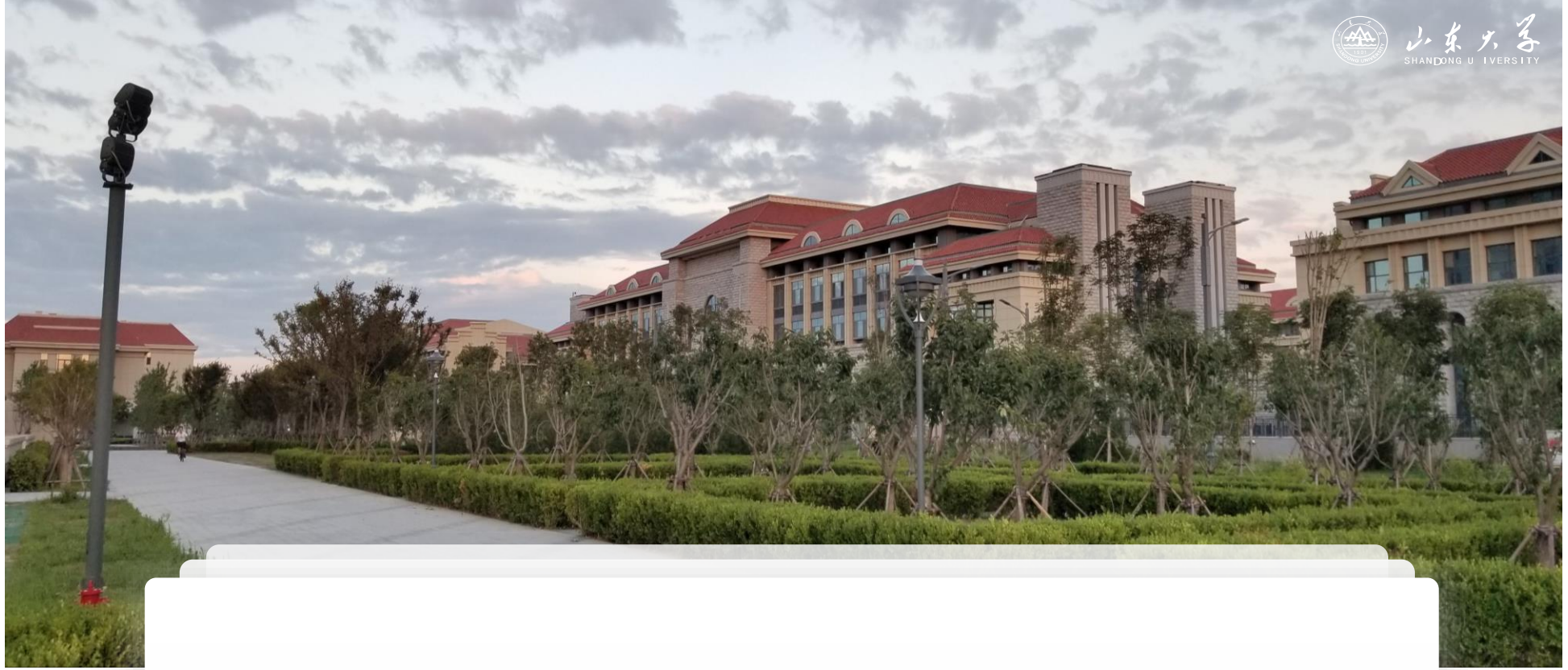
程序设计思维与实践

Thinking and Practice in Programming

动态规划（三） | 内容负责：叶苏伟

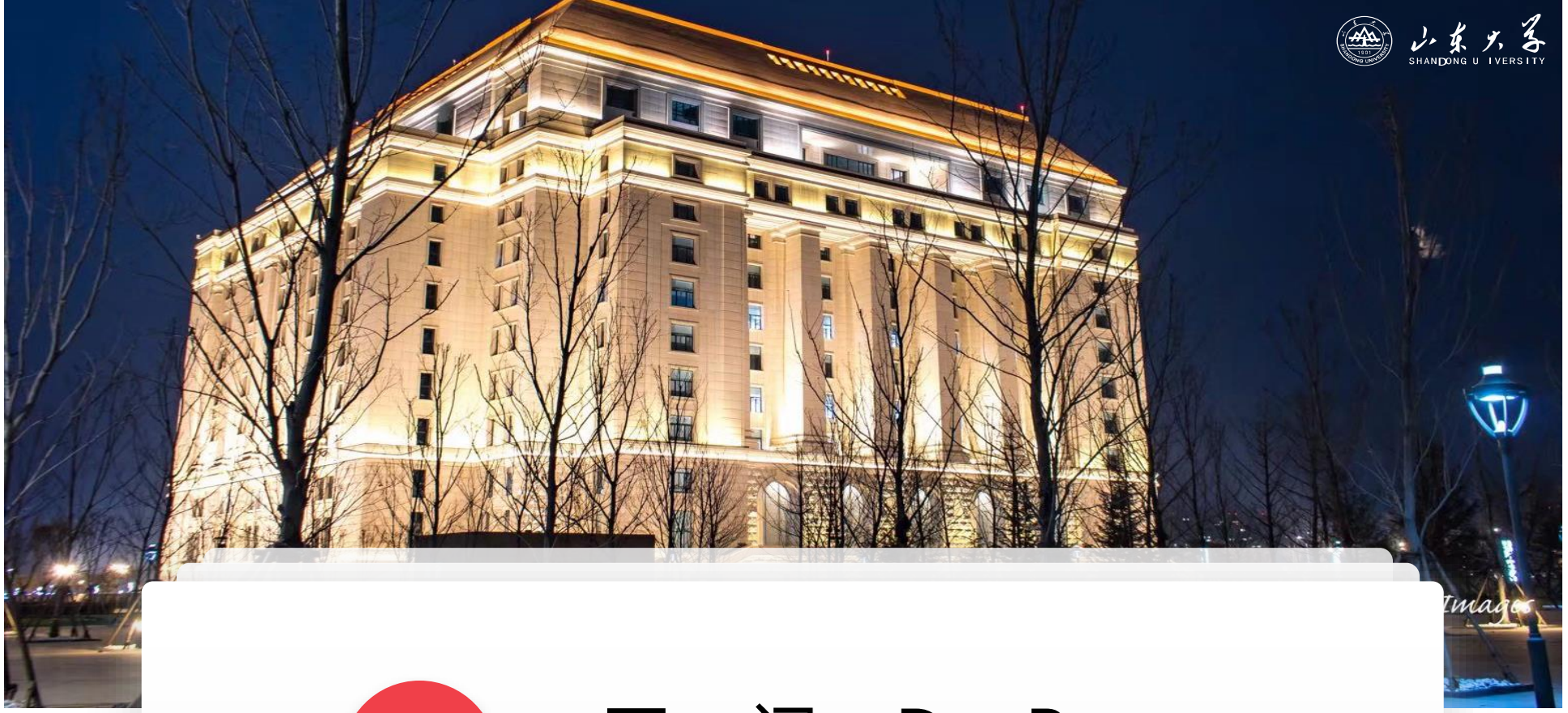


1 课 程 脉 络

The outline of course

课程脉络


动态规划（三）
区间 DP
例题 1
石子合并
例题 2
括号序列
例题 3
最长回文子序列
状压 DP
位运算
例题 4
TSP（变种）
例题 5
作业规划
例题 6
棋盘分割



2 区间DP

Interval

区间DP

- 什么是区间DP?
  - 线性动态规划的一种扩展
  - 在区间上进行的DP，通常是将大区间分割成小区间进行DP
  - 在分阶段划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并过来有很大关系
  - 常见的转移方程类型

$$f[i,j] = max_{i \le k < j}\{f[i,k] + f[k+1,j] + cost\}$$

3 例 1 讲 解

Example 1

例1讲解

- 石子合并问题
  - n堆石子排成一排，现在要将这n堆石子合并成一堆。规定，每次只能选取相邻的两堆进行合并。每次合并的花费为这两堆的石子总数。
  - 求出经过n-1次合并后，所需要的最小花费。
  - n<=100
  - Simple Input:
    - 4
    - 1 3 5 2
  - Simple Output:
    - 22

例1讲解

- 比较容易能够想到的方案：
  - 贪心策略：每次合并相邻两堆和最小的两堆。
  - 对吗？？
  - Example：
    - 3 2 2 3
  - 4 + 7 + 10 = 21 ？？？
  - 5 + 5 + 10 = 20

例1讲解

- 石子合并问题
  - 只有3个数怎么解决?
  - 3 2 5
  - 3 + (2 + 5) + (2 + 5) = 17
  - (3 + 2) + (3 + 2) + 5 = 15
  - ans = 15

例1讲解

- 石子合并问题
  - 考虑4个数的解决方案
  - 3 2 5 4
  - 3 + (2 + 5 + 4) + ans(2,5,4)
  - ans(3,2,5) + (3 + 2 + 5) + 4
  - ans(3,2) + ans(5,4) + (3 + 2) + (5 + 4)
  - 似乎可以归纳出什么?
  - (3 + 2 + 5 + 4) + min(ans(2,5,4),ans(3,2,5),ans(3,2)+ans(5,4))

例1讲解

- 石子合并问题解法：
  - 定义状态 f[i][j] ， 代表区间[i,j]的答案
  - 初始化f[i][i] = 0
  - 递推过程

$$f[i][j] = sum[i][j] + min_{i \le k < j}\{f[i,k] + f[k+1,j]\}$$

  - sum数组也可以利用前缀和实现
  - 复杂度O(n^3)

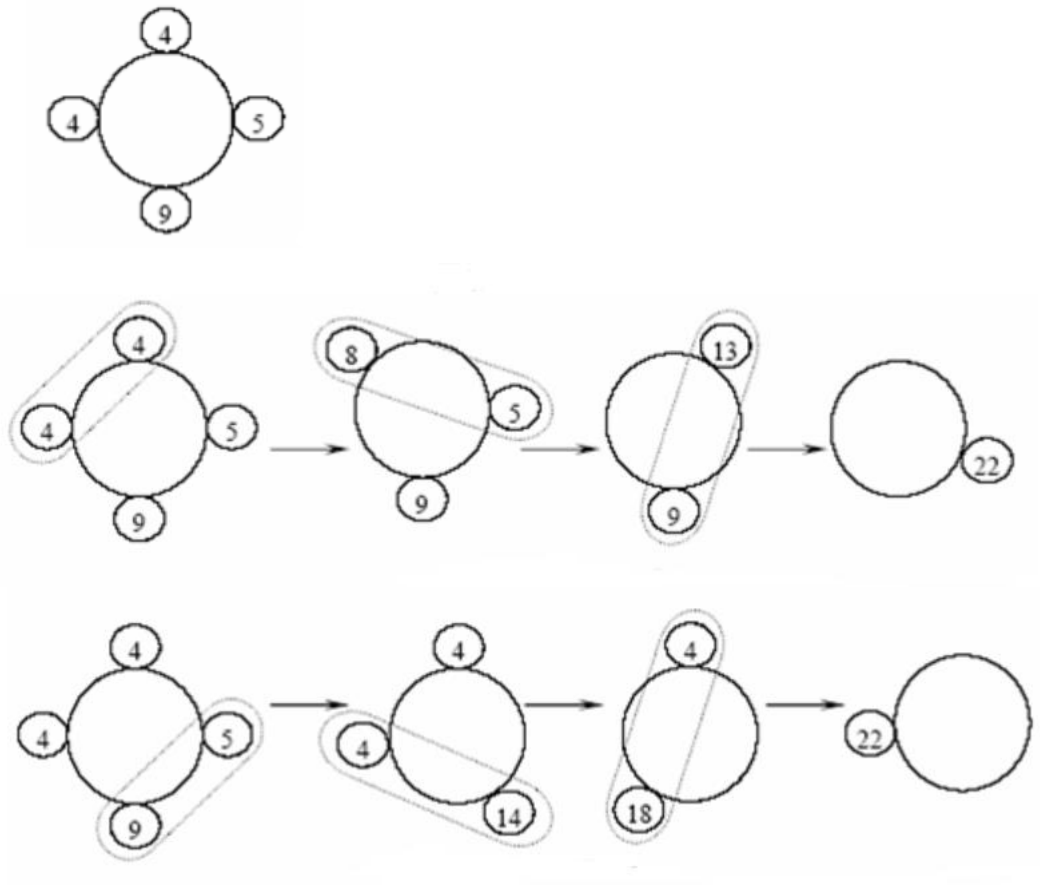
例1讲解

- 石子合并问题代码实现：
  - 1. 记忆化搜索，递归实现。
  - 2. 迭代实现

```
//     初始化过程
memset(f, 63, sizeof f);
for (int i = 1; i <= n; ++i)f[i][i] = 0;
//     处理过程
for (int len = 2; len <= n; ++len) {
    for (int i = 1; i <= n - len + 1; ++i) {
        int l = i, r = i + len - 1;
        for (int k = l; k < r; ++k) {
            //...
        }
    }
}
```

例1拓展

- 如果石子堆不是线性排列的，而是环形排列的，怎么处理？

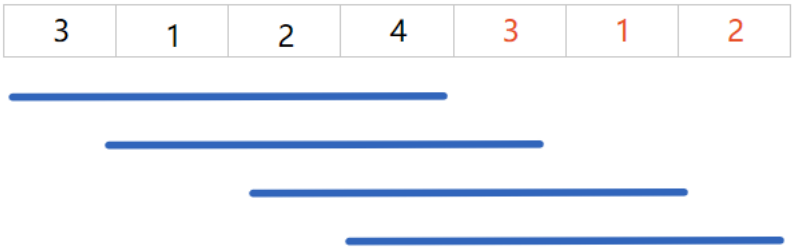


例1拓展

- 方法1
  - 可以看到一定会有一个分断点，在这个点出将整个环拆开，合并后的答案就是最终答案。
  - 枚举断点，得到的答案就是所有枚举方式的最小值。
  - 时间复杂度O(n^4)

例1拓展

- 方法2
  - 思想 – “拆环为链”
  - 例：3 1 2 4
  - 处理方法



  - 和刚刚的思想类似，只是在处理的时候直接扩展数组即可，数组的长度变为2*n-1。
  - 时间复杂度降为O(n^3)

4 例 2 讲 解

Example 2

例2讲解

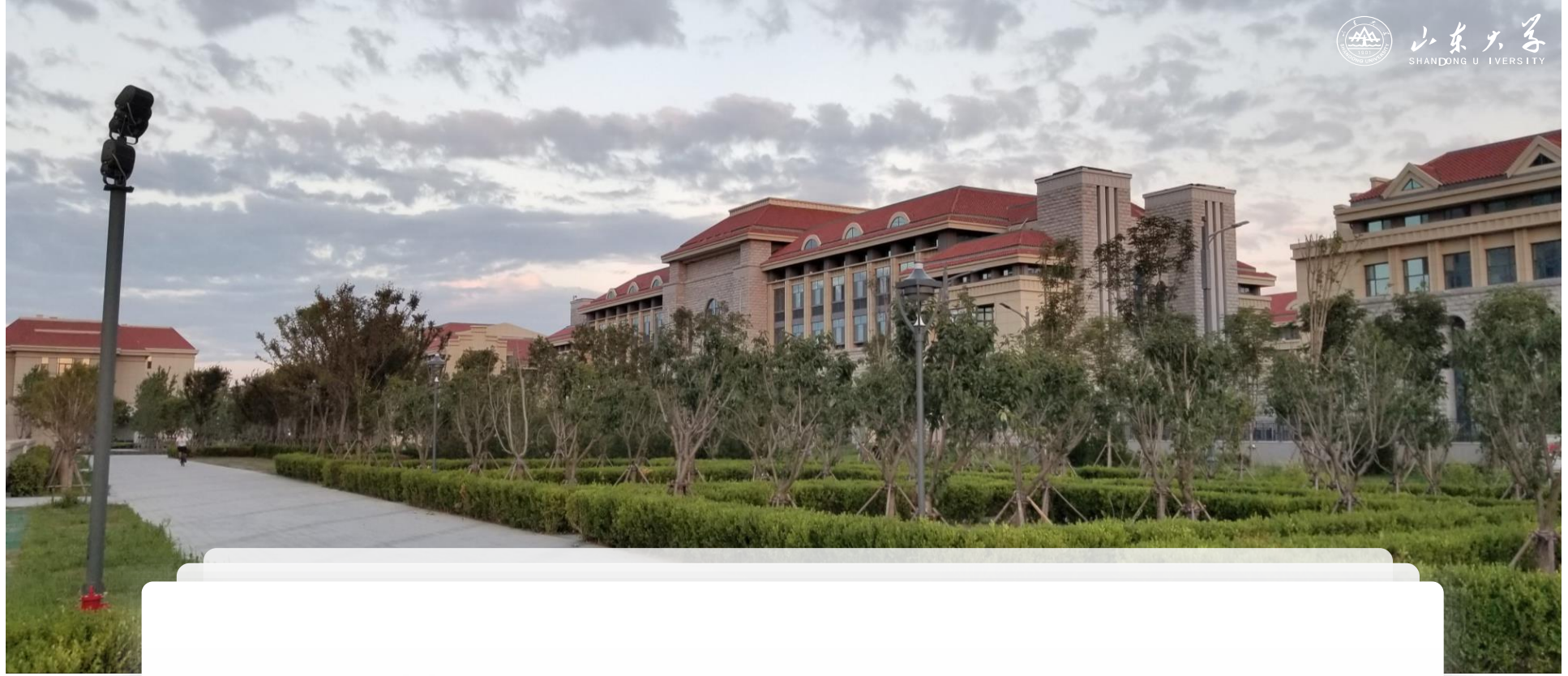
- 括号序列问题：
  - 定义一个合法的括号序列
  - 1. 空序列合法
  - 2. S合法，那么(S)和[S]合法
  - 3. 假设A和B合法，那么AB合法
  - 合法序列 - (), [], (()), ([]), ()[], ()[()]
  - 非法序列 - (, [,],)(, (]), ([)，([)]
  - 给定一个序列，求**最少**添加多少个括号，才能得到一个合法序列。
  - Simple Input: ([()[
  - Simple Output: 3

例2讲解

- 思考：
  - 如果用DP的思想来解决问题，应该怎样去做。
  - 对于本题，状态应该如何去设置。
  - 我们定义：
  - f[i][j]表示子序列[i,j]变成合法序列需要添加的最少括号的数量。
  - 最终答案就应该是f[1][n]

例2讲解

- 思考：
  - 长度为1的串的答案一定是1，长度为0的串的答案一定是0。
  - 对于某个子序列S = [si,sj]
  - 1）f[i][j]的答案可以由子问题更新而来，即
    - f[i][j] = min{f[i][k] + f[k+1][j]},(i≤k<j)
  - 2）若S形如[S’]或(S’) ,那么f[i][j] = min{f[i+1][j-1]}
  - 3）若S形如[S’ 或(S’ ，那么f[i][j] = min{f[i+1][j] + 1}
  - 4）若S形如S’]或S’)，那么f[i][j] = min{f[i][j-1] + 1}
  - **上述四种情况取min，得到的答案就是子序列的答案f[i][j]**
  - 注：3，4情况包含于1。可以看作两种理解方式，即两端的括号既可以左右两端匹配。也可以理解为第一种情况的k=1或k=j-1时的情况。



5 例 3 讲 解

Example 3

最长回文子序列问题

- (Longest Palindromic Subsequence，LPS)
- 子串和子序列的区别
  - 子串是指字符串中连续的一段子字符串
  - 子序列不一定要连续
  - 例如：“awbcdewgh”
  - 子串 – awbc，awbcd，wbcde
  - 子序列 – abc，abcd，abcdeh

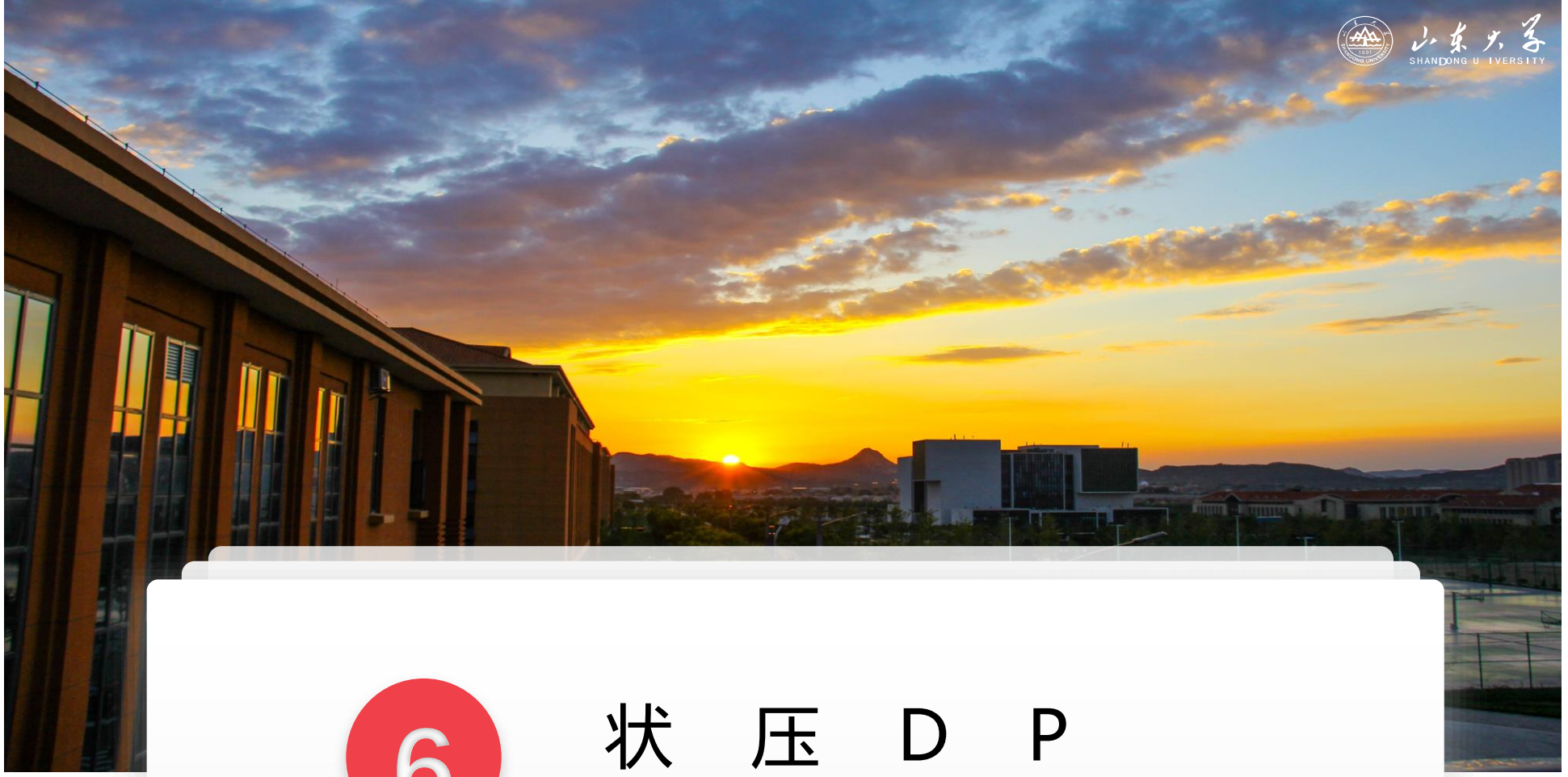
- 回文序列的概念
  - 即字符串正向读与反向读均相同
  - 例如：
    - a，aa，aba，aaabbbaaa，aabbaa 等

最长回文子序列问题

- 如何求最长回文子串？
  - 长度为1的一定是
  - 长度为2的需要判断是否相同
  - 设f[i][j]（bool类型）表示子区间[i,j]是否回文
  - 对于长度大于2的子区间[i,j]
    - f[i][j] = f[i+1][j-1]&(str[i]==str[j])
  - 答案为max{j-i+1}(j>i,f[i][j]==1)

最长回文子序列问题

- 如何求最长回文子序列?
  - 同样，长度为1的一定是
  - 定义f[i][j]表示区间[i,j]内的最长回文子序列的长度。
  - 1）可以由已经求得的规模更小的更新而来
    - f[i][j] = max(f[i+1][j],f[i][j-1])
  - 2）如果str[i]==str[j]
    - f[i][j] = f[i+1][j-1] + 2
  - f[1][n]即是所求的答案
  - 时间复杂度O(n^2)



6 状 压 D P

Bitmasks

状压 DP

- DP 主要难点之一在于状态的表示
- 尤其对于某些题目来说，状态尤其复杂，如果使用先前的多维数组来表示，将会导致数组维度非常大，可操作性非常低
- 因此我们考虑使用一些编码技术，比如二进制编码，用一个数字来表示一个状态，实现状态压缩的目的
- 在动态规划中，使用特定编码技术来压缩状态即为状压 DP

子集表示

- 一包含三个物品的集合，请问如何表示它的子集

- 方法一，多维度数组 ， f[2][2][2]。

- 方法二，状态压缩，int X。

位运算

- 查询状态 S 中是否有编号为 i 的点

```
if(S & (1 << i)) return true;
else return false;
```

- 在状态 S 中添加编号为 i 的点

```
S = S | (1 << i)
```

- 在状态 S 中删去编号为 i 的点

```
if(S & (1 << i))          // S 中有编号为 i 的点
    S = S ^ (1 << i);     // 删去 S 中编号为 i 的点
```

内建函数

- 下述均为内建函数，经过了编译器的高度优化，运行速度十分快

```
/*位运算*/
x = 433 -> 110110001 // unsigned int x

/* 计算 x 的二进制中有多少个 1 */
__builtin_popcount(x) -> 5

/* 判断 x 的二进制中 1 的个数的奇偶性 */
__builtin_parity(x) -> 1

/* 计算 n 的二进制末尾最后一个 1 的位置，位置编号从 1 开始 */
__builtin_ffs(x) -> 1

/* 计算 n 的二进制末尾连续 0 的个数 */
__builtin_ctz(x) -> 0

/* 返回前导 0 的个数 */
__builtin_clz(x) -> 23 = 32 - 9

/* 计算 n 的二进制开头第一个 1 的位置，位置编号从 1 开始 */
32 - __builtin_clz(x) = 9

/* 在函数名末尾添加 ll 使参数类型变为 unsigned long long */
__builtin_clzll(x) -> 55
```

8 例 5 讲 解

Example 5

例5讲解

- 题意
  - ZJM 共有 n 个作业，每个作业对应截止时间与完成所需时间。作业超过截止时间一天，则扣一分。
  - 需要给出最少扣的分数与作业完成方案，若有多个方案，输出字典序最小的一个。 $(1 \leq n \leq 15)$
- 举例
  - 3
  - Computer 3 3
  - English 20 1
  - Math 3 2
- 答案
  - 2
  - Computer
  - Math
  - English

例5讲解

- 如何思考？
  - 能否贪心？
    - 作业每超过截止时间一天，需要扣一分，情况复杂，难以贪心
  - 数据范围非常小（$1 \leq n \leq 15$）
    - 考虑状态压缩
    - 令 S 表示当前完成的作业集合

例5讲解

- 状态定义
  - f[S] 表示完成 S 作业集合后被扣的最少分数
- 转移方程
  - sum = S 作业集合对应的总时间
  - f[S|(1<<x)] = f[S] + tmp（作业 x 被扣的分数）
  - c[x] = 作业 x 完成所需时间
  - d[x] = 作业 x 的 DDL
  - tmp = max (sum + c[x] – d[x], 0)

例5讲解

- 如何保证字典序最小
  - 首先对作业按字典序排序
  - 状态 S 越小则其对应的字典序越小
  - 因此 S 从小到大枚举，x 从小到大枚举即可保证字典序最小
- 如何输出方案
  - 对每个状态 S 记录 pre[S] 表示其对应的最后一个作业
  - pre[S] 递归输出

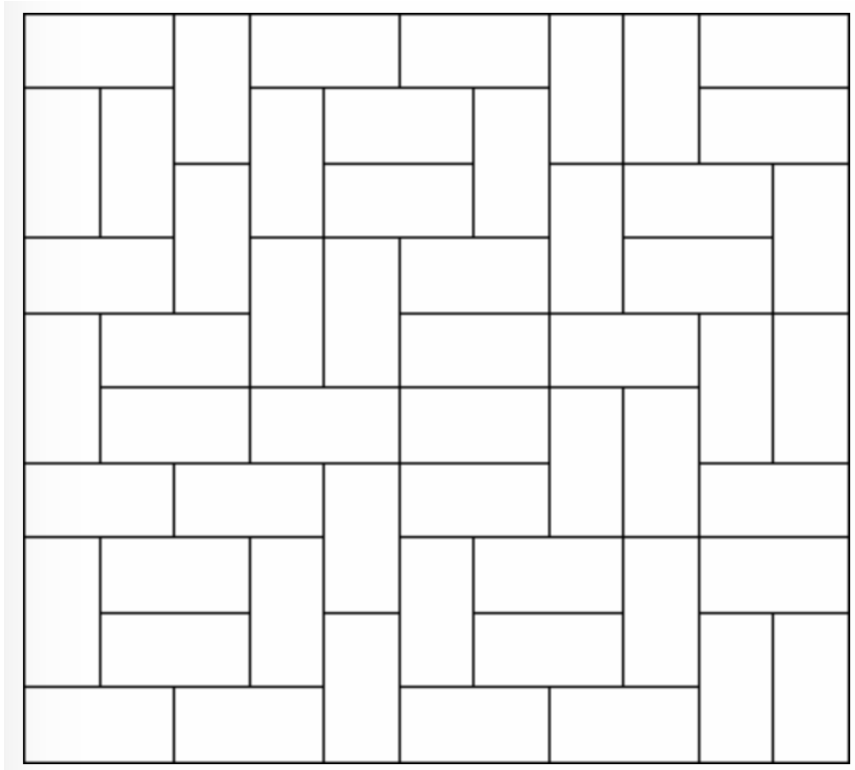
9

例 6 讲 解

Example 6

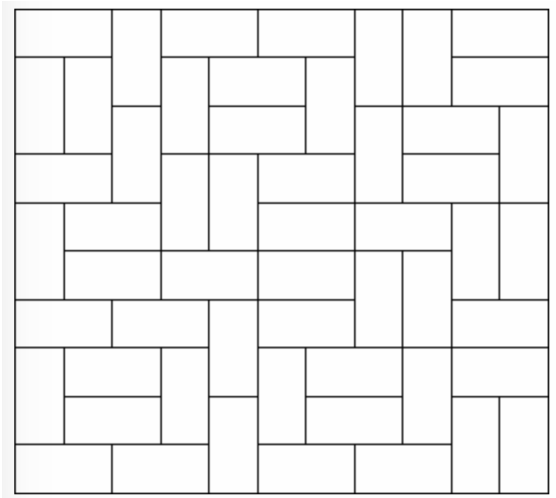
例6讲解

- 题意（POJ-2411）
  - 将高为 h 宽为 w 的棋盘分割成若干个宽和高分别为 1 和 2 的长方形，有多少种方案。（$1 \le h, w \le 11$）



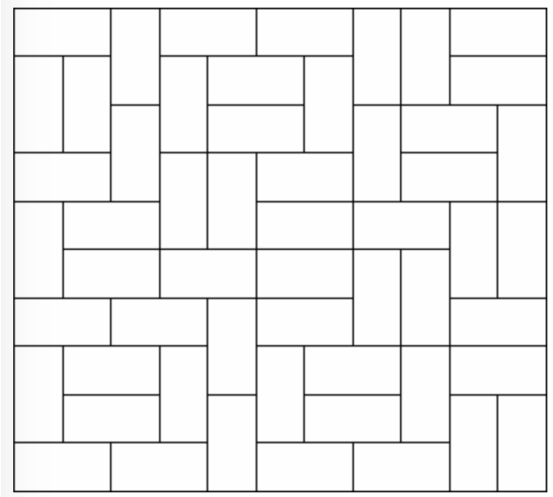
例6讲解

- 思路
  - 数据范围很小，考虑能否进行状压 DP？
- 状态
  - 观察图像，发现位置可以分为两种
    - 不会向下延伸
    - 向下延伸
  - 思考能否对于每一个位置用 0、1 代替



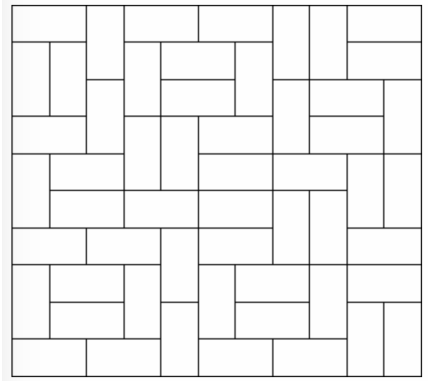
例6讲解

- 状态
  - 根据格子是否会向下延伸，对其用 0、1 分别表示
  - 平躺的长方形 / 竖直长方形的下半部分，即为 0
  - 竖直长方形的上半部分，即为 1
  - f[i][S] 表示填充到了第 i 行，第 i 行填充状态为 S 的方案数



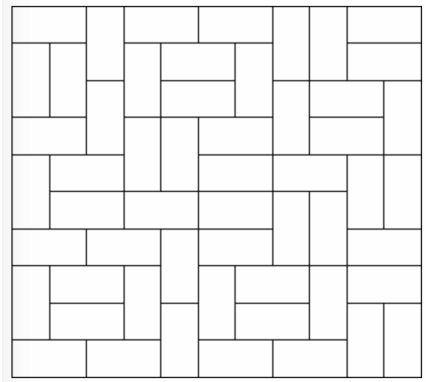
例6讲解

- 状态转移
  - f[i][S1] += f[i-1][S2]，当且仅当下述两个条件成立
    - S1 & S2 == 0
      - 每个数字 1 下方必须为 0
      - 竖着的 1*2 长方形需要补全
    - S1 | S2 == S3
      - S3 需保证每一段连续的 0 都必须为偶数个
      - 连续的 0 表示若干个横着的 1*2 长方形



例6讲解

```
f[0][0] = 1;
for (int i = 1; i <= h; i++) {// 第i行
    for (int S1 = 0; S1 < (1 << w); S1++) {
        for (int S2 = 0; S2 < (1 << w); S2++) {
            if ((S1 & S2) != 0)continue;
            if (!check( x: S1 | S2)) continue;
            f[i][S1] += f[i - 1][S2];
        }
    }
}
```



感谢收听

Thank You For Your Listening