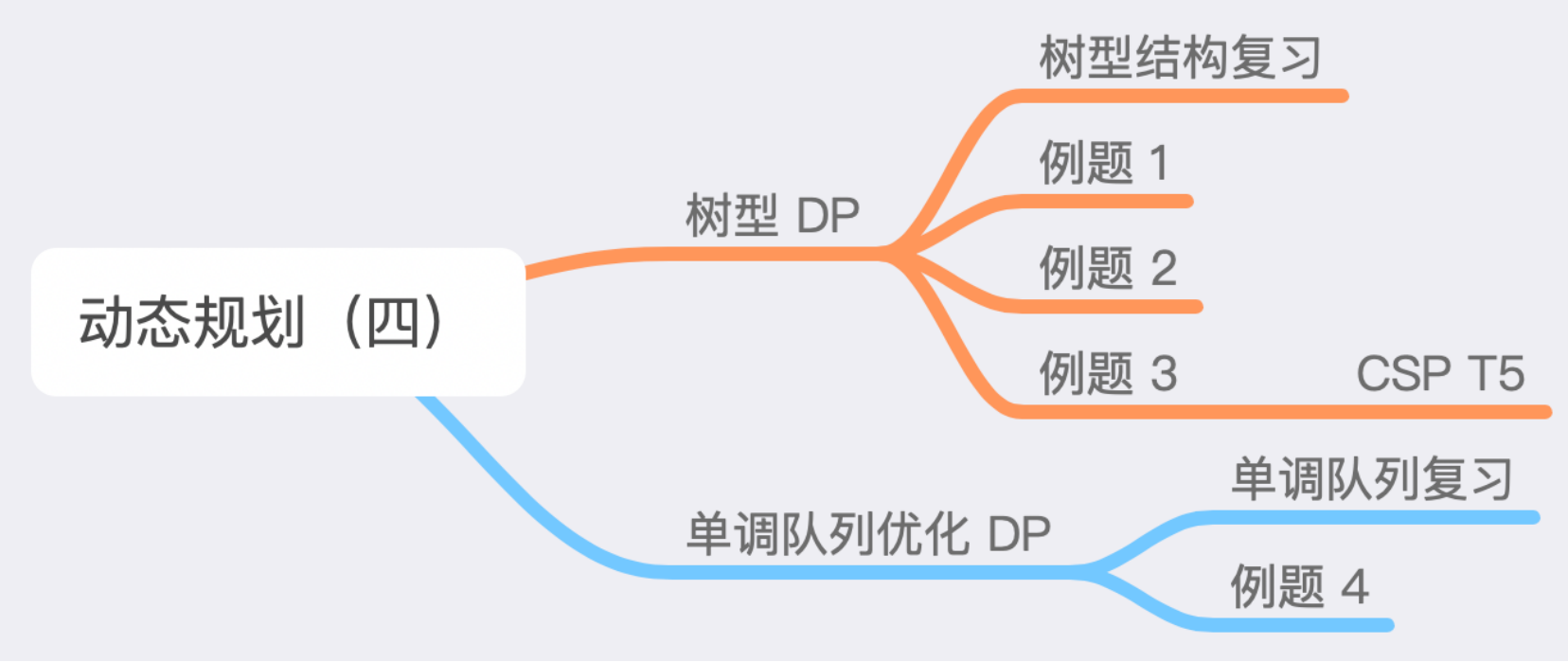
程序设计思维与实践

Thinking and Practice in Programming

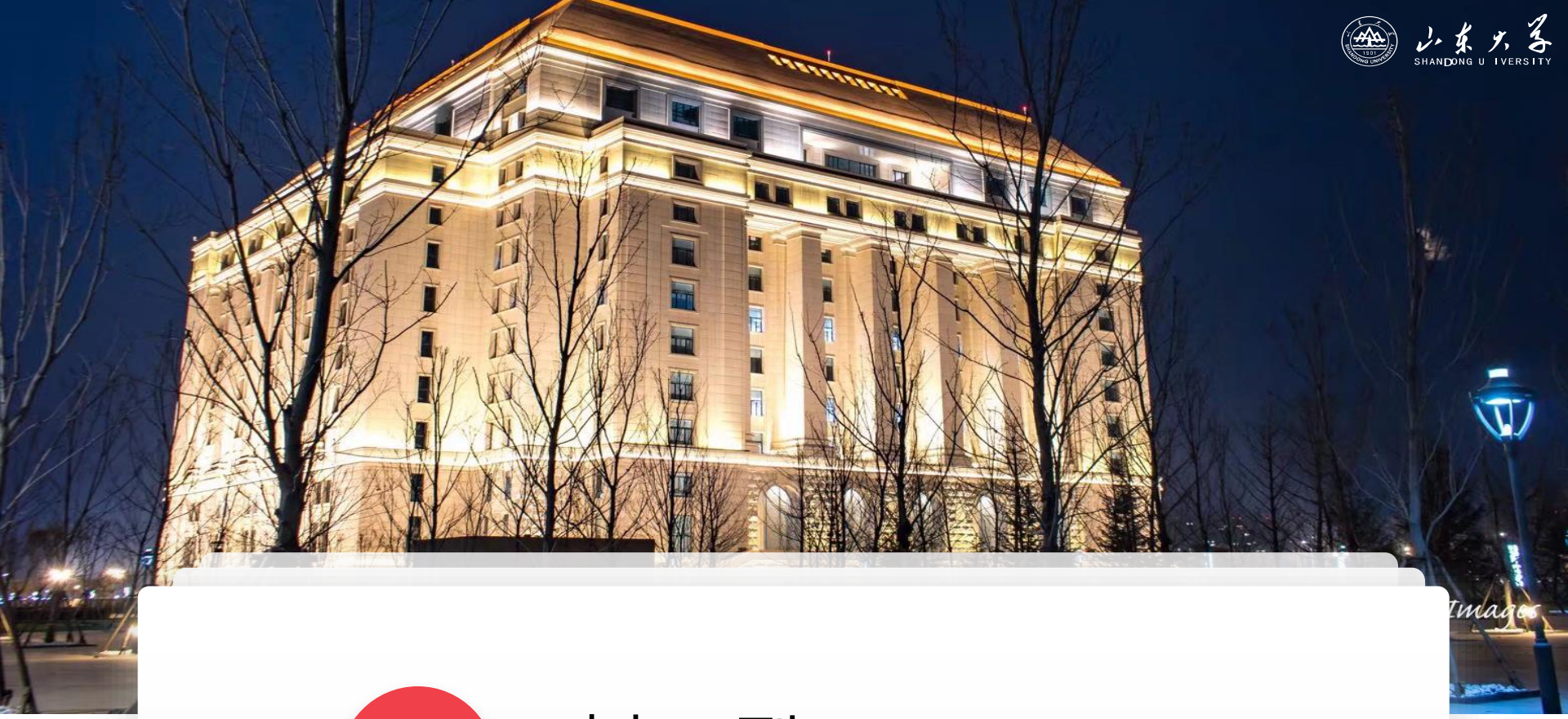
动态规划（四） | 内容负责 师浩晏

课程脉络



动态规划回顾

- 动态规划常见模型
  - 线性型
  - 坐标型
  - 背包型
  - 区间型
  - 状态压缩型
  - **树型**
  - **DP 的单调队列优化**
  - DP 的矩阵快速幂优化



1

树　型　D　P

Tree Dynamic Programming

树型DP

- 什么是树型DP?
  - 就是在树上进行的 DP。

  - 有根树是天然的DAG，因此树型DP一般是在**有根树**上进行的。
  - 由于树固有的递归性质，树型 DP 一般都是**递归**进行的。

树型DP

- **回忆树形结构**
  - 如何遍历一棵有根树？
  - 如何存储树的结构？
  - 如何求树的直径？
  - （Week6内容）

遍历方式

- 图的遍历
  - 深度优先遍历（DFS）
  - 广度优先遍历（BFS）
  - 如果使用邻接矩阵，时间复杂度为$O(n^2)$
  - 如果使用邻接表/链式前向星，时间复杂度为$O(m)$

```cpp
int G[maxn][maxn];
bool vis[maxn];
void dfs(int u) {
    // ...
    for(int i=0;i<n;i++) { // 检查了可能不存在的边
        if(G[u][i] != 0 && !vis[G[u][i]]) {
            vis[G[u][i]] = true;
            dfs(G[u][i]);
        }
    }
    // ...
}
```

```cpp
vector<int> G[maxn];
void dfs(int u) {
    // ...
    for(int i=0, n=G[u].size(); i<n; i++) {
        if(!vis[G[u][i]]) { // 直接检查邻接的点
            vis[G[u][i]] = true;
            dfs(G[u][i]);
        }
    }
    // ...
}
```

- 二叉树的遍历
  - 先序遍历
  - 中序遍历
  - 后序遍历
  - 层次遍历

```cpp
// 前向星
void dfs(int u) {
    // ...
    for(int i=head[u]; i!=-1; i=Edges[i].nxt) {
        if(!vis[Edges[i].v]) { // 直接检查邻接的点
            vis[Edges[i].v] = true;
            dfs(Edges[i].v);
        }
    }
    // ...
}
```

16

- 存图的目的：
  - 将图的结构完整地保留下来
  - 能通过保存在内存中的信息完整地还原它
- 图与树的结构
  - 图使用点与边来表示实体之间的邻接/关联/联通等关系
    每条边可能会用权值来量化这些关系
  - 树是一种特殊的图，边数等于点数减一

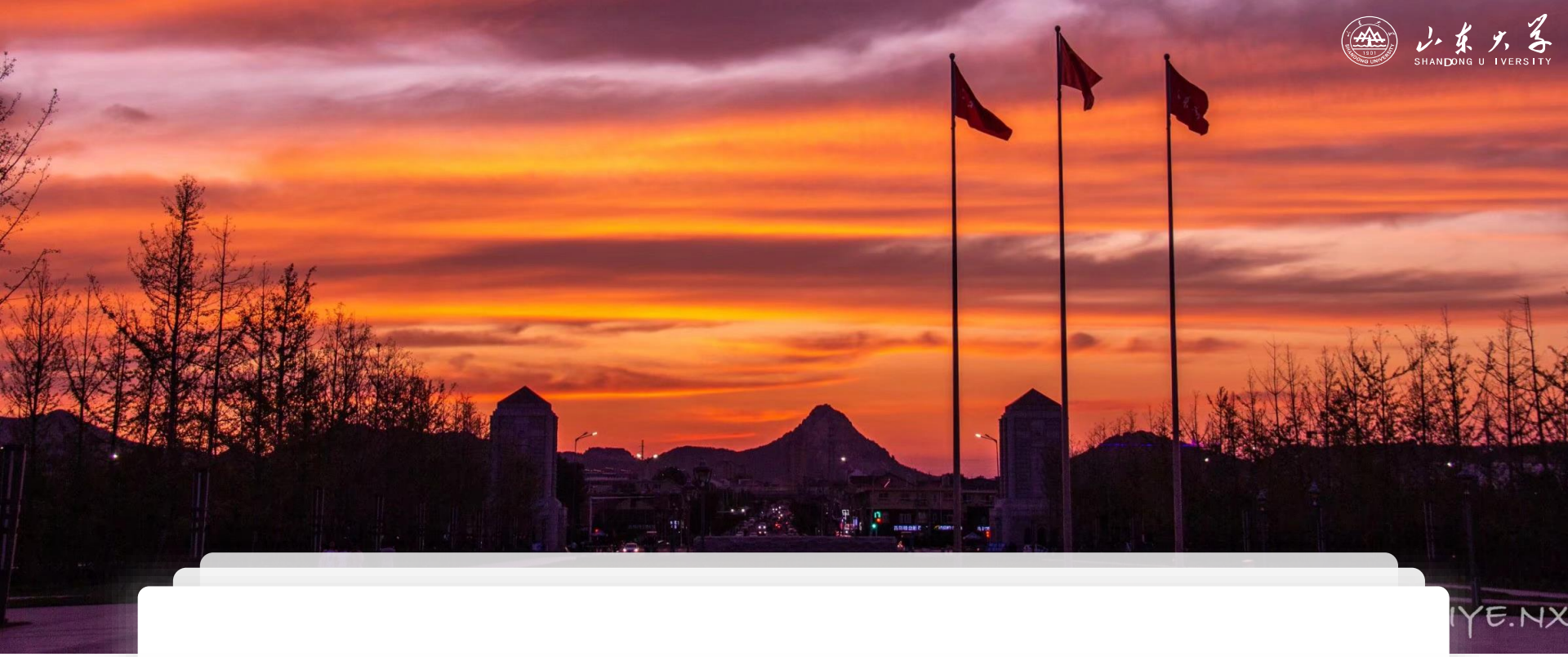
树型DP

- **回忆树型结构**
  - 无根树转有根树（假定输入仅给出树的各条边）
  - fa数组存储节点的父节点

```
const int maxn = 1e5 + 10;
int fa[maxn], n;
vector<int> v[maxn];
```

```
cin >> n;
for (int i = 1; i < n; ++i) {
    int x, y;
    cin >> x >> y;
    v[x].push_back(y);
    v[y].push_back(x);
}
dfs( u: 1,  p: -1);//假定以1号点为根
```

```
void dfs(int u, int p) {
    int d = v[u].size();
    for (int i = 0; i < d; ++i) {
        if (v[u][i] != p) {
            fa[v[u][i]] = u;
            dfs(v[u][i], u);
        }
    }
}
```



2 例 题 讲 解

Examples

例1 树的直径

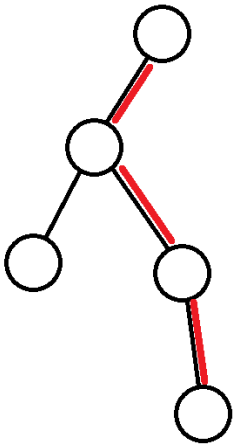
- 给出一棵树，求树的直径长度（最长简单路径），$n \leq 10^5$
- 回忆Week6讲过的BFS/DFS做法

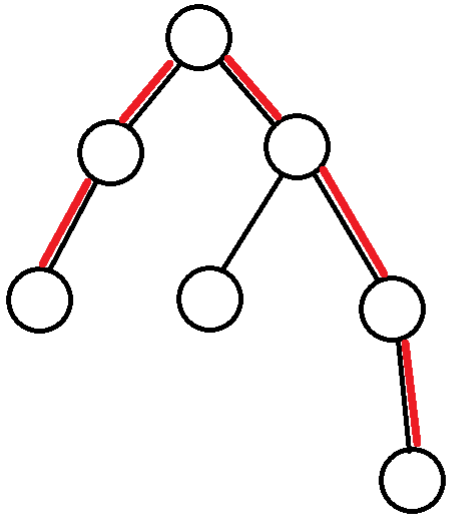
做法：
- 首先明确一点是树的直径一定是某两个叶子之间的距离
- 从树中任选一个点开始遍历这棵树，找到一个距离这个点最远的叶子，然后再从这个叶子开始遍历，找到离这个叶子最远的另一个叶子，他俩之间的距离就是树的直径。
- 两次遍历即可求的树的直径。
- 遍历可以用DFS也可以用BFS，找到距离起点最远的叶子结点就好

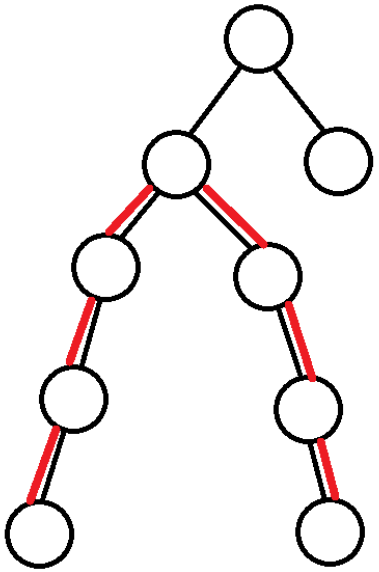
- 这里我们关心树型DP的做法。

例1 树的直径

- 确定状态：diam[i] 代表编号为 i 的 节点为根的子树中，树的直径
- 如何转移？
- 有根树上的直径有几种情况？
  - 由根节点到叶子的一条路径
  - 经过根节点的一条路径
  - 不经过根节点的一条路径







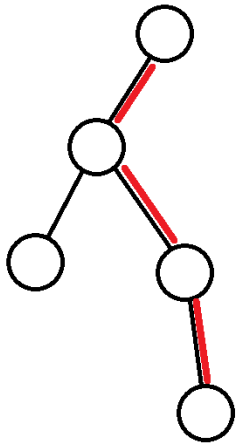
例1 树的直径

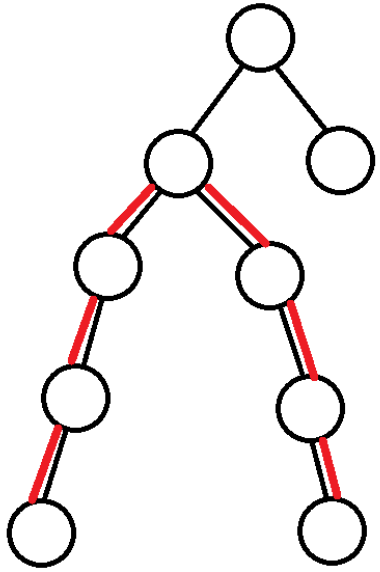
- 有根树上的直径有几种情况？
  - 由根节点到叶子的一条路径
  - 经过根节点的一条路径
  - 不经过根节点的一条路径
- 如果一个节点的所有孩子的子树直径都有了，是否可以直接求出这个节点下面子树直径？
- 情况1：求不出来
- 情况2：求不出来
- 情况3：枚举孩子的子树直径，取最大值即可

- 再添加一个状态：far[i]，代表从 i 号节点向下最深能走多远

例1 树的直径

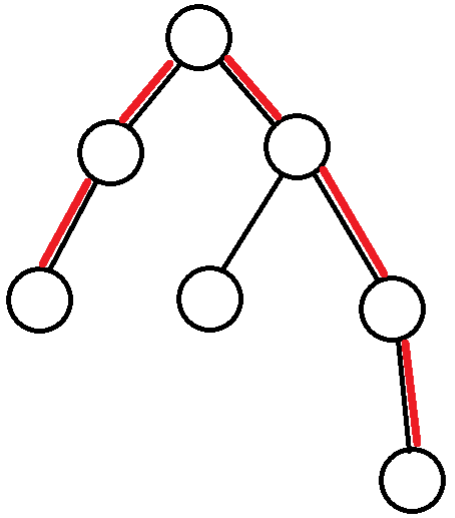
- 如何求 far[i]
  - 对于 i 的子节点 j，从 i 向下经过 j 的最远距离dis(i, j)+far[j]
  - 枚举 j 取最大值
- 求出diam[i] 转移方案：
  - 由根节点到叶子的一条路径
  - 即far[i]
  - 不经过根节点的一条路径
  - 枚举子节 j 点的diam[j]，取最大值





例1 树的直径

- 求出diam[i] 转移方案：
  - 经过根节点的一条路径
  - 枚举两个子节点
  - 时间复杂度过高
  - 寻找最远的子树和第二远的子树
  - 求和



```
// m1 m2 分别代表最大值和次大值
int m1 = 0, m2 = 0;
for (auto &i : edge[x]) {
    if (i == f) {
        continue;
    }
    if (far[i] + 1 > m2) {
        m2 = far[i] + 1;
    }
    if (m2 > m1) {
        swap(m1, m2);
    }
}
```

例1 扩展：直径的数量

- 如果要求出直径的数量呢？
- 使用cntd[i]记录节点 i 下面子树直径数量
- cntf[i] 记录节点 i 向下最远距离数量
- 需要考虑最大值只有一个和不止一个的情况

```
// 直径经过根节点的方案个数
long long cntn = 0;

if (m1 == m2) {
    long long cnt = 0;
    // cnt记录最大值的数量
    for (auto &i : edge[x]) {
        if (i == f) {
            continue;
        }
        if (far[i] + 1 == m1) {
            cntn += cnt * cntf[i];
            cnt += cntf[i];
        }
    }
```

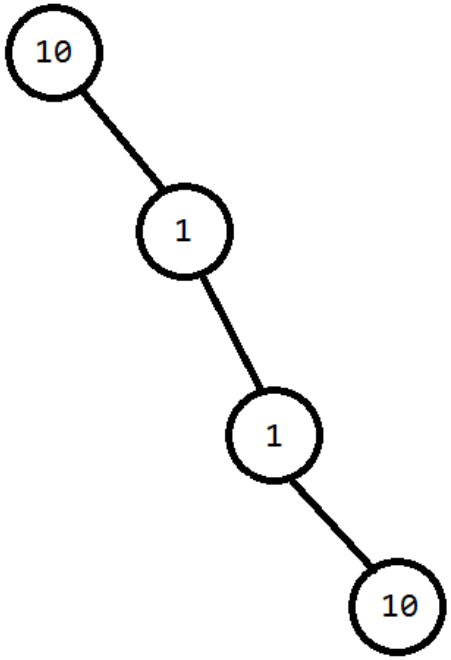
```
} else {
    // cnt1 记录最大值的数量，cnt2记录次大值数量
    long long cnt1 = 0, cnt2 = 0;
    for (auto &i : edge[x]) {
        if (i == f) {
            continue;
        }
        if (far[i] + 1 == m1) {
            cnt1 += cntf[i];
        }
        if (far[i] + 1 == m2) {
            cnt2 += cntf[i];
        }
    }
    cntn = cnt1 * cnt2;
}
```

例2 没有上司的舞会

- 【经典问题】“没有上司的舞会”问题
- 题意
  - 公司共有n（n≤6000）位员工。公司要举行一个舞会。
  - 为了让到会的每个人不受他的**直接**上司约束而能玩得开心，公司领导决定：
    - 如果邀请了某个人，那么一定不会再邀请他的直接的上司，但该人的上司的上司，上司的上司的上司……都可以邀请。
  - 已知每个人最多有**唯一**的一个上司。
  - 公司的每个人参加晚会都能为晚会增添一些气氛，求一个邀请方案，使气氛值的和最大。

例2 没有上司的舞会

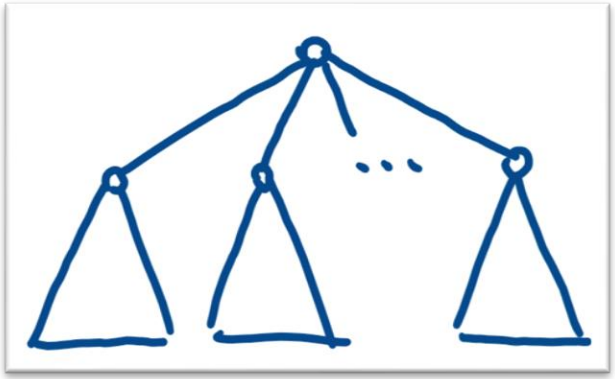
- 贪心，隔一级取一级是否可行？
- 反例：



例2 没有上司的舞会

- 定义状态：
  - 定义 f[i][0/1] 表示以 i 为节点的**子树**所能得到的最大的气氛值。
  - f[i][0] 表示以 i 为子树且**未选择** i 号节点；
  - f[i][1] 表示以 i 为子树且**选择了** i 号节点
  - 最终答案就是 max(f[root][0], f[root][1])

- 注意，子树的求解过程能够很好地体现dp过程中最优子结构的性质。



例2 没有上司的舞会

- 状态转移过程

$$f[i][0] = \sum max\{f[x][1], f[x][0]\}$$

$$f[i][1] = a_i + \sum f[x][0]$$

  - 上司不参加，下属可以参加，也可以不参加。
  - 上司参加，下属只能不参加。

- 枚举量x为当前节点i到能直接到达的所有的子节点。

例3 城市规划

- CSP 201909-5 城市规划（$N \leq 5 * 10^4, M \leq 10^4, K \leq 10^2, c \leq 10^5$）

【题目描述】

有一座城市，城市中有 N 个公交站，公交站之间通过 $N-1$ 条道路连接，每条道路有相应的长度。保证所有公交站两两之间能够通过一唯一的通路互相达到。

两个公交站之间路径长度定义为两个公交站之间路径上所有边的边权和。

现在要对城市进行规划，将其中 M 个公交站定为“重要的”。

现在想从中选出 K 个节点，使得这 K 个公交站两两之间路径长度总和最小。输出路径长度总和即可（节点编号从 1 开始）。

【输入格式】

从标准输入读入数据。

第 1 行包含三个正整数 N，M 和 K 分别表示树的节点数，重要的节点数，需要选出的节点数。

第 2 行包含 M 个正整数，表示 M 个重要的节点的节点编号。

接下来 $N-1$ 行，每行包含三个正整数 a，b，c，表示编号为 a 的节点与编号为 b 的节点之间有一条权值为 c 的无向边。每行中相邻两个数之间用一个空格分隔。

【输出格式】

输出到标准输出。

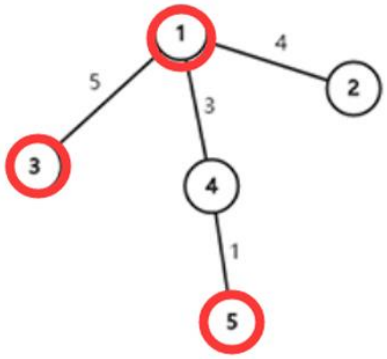
输出只有一行，包含一个整数表示路径长度总和的最小值。

【样例输入】

```
5 3 2
1 3 5
1 2 4
1 3 5
1 4 3
4 5 1
```

【样例输出】

```
4
```



例3 城市规划

- csp201909-5 城市规划
- 如果对这个题没有思路，怎么处理？

【子任务】

测试点	N	M	K	c
1,2	$\le 2,000$	$\le 2,000$	≤ 2	$\le 10^5$
3,4	$\le 5\times 10^4$	≤ 16	≤ 16	
5,6,7	$\le 2,000$	$\le 2,000$	$\le 10^2$	
8,9,10	$\le 5\times 10^4$	$\le 10^4$		

对于所有的数据，$1 \le a,b \le N$，$M \le N$，$K \le M$

- 先从最简单的情况开始分析。能做多少做多少。
- 前20分：
  - 如果 K = 1或 K = 0，则 ans = 0
  - 如果 K = 2，转化为树上给定点集中找出路径最短的两个点。可以采用 dfs 或 bfs 的方式求得。（m不大，可以枚举每个点进行O(n)的遍历）

```
if (k <= 1)cout << 0 << endl;
else if (k == 2)work1();//处理过程
```

例3 城市规划

- csp201909-5 城市规划

【子任务】

测试点	N	M	K	c
1,2	$\leq 2,000$	$\leq 2,000$	≤ 2	$\leq 10^5$
3,4	$\leq 5\times 10^4$	≤ 16	≤ 16	
5,6,7	$\leq 2,000$	$\leq 2,000$	$\leq 10^2$	
8,9,10	$\leq 5\times 10^4$	$\leq 10^4$		

对于所有的数据，$1 \leq a,b \leq N$，$M \leq N$，$K \leq M$

- 接下来的20分：
  - 特点：m 和 k 都很小，只有 16（2^16 = 65,536）
  - 过程：
    - 预处理出 M 个节点两两间的距离 $O(N * M)$
    - 对 M 进行子集枚举，若子集内元素个数恰好为 K，则更新 ans，$O(2^M * M^2) \approx 2e7$

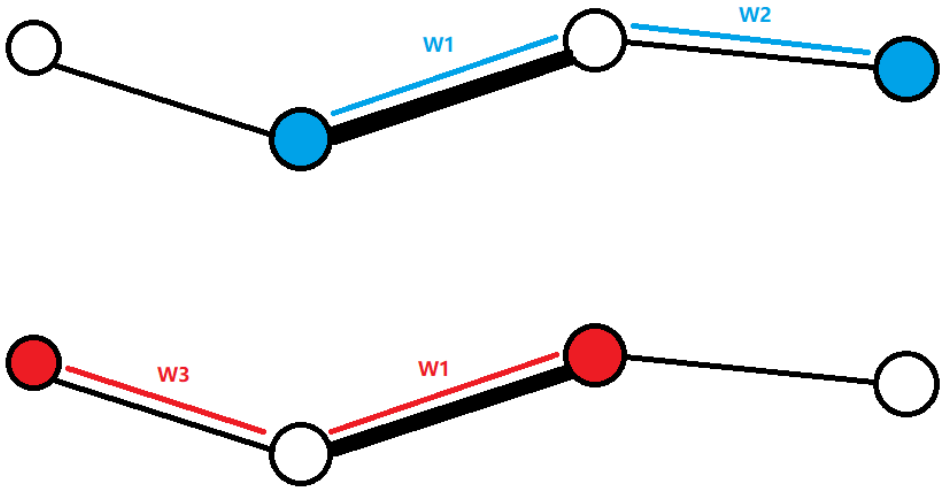
例3 城市规划

- 100分的做法。
- 此时数据范围为 n≤5e4，M≤1e4，K≤100。

- “树上背包”问题。
- 初步的思路：
  - 定义 f[u][s] 表示以点 u 为根的子树中选取 s 个节点，此时子树中所有边对答案的贡献。
  - 转移利用节点与直接节点之间的关系，进行求解。
  - 最终答案就是 f[root][K]

例3 城市规划

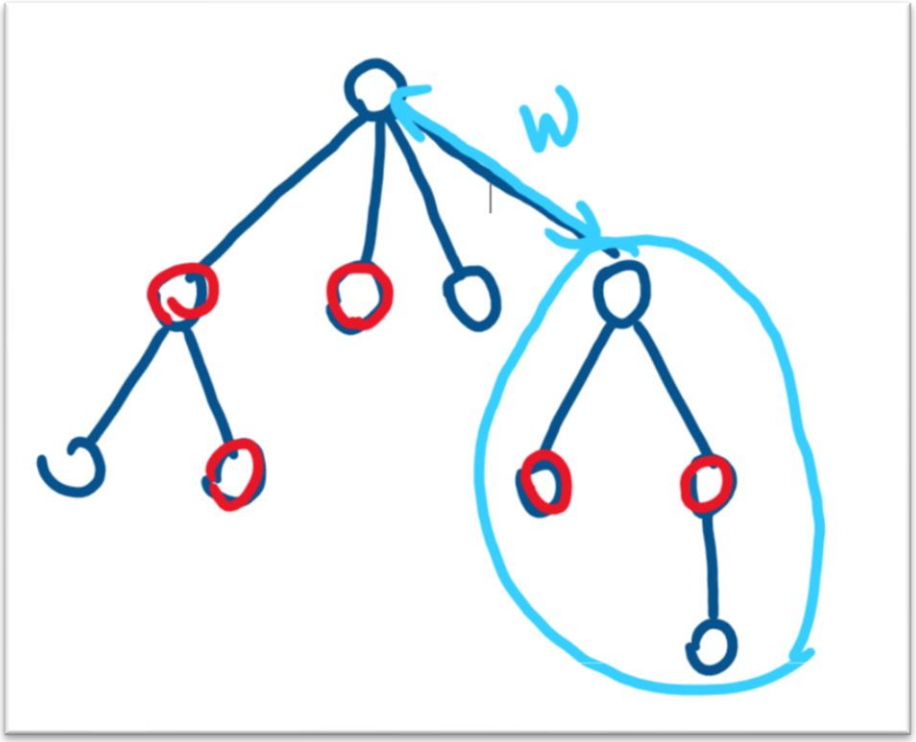
- 状态设计
- 什么是一条边的贡献
- 假设有两条路径经过边W1，则W1的贡献就是2 * W1



- 为了便于分析，我们把一条路径拆分成一条一条边，然后以边为单位计算贡献

例3 城市规划

- 如何计算一条边的贡献？

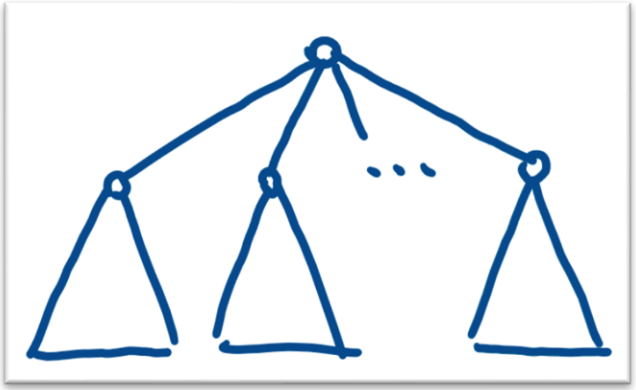


- 假设一共需要选取5个点
- 如同，对于标记为W的边
- W下面的子树，选取了2个点
- W的另一端，选取了5-2=3个点

- 那么当前这条边的“贡献”值为 2*3*w。

例3 城市规划

- 状态转移方程（初步）
  - 对于每一个正在被更新的节点 u，与他直接相连的点（除了父节点）可能很多。
  - 以 u 为根的这个子树依赖于内部更小的子树。
  - 每个子树，都可以在其中选取**有限**个点。
  - 不同的子树中选的数量可以不同。
  - 类似多重背包



  - 如果是100分的话，已经默认需要滚动数组了
  - 按照前文所述的状态，对于每个节点而言只有1维来记录有关选取情况的信息。

例3 城市规划

- 多重背包回忆（非二进制版本）
  - n 类物品，第 i 类物品有 a_i 个，重量为 w_i，价值为 v_i，背包最大重量为 m，求装满背包的最大价值。

```
for(int i = 1; i <= n; i++) {
    for(int j = 0; j <= m; j++) {
        f[i][j] = f[i-1][j];
        for(int k = 0; k <= a[i]; k++)
            if(j >= k*w[i])
                f[i][j] = max(f[i][j], f[i-1][j-k*w[i]] + k*v[i]);
    }
}
```

  - 滚动数组

```
for(int i = 1; i <= n; i++) {
    for(int j = m; j >= 0; j--) {
        for(int k = 0; k <= a[i]; k++)
            if(j >= k*w[i])
                f[j] = max(f[j], f[j-k*w[i]] + k*v[i]);
    }
}
```

例3 城市规划

- 类别本题的树上多重背包
  - $f[u][s]$ 表示以点 u 为根的子树中选取 s 个节点时，对最终答案的贡献。
  - 贡献如何计算？
    - 计算一条边被经过的次数
  - 如何更新 $f[u][s]$？
    - 假如 u 一共有 n 个子节点，每个子节点的子树中有 a[v] 个关键节点
    - 在子节点 v 中选取 x 个关键节点，对答案的贡献为 $f[v][x] + x * (k - x) * w(u, v)$

例3 城市规划

- 状态转移方程
  - 计算以u为根节点的子树的答案
  - 假设当前枚举到以v为根节点的子树，边<u, v>连接着它们
  - 枚举子树v中选取点的个数 j

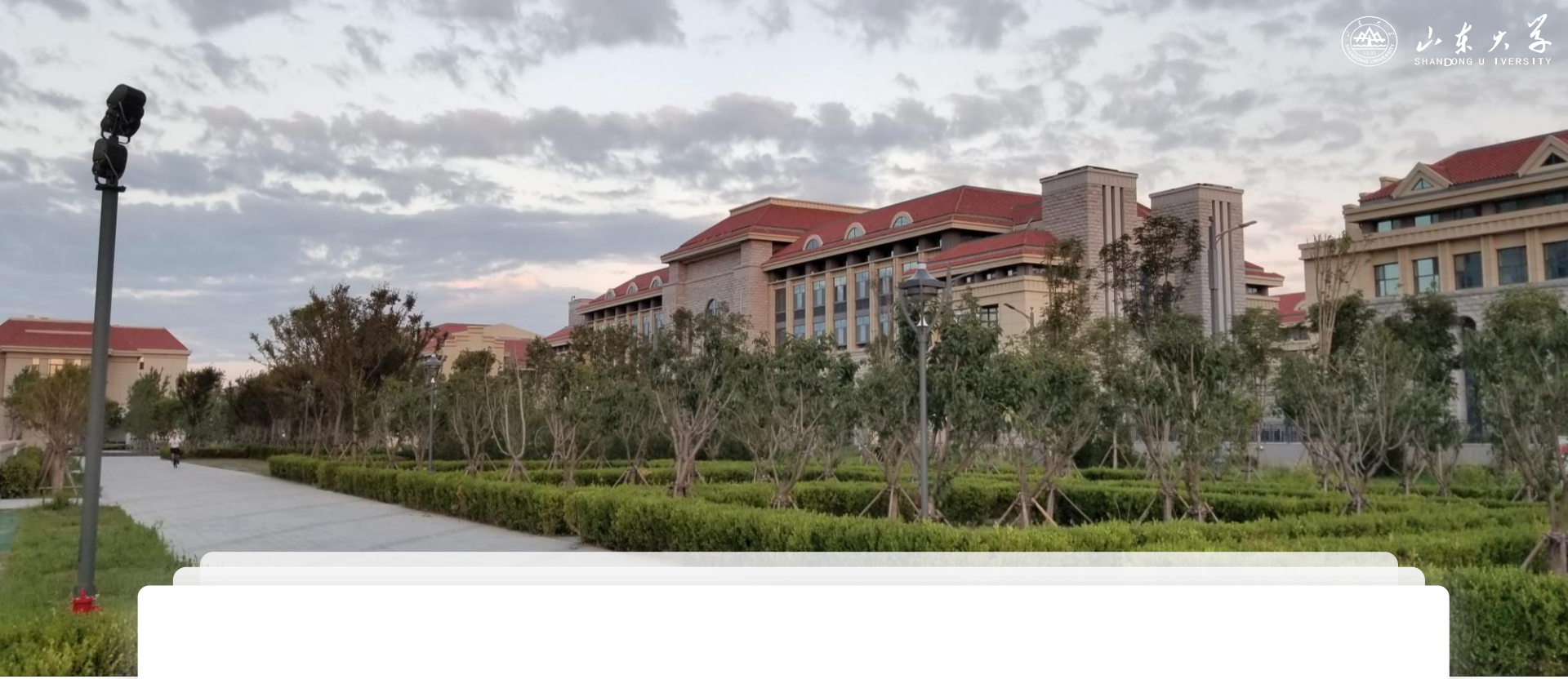
$$f[u][s] = min\{f[v][j] + f[u][s-j] + j*(K-j)*w\}$$

  - f[v][j]：子树v的最小贡献
  - f[u][s-j]：选取剩下s-j个点的最小开销
  - j *（K – j）* w：边<u, v> 的贡献
  - 注意：s 应为**逆序**

例3 城市规划

- 多重背包无需优化，逐个枚举即可。
- 核心代码：

```
for (int i = head[u]; i; i = e[i].next) {
    if (e[i].to == fa)continue;
    dfs(e[i].to, u);
    ct[u] += ct[e[i].to];
    for (int j = min(k, ct[u]); j >= 0; --j) {
        for (int l = 0; l <= j && l <= min(k, ct[e[i].to]); ++l) {
            f[u][j] = min(f[u][j], f[e[i].to][l] + f[u][j - l] + 1ll * l * (k - l) * e[i].vi);
        }
    }
}
```



3 DP优化

Optimization of Dynamic Programming

DP优化方法

- 常见的DP优化方法有以下几种：
  - 滚动数组优化（针对空间）
  - 数据结构优化（包括但不限于前缀和/差分数组、单调栈/队列、线段树/树状数组、set/map等数据结构）
  - 状态设计优化
  - 斜率优化*
  - 四边形不等式优化*
- 由于课时有限，后面三种优化方法不展开讲解，有兴趣的同学可以自行查阅资料。

滚动数组

- 在0-1背包和完全背包问题中，我们使用了一维数组解决了二维状态的问题

  - 滚动数组
    - 如果只用一个数组 f[V]，能不能保证第 i 次循环结束后 f[j] 中表示的就是我们定义的状态 f[i][j]?
    - f[i][j] 是由 f[i-1][j] 和 f[i-1][j-w[i]] 两个子问题递推过来
    - 只需要保证枚举顺序从 j = V → 0 即可
    - 滚动数组的关键点是：逆序!

      如果是顺序的话则会出现同一件物品选多次的情况。

- 如果转移顺序不保证顺序/逆序，怎么办呢?

滚动数组

- 假设转移方程为 $f[i][j] += f[i-1][(j*a[i])\%m]$ ，正向枚举和逆向枚举都不能实现一维数组转移。
- 但并不是没有优化空间的余地，发现转移时只有当前行和上一行是用得到的，因此只需要用两个数组记上一行和当前行即可。

```
int f[2][MAXM]; // f[i & 1]是当前行，f[i & 1 ^ 1]是上一行
for (int i = 1; i <= n; i++) {
    fill(f[i & 1], f[i & 1] + m, 0);
    for (int j = 0; j < m; j++)
        f[i & 1][j] = f[i & 1 ^ 1][(j * a[i]) % m];
}
```

```
vector<int>f(m, 0); //前一行
for (int i = 1; i <= n; i++) {
    vector<int>d(f.size(), 0); //新的一行
    for (int j = 0; j < m; j++)
        f[j] = d[(j * a[i]) % m];
    f.swap(d);
}
```

前缀和优化

- 前缀和优化适用于转移方程为一段区间和的方程，例如

$$f[i][j] = \sum_{k=l[j]}^{r[j]} f[i-1][k]$$

- 设 $sum[j]$ 为上一行的前缀和，则转移方程变为：

$$f[i][j] = sum[r[j]] - sum[l[j]-1]$$

- 如果转移方程改成 $f[i][j] = \sum_{k=l[j]}^{r[j]} (k - l[j]) * f[i-1][k]$ 呢？
- 设 $sum2[j]$ 为 $j * f[i-1][j]$ 的前缀和，将 $l[j]$ 提取出来

单调队列优化

- 什么是单调队列优化 DP?
  - 其实就是 DP 的转移方程是一个线性转移式
    - 例如：$f[i] = \min(f[i-k]), 1 \le k \le 10$
  - DP 值由一个窗口内的最 小 / 大 值决定
    - 因此可以使用滑动窗口的思想进行时间复杂度优化
    - 即使用单调队列 $O(1)$ 求取一个窗口内的最小值

- 时间复杂度优化：$O(nk) \Rightarrow O(n)$，k 为窗口大小

单调队列复习

- 回忆单调队列

- 由于单调队列 **可以队首出队** 以及 **前面的元素一定比后面的元素先入队** 的性质，使得它可以维护局部的单调性
- 当队首元素不在区间之内则可以出队

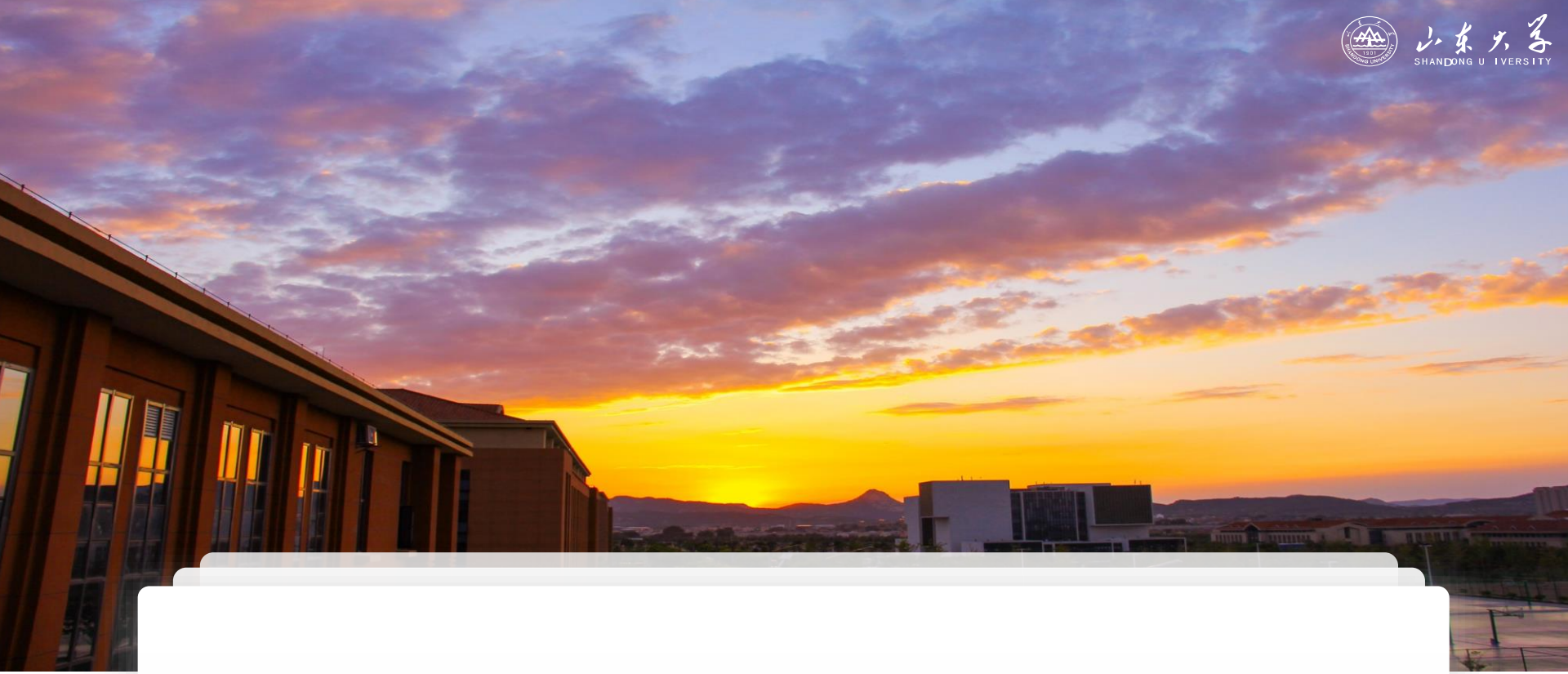
- 单调队列 = 单调 + 队列
- 单调
  - 单调递增 1 2 3 4 5 6 7...
  - 单调递减 7 6 5 4 3 2 1...
  - 单调非减 1 1 2 3 4 4 5...
  - 单调非增 7 7 6 5 4 4 3...
- 队列
  - 一种线性数据结构，支持 push 和 pop 操作
  - 已经在 「数据结构与算法」课程中学习过
  - 满足先进先出 FIFO 原则
- 单调队列 = 队列里的元素满足 **出队** 顺序的 **单调性**

```
INITIALIZE queue
FOR each element u DO
  WHILE queue.size() > 0 AND queue.front() does not belong to the interval DO
    queue.pop_front()
  END
  WHILE queue.size() > 0 AND queue.back() > u DO
    queue.pop_back()
  END
  queue.push(u)
END
```

单调队列复习

- 滑动窗口问题
  - 查找一个窗口内最小值，局部最小值问题

Window position	Minimum value	Maximum value
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7



4 例 4 讲 解

Example 4

例4 最大区间和III

- 最大区间和III
- 输入一个长度为 n 的整数序列，从中找出一段**不超过 m** 的**连续**子序列（区间），使得这个序列的和最大。

  - 6 3
  - 1 -3 **5 1** -2 3

  - ans = 6

例4 最大区间和III

- 这个题和“动态规划（一）”中的最大区间和 I 很像。
- 但多了一个长度不超过 m 的限制。

- 当时的解决方案。

  - 贪心： O(n^3) -> O(n^2) -> O(n)
  - O(n^2)：维护前缀和 sum[]，假如我们选了 [L, R]，那么答案就是：
    - ans = sum[R] – sum[L-1]
  - O(n)：观察上述答案表达式，若固定 R 不变，则让 ans 最大，则可让 sum[L-1] 最小（1 <= L-1 < R），这个值无需每次遍历，维护即可。

不用贪心，用动态规划的思想做，怎么列方程？

定义 dp_i -- 表示 i 为区间右端点时，可以取到的最大和

- $dp_i = \max(dp_{i-1} + a_i,\ a_i)$
- 要么就取 ai / 要么把 ai 与前面的并在一起

例4 最大区间和III

- 添加了限制之后，又该如何去做？

- 同样，我们可以这样设定状态：
  - f[i] 表示以 i 为结尾的最大区间和。
  - 答案为 max{f[i]}

- 转移方程可列为：

$$f[i] = sum[i] - min\{sum[k]\}(i - m \leq k \leq i)$$

  - 仅仅多了一个 k 的限制

例4 最大区间和III

- 此时的时间复杂度为 O(n*m)。
- 因为减数已经有了位置的限定。
- 不能用一个变量来记录（维护）。

- 回忆“滑动窗口”。
- 这个过程像不像一个长度为m的窗口不断向右滑动，每次取窗口中的最小值？？
- 我们采用类似滑动窗口的思路，维护一个最大容量为 m 的单调队列，就能够将每次 m 的遍历优化到常数。
- 优化后的时间复杂度仍为 O(n)。

例4 最大区间和III

- 代码实现

```
ll ans = 0;
for(int i = 1; i <= n; i++) sum[i] = sum[i-1] + b[i];
q.push_back(0);
for(int i = 1; i <= n; i++) {
    while(!q.empty() && sum[q.back()] > sum[i])
        q.pop_back();
    q.push_back(i);
    while(!q.empty() && i - m > q.front())
        q.pop_front();
    ans = max(ans, sum[i] - sum[q.front()]);
}
```

单调队列优化 DP 延伸

- 多重背包
  - 单调队列优化 DP 可以将多重背包从二进制拆分的 $O(m\sum_{i=1}^{n}\log(k_i))$优化到 $O(nm)$
  - 其中，n 为物品类别数，m 为背包大小，k_i 为第 i 类物品的个数
  - 感兴趣的同学可以查看这篇文章（不作要求）
    - https://oi-wiki.org/dp/opt/monotonous-queue-stack/
- 至此，3 大常见背包均可在 $O(nm)$ 内实现

感谢收听

Thank You For Your Listening