动态规划

姜汉

目　　录

- 动态规划概念
- 最长上升子序列
- 最长公共子序列
- 子段和问题
- 有关构造的问题
- 背包问题
- 树形Dp

动态规划

- 动态规划的实质就是通过保存计算过的状态，来避免递归的重叠子问题，去除冗余计算。
  - Fibonacci sequence
- 动态规划的性质
  - 决策和阶段
  - 无后效性
  - 最优化原理
- 动态规划转移方程
- 递推和记忆化搜索

动态规划的四个公式（1）

定义w(i, j) (1<=i, j<=n)，已知d[0]，

状态转移方程：

$e[j] = \min\{d[i]+w(i, j)\}$ $(0<=i<j)$ for $1<=j<=n$

d[i]可以由e[i]在常数时间内得出

least weight subsequence problem

optimum paragraph formation problem

the problem of finding a minimum height B-tree

the modified edit distance problem

(or sequence alignment with gaps)

动态规划的四个公式（2）

定义d[i][0]和d[0][j] ($0 \le i, j \le n$)

$$e[i][j] = \min\{d[i-1][j]+x_i,\ d[i][j-1]+y_i,\ d[i-1][j-1]+z_{ij}\}\quad（1 \le i, j \le n）$$

x_i, y_i, z_{ij}可以在常数时间内得出

d[i][j]可以由e[i][j]在常数时间内得出

string edit distance problem

the longest common subsequence problem

the sequence alignment problem

the sequence alignment with linear gap (variation)

动态规划的四个公式（3）

定义$w(i, j)$其中$1<=i<j<=n$和$c[i][i]=0$ $1<=i<=n$

$$c[i][j] = w(i, j)+\min\{c[i][k-1]+c[k][j]\}$$
$$i<k<=j \text{ for } 1<=i<j<=n$$

optimal binary search trees

the maximum perimeter inscribed polygon problem

the construction of optimal t-ary tree

动态规划的四个公式（4）

定义w(i, j)，其中$0<=i<j<=n$， 已知d[i][0]和 d[0][j] ($0<=i, j<=n$)

$$e[i][j] = \min_{0<=i'<i\ \ 0<=j'<j}\{d[i'][j']+w(i'+j', i+j)\} \qquad \text{for } 1<=i, j<=n$$

d[i][j]可以由e[i][j]在常数时间内得出。

最长上升子序列

- 给定序列a1, a2, …, an，找到一个最长的子序列a_{b1}, a_{b2}, …, a_{bm}，使得bi<bj，且$a_{bi}<a_{bj}$
- 序列 {1, 7, 3, 5, 9, 4, 8}
- LIS = {1, 3, 5, 8}

最长上升子序列

- dp[i]表示以i结尾时，最长上升子序列的长度
- $dp[i] = Max\{1, Max\{dp[j]+1, j<i, a[j]<a[i]\}\ \}$
- 初始条件：
- $dp[i] = 1 \qquad\qquad 1<=i<=n$
- dp[]中的最大值就是LIS的长度
- s[i]记录得到dp[i]的j值
- 用s[]构造LIS

最长上升子序列

- 在dp的状态转移过程中，去除无用的状态。
- 维持决策单调性
- dp[x]<dp[y]且a[x]<a[y] (x<y)
- 用单调队列二分查找将复杂度降为O(nlogn)

最长上升子序列

```
int find(int x) {
    int left=1, right=len, mid;
    while (left<=right) {
        mid=(left+right)/2;
        if (x>dp[mid])
left=mid+1;
        else if (x<dp[mid])
right=mid-1;
        else return mid;
    }
    return left;
}
```

```
int main() {
    …输入
    len=1; dp[1]=a[0];
    for (int i=1; i<n; i++) {
        int j=find(a[i]);
        dp[j]=a[i];
        if (j>len) len=j;
    }
    printf("%d\n", len);
}
```

最长公共子序列

- 给定两个序列{a1, a2, …, an}和{b1, b2, …, bn}，求最长公共子序列
- Str1 = “abcfbc”
- Str2 = “abfcab”
- 最长公共子序列是“abcb”和 “abfc”

最长公共子序列

- dp[i][j]表示第一个字符串前i位和第二个字符串前j位的最长公共子序列长度
- dp[i][j] = dp[i-1][j-1]+1　　　　str[i]==str[j]
- dp[i][j] = Max{dp[i-1][j], dp[i][j-1]}str[i]!=str[j]
- 初始条件：
- dp[0][i]=0　　　　0<=i<=len1
- dp[j][0]=0　　　　0<=j<=len2

最长公共子序列

```
for(i=0; i<=len1; i++) dp[i][0]=0;
for(j=0; j<=len2; j++) dp[0][j]=0;
for(i=1; i<=len1; i++)
        for(j=1; j<=len2; j++)
            if (str1[i]==str2[j]) dp[i][j] = dp[i-1][j-1]+1;
            else
                dp[i][j] = (dp[i-1][j]>dp[i][j-1]) ? (dp[i-1][j]) : (dp[i][j-1]);
```

练习题：Common Subsequence (toj1683)．

子段和问题

- 最大子段和问题
- 最大子矩阵和
- 最大m子段和

最大子段和问题

给定由N个整数(可能为负整数)组成的序列a1, a2, ..., an，求该序列形如

Sum{ak} i<=k<=j

的子段和的最大值。

最大子段和问题

dp[j]以第j个数为结尾的最大子段和

$dp[j-1]>0$时，必然有$dp[j] = dp[j-1]+a[j]$

否则$dp[j] = a[j]$

$dp[j] = Max\{dp[j-1]+a[j], a[j]\} \quad 1<=j<=n$

最大子段和问题

```
int maxSum(int b[], int n) {
    int res=-999999, i, s=0;
    for (i=1; i<=n; i++) {
        if (s>0) s+=b[i]; else s=b[i];
        if (s>res) res=s;
    }
    return res;
}
```

最大子矩阵和

拓展到二维情况
转化为一维的最大子段和问题

最大子矩阵和

```
for (int i=1; i<=n; i++) {
    memset(b, 0, sizeof(b));
    for (int j=i; j<=n; j++) {
        for (int k=1; k<=n; k++)
            b[k]+=a[j][k];
        tmp=maxSum();
        if (tmp>ans) ans=tmp;
    }
}
```

最大m子段和

给定由N个整数组成的序列以及一个整数m，要求确定序列A[]的m个不相交子段，使这m个子段的总和达到最大。

最大m子段和

- dp[i][j]表示包含第i个数的前i个数划分为j个子段和的最大值
- dp[i][j] = Max{dp[i-1][j]+a[i],
  Max{dp[k][j-1]+a[i]} (j-1<=k<i)}
  1<=j<=m   j<=i<=n

- dp[i-1][j]+a[i]表示第j个子段包含a[i]
- Max{dp[k][j-1]+a[i]}表示第j个子段只有a[i]

最大m子段和

- f[i][j]表示前i个数划分为j个子段的最大值
- f[i][j]中可以不包含a[i]!!
- f[i][j] = Max{f[i-1][j], dp[i][j]}

- dp[i][j] = Max{dp[i-1][j], f[i-1][j-1]} + a[i]

最大m子段和

```
memset(Dp, 0x88, sizeof(Dp));
memset(F, 0x88, sizeof(F));
Dp[0]=F[0]=0;
for (int i=1; i<=N; i++)
    for (int j=Min(i, M); j>=1; j--)
        F[j]=Max(F[j], Dp[j-1]), F[j]+=A[i],
        Dp[j]=Max(Dp[j], F[j]);
```

有关构造的问题

- 例1 (Ural_1081) 序列只包含0, 1元素，有一定长度N(0 < N < 44)，且没有2个1相邻（例如110这个长度为3的序列是不合法的，而0101是长度为4的合法序列）。所有合法的序列是按照字典升序编号排的，编程找出第K(0 < K < 10^9)个这样的序列。
- 输入：3 1
- 输出：000

有关构造的Dp - 例1

- dp[i][0]长度为i的以0为开头的合法序列数
- dp[i][1]长度为i的以1为开头的合法序列数
- dp[i][0] = dp[i-1][0]+dp[i-1][1]
- dp[i][1] = dp[i-1][0]

有关构造的问题 – 例1

```
memset(s, 0, sizeof(s));
s[1][1]=s[1][0]=1;
for (int i=2; i<=44; i++) {
        s[i][0]=s[i-1][1]+s[i-1][0];
        s[i][1]=s[i-1][0];
}
scanf("%d%d", &k, &n);
if (n>s[k][0]+s[k][1]) {
            printf("-1\n");
            return 0;
 }
for (int i=k; i>=1; i--) {
            if (n<=s[i][0]) {
            printf("0");
            }
            else {
                        printf("1");
            if (i!=1) printf("0");
            n=n-s[i][0];
            i--;
            }
}
```

有关构造的问题

- 例2 (Ural_1586)
- 定义这样一种数：Threeprime，指对于它的任意连续3位上的数字，都构成一个3位的质数。求对于一个n位数，存在多少个Threeprime数。
- 读入：一个整数n(3<=n<=10000)。
- 输出：即总数mod 10^9+9。

有关构造的问题 - 例2

- 求出所有3位的素数
- d[i][j]表示i位数字，以素数j为开头的Threeprime的个数
- d[i][j] = Sum{d[i-1][x]}
- x为j的后两位为开头的3位素数

背包问题

背包问题是非常常见的Dp问题，而且有很多变化，题目描述也可以变化多端。背包的主要类型：01背包、完全背包、多重背包、分组背包、混合背包、依赖背包、…

背包问题详见《背包九讲》

01背包

- 给定N件物品和一个容量为V的背包，物品i的重量为w[i]，价值为c[i]，求解如何选择装入背包的物品，使得背包中的物品总价值最大。

01背包

- dp[i][j]表示前i个物品放入容量为j的背包得到的最大价值
- dp[i][j] = Max{dp[i-1][j], dp[i-1][j-w[i]]+c[i]}   j>=w[i]
- dp[i][j] = dp[i-1][j]   j<w[i]

01背包

```
for (i=1; i<=n; i++)
        for(j=w[i]; j<=v; j++)
            dp[i][j] = Max{dp[i-1][j], dp[i-1][j-w[i]]+c[i]};
```

01背包

- dp[]初始化的时候需要注意
- 恰好装满背包：　dp[0] = 0　dp[1..v] = -INF
- 在不装入任何物品的时候只有dp[0] = 0为合法解
- 不需要装满背包：dp[0..v] = 0
- 每个容量都是合法解

01背包优化

- 效率
  - 时间复杂度 O(N*V)
  - 空间复杂度 O(N*V)
- 空间优化：一维数组
- 时间优化：跳点

01背包优化

- 空间上可以只使用一位数组
- dp[i][j] = Max{dp[i-1][j], dp[i-1][j-w[i]]+c[i]}
- dp[][]中第i行只与i-1行有关
- （滚动数组）

```
for (i=1; i<=n; i++)
    for(j=v; j>=w[i]; j--)
        dp[j] = Max{dp[j], dp[j-w[i]]+c[i]};
```

01背包优化

- 若背包容量和物品体积都很大(10^9)
- 一位数组不能承受怎么办？
- 跳点法

01背包优化

- 对于dp[i][j]中的第i个物品来说，只有在确切使用了前i-1个物品的时候，那个关键的点才是有效的。
- 即只有在发生状态转移的时候

01背包优化

- 我们可以只记录发生状态转移的关键点

- 记录(w_i, c_i)表示重量为w_i，价值为c_i
- 若$w_i<w_j$ $c_i>c_j$ (w_j, c_j)为无用点

- 这样$w_1<w_2<\ldots<w_l$ $c_1<c_2<\ldots<c_l$

多重背包

- 给定N种物品和一个容量为V的背包，物品i的重量为w[i]，价值为c[i]，每种物品有k[i]件，求解如何选择装入背包的物品，使得背包中的物品总价值最大。

- 转化为01背包

多重背包 - 例1

- 例1 (Poj_1742) 现在有n中面值的货币，每种货币的面值为v[i]，数量为c[i]。问这些货币能组成多少种面值小于m的方案。

输入：	输出：
3 10	8
1 2 4 2 1 1	4
2 5	
1 4 2 1	
0 0	

多重背包 – 例1

- dp[M][2]记录得到M面值时的两个状态
- dp[j][0]表示达到面值j时用的最后一种货币的面值。
- dp[j][1]表示达到面值j时用的最后一种货币的数量。
- f[j]表示面值j是否可以达到。
- f[0] = true；其他都是false

多重背包 - 例1

- dp[j][]是由dp[j-v[i]][]得来的
- j必须没有达到过
- j-v[i]必须达到过
- 使用i的次数不能超过c[i]次
- (!f[j])&&(f[j-v[i]])&&
- !(dp[j-v[i]][0]==i&&dp[j-v[i]][1]==c[i])

背包应用

- (Hdu_3449) 顾客有钱w，有n种盒子，每种盒子的花费是p[i]，里面有m[i]件物品，每件物品的花费是c[j]，价值是v[j]。买物品前必须先买对应的盒子。如何选取使得购买的价值最大。
  输入：　　　　　　　　　　　　输出：210
  3 800
  300 2 30 50 25 80
  600 1 50 130
  400 3 40 70 30 40 35 60

背包应用

- 枚举每种盒子购买的情况，转化为分组背包。
- 对于每种盒子而言，如果没有盒子的花费，就是01背包。
- 对背包进行变形。。

背包应用

dp[i][]表示前i个盒子总体购买的最好情况
f[i][]表示买当前盒子的最好情况

将dp[i][p[i]…w]的部分取出作为f[i][]

对于f[i][]就相当于普通的01背包

将dp[i][p[i]..w]部分与f[i][]进行比较

树形Dp

- 例1 (Poj_1463) 给你一个树状图，在每个节点上派士兵，每个士兵能够控制到相邻边。问最少派多少士兵能够控制全图。
- 输入： 输出：
- 4 1
- 0:(1) 1
- 1:(2) 2 3
- 2:(0)
- 3:(0)

树形Dp

- dp[i][0]表示该点派士兵
- dp[i][1]表示该点不派士兵
- $dp[u][0] = 1+\text{Sum}\{ \text{Min}(dp[v][0], dp[v][1]) \}$
- $dp[u][1] = \text{Sum}\{ \text{Min}(dp[v][0], dp[v][1]) \} + \text{Min}\{ |dp[v][0]-dp[v][1]| \}$
- v为u的孩子

树形Dp

- 例2 (Poj_3659) John要建电话亭，共n个点，用n-1条边相连。只能在点上建电话亭，每个电话亭可以服务该点和相邻点。问最少建立多少个电话亭？

  输入：
  5
  1 3
  5 2
  4 3
  3 5

  输出：
  2

树形Dp

- dp[i][0]表示取i点的状态
- dp[i][1]表示不取i点，该点依赖孩子
- dp[i][2]表示不取i点，该点依赖父亲