
    #输入一维还是二维点平面
    model=input("Please choose model of '1' or '2':").split()[0]
    S=[]
    #一维点对
    if(model == '1'):
        point=input("Please input a group of number in order:\n").split()
        #如果输入空点对
        if(len(point)==0):
            raise ValueError("您输入了空点对！")
        #转换类型
        for i in range(len(point)):
            S.append(int(point[i]))
        #输出最近点对
        cpair1(S)
    #二维点对
    elif(model == '2'):
        #输入点数
        n=int(input("Please input how many points:\n"))
        if(n==0):
            raise ValueError("您输入了0个点！")
        for i in range(n):
            words=f"please input the No.{i+1} point (like: x y) in x order:"
            point=input(words).split()
            p=Point1(int(point[0]),int(point[1]),i)
            S.append(p)
        #找到最近的一对点对
        best=cpair2(S)
        print(f"The closest points are ({best.a.x},{best.a.y}) and ({best.b.x},{best.b.y}).")
        print(f"And the distance is {best.dist**0.5}.")
    else:
        raise ValueError("没有这个选项！")
```

```
if __name__ == "__main__":
    #异常处理
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

第三章 动态规划

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解，但与分治法不同的是，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相互独立的。
- 动态规划算法的步骤：
  - ①找出最优解的性质，并刻画其结构特征
  - 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
- 动态规划算法的两个基本要素：**最优子结构**与**重叠子问题**
  - 最优子结构性质：问题的最优解包含子问题的最优解
  - 重叠子问题：在用递归算法自顶向下求解问题时，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
  - 无后效性：一个问题被划分阶段后，阶段I中的状态只能由I+1中的状态通过状态转移方程得来，与其他状态没有关系，特别是与未发生的状态没有关系
- 动态规划算法有一个变形方法——备忘录方法，这种方法不同于动态规划算法“自底向上”的填充方向，而是“自顶向下”的递归方向，为每一个解过的子问题建立一个记录项（备忘录）以备需要时查看，也可以避免相同子问题的重复求解

3.1 矩阵连乘问题

- $$m(i,j)=\begin{cases}0 & i=j \\ \min\limits_{i\le k<j}\{m(i,k)+m(k+1,j)+p_{i-1}p_kp_j\} & i<j\end{cases}$$
- m(i,j)是指从A[i]到A[j]（1≤i≤j≤n）的最少数乘次数
- 矩阵可乘条件：A的列数等于B的行数，若A是一个p×q矩阵，B是一个q×r矩阵，则AB总共需要pqr次数乘。

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 矩阵连乘问题
"""

#计算最优值
def matrixChain(p,m,s):
    #m[i][j]表示A[i]到A[j]所需的最少数乘次数
    #s[i][j]表示A[i]到A[j]所需的最少数乘法对应的分隔位置
    n=len(p)-1
    for r in range(2,n+1):
        for i in range(1,n-r+2):
            #沿斜线方向递进
            j=r+i-1
            m[i][j]=m[i+1][j]+p[i-1]*p[i]*p[j]
            s[i][j]=i
            k=i+1
            #寻找i到j间最优分隔k
            while(k<j):
                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
                k=k+1

#根据S递归输出
def traceback(s,i,j):
    if(i==j):
        print(f"A[{i}]",end="")
        return ;
    print("(",end="")
    traceback(s,i,s[i][j])
    traceback(s,s[i][j]+1,j)
    print(")",end="")



def main():
    p=[]
    y=0
    #输入矩阵个数
    n=input("Please iuput the number of matrix:").split()
```

```
    #异常处理
    if(len(n)==0):
        raise ValueError("您输入了空矩阵！")
    n=int(n[0])
    #输入每个矩阵的信息
    for i in range(n):
        s=input(f"Input No.{i+1} Matrix size,eg:5 5\n").split()
        #判断是否能与前一项相乘
        if(len(p)>=1):
            if(y!=int(s[0])):
                raise ValueError("您输入的矩阵不能相乘！")
        x,y=int(s[0]),int(s[1])
        p.append(x)
    p.append(y)
    m=[]
    s=[]
    for i in range(n+1):
        m.append([0]*(n+1))
        s.append([0]*(n+1))
    matrixChain(p,m,s)
    traceback(s,1,n)
    print("\nCount times:",m[1][n])




if __name__ =="__main__":
    #异常处理
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

3.3 最长公共子序列

- 定理：设序列X={$x_1,x_2,\ldots,x_m$}和Y={$y_1,y_2,\ldots,y_n$}的最长公共子序列为Z={$z_1,z_2,\ldots,z_k$}，则
  (1)若$x_m=y_n$，则$z_k=x_m=y_n$，且Z_{k-1}是X_{m-1}和Y_{n-1}的最长公共子序列。
  (2)若$x_m\neq y_n$且$z_k\neq x_m$，则Z是X_{m-1}和Y的最长公共子序列。
  (3)若$x_m\neq y_n$且$z_k\neq y_n$，则Z是X和Y_{n-1}的最长公共子序列。

建立递归关系：

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

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 最长公共子序列问题
"""

def IcsLength(x,y,b):
    m=len(x)
    n=len(y)
    #初始化
    c=[]
    for j in range(m+1):
        c.append([0]*(n+1))
    #逐个比较
    for i in range(1,m+1):
        for j in range(1,n+1):
            #如果相等那么此时的最长公共长度为去除该位置的最长公共长度+1
            if(x[i-1]==y[j-1]):
                c[i][j]=c[i-1][j-1]+1
                #记录c[i][j]的值是第一类子问题的解得到的
                b[i][j]=1
```

```
            #如果对应位置不相等，则比较两个序列去掉这个不等值后哪边的最长子序列会更长
            elif(c[i-1][j]>=c[i][j-1]):
                c[i][j]=c[i-1][j]
                b[i][j]=2
            else:
                c[i][j]=c[i][j-1]
                b[i][j]=3
    return c[m][n]

#根据b[i][j]输出最长子序列
def Ics(i,j,x,b):
    if(i==0 or j==0):
        return ;
    #如果是第一类子问题的解，则说明该位置是公共部分
    if(b[i][j]==1):
        Ics(i-1,j-1,x,b)
        print(x[i-1],end="")
    #如果是第二类子问题的解，则说明此时Zk≠Xm
    elif(b[i][j]==2):
        Ics(i-1,j,x,b)
    #Zk≠Yn
    else:
        Ics(i,j-1,x,b)

def main():
    #输入字符串
    A=input("Please input No.1 Ics:").split()
    B=input("Please input No.2 Ics:").split()
    b=[]
    for i in range(len(A)+1):
        b.append([0]*(len(B)+1))
    print("The longest length:",IcsLength(A,B,b))
    Ics(len(A),len(B),A,b)

if __name__=="__main__":
    #异常处理
    try:
        main()
    except Exception as e:
        print("您的输入不会合法！出错信息如下：")
        print(e)
```

3.4 凸多边形最优三角剖分

- 和矩阵连乘相似

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 凸多边形最优三角剖分问题
"""
from isConvex import isConvex

#计算最优值
def minWeightTriangulation(n,t,s,v):
    #t[i][j]是凸子多边形vi-1,vi,...,vj的最优三角剖分对应的权函数值
    for r in range(2,n+1):
        for i in range(1,n-r+2):
            j=r+i-1
            t[i][j]=t[i+1][j]+weight(i-1,i,j,v)
            s[i][j]=i
            k=i+1
            #遍历i到j的所有边
            while(k<j):
                u=t[i][k]+t[k+1][j]+weight(i-1,k,j,v)
                if(u<t[i][j]):
                    t[i][j]=u
                    s[i][j]=k
                k=k+1

#根据s输出划分结果
def traceback(s,i,j):
    if(i==j):
        print(f"B[{i}]",end="")
        return ;
    print("(",end="")
```

```
    traceback(s,i,s[i][j])
    traceback(s,s[i][j]+1,j)
    print(")",end="")

#根据距离计算权重
def weight(i,j,k,v):
    return dist(i,j,v)+dist(i,k,v)+dist(k,j,v)

#计算距离
def dist(i,j,v):
    return (v[i][0]-v[j][0])**2+(v[i][1]-v[j][1])**2

def main():
    v=[]
    #可选择手动输入和使用默认值
    ans=input("Do you want to use default v[]:(y / n )")
    if(ans=="y" or ans=="Y"):
        v=[[6,1],[13,1],[16,4],[13,7],[6,7],[3,4]]
        graph="""-----@######@-------\n----#--------#------\n---#----------#-----\n--@------------@----\n---#----------#-----\n----#---
-----#------\n-----@######@-------\n"""
        print(graph)
        for i in v:
            print(f"({i[0]},{i[1]})",end=" ")

    elif(ans=="n" or ans=="N"):
        n=int(input("Please input the number of points:\n"))
        if(n==0):
            raise ValueError("您输入了0! ")
        for i in range(n):
            a=input(f"Input X and Y of No.{i+1} point:(eg:X Y)\n").split()
            v.append([int(a[0]),int(a[1])])

    else:
        raise ValueError("对不起没有这个选项! ")
    #判断是不是图多边形
    if(not isConvex(v)):
        raise ValueError("您输入的不是凸多边形! 请确认是否按顺序输入! ")
    t=[]
    s=[]
    n=len(v)
    #初始化
    for i in range(n):
        t.append([0]*(n))
        s.append([0]*(n))
    minWeightTriangulation(n-1,t,s,v)
    traceback(s,0,n-1)

if __name__=="__main__":
    #异常处理
    try:
        main()
    except Exception as e:
        print("您的输入不合法! 出错信息如下: ")
        print(e)
```

判断是否为凸多边形

PYTHON

```
#判断是否为凸多边形
'''
计算直线表达式
param vertex1: 前一个顶点
param vertex2: 后一个顶点
return (type, param): 返回直线的类别及其描述参数
'''
def kb(vertex1, vertex2):
    x1 = vertex1[0]
    y1 = vertex1[1]
    x2 = vertex2[0]
    y2 = vertex2[1]

    if x1==x2:
        return (0, x1)      # 0-垂直直线
    if y1==y2:
        return (1, y1)      # 1-水平直线
    else:
        k = (y1-y2)/(x1-x2)
```

```
        b = y1 - k*x1
        return (2, k, b)    # 2-倾斜直线

'''
判断是否为凸多边形
param vertexes: 构成多边形的所有顶点坐标列表，如[[0, 0], [50, 0], [0, 50]]
return convex: 布尔类型，为True说明该多边形为凸多边形，否则为凹多边形
'''
def isConvex(vertexes):
    # 默认为凸多边形
    convex = True

    # 多边形至少包含三个顶点
    l = len(vertexes)
    if l<3:
        raise ValueError("多边形至少包含三个顶点！")

    # 对每两个点组成的直线做判断
    for i in range(l):
        pre = i
        nex = (i+1)%l

        # 得到直线
        line = kb(vertexes[pre], vertexes[nex])

        # 计算所有点和直线的距离（可能为正也可能为负）
        if line[0]==0:
            offset = [vertex[0]-vertexes[pre][0] for vertex in vertexes]
        elif line[0]==1:
            offset = [vertex[1]-vertexes[pre][1] for vertex in vertexes]
        else:
            k, b = line[1], line[2]
            offset = [k*vertex[0]+b-vertex[1] for vertex in vertexes]

        # 计算两两距离的乘积，如果出现负数则存在两个点位于直线两侧，因此为凹多边形
        for o in offset:
            for s in offset:
                if o*s<0:
                    convex = False
                    break
            if convex==False:
                break

        if convex==False:
            break

    # 打印判断结果
    if convex==True:
        print("该多边形为凸多边形！")
    else:
        print("该多边形为凹多边形！")

    return convex
```

3.9 0-1背包问题

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

$$m(n,j)=\begin{cases}v_n & j\geq w_n\\ 0 & 0\leq j<w_n\end{cases}$$

- 其中m(i,j)是指背包容量为j，可选择物品为i，i+1，…，n时0-1背包问题的最优值

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 0-1背包问题--动态规划
"""

#跳跃点法
def knapsack_Pro(n,v,w,C,p,x):
    #head指向每一阶段跳跃点集合的开始
    head=[0 for i in range(n+1)]
    p[0][0],p[0][1]=0,0
```

```
    left,right,pnext,head[1]=0,0,1,1
    for i in range(n):
        k=left
        for j in range(left,right+1):
            if(p[j][0]+w[i]>C):
                break
            y=p[j][0]+w[i]
            m=p[j][1]+v[i]
            #重量小于此数的跳跃点直接加进来，不会被支配
            while(k<=right and p[k][0]<y):
                p[pnext][0]=p[k][0]
                p[pnext][1]=p[k][1]
                pnext+=1
                k+=1
            #两个if判断新产生的点能否加入p
            if(k<=right and p[k][0]==y):
                if(m<p[k][1]):
                    m=p[k][1]
                k+=1
            if(m>p[pnext-1][1]):
                p[pnext][0]=y
                p[pnext][1]=m
                pnext+=1
            #取出可以支配的点
            while(k<=right and p[k][1]<=p[pnext-1][1]):
                k+=1

        #上面break后
        while(k<=right):
            p[pnext][0]=p[k][0]
            p[pnext][1]=p[k][1]
            pnext+=1
            k+=1

        left=right+1
        right=pnext-1
        head[i+1]=pnext
    traceback_Pro(n,w,v,p,head,x)

def traceback_Pro(n,w,v,p,head,x):
    j=p[head[n]-1][0]
    m=p[head[n]-1][1]
    print("max value:",m,"max weight:",j)
    for i in range(n)[::-1]:
        for k in range(head[i],head[i+1]-1):
            if(p[k][0]+w[i]==j and p[k][1]+v[i]==m):
                x[i]=1
                j=p[k][0]
                m=p[k][1]
                break


def knapsack(v,w,C,m):
    #m[i][j]指背包容量为j，可选择物品为i，i+1，...，n时的0-1背包问题的最优值
    n=len(v)-1
    #只剩一个物品的情况
    for j in range(C):
        m[n][j] = v[n] if j>=min(w[n]-1,C) else 0
    #普通情况
    for i in range(1,n)[::-1]:
        for j in range(C):
            m[i][j] = max(m[i+1][j],m[i+1][j-w[i]]+v[i]) if j>w[i]-1 else m[i+1][j]
    #第一件物品
    if(n>0):
        m[0][C-1]=m[1][C-1]
        if C-1>=w[0]:
            m[0][C-1]=max(m[0][C-1],m[1][C-1-w[0]]+v[0])

def traceback(m,w,C,x):
    c=C-1
    for i in range(len(w)-1):
        #没选物品i则x[i]=0
        if (m[i][c]==m[i+1][c]):
            x[i]=0
        else:
            x[i]=1
            c -= w[i]
```

```
    #对于最后一个物品
    x[len(w)-1]=1 if m[len(w)-1][c]>0 else 0

#输出格式
def cout(x,v,w):
    total_v=0
    total_w=0
    print("Choose:")
    for i in range(len(v)):
        if x[i]==1:
            print(f"No.{i+1} item: value is {v[i]} , weight is {w[i]}")
            total_v +=v[i]
            total_w +=w[i]
    print(f"total value: {total_v}")
    print(f"total weight: {total_w}")

def main():
    v=[]    #物品的价值列表
    w=[]    #物品的重量列表
    #输入物品数量
    n=input("Please input the number of items:\n")
    if(n=="" or n=="0"):
        raise ValueError("您输入了空值或0! ")
    else:
        n=int(n)
    x=[0 for i in range(n+1)]
    #选择两种算法（课本上的）
    ans=input("Choose Knapsack or Knapsack_Pro?(1 or 2)\n").split()[0]
    if ans=='1':
        m=[]    #m(i,j)指背包容量为j，可选择物品为i，i+1，...，n时的0-1背包问题的最优值
        for i in range(n):
            item=input(f"please input No.{i+1} item's value(v) and weight(w):(eg:v w)\n").split()
            v.append(int(item[0]))
            w.append(int(item[1]))
        C=int(input("Please input the max weight of bag:\n"))
        if(C<=0):
            raise ValueError("背包容量不能≤0")
        for i in range(n):
            m.append([0]*C)
        knapsack(v,w,C,m)
        traceback(m,w,C,x)
        cout(x,v,w)
    elif ans=='2':
        for i in range(n):
            item=input(f"please input No.{i+1} item's value(v) and weight(w):(eg:v w)\n").split()
            v.append(float(item[0]))
            w.append(float(item[1]))
        #初始化
        p=[[0 for i in range(2)]for j in range(n*n)]
        C=float(input("Please input the max weight of bag:\n"))
        if(C<=0):
            raise ValueError("背包容量不能小于等于0")
        if(n==1):
            if(w[0]<=C):
                x[0]=1
            else:
                x[0]=0
        else:
            knapsack_Pro(n,v,w,C,p,x)
        for i in range(n):
            if(x[i]==1):
                print("choose: value:",v[i],"weight:",w[i])
    else:
        raise ValueError(f"您输入了{ans}没有该选项! ")

if __name__=="__main__":
    #异常处理
    try:
        main()
    except Exception as e:
        print("您的输入不合法! 出错信息如下: ")
        print(e)
```

3.10 最优二叉搜索树

- 二叉搜索树：存储于每个结点中的元素x大于其左子树中任一结点所存储的元素，小于其右子树中任一结点所存储的元素

第四章 贪心算法

- 贪心算法：总是做出在当前看来最好的选择，也就是说贪心算法并不从整体最优考虑它所作出的选择只是在某种意义上的局部最优选择。
- 使用贪心算法需满足：
  - 贪心选择性：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
  - 最优子结构性质：当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质
- 贪心算法适合的问题：有n个输入，其解就由这n个输入满足某些事先给定的约束条件的某个子集组成，而把满足约束条件的子集称为该问题的可行解。显然可行解一般来说是不唯一的。那些使目标函数取极值的可行解，成为最优解。
- 贪心算法是一种分级处理方法，它首先根据题意，选取一种度量标准，然后按这种度量标准对这n个的输入排序，并按序依次输入，如果不满足条件，则不把此输入加到解当中。
- 贪心算法设计求解的核心问题：选择能产生问题最优解的最优量度标准。
- 贪心算法正确性的证明：
  - ①证明算法所求的问题具有优化子结构
  - ②证明算法所求解的问题具有贪心选择性
  - ③算法按照②种的贪心选择性进行局部最优选择

4.2 活动安排问题

- 为了选择最多的相容活动，每次选择fi最小的活动，使能够选择更多的活动
- 度量标准：按照结束时间的非减序排列
- 如果有序，则O(n)，如果无序，O(nlogn)

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 活动安排问题
"""

#活动类，每个活动包括开始时间和结束时间
class activity():
    def __init__(self,ss,ff):
        self.s=ss
        self.f=ff


def greedySelector(arr,a):
    n=len(arr)-1
    a[0]=True
    j=0
    count=1
    #满足开始时间大于上一个活动的结束时间的加入（设为True）
    #O(n)
    for i in range(1,n+1):
        if(arr[i].s>=arr[j].f):
            a[i]=True
            j=i
            count+=1
        else:
            a[i]=False
    return count

def main():
    activities=[]
    #输入数据
    n=int(input("please input the number of activities:\n"))
    #异常处理
    if(n==0):
        raise ValueError("您输入了0！")
    print("Use greedy selector , activities should be ordered by the end_time.")
    for i in range(n):
        item=input("please input the begin-time and end-time:(eg: 3 6)\n").split()
        if(len(item)!=2):
            raise ValueError("您输入的数据个数不合法！")
        s=activity(float(item[0]),float(item[1]))
        activities.append(s)
    #以结束时间非减序排序
    activities=sorted(activities,key=lambda x:x.f)
    #初始化选择集合a
    a=[False for i in range(n)]
    count=greedySelector(activities,a)
    print("Maximum number of activities:",count)
    print("Choose:",a)

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
```

```
        print("您的输入不合法！出错信息如下：")
        print(e)
```

4.3 最优装载问题

- O(nlogn)

4.4 哈夫曼编码

- 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

PYTHON

```python
#构建二叉树类型
class BinaryTree:
    def __init__(self,data,left,right,code):
        self.data=data
        self.left=left
        self.right=right
        self.code=code

    def getdata(self):
        return self.data

#哈夫曼树
class Huffman:
    def __init__(self,tree,ww):
        self.tree=tree
        self.w=ww

    def getweight(self):
        return self.w

def huffmanTree(f):
    #f是出现频率权值字典
    H=[]
    n=len(f)
    #根据value对键进行从大到小排序
    for i in sorted(f,key=f.__getitem__,reverse=True):
        tree = BinaryTree(i,0,0,"")
        w = Huffman(tree,f[i])
        H.append(w)

    for i in range(1,n):
        #取出最后两位
        x=H.pop()
        y=H.pop()
        #取权重小的做左孩子，大的是右孩子
        t=BinaryTree(i,x.tree if x.w<y.w else y.tree,y.tree if y.w>x.w else x.tree,"")
        h=Huffman(t,x.w+y.w)
        H.append(h)
        #根据权重从大到小排序
        H=sorted(H,key=lambda x:x.w,reverse=True)

    return H.pop()
```

PYTHON

```python
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 哈夫曼编码
"""

#构建二叉树类型
class BinaryTree:
    def __init__(self,data,left,right,code):
        self.data=data
        self.left=left
        self.right=right
        self.code=code

    def getdata(self):
        return self.data

#哈夫曼树
class Huffman:
    def __init__(self,tree,ww):
        self.tree=tree
```

```
        self.w=ww

    def getweight(self):
        return self.w

#计算权重
def makedict(s):
    dic={}
    for i in s:
        if i not in dic.keys():
            dic[i]=1
        else:
            dic[i]+=1
    return dic

def huffmanTree(f):
    #f是出现频率权值字典
    H=[]
    n=len(f)
    #根据value对键进行从大到小排序
    for i in sorted(f,key=f.__getitem__,reverse=True):
        tree = BinaryTree(i,0,0,"")
        w = Huffman(tree,f[i])
        H.append(w)

    for i in range(1,n):
        #取出最后两位
        x=H.pop()
        y=H.pop()
        #取权重小的做左孩子，大的是右孩子
        t=BinaryTree(i,x.tree if x.w<y.w else y.tree,y.tree if y.w>x.w else x.tree,"")
        h=Huffman(t,x.w+y.w)
        H.append(h)
        #根据权重从大到小排序
        H=sorted(H,key=lambda x:x.w,reverse=True)

    return H.pop()

def listall(h):
    m=[]
    k=[]
    left,right=h.tree.left,h.tree.right
    rcode="1"
    lcode="0"
    m.append(right)
    right.code+=rcode
    m.append(left)
    left.code+=lcode
    while(len(m)>0):
        #如果存在左孩子（左右必同时存在）
        if(m[-1].left):
            a=m.pop()
            c=a.code
            m.append(a.right)
            a.right.code=c+rcode
            m.append(a.left)
            a.left.code=c+lcode
        else:
            b=m.pop()
            k.append(b)
    return k

def back(hfmcode,filename):
    ans=input(f"Do you want to decode '{filename}'? (y/n) \n")
    if(ans!="y" and ans!='Y'):
        return;
    #读取要解压缩的文件
    with open(filename,'r') as f:
        s=f.read()
    st=""
    #键和值交换形成新字典
    new_dict = {v:k for k,v in hfmcode.items()}
    #写入新文件
    with open('解压缩.txt','w') as f:
        for i in s:
            st+=i
            if(st in hfmcode.values()):
```

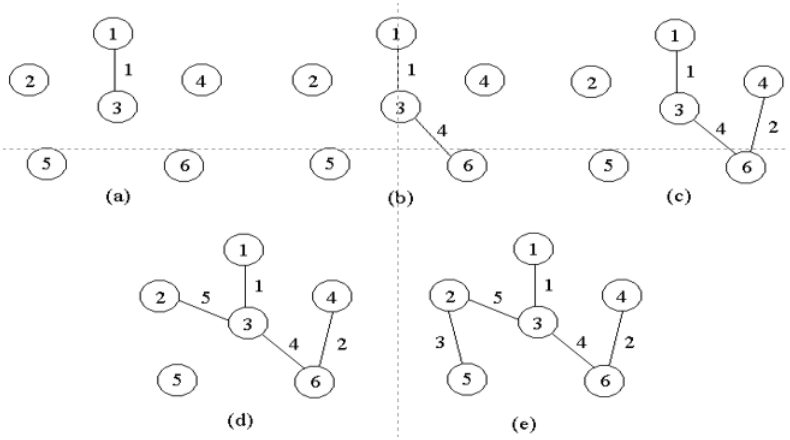
```
                f.write(new_dict[st])
                st=""
    print("=="*10)
    print("ok!Please check the file: '解压缩.txt'")
    print("=="*10)

def main():
    filename1="测试用例.txt"
    filename2='编码后.txt'
    #可以选择读文件和输入字符串
    s=input(f"Do you want to search {filename1}!  (y/n) \n")
    if(s=="y" or s=="Y"):
        #读文件
        with open(filename1,'r') as f:
            s=f.read()
        #权值字典
        dic=makedict(s)
        print("权值: ",dic)
        #构建哈夫曼树
        hTree=huffmanTree(dic)
        #编码
        k=listall(hTree)
        print("哈夫曼编码: ")
        for i in k:
            print(i.data,i.code)
        #存储值对应的编码
        hfmcode={}
        for i in k:
            hfmcode[i.data]=i.code
        #写入哈夫曼编码
        with open(filename2,'w') as f:
            for i in s:
                string=hfmcode[i]
                f.write(string)
        print("=="*10)
        print(f"ok!Please check the file: '{filename2}'")
        print("=="*10)
        back(hfmcode,filename2)

    else:
        s=input("Please input the string:")
        dic=makedict(s)
        print(dic)
        hTree=huffmanTree(dic)
        k=listall(hTree)
        for i in k:
            print(i.data,i.code)


if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print("您的输入不合法! 出错信息如下: ")
        print(e)
```

4.5 单源最短路径

- 设置顶点集合S并不断地作贪心选择来扩充这个集合，一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知，初始时，S中仅含有源。设u是G的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u的特殊路径，并用数组dist记录当前每个顶点所对应的最短特殊路径长度。Dijkstra算法每次从V-S中取出具有最短特殊路径长度的顶点u，将u添加到S中，同时对数组dist作必要的修改。一旦S包含了所有V中顶点，dist就记录了从源到所有其他顶点之间的最短路径长度

4.6 最小生成树

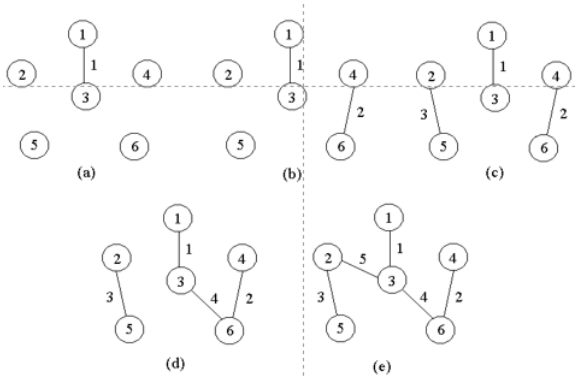
- 设G =(V,E)是无向连通带权图，即一个网络。E中每条边(v,w)的权为`c[v][w]`。如果G的子图G'是一棵包含G的所有顶点的树，则称G'为G的生成树。生成树上各边权的总和称为该生成树的耗费。在G的所有生成树中，耗费最小的生成树称为G的最小生成树
- 最小生成树性质（MST性质）：设G=(V,E)是连通带权图，U是V的真子集。如果(u,v)∈E，且u∈U，v∈V-U，且在所有这样的边中，(u,v)的权为`c[u][v]`最小，那么一定存在G的一颗最小生成树，它以(u,v)为其中一条边
- Prim算法：
  - 首先置S={1}，然后，只要S是V的真子集，就作如下的贪心选择：
  - 选取满足条件i∈S，j∈V-S，且`c[i][j]`最小的边，将顶点j添加到S中。这个过程一直进行到S=V时为止。
  - 在这个过程中选取到的所有边恰好构成G的一棵最小生成树。



PYTHON

```python
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 最小生成树-Prim算法
"""

def prim(n,c):
    #初始化Prim算法的数组
    s=[1]
    p=[1]
    lowcost=[float('inf') for i in range(n)]
    m=1
    #遍历S中的点
    for r in range(1,n):
        ns=len(s)
        for t in range(ns):
            i=s[t]
            for j in range(1,n+1):
                #如果不在S中，且最短则记录
                if(j not in s) and (c[i][j]<lowcost[m]):
                    lowcost[m]=c[i][j]
                    k=j
                    u=i
        m+=1
        s.append(k)
        p.append(u)

    for i in range(1,len(s)):
        print(s[i],p[i],c[s[i]][p[i]])

def main():
    #输入点数
    n=int(input("Please input the number of points:\n"))
    #初始化边长
    c=[[float('inf') for i in range(n+1)] for j in range(n+1)]
    for i in range(1,n+1):
        c[i][i]=0
    if(n<=1):
        raise ValueError(f"You input {n} point.")
    #输入边长
    g=input("Please input the p1,p2 and weight,like: 1 2 4\nInput end to end.\n")
    while(g!='end'):
        a=g.split()
        i=int(a[0])
        j=int(a[1])
        w=float(a[2])
        c[i][j]=w
        c[j][i]=w
        g=input("Please input the p1,p2 and weight,like: 1 2 4\nInput end to end.\n")

    prim(n,c)

if __name__=="__main__":
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

- Kruskal算法：
  - 首先，将G的n个顶点看成n个孤立的连通分支。将所有的边按权从小到大排序。
  - 然后，从第一条边开始，依边权递增的顺序查看每一条边，并按下述方法连接2个不同的连通分支：
  - 当查看到第k条边(v,w)时，如果端点v和w分别是当前2个不同的连通分支T1和T2中的顶点时，就用边(v,w)将T1和T2连接成一个连通分支，然后继续查看第k+1条边；
  - 如果端点v和w在当前的同一个连通分支中，就直接再查看第k+1条边。
  - 这个过程一直进行到只剩下一个连通分支时为止。



PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 最小生成树-Kruskal算法
"""

class Edge:
    def __init__(self,u,v,w):
        self.u=u
        self.v=v
        self.w=w

class EdgeNode:
    def __init__(self,p,g):
        self.id=p
        self.g=g


def kruskal(n,e):
    e=sorted(e,key=lambda x: x.w)
    en=len(e)
    s=[0]
    e[0].u.g,e[0].v.g=0,0
    for j in range(en):
        if(j not in s) and (e[j].u.g!=e[j].v.g):
            m=e[j].u.g if e[j].u.g<e[j].v.g else e[j].v.g
            for eachedge in e:
                if (eachedge.u==e[j].u or eachedge.v==e[j].v or eachedge.u==e[j].v or eachedge.v==e[j].u) and
(eachedge.u.g==eachedge.v.g):
                    m=min(eachedge.u.g,eachedge.v.g,m)
                    eachedge.u.g=eachedge.v.g=m
            e[j].u.g=e[j].v.g=m
            s.append(j)

    for i in range(len(s)):
        print(e[s[i]].u.id,e[s[i]].v.id,e[s[i]].w)


def main():
    #输入点数
    n=int(input("Please input the number of points:\n"))
    if(n<=1):
        raise ValueError(f"You input {n} point.")
    #输入边长
    e=[]
    p={}
    g=input("Please input the p1,p2 and weight,like: 1 2 4\nInput end to end.\n")
    aa,bb=n,n+1
    while(g!='end'):
        a=g.split()
        i,j,w=int(a[0]),int(a[1]),float(a[2])
        if(i not in p.keys()):
            p[i]=EdgeNode(i,aa)
        if(j not in p.keys()):
            p[j]=EdgeNode(j,bb)
```

```
        e.append(Edge(p[i],p[j],w))
        g=input("Please input the p1,p2 and weight,like: 1 2 4\nInput end to end.\n")
        aa+=1
        bb+=1

    kruskal(n,e)

if __name__=="__main__":
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

第五章 回溯法

- 回溯法是一个既带有系统性又带有跳跃性的搜索算法。它在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该节点为根的子树的搜索，逐层向其祖先结点回溯，否则，进入该子树，继续按深度优先策略搜索
- 具有剪枝函数的深度优先生成法
- 扩展结点：正在产生儿子的结点称为扩展结点
- 活结点：自身已生成但其儿子还没有全部生成的结点
- 回溯法的步骤：
  - (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)以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索
- 子集树：当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树称为子集树。通常有2n个叶子节点，其节点总个数为2n+1-1。如：0-1背包问题
- 排列树：当所给的问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有n！个叶子节点。如：旅行售货员问题。
- 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衷。

5.2 装载问题

- 如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。
  - (1)首先将第一艘轮船尽可能装满；
  - (2)将剩余的集装箱装上第二艘轮船。
- 解空间：子集树
- 可行性约束函数(选择当前元素)：
- 上界函数(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

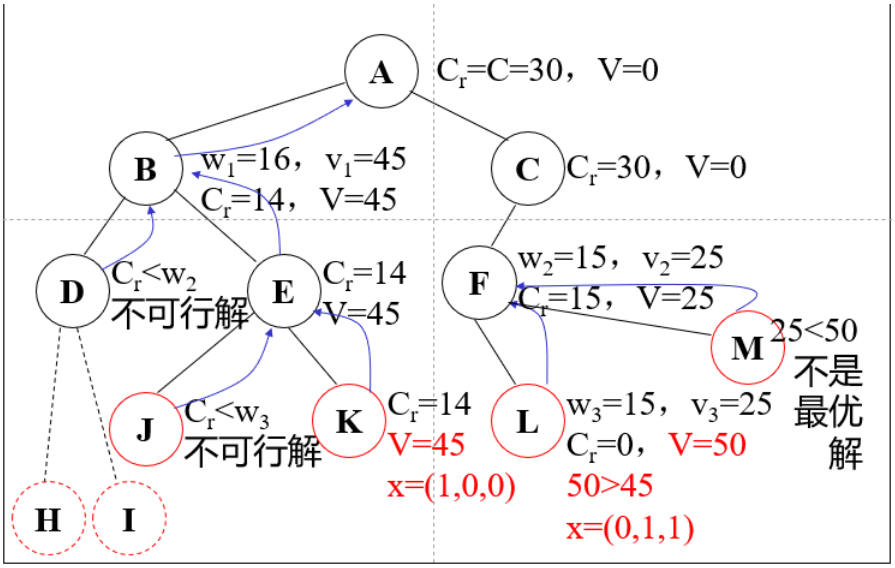
PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 装载问题（回溯法）
"""

def backtrack(i,c):
    global w,bestx,x,bestw,r,cw
    if(i>len(w)-1):
        if(cw>bestw):
            for j in range(1,len(w)):
                bestx[j]=x[j]
            bestw=cw
        return ;

    #逐层搜索子树
    r-=w[i]
    if(cw+w[i]<=c):
        x[i]=1
        cw+=w[i]
        backtrack(i+1,c)
        cw-=w[i]
    if(cw+r>bestw):
```

```
        x[i]=0
        backtrack(i+1,c)
    r+=w[i]



def main():
    global w,bestx,x,bestw,r,cw
    w=[0]
    m=input("Please input the weight of each items:(eg:1 2 3 4 5)\n").split()
    n=len(m)    #物品数量
    if(n==0):
        raise ValueError("物品数量不能为空！")
    r=0         #剩余的物品容量
    #转换w类型并初始化r
    for i in range(n):
        w.append(int(m[i]))
        r+=w[i+1]
    c1=int(input("Please input the size of No.1 ship:\n"))      #第一艘船载重量
    c2=int(input("Please input the size of No.2 ship:\n"))      #第二艘船载重量

    x=[0 for i in range(n+1)]          #记录路径
    bestx=x[:]                        #最优路径
    bestw,cw=0,0                      #最优载重量，当前载重量
    #尽可能的装满第一个
    backtrack(1,c1)
    #print(bestx)
    cw2=0
    for i in range(1,len(bestx)):
        if(bestx[i]==0):
            cw2+=w[i]
    if(cw2>c2):
        print("不能由两艘船装完！")
        return ;
    else:
        for i in range(1,len(bestx)):
            if(bestx[i]==1):
                print(f"第{i}个物品，重量{w[i]},装入第1艘船")
            else:
                print(f"第{i}个物品，重量{w[i]},装入第2艘船")

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

5.6 0-1背包问题

- n=3, C=30, w={16, 15, 15}, v={45, 25, 25}
-



PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
```

```
Title: 0-1背包问题（回溯法）
"""

class Q:
    def __init__(self,_id,qq):
        self.id=_id
        self.d=qq

def bound(i):
    global bestp,cw,cp,n,p,c,w,x,bestx
    cleft=c-cw
    bound=cp
    while(i<=n and w[i]<=cleft):
        cleft-=w[i]
        bound+=p[i]
        i+=1
    #贪心
    if(i<=n):
        bound+=p[i]*cleft/w[i]
    return bound


def backtrack(i):
    global bestp,cw,cp,n,p,c,w,x,bestx
    #到达叶结点
    if(i>n):
        if(cp>bestp):
            for j in range(1,n+1):
                bestx[j]=x[j]
        bestp=cp
        return ;
    if(cw+w[i]<c):
        x[i]=1
        cw+=w[i]
        cp+=p[i]
        backtrack(i+1)
        cp-=p[i]
        cw-=w[i]
    if(bound(i+1)>bestp):
        x[i]=0
        backtrack(i+1)


def main():
    global bestp,cw,cp,n,p,c,w,x,bestx
    pp=input("Please input the price of each items.(eg:1 2 3 4 5)\n").split()
    ww=input("Please input the weight of each items.(eg:1 2 3 4 5)\n").split()
    if(len(pp)!=len(ww)):
        raise ValueError("您的输入长度不一致！")
    n=len(pp)
    c=float(input("Please input the size of bag:\n"))
    cw=0        #当前重量
    cp=0        #当前价值
    bestp=0     #当前最优价值
    x=[0 for i in range(n+1)]       #初始化临时选择方案
    bestx=x[:]                      #初始化最优选择方案
    p=[0]       #价值列表
    w=[0]       #重量列表
    #单位重量价值
    #初始化
    q=[Q(0,0) for i in range(n)]
    for i in range(n):
        pp[i]=float(pp[i])
        ww[i]=float(ww[i])
        q[i].d=pp[i]/ww[i]
        q[i].id=i
    q=sorted(q,key=lambda x:x.d)[::-1]                #从大到小排序
    for i in range(n):
        p.append(pp[q[i].id])
        w.append(ww[q[i].id])
    #回溯
    backtrack(1)
    #打印输出
    print("Max price:",bestp,"包括：")
    for i in range(len(bestx)):
        if(bestx[i]==1):
            print(f"第{q[i-1].id+1}个,价值：{pp[q[i-1].id]},重量：{ww[q[i-1].id]}")
```

```
if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

第六章 分支限界法

- 分支限界法以**广度优先**或以**最小耗费(最大效益)优先**的方式搜索解空间树。其搜索策略是：在扩展结点处，先生成其所有儿子结点，然后再从当前的活结点表中选择下一个扩展结点。为了有效地选择下一扩展结点，加速搜索的进程，在每一格活结点处，计算一个函数值（限界），并根据函数值，从当前活结点表中，选择一个最有利的结点作为扩展结点，使搜索朝着解空间上最优解的分支推进，以便尽快找出一个最优解。
- 常见两种分支限界法：①队列式（FIFO/LIFO）分支限界法②优先队列式分支限界法
- 与回溯法对比：
  - （1）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。
  - （2）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。

不同点	回溯法	分支限界法
求解目标	找出树中满足约束条件的所有解	找出满足约束条件的一个解或找出使目标函数达到极大(小)的最优解
搜索方式	深度优先	广度优先或最小耗费优先
扩展结点	扩展结点变为活结点后又可成为扩展结点	每个活结点只有一次机会成为扩展结点
树结点的生成顺序	生成最近一个有希望结点的单个子女	选择其中最有希望的结点,并生成它的所有子女
	随机性	活结点表,搜索朝着解空间树上有最优解的分支推进

6.5 0-1背包问题

PYTHON

```
class Q:
    def __init__(self,_id,qq):
        self.id=_id
        self.d=qq

class BBnode:
    def __init__(self,par,ch):
        self.par=par
        self.ch=ch

class HeapNode:
    def __init__(self,bNode,up,pp,ww,lev):
        self.liveNode=bNode
        self.up=up
        self.p=pp
        self.w=ww
        self.lev=lev

#插入队列
def addlivenode(heap,up,pp,ww,lev,par,ch):
    b=BBnode(par,ch)
    node=HeapNode(b,up,pp,ww,lev)
    heap.append(node)

#上界函数，贪心
def bound(i):
    global cw,cp,n,p,c,w
    cleft=c-cw
    bound=cp
    while(i<=n and w[i]<=cleft):
        cleft-=w[i]
        bound+=p[i]
        i+=1

    if(i<=n):
```

```
        bound+=p[i]*cleft/w[i]
    return bound

def knapsack():
    global bestp,cw,cp,n,p,c,w,bestx
    i=1
    up=bound(i)
    heap=[]
    cnode=BBnode(None,None)
    while(i!=n+1):
        #左孩子
        cleft=cw+w[i]
        if(cleft<c):
            if(cp+p[i]>bestp):
                bestp=cp+p[i]
            addlivenode(heap,up,cp+p[i],cw+w[i],i+1,cnode,True)
        #右孩子
        up=bound(i+1)
        if(up>=bestp):
            addlivenode(heap,up,cp+p[i],cw+w[i],i+1,cnode,False)
        #取下一扩展结点
        node=heap.pop(0)
        #更新数据
        cnode=node.liveNode
        cw=node.w
        cp=node.p
        up=node.up
        i=node.lev

    #最优解
    for j in range(1,n+1)[::-1]:
        bestx[j]=1 if cnode.ch==True else 0
        cnode=cnode.par
```

实现

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 0-1背包问题（分支限界法）
"""

class Q:
    def __init__(self,_id,qq):
        self.id=_id
        self.d=qq

class BBnode:
    def __init__(self,par,ch):
        self.par=par
        self.ch=ch

class HeapNode:
    def __init__(self,bNode,up,pp,ww,lev):
        self.liveNode=bNode
        self.up=up
        self.p=pp
        self.w=ww
        self.lev=lev

#插入队列
def addlivenode(heap,up,pp,ww,lev,par,ch):
    b=BBnode(par,ch)
    node=HeapNode(b,up,pp,ww,lev)
    heap.append(node)

#上界函数，贪心
def bound(i):
    global cw,cp,n,p,c,w
    cleft=c-cw
    bound=cp
    while(i<=n and w[i]<=cleft):
        cleft-=w[i]
        bound+=p[i]
        i+=1
```

```
    if(i<=n):
        bound+=p[i]*cleft/w[i]
    return bound

def knapsack():
    global bestp,cw,cp,n,p,c,w,bestx
    i=1
    up=bound(i)
    heap=[]
    cnode=BBnode(None,None)
    while(i!=n+1):
        #左孩子
        cleft=cw+w[i]
        if(cleft<c):
            if(cp+p[i]>bestp):
                bestp=cp+p[i]
            addlivenode(heap,up,cp+p[i],cw+w[i],i+1,cnode,True)
        #右孩子
        up=bound(i+1)
        if(up>=bestp):
            addlivenode(heap,up,cp+p[i],cw+w[i],i+1,cnode,False)
        #取下一扩展结点
        node=heap.pop(0)
        #更新数据
        cnode=node.liveNode
        cw=node.w
        cp=node.p
        up=node.up
        i=node.lev

    #最优解
    for j in range(1,n+1)[::-1]:
        bestx[j]=1 if cnode.ch==True else 0
        cnode=cnode.par

def main():
    global bestp,cw,cp,n,p,c,w,bestx
    #输入
    pp=input("Please input the price of each items.(eg:1 2 3 4 5)\n").split()
    ww=input("Please input the weight of each items.(eg:1 2 3 4 5)\n").split()
    if(len(pp)!=len(ww)):
        raise ValueError("您的输入长度不一致！")
    n=len(pp)
    c=float(input("Please input the size of bag:\n"))
    cw=0        #当前重量
    cp=0        #当前价值
    bestp=0     #当前最优价值
    bestx=[0 for i in range(n+1)]   #最优解初始化
    p=[0]
    w=[0]
    q=[Q(0,0) for i in range(n)]     #单位重量价值
    allp=0    #总价值
    allw=0    #总重量
    #单位重量价值列表
    for i in range(n):
        pp[i]=float(pp[i])
        ww[i]=float(ww[i])
        allp+=pp[i]
        allw+=ww[i]
        q[i].d=pp[i]/ww[i]
        q[i].id=i
    q=sorted(q,key=lambda x:x.d)[::-1]                #从大到小排序
    for i in range(n):
        p.append(pp[q[i].id])
        w.append(ww[q[i].id])
    #如果能直接全装
    if(allw<c):
        print(f"All in! Total price is {allp}!")
        return ;

    knapsack()
    print("Max price:",bestp,"包括：")
    for i in range(len(bestx)):
        if(bestx[i]==1):
            print(f"第{q[i-1].id+1}个,价值：{pp[q[i-1].id]},重量：{ww[q[i-1].id]}")

if __name__ == "__main__":
```

```
    try:
        main()
    except Exception as e:
        print("您的输入不合法!出错信息如下: ")
        print(e)
```

6.6 装载问题

PYTHON

```
"""
Copyright: Copyright (c) 2019
Author: Justlovesmile
Title: 装载问题（分支限界法）
"""

class Node:
    def __init__(self,parent,isleftchild,weight):
        self.parent=parent
        self.islchild=isleftchild
        self.weight=weight

def maxloading(c):
    global w,bestx,bestw,r,cw,n
    i=1
    r-=w[i]
    cnode=Node(None,None,-1)    #当前结点
    q=[cnode]
    while(True):
        #左结点
        cleft=cw+w[i]
        if(cleft<=c):
            enode=Node(cnode,True,cleft)
            if(cleft>bestw):
                bestw=cleft
                bestx=enode
            if(i<n):
                q.append(enode)
            if(i==n):
                return;
        #右结点
        if(cw+r>bestw and i<n):
            enode=Node(cnode,False,cw)
            q.append(enode)
        #出队列
        cnode=q.pop(0)
        cw=cnode.weight
        if(cw==-1):
            if(len(q)==0):
                return ;
            q.append(Node(None,None,-1))
            cnode=q.pop(0)
            cw=cnode.weight
            i+=1
            r-=w[i]

def main():
    global w,bestx,bestw,r,cw,n
    w=[0]
    m=input("Please input the weight of each items:(eg:1 2 3 4 5)\n").split()
    n=len(m)    #物品数量
    if(n==0):
        raise ValueError("物品数量不能为空！")
    r=0         #剩余的物品容量
    bestw,cw=0,0
    #转换w类型并初始化r
    for i in range(n):
        w.append(int(m[i]))
        r+=w[i+1]
    allweight=r    #总重量
    x=[0 for i in range(n+1)]
    tx=[]
    c1=int(input("Please input the size of No.1 ship:\n"))      #第一艘船载重量
    c2=int(input("Please input the size of No.2 ship:\n"))      #第二艘船载重量

    maxloading(c1)
    if(bestw+c2<allweight):
        print("不能由两艘船装完！")
```

```
        return;
    for i in range(1,n+1)[::-1]:
        if(bestx.islchild==True):
            tx.append(1)
        elif(bestx.islchild==False):
            tx.append(0)
        bestx=bestx.parent
    for i in range(len(tx)):
        x[i+1]=tx[::-1][i]
    print(f"第一艘船载重量{bestw}，包括：")
    for i in range(1,n+1):
        if(x[i]==1):
            print(f"第{i}个集装箱")
    print(f"第二艘船载重量{allweight-bestw},包括：")
    for i in range(1,n+1):
        if(x[i]==0):
            print(f"第{i}个集装箱")

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print("您的输入不合法！出错信息如下：")
        print(e)
```

知识点整理

- 算法的特征：**输入，输出，确定性，有限性**
- θ记号在算法复杂性的表示法中表示**紧致界**
- 由分治法产生的子问题往往是**原问题的较小模式**，这就为使用**递归**提供了方便
- 建立计算模型的目的：**为了使问题的计算复杂性分析有一个共同的客观尺度**
- 基本计算模型：`RAM`,`RASP`,`TM`
- 拉斯维加斯算法找到的解一定是**正确**的
- 贪心算法常采用**自顶向下**的方式求解最优解
- 采用高级语言的主要好处：

  > ①更接近算法语言，易学，易掌握②提供了结构化程序设计的环境和工具，使得设计出的程序可读性高，可维护性强，可靠性高③不依赖于机器语言，因此写出的程序可移植性好，重用率高③自动化程度高，开发周期短，程序员可以集中精力从事更重要的创造性劳动，提高程序质量

- 贪心算法的特点：

  > ①不能保证最后求得的解是最优的②策略易发现，运用简单，被广泛运用③策略多样，结果也多样④常用到辅助算法：排序

- **平衡子问题**思想：通常分治法在分割原问题，形成若干子问题时，这些子问题的规模都大致相同
- Prim算法采用**贪心策略**求解**最小生成树问题**，其时间复杂度是O(n^2)。
- 动态规划算法适用于解具有**某种最优性质**的问题。
- 贪心算法做出的选择只是适用于在某种意义上的**局部最优**选择
- 在动态规划算法中，通常不同子问题的个数随问题规模呈**多项式**级增长
- 动态规划是解决**多阶段决策过程**的最优化问题
- **选择能产生最优解的贪心准则**是设计贪心算法的核心问题
- 分支限界法常以**广度优先**或以**最小耗费（最大效益）优先**的方式搜索问题的解空间树
- 为什么用分治法设计的算法一般有递归调用？因为子问题的规模还很大时，必须继续使用分治法，反复分治，必然要用到递归
- 请简述分支限界法找最优解比回溯法高的原因：在分支限界法中，每一个点只有一次机会称为扩展结点。
- 回溯法的算法框架按照问题的解空间一般分为**子集树**算法框架（如解0-1背包问题）和**排列树**算法框架（如解批处理作业调度问题）
-

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数

- 常见的多项式阶：`O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)`
- 回溯和分支限界：
  - 相同点：都是以构造一颗状态空间树为基础的，树的结点反映了对一部分解所作的特定选择
  - 不同点：①他们处理的问题类型不同，回溯法的求解目标是找出T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使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。②生成状态空间树的顺序不同③对节点存储的常用数据结构以及节点存储特性也各不相同，除由搜索方式决定的不同的存储结构外，分支限界法通常需要存储一些额外的信息以利于进一步地展开搜索
- 规则$O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$ 的证明：
  - 对于任意$f_1(n) \in O(f(n))$，存在正常数c_1和自然数n_1，使得对所有$n \geq n_1$，有$f_1(n) \leq c_1 f(n)$。
  - 类似地，对于任意$g_1(n) \in O(g(n))$，存在正常数c_2和自然数n_2，使得对所有$n \geq n_2$，有$g_1(n) \leq c_2 g(n)$。
  - 令$c_3=\max\{c_1, c_2\}$，$n_3=\max\{n_1, n_2\}$，$h(n)=\max\{f(n),g(n)\}$。
  - 则对所有的$n \geq n_3$，有
  - $f_1(n)+g_1(n) \leq c_1 f(n)+c_2 g(n)$
    $\leq c_3 f(n)+c_3 g(n)=c_3(f(n)+g(n))$
    $\leq c_3 2\max\{f(n),g(n)\}$
    $=2c_3 h(n)=O(\max\{f(n),g(n)\})$.
- **分治（Divid and Conquer）递推关系的解**

  证明：
  $$
  \begin{aligned}
  &f(n)=2f(n/2)+bn\log n\\
  &=2(2f(n/2^2)+b(n/2)\log(n/2))+bn\log n\\
  &=2^2 f(n/2^2)+bn\log(n/2)+bn\log n\\
  &=2^2(2f(n/2^3)+b(n/2^2)\log(n/2^2))+bn\log(n/2)+bn\log n\\
  &=2^3 f(n/2^3)+bn\log(n/2^2)+bn\log(n/2)+bn\log n\\
  &\cdots\\
  &=2^k f(n/2^k)+bn(\log(n/2^{k-1})+\log(n/2^{k-2})+\ldots+\log(n/2^{k-k}))\\
  &=dn+bn\sum_{j=1}^{k}\log 2^j\\
  &=dn+bn\frac{k(k+1)}{2}\\
  &=dn+\frac{bn\log^2 n}{2}+\frac{bn\log n}{2}
  \end{aligned}
  $$
- NP问题是指还未被证明是否存在多项式算法能够解决的问题，而其中NP完全问题又是最有可能不是P问题的问题类型。所有的NP问题都可以用多项式时间划归到他们中的一个。所以显然NP完全的问题具有如下性质：它可以在多项式时间内求解，当且仅当所有的其他的NP - 完全问题也可以在多项式时间内求解。