KMP 贪心算法 动态规划

吴益强

C 版

日期：2022 年 5 月 13 日

1 KMP

1.1 KMP 算法思路

KMP 算法的核心是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。具体实现就是通过一个 next() 函数实现，函数本身包含了模式串的局部匹配信息。KMP 算法的时间复杂度 O(m+n)

1.2 KMP 实现

C 版

```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        // 如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            // 如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，
               j = next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
```

```
    else
        return -1;
}
```

Python 版

```
def kmp(a,b,next):
    la,lb=len(a),len(b)
    pa=pb=0
    while pa<la and pb<la:
        if pb == -1 or a[pa]==b[pb]:
            pa,pb=pa+1,pb+1
        else:
            pb = next[pb]
    if pb == lb:
        return pa-pb
    else:
        return -1

def countnext(b):
    i,k,lb = 0,-1,len(b)
    next = [-1 for i in range(lb)]
    while i<lb-1:
        if k == -1 or b[i]==b[k]:
            next[i+1]=k+1
            i+=1
            k+=1
        else:
            k = next[k]
    return next

n = input()
b = input()
next = countnext(b)
ans = kmp(n,b,next)
print(ans)
```

2 贪心算法

2.1 贪心算法思路

在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解

2.2 雷达

C 版

```
#include<algorithm>
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
struct Radar{
    double start,end;
    bool friend operator < (Radar a,Radar b){
        return a.start < b.start;
    }
}radar[1005];
int main(){
    int n,d,x,y,m,num,flag;
    double l,r;
    m = 1;
    while(scanf("%d%d",&n,&d)){
        if(!n && !d)
            break;
        flag = true;
        for(int i = 0;i < n;i++){
            scanf("%d%d",&x,&y);
            if(y > d)
                flag = false;
            radar[i].start = x - sqrt(d * d - y * y);
            radar[i].end = x + sqrt(d * d - y * y);
        }
        if(!flag){
            printf("Case %d: -1\n",m++);
            continue;
        }
        sort(radar,radar + n);
        num = 1,l = radar[0].start,r = radar[0].end;//切记l和r是double类型!!!!!
        for(int i = 1;i < n;i++){
            if(radar[i].start >= l && radar[i].end <= r){//对应情况1
                l = radar[i].start;
                r = radar[i].end;
            }
            else if(radar[i].start <= r && radar[i].end >= r)//对应情况2
                l = radar[i].start;
            else if(radar[i].start > r){//对应情况3
                l = radar[i].start;
                r = radar[i].end;
                num++;
            }
```

```
        }
        printf("Case %d: %d\n",m++,num);
    }
    return 0;
}
```

2.3 圣诞老人的礼物

C 版

```
#include<iostream>
#include<algorithm>
#include<cstdio>
using namespace std;
struct Box
{
    int v; //价值
    int w; //重量
    double vw; //单位重量的价值
}Bs[101];
bool operator < (const Box& a,const Box& b)
{
    return a.vw<b.vw; //按单位重量价值由小到大排列
}
int main()
{
    int n,W;
    scanf("%d%d",&n,&W);
    for(int i=0;i<n;i++)
    {
        scanf("%d%d",&Bs[i].v,&Bs[i].w);
        Bs[i].vw=1.0*Bs[i].v/Bs[i].w;
    }
    sort(Bs,Bs+n);
    double sum=0,s=0;
    for(int i=n-1;i>=0;i--)
    {
        if(Bs[i].w<=W-s)
        {
            sum+=Bs[i].v;
            s+=Bs[i].w;
        }

        else
        {
            sum+=(W-s)*Bs[i].vw;
            break;
```

```
        }

    }
    printf("%.1lf\n",sum);
    return 0;

}
```

3 动态规划

3.1 动态规划

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。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。

3.2 能解决的问题

1、问题具有最优子结构性质。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质。

2、无后效性。当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系。

3.3 DP 的典型应用——DAG 最短路

给定一个城市的地图，所有的道路都是单行道，而且不会构成环。每条道路都有过路费，问您从 S 点到 T 点花费的最少费用。

解决办法：拓扑排序

3.4 DP 优势

无论是 DP 还是暴力，我们的算法都是在可能解空间内，寻找最优解。

DP 自带剪枝。DP 的核心思想：尽量缩小可能解空间。　　在暴力算法中，可能解空间往往是指数级的大小；如果我们采用 DP，那么有可能把解空间的大小降到多项式级。

一般来说，解空间越小，寻找解就越快。这样就完成了优化。

3.5 例题：最长上升子序列

```
#include <stdio.h>
#include <algorithm>
using namespace std;
```

```
int N,last[1005],n[1005];

int main()
{
    //freopen("1.txt","r",stdin);
    scanf("%d",&N);
    last[0] =1;
    for(int i=0;i<N;i++){
        scanf("%d",&(n[i]));
    }

    for(int i=1;i<N;i++){
            last[i] =1;
        for(int j=i-1;j>=0;j--){
            if(n[j]<n[i]){
                last[i]=max(last[i],last[j]+1);
            }
        }
    //printf("i=%d len=%d\n",i,last[i]);
    }
    int m = -1;
    for(int i=0;i<N;i++){
        m = max(m,last[i]);
    }
    printf("%d",m);
    return 0;
}
```

动态规划初步 · 各种子序列问题

3.6 POJ3624 背包

```
Bessie has gone to the mall's jewelry store and spies a charm bracelet. Of
    course, she'd like to fill it with the best charms possible from the N (1
     N   3,402) available charms. Each charm i in the supplied list has a weight
     Wi (1   Wi   400), a 'desirability' factor Di (1   Di   100), and can be
    used at most once. Bessie can only support a charm bracelet whose weight is
    no more than M (1   M   12,880).

Given that weight limit as a constraint and a list of the charms with their
    weights and desirability rating, deduce the maximum possible sum of ratings.
Input

* Line 1: Two space-separated integers: N and M

* Lines 2..N+1: Line i+1 describes charm i with two space-separated integers:
    Wi and Di
```

```
Output

* Line 1: A single integer that is the greatest sum of charm desirabilities
    that can be achieved given the weight constraints

Sample Input
4 6
1 4
2 6
3 12
2 7

Sample Output
23
```

```cpp
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<vector>
using namespace std;
#define mem(a,n) memset(a,n,sizeof(a))
typedef long long LL;
const int N=13000;
const int INF=0x3f3f3f3f;
int dp[N];
int main()
{
    int n,m;
    while(~scanf("%d%d",&n,&m))
    {
        mem(dp,0);
        int w,d;
        for(int i=0; i<n; i++)
        {
            scanf("%d%d",&w,&d);
            for(int j=m; j>=0; j--)
                if(j>=w)
                {
                    dp[j]=max(dp[j],dp[j-w]+d);
                    printf("dp[%d]=%d\n",j,dp[j]);
                }
        }
        printf("%d\n",dp[m]);
    }
    return 0;
}
```