区间DP

```
线性DP一般是从初态开始，沿着阶段某个方向递推，到计算出目标的状态
区间DP其实也属于线性DP的一种，它是以“区间长度”作为DP的转移状态，
使用左右端点作为维度，在区间DP中，一个状态由若干个比它小的区间所
代表的状态转移，所以区间DP的转移一般是划分区间的转移
```

##例题： 石子合并 NOI1995/CH5301

```
[思路]：若最初的第l堆和第r堆石子被合并成一堆，则说明[l, r]之间的石子每堆都被合并了
这样lr才会相邻,所以在任意时刻一堆石子都可以用一个闭区间[l, r]来描述，表示这堆石子
是由最初的l~r堆石子合并而成的，质量为
```

$$\sum_{i=l}^{n} A_l$$

存在一个整数k

$$1 \leq k < r$$

在这堆石子形成前，现有第l~k堆石子被合并，第k + 1 ~ r堆石子被合并，那么才会有[l, r]被合并

```
对应到动态规划的转移：就代表着长度较小的两个区间向一个更长的区间进行转移
所以可得划分点k为转移的决策，其区间长度len作为dp的状态
所以我们可以使用dp[l][r]来代表动态规划的状态，其含义是将最初的第l堆石子和第r堆石子进行合
并，消耗的最小的体力
所以可得如下状态转移方程
```

$$F[l,r] = \min_{l \leq k < r} (F[l,k] + F[k+1,r]) + \sum_{i=l}^{r} Ai$$

附上核心代码：

```
memset(f, 0x3f, sizeof(f));
for(int i = 1; i <= n; i ++){
    f[i][i] = 0;
    sum[i] = sum[i - 1] + arr[i];///预处理前缀和
}
for(int len = 2; len <= n; len ++){
    for(int l = 1; l <= n - len + 1; l ++){
        int r = l + len - 1;
```

```
        for(int k = l; k < r; k ++){
            f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r]);
        }
        f[l][r] += sum[r] - sum[l - 1];
    }
}
```

附上AC代码：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 305;
int arr[MAXN], sum[MAXN], dp[MAXN][MAXN];
int n;
int main(){
    ios::sync_with_stdio(false);
    cin >> n;
    for(int i = 1; i <= n; i ++){
        cin >> arr[i];
    }
    memset(dp, 0x3f, sizeof(dp));
    for(int i = 1; i <= n; i ++){
        sum[i] = sum[i - 1] + arr[i];
        dp[i][i] = 0;
    }
    for(int i = 2; i <= n; i ++){
        for(int l = 1; l <= n - i + 1; l ++){
            int r = l + i - 1;
            for(int k = l; k < r; k ++){
                dp[l][r] = min(dp[l][r], dp[l][k] + dp[k + 1][r]);
            }
            dp[l][r] += (sum[r] - sum[l - 1]);
        }
    }
    cout << dp[1][n] << endl;
    return 0;
}
```

例题： FJUT2385 环形石子合并

```
[思路]：就是跟上一题一样的模板，把数组再覆盖一遍，然后跑上述算法就可以了
```

附上代码：

```
#include <bits/stdc++.h>
#define debug
using namespace std;
const int MAXN = 305;
int arr[MAXN * 2], sum[MAXN * 2], dp[MAXN * 2][MAXN * 2];
int n;
int main(){
    ios::sync_with_stdio(false);
    cin >> n;
    for(int i = 1; i <= n; i ++){
        cin >> arr[i];
    }
    for(int i = n + 1; i <= 2 * n; i ++){
        arr[i] = arr[i - n];
    }
    #ifdef debug
    for(int i = 1; i <= 2 * n; i ++){
        cout << arr[i] << " ";
    }
    cout << endl;
    #endif
    memset(dp, 0x3f, sizeof(dp));
    for(int i = 1; i <= 2 * n; i ++){
        sum[i] = sum[i - 1] + arr[i];
        dp[i][i] = 0;
    }
    int ans = 0x7fffffff;
    for(int i = 2; i <= n; i ++){
        for(int l = 1; l <= 2 * n - i + 1; l ++){
            int r = l + i - 1;
            for(int k = l; k < r; k ++){
                dp[l][r] = min(dp[l][r], dp[l][k] + dp[k + 1][r]);
            }
            dp[l][r] += (sum[r] - sum[l - 1]);
        }
    }
    for(int i = 1; i <= n; i ++){
        ans = min(ans, dp[i][i + n - 1]);
    }
    cout << ans << endl;
    return 0;
}
```

数据结构优化DP

> 有些情况下，dp的状态转移可能要从前n个中的最小值，或者前n个中的最大值来进行转移
> 朴素想法的解法就是每次从前n个扫描一遍即

$$O(n)$$

其实有些情况下可以使用数据结构来优化DP的时间复杂度 比如线段树进行维护最大值最小值，即复杂度可以变为

$$O(log_2(n))$$

例题：POJ2376/3171 FJUT1231

[思路1]： 其实这个题目就是一个DP加上线段树维护即可AC， 根据题意容易得知，需要区间全覆盖，那么我们可以考虑先按L从小到大排序维护从[S, R]所花费的最小价值，可得状态为dp[i]代表从[s, i]区间覆盖的最小价值。可得状态转移方程为 dp[i] = min(dp[i], quert_min(l - 1, i - 1)) + w) 即可AC

附上代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
typedef long long LL;
const int MAXN = 1e5 + 15;
const LL inf = __LONG_LONG_MAX__;
int n, s, e;
struct Segtree{
    LL arr[MAXN];
    struct NODE{
        int l, r;
        LL mins;
    }tree[MAXN << 2];
    void build(int root, int l, int r){
        tree[root].l = l, tree[root].r = r;
        if(l == r){
            tree[root].mins = arr[l];
            return ;
        }
        int mid = (l + r) >> 1;
        build(root << 1, l, mid);
        build(root << 1 | 1, mid + 1, r);
        tree[root].mins = min(tree[root << 1].mins, tree[root << 1 | 1].mins);
        return ;
    }
    void update(int root, int num, LL val){
        if(tree[root].l == num && tree[root].r == num){
            tree[root].mins = val;
            return ;
        }
        int mid = (tree[root].l + tree[root].r) >> 1;
        if(num <= mid){
            update(root << 1, num, val);
        }
        else{
            update(root << 1 | 1, num, val);
        }
```

```
        tree[root].mins = min(tree[root << 1].mins, tree[root << 1 | 1].mins);
        return ;
    }
    LL query(int root, int l, int r){
        if(tree[root].l >= l && tree[root].r <= r){
            return tree[root].mins;
        }
        int mid = (tree[root].l + tree[root].r) >> 1;
        if(r <= mid){
            return query(root << 1, l, r);
        }
        else if(l > mid){
            return query(root << 1 | 1, l, r);
        }
        else{
            return min(query(root << 1, l, mid), query(root << 1 | 1, mid + 1,
r));
        }
    }
}seg;
struct Person{
    int l, r;
    LL value;
    friend bool operator< (const Person &a, const Person &b){
        if(a.l == b.l){
            return a.value < b.value;
        }
        else{
            return a.l < b.l;
        }
    }
}arr[MAXN];
LL dp[MAXN];
int main(){
    ios::sync_with_stdio(false);
    cin >> n >> s >> e;
    s += 2, e += 2;
    for(int i = 0; i < n; i ++){
        cin >> arr[i].l >> arr[i].r >> arr[i].value;
        arr[i].l += 2, arr[i].r += 2;
    }
    for(int i = s; i <= e; i ++){
        seg.arr[i] = inf;
        dp[i] = inf;
    }
    for(int i = 0; i < s; i ++){
        seg.arr[i] = 0;
        dp[i] = 0;
    }
    seg.build(1, 1, e);
    sort(arr, arr + n);
    for(int i = 0; i < n; i ++){
        if(seg.query(1, arr[i].l - 1, arr[i].r - 1) != inf){
            dp[arr[i].r] = min(dp[arr[i].r], seg.query(1, arr[i].l - 1, arr[i].r -
```

```
1) + arr[i].value);
            seg.update(1, arr[i].r, dp[arr[i].r]);
        }
    }
    if(dp[e] == inf){
        cout << "-1\n";
    }
    else{
        cout << dp[e] << endl;
    }
    return 0;
}
```

##例题2 FJUT3714

题目地址：http://www.fjutacm.com/Contest.jsp?cid=659#P3

```
题意：该题的意思是给出一个t，代表t组，然后有n个数组，你可以选m个，每个选择的相邻的数字之间的下标差不能超过k
即i = 1， k = 3
那么[2, 4]的数都能选择1.
那么我们可以得到一个状态转移方程、
设状态为dp[i][j] 第一维i为当前选择了几个数字，第二维j为当前选择的是哪一个数字的下标
故可得状态转移方程为
```

$$dp[i][j] = \max(dp[i-1][j](j \in [j-k, j-1])) + dp[1][j]$$

那么我们可以得到一个O(n ^ 3)的dp 但是参考题目数据是无法通过该题的，可以采用ST表优化，优化查询

$$\max(dp[i-1][j-k], dp[i-1][j-1])$$

复杂度可以降为

$$O(n^2 log_2(n))$$

那么在比赛的时候可以通过该题 其实正解是可以优化到O(n ^ 2)的，使用单调队列优化该DP

先附上ST表优化的代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
using namespace std;
typedef long long ll;
const int MAXN = 3005;
int t;
ll dp[MAXN][MAXN];
ll f[MAXN][20];
```

```
void ST_prework(int n, int step){
    for(int i = 1; i <= n; i ++) f[i][0] = dp[step - 1][i];
    int t = log(n) / log(2) + 1;
    for(int j = 1; j < t; j ++){
        for(int i = 1; i <= n - (1 << j) + 1; i ++){
            f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
        }
    }
}
ll ST_query(int l, int r){
    int k = log(r - l + 1) / log(2);
    return max(f[l][k], f[r - (1 << k) + 1][k]);
}
int main(){
    scanf("%d", &t);
    while(t --){
        int n, m, k;
        scanf("%d%d%d", &n, &m, &k);
        for(int i = 1; i <= n; i ++){
            scanf("%lld", &dp[1][i]);
        }
        for(int i = 2; i <= m; i ++){
            ST_prework(n, i);
            for(int j = i; j <= n; j ++){
                int l = j - k;
                if(l <= 0){
                    l = 1;
                }
                dp[i][j] = ST_query(l, j - 1) + dp[1][j];
            }
        }
        ll re = 0;
        for(int i = 1; i <= n; i ++){
            re = max(re, dp[m][i]);
        }
        printf("%lld\n", re);
    }
    return 0;
}
```

以下为单调队列优化：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <queue>
using namespace std;
```

```
typedef long long ll;
const int MAXN = 3005;
int t;
ll dp[MAXN][MAXN];
deque<ll>que;
int main(){
    scanf("%d", &t);
    while(t --){
        int n, m, k;
        scanf("%d%d%d", &n, &m, &k);
        for(int i = 1; i <= n; i ++){
            scanf("%lld", &dp[1][i]);
        }
        for(int i = 2; i <= m; i ++){
            while(!que.empty()){
                que.pop_back();
            }
            int l = i - k;
            if(l <= 0){
                l = 1;
            }
            for(int j = l; j < i; j ++){
                if(que.empty()){
                    que.push_back(j);
                }
                else{
                    if(dp[i - 1][j] >= dp[i - 1][que.front()]){
                        while(!que.empty()){
                            que.pop_back();
                        }
                        que.push_back(j);
                    }
                    else{
                        while(dp[i - 1][j] >= dp[i - 1][que.back()]){
                            que.pop_back();
                        }
                        que.push_back(j);
                    }
                }
            }
            for(int j = i; j <= n; j ++){
                if(que.front() < j - k){
                    que.pop_front();
                }
                dp[i][j] = dp[i - 1][que.front()] + dp[1][j];
                if(dp[i - 1][j] >= dp[i - 1][que.front()]){
                    while(!que.empty()){
                        que.pop_back();
                    }
                    que.push_back(j);
                }
                else{
                    while(dp[i - 1][j] >= dp[i - 1][que.back()]){
                        que.pop_back();
```

```
                    }
                    que.push_back(j);
                }
            }
        }
        ll re = 0;
        for(int i = 1; i <= n; i ++){
            re = max(re, dp[m][i]);
        }
        printf("%lld\n", re);
    }
    return 0;
}
```