《L4-CSP-J 复赛进阶算法与数据结构》

Day08 动态规划-Dynamic Programming

区间动态规划

主讲人：小武老师

可达信奥_小武老师_keda.ac

区间DP

区间DP属于线性DP的一种。一般以区间长度作为DP的阶段，以区间的左右端点作为状态的维度。一个状态通常由被它包含且比它更小的区间状态转移而来。阶段（长度）、状态（左右端点）、决策三者按照由外到内的顺序构成三层循环。

可达信奥_小武老师_keda.ac

DP的三要素

状态、阶段和决策是构成DP算法的三要素，而子问题重叠性、无后效性和最优子结构性质是问题能用DP求解的三个基本条件。动态规划是解决多阶段决策过程最优化问题的一种方法。

状态

某一阶段的出发位置称为状态。通常一个阶段包含若干状态。

阶段

把问题分成几个相互联系的有顺序的几个环节，这些环节即称为阶段。

决策/转移方程

从某阶段的一个状态演变到下一个阶段某状态的选择。也就是从哪来要到哪去。前一阶段的终点就是后一阶段的起点，前一阶段的决策选择导出了后一阶段的状态，这种关系描述了由i阶段到i+1阶段状态的演变规律，称为状态转移方程。

区间DP

顾名思义：区间dp就是在区间上进行动态规划，求解一段区间上的最优解。主要是通过合并小区间的最优解进而得出整个大区间上最优解的dp算法。板子如下：

```
for(int len = 1; len <= n; len++){ //枚举每个小区间
    for(int j = 1; j+len-1 <= n; j++){ //枚举起点，ends <= n
        int ends = j+len - 1;
        for(int i = j; i < ends; i++){  //枚举分割点，更新小区间最优解
            dp[j][ends] = min(dp[j][ends],dp[j][i]+dp[i+1][ends]+something);
        }
    }
}
```

例题选讲

P0202　石子合并

P0297. 括号配对

P0483. 石子合并之圆形操场

P0486. 回文

可达信奥_小武老师_keda.ac

P0202 石子合并

定义dp[i][j] 为将第i堆石子到第j堆石子合并成一堆石子的最小代价

1	3	5	2	4

dp[i][j]

	1	2	3	4	5
1	0	4	?1	?4	?6
2		0	8	?2	?5
3			0	7	?3
4				0	6
5					0

```
dp[1][3]= min{dp[1][1]+dp[2][3]+sum(1,3),
              dp[1][2]+dp[3][3]+sum(1,3)}
= min{17,13}
= 13
```

```
dp[2][4]= min{dp[2][2]+dp[3][4]+sum(2,4),
              dp[2][3]+dp[4][4]+sum(2,4)}
= min{17,18}
= 17
```

```
dp[3][5]= min{dp[3][3]+dp[4][5]+sum(3,5),
              dp[3][4]+dp[5][5]+sum(3,5)}
= min{17,18}
= 17
```

P0202 石子合并

定义dp[i][j] 为将第i堆石子到第j堆石子合并成一堆石子的最小代价

1	3	5	2	4

dp[i][j]

	1	2	3	4	5
1	0	4	13	?4	?6
2		0	8	17	?5
3			0	7	17
4				0	6
5					0

```
dp[1][4]= min{dp[1][1]+dp[2][4]+sum(1,4),
              dp[1][2]+dp[3][4]+sum(1,4),
              dp[1][3]+dp[4][4]+sum(1,4),
}
= min{28,22,24}

= 22
```

```
dp[2][5]= min{dp[2][2]+dp[3][5]+sum(2,5),
              dp[2][3]+dp[4][5]+sum(2,5)},
              dp[2][4]+dp[5][5]+sum(2,5)}
= min{31,28,31}
= 28
```

P0202 石子合并

定义dp[i][j] 为将第i堆石子到第j堆石子合并成一堆石子的最小代价

1	3	5	2	4

dp[i][j]

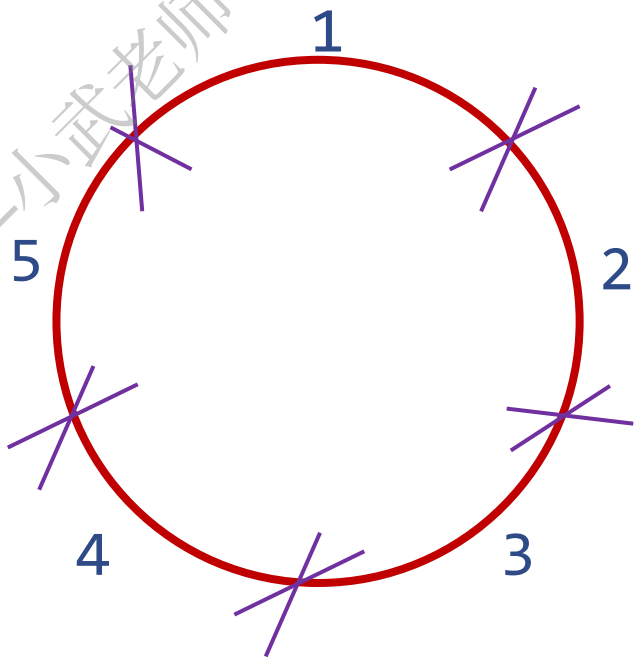
	1	2	3	4	5
1	0	4	13	22	?6
2		0	8	17	28
3			0	7	17
4				0	6
5					0

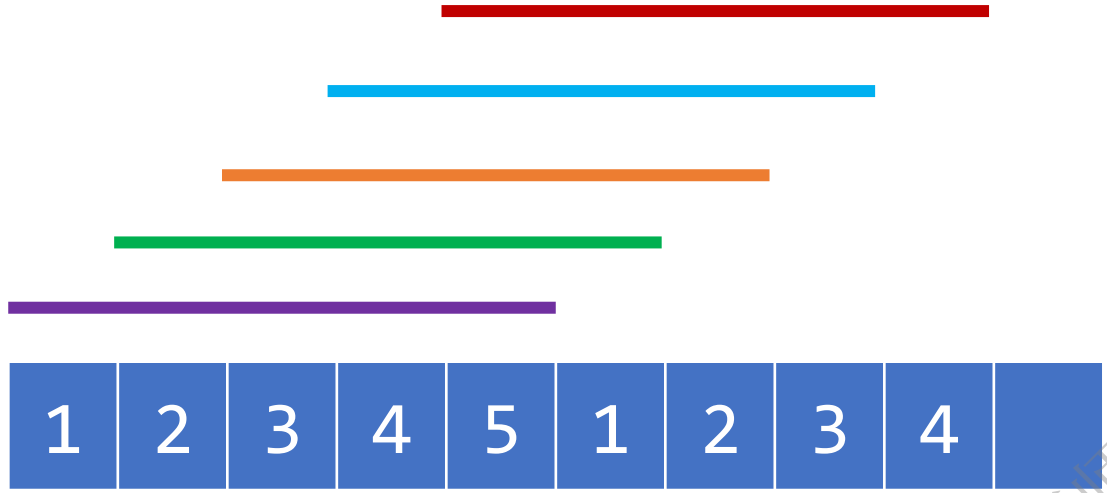
```
dp[1][5]= min{dp[1][1]+dp[2][5]+sum(1,5),
              dp[1][2]+dp[3][5]+sum(1,5),
              dp[1][3]+dp[4][5]+sum(1,5),
              dp[1][4]+dp[5][5]+sum(1,5),
}
= min{43,36,34,37}

= 34
```

P0202 石子合并

```
int main(){
    int n;
    scanf("%d",&n);
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= n; i++){
        scanf("%d",&num[i]);
        s[i]=s[i-1]+num[i];
    }
    for(int len = 2; len <= n; len++){ // 区间长度
        for(int i = 1; i+len-1 <= n; i++){ //区间起点到终点
            int l = i, r = i+len-1;
            dp[l][r] = INF;//初始化
            for(int k = l; k < r; k++){ //枚举分割点
                dp[l][r] = min(dp[l][r],dp[l][k]+dp[k+1][r]+s[r]-s[l-1]);
            }
        }
    }
    printf("%d",dp[1][n]);
    return 0;
}
```

P0483 石子合并-圆形操场






```
dp_min[l][r] = min(dp_min[l][r], dp_min[l][k]+dp_min[k+1][r]+sum[r]-sum[l-1]);
dp_max[l][r] = max(dp_max[l][r], dp_max[l][k]+dp_max[k+1][r]+sum[r]-sum[l-1]);
```

P0483 石子合并-圆形操场

```
    cin >> n;
    for(int i = 1; i <= n; i++){
        cin >> a[i]; a[i+n] = a[i]; // 2*n 数组
    }
    for(int i = 1; i <= 2*n; i++){
        sum[i] = sum[i-1] + a[i];
        dp_min[i][i] = dp_max[i][i] = 0;
    }
    for(int len = 2; len <= n; len++){ //枚举区间长度
        for(int l = 1; l+len-1 <= n*2; l++){ //枚举左端点
            int r = l + len - 1; //计算右端点
            for(int k = l; k < r; k++){ // 枚举分界点
                dp_min[l][r] = min(dp_min[l][r], dp_min[l][k]+dp_min[k+1][r]+sum[r]-sum[l-1]);
                dp_max[l][r] = max(dp_max[l][r], dp_max[l][k]+dp_max[k+1][r]+sum[r]-sum[l-1]);
            }
        }
    }
    int ans_min = INF, ans_max = -INF;
    for(int i = 1; i <= n; i++){
        ans_min = min(ans_min, dp_min[i][i+n-1]);
        ans_max = max(ans_max, dp_max[i][i+n-1]);
    }
    cout << ans_min << endl << ans_max;
    return 0;
}
```

P0297. 括号配对

dp[i][j] 表示字符串s子区间[i,j]的最长正则括号子序列的长度

S_i	S_{i+1}	S_{i+2}	···	S_{j-2}	S_{j-1}	S_j

Step 1

```
dp[i][j] = dp[i+1][j-1]+2
```

((()))

Step 2

```
dp[i][j] = max(dp[i][k]+dp[k+1][j])
```

((()))[()()()

()()()

(1) dp[0][5] = dp[1][4] + 2

(2) 枚举k 当k=1 时，dp[0][5] = dp[0][1] + dp[2][5] = 6

P0297. 括号配对

```
while(cin >> s){
    if(s == "end") break;
    memset(dp, 0, sizeof(dp));
    int n = s.length();
    for(int len = 2; len <= n; len++){
        for(int l = 0; l+len-1 < n; l++){
            int r = l+len-1;
            if((s[l] == '[' && s[r] == ']' ) || (s[l] == '(' && s[r] == ')'))
                dp[l][r] = dp[l+1][r-1] + 2;
            for(int k = l; k < r ; k++){
                dp[l][r] = max(dp[l][r], dp[l][k]+dp[k+1][r]);
            }
        }
    }
    cout << dp[0][n-1] << endl;
}
```

P0486. 回文

dp[i][j] 表示将[i,j]区间的字符串转化成回文字符串的最小成本。

s[i]=s[j]　　abcda

S_i	S_{i+1}	S_{i+2}	...	S_{j-2}	S_{j-1}	S_j

```
dp[i][j] = dp[i+1][j-1]
```

s[i]≠s[j]　　abcdc

bcdc　　abcdca

abcd　　cabcdc

dp[i+1][j]+w[i]

w[i]表示添加或删除s[i]的最小成本

dp[i][j-1]+w[j])

w[j]表示添加或删除s[j]的最小成本

```
dp[i][j] = min(dp[i+1][j]+w[i], dp[i][j-1]+w[j])
```

P0486. 回文

dp[i][j] 表示将[i,j]区间的字符串转化成回文字符串的最小成本。

```
for(int i = 1; i <= n; i++) {
        char c;
        int k1, k2;
        cin >> c >> k1 >> k2;
        w[c-'a'] = min(k1,k2);
    }
    for(int i = m - 1; i >= 0; i--){
        for(int j = i+1; j < m; j++){
            if(s[i] == s[j]){
                dp[i][j] = dp[i+1][j-1];
            }else{
                dp[i][j] = min(dp[i+1][j]+w[s[i]-'a'], dp[i][j-1]+w[s[j]-'a']);
            }
        }
    }
    cout << dp[0][m-1] << endl;
    return 0;
```

课后习题与实验

Talk is cheap, show me the code !

下节课见啦！