问题笔记

背包问题

01背包

- **背包容量**：给定一个容量为 W 的背包。
- **物品**：有 n 个物品，每个物品有重量 w_i 和价值 v_i。
- **目标**：选择一些物品装入背包，使得总重量不超过 W，且总价值最大。
- **特点**：01选择，每个物品要么选（1），要么不选（0），不能分割。

动态规划解法

1. 定义状态
   - `dp[i][j]`：前 `i` 个物品在容量为 `j` 时的最大价值。
2. 状态转移方程
   - `dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])`
3. 初始化
   - `dp[0][j] = 0`（没有物品时价值为0）
   - `dp[i][0] = 0`（容量为0时价值为0）
4. 最终结果
   - `dp[n][W]` 即为最大价值。

优化空间复杂度

使用一维数组 `dp[j]` 代替二维数组，从后向前更新：

```
int knapsack(int W, vector<int>& weights, vector<int>& values) {
    // 初始化 dp 数组，大小为 W+1，初始值为 0
    vector<int> dp(W + 1, 0);
    // 动态规划求解
    for (int i = 0; i < weights.size(); ++i) { // 遍历每个物品
        for (int j = W; j >= weights[i]; --j) { // 从后向前更新 dp 数组
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }
    return dp[W];
}
int main() {
    int W = 50; // 背包容量
    vector<int> weights = {10, 20, 30}; // 物品重量
    vector<int> values = {60, 100, 120}; // 物品价值
```

```
    int n = weights.size(); // 物品数量
    // 调用函数求解
    int result = knapsack(W, weights, values);
    cout << "最大价值为: " << result << endl; // 输出结果
    return 0;
}
```

最终时间复杂度为$O(nW)$，空间复杂度为$O(W)$。

> **为什么内层循环j从大到小?**
>
> 在第i次循环中，计算的 `dp[j]` 为 `dp[i][j]`，要使用 `dp[i-1][j]` 和 `dp[i-1][j-w[i]]`。如果j从小到大遍历，此时的 `dp[j-w[i]]` 已经变成了 `dp[i][j-w[i]]`。

需要恰好装满时的初始化

仅初始化 `dp[0] = 0`，其余均初始化为$-\infty$，表示这些容量在初始状态下无法被恰好装满，是非法的。状态转移方程不变。同时注意需要进行前驱状态是否合法的检查。

```
int exactKnapsack(int W, vector<int>& weights, vector<int>& values) {
    vector<int> dp(W + 1, INT_MIN); // 初始化为极小值
    dp[0] = 0; // 容量0时恰好装满，价值为0
    for (int i = 0; i < weights.size(); ++i) {
        for (int j = W; j >= weights[i]; --j) {
            if (dp[j - weights[i]] != INT_MIN) { // 检查前驱状态是否合法
                dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
            }
        }
    }
    // 返回结果：若无法装满则返回0，否则返回最大价值
    return (dp[W] == INT_MIN) ? 0 : dp[W];
}
```

这里的 `INT_MIN` 定义如下:

```
#include <climits>
#define INT_MIN     (-2147483647 - 1)
```

完全背包

与01背包问题的区别是，每个物品的数量不是1而是无穷多个，因此每个物品都可能被选择多次。
定义一个二维数组 `dp[i][j]`，其中 `dp[i][j]` 表示前 `i` 种物品在背包容量为 `j` 时的最大价值。对于每种物品 `i`，我们有两种选择:

1. **不选择**该物品：`dp[i][j] = dp[i-1][j]`
2. **选择**该物品：`dp[i][j] = dp[i][j - w[i]] + v[i]`

因此，状态转移方程为：`dp[i][j] = max(dp[i-1][j], dp[i][j - w[i]] + v[i])`

优化空间复杂度

由于 `dp[i][j]` 只依赖于 `dp[i-1][j]` 和 `dp[i][j - w[i]]`，我们可以将二维数组优化为一维数组 `dp[j]`，其中 `dp[j]` 表示背包容量为 `j` 时的最大价值。
优化后的状态转移方程： `dp[j] = max(dp[j], dp[j - w[i]] + v[i])`

```
int knapsack(int W, vector<int>& weights, vector<int>& values) {
    vector<int> dp(W + 1, 0);
    for (int i = 0; i < weights.size(); ++i) {
        for (int j = weights[i]; j <= W; ++j) {
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }
    return dp[W];
}
```

> 内层循环j从小到大
>
> 与01背包问题不同，这里的`dp[j]`为`dp[i][j]`，计算要用到`dp[i][j-w[i]]`，因此要正向循环`j`。

时间复杂度为$O(nW)$，空间复杂度为$O(W)$。

二维费用的背包问题

二维费用的背包问题是背包问题的一个扩展。与普通的背包问题不同，二维费用的背包问题中，每个物品不仅有一个重量限制，还有一个额外的费用限制。给定一个容量为 `W` 的背包和一个费用限制为 `C`，以及 `n` 种物品，每种物品有一个重量 `w[i]`、一个费用 `c[i]` 和一个价值 `v[i]`。我们的目标是从这些物品中选择一些放入背包，使得背包中的物品总重量不超过 `W`，总费用不超过 `C`，且总价值最大。

解法

1. 状态定义：定义一个三维数组 `dp[i][j][k]`，其中 `dp[i][j][k]` 表示前 `i` 种物品在背包容量为 `j` 且费用限制为 `k` 时的最大价值。
2. 状态转移方程：对于每种物品 `i`，我们有两种选择：
   - **不选择**该物品：`dp[i][j][k] = dp[i-1][j][k]`
   - **选择**该物品：`dp[i][j][k] = dp[i-1][j - w[i]][k - c[i]] + v[i]`
     因此，状态转移方程为：`dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j - w[i]][k -`

`c[i]] + v[i])`

3. 初始化：
    - `dp[0][j][k] = 0`：表示没有物品时，背包的最大价值为0。
    - `dp[i][0][k] = 0`：表示背包容量为0时，最大价值为0。
    - `dp[i][j][0] = 0`：表示费用限制为0时，最大价值为0。

优化

由于 `dp[i][j][k]` 只依赖于 `dp[i-1][j][k]` 和 `dp[i-1][j - w[i]][k - c[i]]`，我们可以将三维数组优化为二维数组 `dp[j][k]`，其中 `dp[j][k]` 表示背包容量为 `j` 且费用限制为 `k` 时的最大价值。
优化后的状态转移方程：`dp[j][k] = max(dp[j][k], dp[j - w[i]][k - c[i]] + v[i])`

```
int knapsack(int W, int C, vector<int>& weights, vector<int>& costs, vector<int>& values)
{
    vector<vector<int>> dp(W + 1, vector<int>(C + 1, 0));
    for (int i = 0; i < weights.size(); ++i) {
        for (int j = W; j >= weights[i]; --j) {
            for (int k = C; k >= costs[i]; --k) {
                dp[j][k] = max(dp[j][k], dp[j - weights[i]][k - costs[i]] + values[i]);
            }
        }
    }
    return dp[W][C];
}
```

时间复杂度为$O(nWC)$，空间复杂度为$O(WC)$。

变式：计算最小值的动态规划

例题：洛谷1679
输入一个整数m，将其分解为n个四次方数之和，求n的最小值。
思路：显然n有最大值m，即分解为m个$1=1^4$。等效为完全背包问题，且要求背包恰好装满（用一系列四次方数）。价值等效为装入数的数目，求价值最小值。
代码如下：

```
const int MAX = numeric_limits<int>::max();
int main() {
    int m;
    cin >> m;
    int num = pow(m, 0.25) + 1;  // 计算可装入的最大元素。注意防止下面循环w[j]溢出，再加
一

    vector<int> w(num + 1, 0);  // weight数组
    for (int i = 1; i <= num; ++i) {
        w[i] = pow(i, 4);
    }
```

```
    vector<int> dp(m + 1, MAX);  // 除dp[0]外均初始化为int最大值
    dp[0] = 0;
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; w[j] <= i; ++j) {  // 正向遍历
            if (dp[i - w[j]] != MAX) {  // 合理性检验
                dp[i] = min(dp[i], dp[i - w[j]] + 1);  // 附加value为1，求最小值
            }
        }
    }
    cout << dp[m];
    return 0;
}
```

DFS与回溯

排列数A_n^k

问题：给出n个数：$0, 1, 2, \cdots, n-1$，输出所有可能的排列。

```
const int n = 5, k = 3;  // 当k == n时为全排列
vector<vector<int>> ans;  // 保存结果
vector<int> flag(n, 0);  // 标记数组，是否已被使用
/**
 * @brief 递归地解决n个数的全排列：0 ~ n-1.
 * @param wk 工作集，保存已排列的元素，wk.size()为递归深度
 */
void dfs(vector<int>& wk) {
    if (wk.size() == k) {  // 已经处理完所有元素
        ans.push_back(wk);
        return;
    }
    for (int i = 0; i < n; ++i) {
        if (flag[i] == 0) {
            flag[i] = 1;
            wk.push_back(i);  // 如果要使得保存结果为1 ~ n的全排列，则push_back(i + 1)
            dfs(wk);  // 递归
            wk.pop_back();  // 回溯
            flag[i] = 0;
        }
    }
}

int main() {
    vector<int> a;
    a.reserve(n);
    dfs(a);  // 从空数组开始调用，size为0，从下标0号元素开始排列
    cout << "Num: " << ans.size();  // 也可以选择输出所有的排列结果
```

```
    return 0;
}
```

组合数C_n^k

问题：给出n个物品，编号为$0,1,2,\cdots,n-1$，从中选择k个物品，输出所有可能的选择。

> **Tip**
>
> 与排列数的区别是，相同元素集合的不同顺序视为相同的结果。一般情况下为了让结果能够满足字典序排列，每次递归后元素的遍历要从上一个元素的后继开始。

```cpp
const int n = 6, k = 3;
vector<vector<int>> ans;  // 保存结果
vector<int> flag(n, 0);  // 标记数组，是否已被使用
/**
 * @brief 递归地解决n个数（0 ~ n-1）中选择k个的组合问题.
 * @param wk 工作集，保存已选择的元素
 * @param index 已遍历过的最后一个元素下标。法二：也可以改为新递归起始遍历元素。
 */
void dfs(vector<int>& wk, int index) {
    if (wk.size() == k) {  // 已经选择k个元素时结束递归，保存结果
        ans.push_back(wk);
        return;
    }
    for (int i = index + 1; i < n; ++i) {  // 法二：int i = index
        if (flag[i] == 0) {
            flag[i] = 1;
            wk.push_back(i);
            dfs(wk, i);  // 递归。法二：dfs(wk, i + 1)
            wk.pop_back();  // 回溯
            flag[i] = 0;
        }
    }
}

int main() {
    vector<int> a;
    a.reserve(n);
    dfs(a, -1);  // 法二：dfs(a, 0)
    cout << "Num: " << ans.size() << endl;
    return 0;
}
```

2^n种选择

问题：给出n个物品，每个物品都可以选择或不选，输出所有可能的选择。

```
const int n = 5;
vector<string> ans;  // 保存结果
/**
 * @brief 输出给定长度为n的所有可能的01串.
 * @param wk 工作集，保存遍历元素的选择结果。也可以修改为某个与选择有关的操作数
 * @param deep 递归深度，当前需要选择的元素下标。
 */
void dfs(string& wk, int deep) {
    if (wk.size() == n) {  // 已经处理完所有元素
        ans.push_back(wk);
        return;
    }
    wk.push_back('0');  // 不选择该物品。按照先0后1的顺序，结果为按照字典序排列
    dfs(wk, deep + 1);
    wk.back() = '1';  // 回溯，选择该物品
    dfs(wk, deep + 1);
    wk.pop_back();
}

int main() {
    string s;
    dfs(s, 0);  // 从0号元素开始递归
    cout << "Num: " << ans.size() << endl;
    return 0;
}
```

如果题目要求至少需要选择一件物品，那么只有$2^n - 1$种可能。可以给函数添加一个参数 `count` 表示已选择的物品数目。

```
void dfs(string& wk, int deep, int count) {
    if (wk.size() == n) {  // 已经处理完所有元素
        if (num > 0) ans.push_back(wk);  // 至少选择一件物品才保存结果
        return;
    }
    wk.push_back('0');  // 不选择该物品
    dfs(wk, deep + 1, count);
    wk.back() = '1';  // 回溯，并选择该物品
    dfs(wk, deep + 1, count + 1);
    wk.pop_back();
}
```

八皇后问题

八皇后问题是一个经典的回溯算法问题，要求在8×8的国际象棋棋盘上放置8个皇后，使得它们互不攻击。皇后可以攻击同一行、列或对角线上的其他棋子。

思路：

1. 使用一个一维数组记录每行皇后所在的列
2. 递归地解决每行的位置，若安全，则放置，继续解决下一行，若不安全，则**回溯**
3. 边界条件：递归到行数为N时，得到一个安全放置方案，保存结果

示例代码如下。

```
const int N = 8;  // 棋盘为N*N正方形，有N个皇后
vector<vector<int>> solutions;  // 保存结果

bool isSafe(vector<int>& board, int row, int col) {
    // 检查列
    for (int i = 0; i < row; i++) if (board[i] == col) return false;
    // 检查左上对角线
    for (int i = row, j = col; i >= 0 && j >= 0; --i, --j)
        if (board[i] == j) return false;
    // 检查右上对角线
    for (int i = row, j = col; i >= 0 && j < N; --i, ++j)
        if (board[i] == j) return false;
    return true;
}
void solveNQueens(vector<int>& board, int row) {
    if (row == N) {
        solutions.push_back(board);
        return;
    }
    for (int col = 0; col < N; ++col) {  // 对第row行，从第0列开始寻找满足条件的解
        if (isSafe(board, row, col)) {
            board[row] = col;  // 安全，放置
            solveNQueens(board, row + 1);  // 递归
            board[row] = -1;  // 回溯
        }
    }
}
void printSolution(vector<int>& board) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << (board[i] == j ? "Q " : ". ");
        cout << endl;
    }
    cout << endl;
}

int main() {
    vector<int> board(N, -1);  // 初始化
    solveNQueens(board, 0);  // 从第0行开始求解
    cout << "Total solutions: " << solutions.size() << endl;
    for (auto& sol : solutions) printSolution(sol);
```

```
    return 0;
}
```

行和列都是从`0~N`开始遍历，这样求得的解也是按照字典序排列的。

最大子段和

一维问题

题目描述：给出一个长度为 n 的序列 a，选出其中连续且非空的一段使得这段和最大。
来源：洛谷P115

输入格式

第一行是一个整数，表示序列的长度 n。
第二行有 n 个整数，第 i 个整数表示序列的第 i 个数字 a_i。

思路（动态规划）

- 第一个数为一个有效序列
- 如果一个数加上上一个有效序列得到的结果比这个数大，那么该数也属于这个有效序列。
- 如果一个数加上上一个有效序列得到的结果比这个数小，那么这个数单独成为一个新的有效序列
- 在执行上述处理的过程中实时更新当前有效序列的所有元素之和并取最大值。

代码

```
int n, a;
cin >> n >> a;
int ans = 0x80000000, sum = a;  // ans初始化为int最小值，sum初始化为序列第一个元素
for (int i = 1; i < n; ++i) {  // 从第二个元素开始循环
    cin >> a;
    sum = max(sum + a, a);
    ans = max(ans, sum);
}
cout << ans;
```

时间复杂度为$O(n)$，空间复杂度优化为$O(1)$。

二维问题

问题描述：在一个矩阵中找到一个矩形，使得包围的元素之和最大。
来源：洛谷1719

输入格式

第一行是 n，接下来是 n 行 n 列的矩阵。

思路（前缀和+动态规划）

将矩阵压缩，每一行表示该行和上面所有行对应列的元素之和。在保存的矩阵最前面加一行全0。则用两行之差就可以表示若干行元素之和。
最外层用二层循环，遍历所有的行之差组合，内层变成一维数组最大子段和的问题。

代码

```
int n, ans = 0x80000000;
cin >> n;
vector<vector<int>> mat(n + 1, vector<int>(n));  // 添加全0行
for (int i = 1; i <= n; ++i) {
    for (int j = 0; j < n; ++j) {
        cin >> mat[i][j];
        mat[i][j] += mat[i - 1][j];  // 矩阵按行压缩
    }
}
for (int i = 0; i < n; ++i) {  // 注意i和j的遍历范围
    for (int j = i + 1; j <= n; ++j) {
        // 计算第i~j行 (i, j] 的最大区间和
        int sum = mat[j][0] - mat[i][0];  // 用第一个元素初始化sum
        for (int k = 1, a; k < n; ++k) {
            a = mat[j][k] - mat[i][k];
            sum = max(sum + a, a);
            ans = max(ans, sum);
        }
    }
}
cout << ans;
```

最终时间复杂度为$O(n^3)$，空间复杂度为$O(n^2)$。

> **Tip**
>
> 如果将前缀和考虑为以最左上角元素和a_{ij}为对角线的矩形内所有元素之和，这样计算需要四重循环，时间复杂度为$O(n^4)$。

第k小数

输入 n（$1 \leq n < 5000000$）个数字，输出这些数字的第 k 小的数。最小的数是**第 0 小**。

快速排序思路

排序使得第k个数左边的数均不大于该数，右边的数均不小于该数。平均时间复杂度$O(n)$。

```
// 通过qs(v, 0, n - 1, k)调用
void qs(vector<int>& v, int le, int ri, int k) {  // [le, ri]
    while (le < ri) {
        int pvt = v[le];
        int l = le, r = ri;
        while (l < r) {
            while (l < r && pvt <= v[r]) --r;  // 注意这里一定要带等号，否则会死循环
            v[l] = v[r];  // pvt在v[r]
            while (l < r && pvt >= v[l]) ++l;  // 这里也带等号
            v[r] = v[l];  // pvt在v[l]
        }
        v[l] = pvt;
        if (l == k) return;
        else if (l < k) le = l + 1;
        else ri = l - 1;
    }
}
// 通过cout << v[k]输出结果
```

堆维护最小的k个数

时间复杂度$O(n \log k)$。

```
cin >> n >> k;
priority_queue<int> pq;  // 大根堆保存最小的k + 1个数.堆顶为第k小
int a;
for (int i = 0; i <= k; ++i) {
    cin >> a;
    pq.push(a);
}
for (int i = k + 1; i < n; ++i) {
    cin >> a;
    if (a < pq.top()) {
        pq.push(a);
        pq.pop();
    }
}
cout << pq.top();
```

nth_element

在头文件 `<algorithm>` 中，保证以$O(n)$的时间复杂度得到结果。

```
nth_element(v.begin(), v.begin() + k, v.end());
```

最大单调子列长度

问题：给出一个数组，求最大单调（递增/递减/不减……）子序列长度。

最大严格单调递增子列

思路

1. `tails` 数组：
   - `tails` 用于存储当前找到的单调递增子列的最小末尾元素。
   - 它始终保持单调递增，因此可以使用二分查找来快速定位。
2. `lower_bound`：
   - `lower_bound` 是 C++ 标准库中的函数，用于在有序范围内查找*第一个大于或等于*目标值的位置。
   - 它的时间复杂度是 O(log n)。
3. 更新 `tails`：
   - 如果 `num` 比 `tails` 中的所有元素都大，则将其添加到 `tails` 的末尾。
   - 否则，用 `num` 替换 `tails` 中第一个大于或等于 `num` 的元素。
4. 结果：
   - `tails` 的长度就是最长严格单调递增子列的长度。

原理分析

1. 贪心策略：
   - 我们希望找到一个尽可能长的单调递增子列。
   - 为了让子列尽可能长，我们需要让子列的末尾元素尽可能小，这样后续的元素更容易添加到子列中。
2. `tails` 数组的作用：
   - `tails[i]` 表示长度为 `i+1` 的所有单调递增子列中，最小的末尾元素。
   - 通过维护这样一个数组，我们可以快速判断当前元素是否可以扩展某个长度的子列。
3. 二分查找的优化：
   - 由于 `tails` 数组是单调递增的，我们可以使用二分查找来快速定位当前元素应该插入或替换的位置。
   - 这避免了暴力搜索的低效性，将时间复杂度从 $O(n^2)$ 降低到 $O(n \log n)$。

代码

```cpp
int lengthOfLIS(vector<int>& nums) {
    vector<int> tails; // 用于存储单调递增子列的最小末尾元素
    for (int num : nums) {
        // 使用二分查找找到第一个大于等于 num 的位置
        auto it = lower_bound(tails.begin(), tails.end(), num);
```

```
        if (it == tails.end()) {
            tails.push_back(num); // 如果 num 比所有 tails 中的元素都大，则扩展 tails
        } else {
            *it = num; // 否则替换当前元素为 num
        }
    }
    return tails.size(); // tails 的长度就是最长单调递增子列的长度
}
```

最大严格单调递减子列

`tails` 数组用于存储单调递减子列的最大末尾元素，保持单调递减。
对于搜索函数，通过传入 `greater<int>()`，将比较逻辑改为 `>`，即查找第一个**小于或等于**目标值的位置。

```
int lengthOfLDS(vector<int>& nums) {
    vector<int> tails; // 用于存储单调递减子列的最大末尾元素
    for (int num : nums) {
        // 使用二分查找找到第一个小于等于 num 的位置
        auto it = lower_bound(tails.begin(), tails.end(), num, greater<int>());
        if (it == tails.end()) {
            tails.push_back(num); // 如果 num 比所有 tails 中的元素都小，则扩展 tails
        } else {
            *it = num; // 否则替换当前元素为 num
        }
    }
    return tails.size(); // tails 的长度就是最长单调递减子列的长度
}
```

最大单调不减子列

思路调整

1. **比较逻辑**：
   - 对于严格单调递增子列，使用 `lower_bound` 查找第一个**大于或等于**当前元素的位置。
   - 对于单调不减子列，使用 `upper_bound` 查找第一个**大于**当前元素的位置。
2. **更新规则**：
   - 如果当前元素 `num` 大于或等于 `tails` 的最后一个元素，则扩展 `tails`。
   - 否则，替换 `tails` 中第一个大于 `num` 的元素。

代码

```
int lengthOfNonDecreasingSubsequence(vector<int>& nums) {
    vector<int> tails; // 用于存储单调不减子列的最小末尾元素
```

```
    for (int num : nums) {
        // 使用 upper_bound 找到第一个大于 num 的位置
        auto it = upper_bound(tails.begin(), tails.end(), num);
        if (it == tails.end()) {
            tails.push_back(num); // 如果 num 大于或等于所有 tails 中的元素，则扩展 tails
        } else {
            *it = num; // 否则替换当前元素为 num
        }
    }
    return tails.size(); // tails 的长度就是最长单调不减子列的长度
}
```

最大单调不增子列

```
int lengthOfNonIncreasingSubsequence(vector<int>& nums) {
    vector<int> tails; // 用于存储单调不增子列的最大末尾元素
    for (int num : nums) {
        // 使用 upper_bound 和 greater<int>() 找到第一个小于 num 的位置
        auto it = upper_bound(tails.begin(), tails.end(), num, greater<int>());
        if (it == tails.end()) {
            tails.push_back(num); // 如果 num 小于或等于所有 tails 中的元素，则扩展 tails
        } else {
            *it = num; // 否则替换当前元素为 num
        }
    }
    return tails.size(); // tails 的长度就是最长单调不增子列的长度
}
```

Dilworth 定理

> 定理
>
> 在任何有限的偏序集中，**最小链覆盖的大小等于最大反链的大小**。

推论：

- **最长单调递增子列的长度等于将序列划分为单调不增子列的最小数目。**
- **最长单调递减子列的长度等于将序列划分为单调不减子列的最小数目。**

状态压缩动态规划与旅行商问题

状态压缩

状态压缩动态规划（State Compression DP）是一种优化动态规划的技术，主要用于处理状态空间较大的问题。通过将状态表示为**二进制数**或其他紧凑形式，减少存储和计算开销。

旅行商问题（TSP）

给出n个城市之间的距离，要求恰好访问每个城市一次，并最终回到出发城市，求最短距离。
输入：一个$n \times n$的矩阵 `dist` 表示任意两个城市之间的距离。
状态压缩方法：以$n = 4$为例，$N = (1 << 4) = 16 = 10000_{(2)}$。$N - 1 = 15 = 1111_{(2)}$表示4个城市都去过的最终态。
以下为指定起点城市的代码。

```
// dist: n * n，距离矩阵；start: 0 ~ n-1，表示起点城市
double tsp(const vector<vector<double>>& dist, int start) {
    int n = dist.size(), N = 1 << n;  // N = 100...0 (n个0)
    constexpr double MAX = numeric_limits<double>::max();
    // dp[mask][u] 表示从城市 0 出发，经过 mask 表示的城市集合，最后到达城市 u 的最短路径
    vector<vector<double>> dp(N, vector<double>(n, MAX));
    // 初始化：从 start 出发
    dp[1 << start][start] = 0;
    // 遍历所有状态
    for (int mask = 1; mask < N; ++mask) {
        for (int u = 0; u < n; ++u) {
            if (!(mask & (1 << u))) continue;  // u未访问，跳过
            for (int v = 0; v < n; ++v) {
                if (mask & (1 << v)) continue;  // v已访问，跳过
                dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v], dp[mask][u] +
dist[u][v]);
            }
        }
    }
    // 计算最终结果：返回从所有城市回到起点城市 0 的最短路径
    double result = MAX;
    for (int u = 0; u < n; ++u) {
        if (u == start) continue; // 跳过起点，因为已经回到起点
        result = min(result, dp[N - 1][u] + dist[u][start]);
    }
    return result;
}
```

1. `dp` 数组：
   - `dp[mask][u]` 表示从城市 0 出发，经过 `mask` 表示的城市集合，最后到达城市 `u` 的最短路径。
   - `mask` 是一个二进制数，表示已访问的城市集合。例如，`mask = 5`（二进制 `101`）表示访问了城市 0 和城市 2。
2. 状态转移：
   - 对于每个状态 `mask` 和城市 `u`，尝试从 `u` 移动到未访问的城市 `v`，并更新新状态 `mask | (1 << v)` 的最短路径。
3. 结果计算：

- 最终结果是所有城市都被访问过（`mask = (1 << n) - 1`），并返回到起点城市 0 的最短路径。

4. **时间复杂度**：
   - 外层循环遍历所有状态 `mask`，共 `2^n` 种。
   - 内层循环遍历所有城市 `u` 和 `v`，共 `n^2` 次。
   - 总时间复杂度为$O(n^2 \cdot 2^n)$，一般解决规模$n \leq 20$的问题。
5. **空间复杂度**：
   - $O(n \cdot 2^n)$。

变体

- 指定起点，但不需要返回起点：最终计算距离时直接比较 `dp[N - 1][u]` 之间的最小值。
- 从任意城市出发，遍历并返回起点：在主函数中循环一次 `start`。
- 只需要遍历所有点：初始化所有 `dp[1 << u][u] = 0`，最终直接比较 `dp[N - 1][u]` 之间的最小值。
- 从原点出发，遍历平面所有点：初始化所有 `dp[1 << u][u] = distance(O, u)`。

矩形覆盖面积

题目

给出一系列平面内矩形（边和坐标轴平行）的坐标，求覆盖区域的面积。
输入格式：

1. `N`：表示矩形的个数；
2. `N` 行 `x1, y1, x2, y2`：表示每个矩形的左上角坐标和右下角坐标。

参见：洛谷1884

扫描线算法

离散化与差分辅助。

思路

- 事件定义为每个矩形平行于x轴的两条边，按照y坐标从小到大遍历，逐行扫描并计算面积。
- 使用结构体 `Event` 保存事件，事件需要保存该区间的起始和终止x坐标，y坐标与类型：`1` 表示这是一条矩形的下边界，扫描开始；`-1` 表示矩形的上边界，扫描结束。
- 对x坐标离散化。保存所有出现过的x坐标，去重排序，每个矩形边界的区间两个端点可以从中搜索到。
- 差分数组保存对于所有的x坐标，这里被多少个矩形覆盖的数组的差分结果。

代码

```
using ll = long long;
struct Event {
    ll x1, x2, y;
    int type; // 1: start, -1: end
    Event(ll x1, ll x2, ll y, int type) : x1(x1), x2(x2), y(y), type(type) {}
    bool operator<(const Event& other) const { return y < other.y; }
};
int main() {
    int N;
    cin >> N;
    vector<Event> events;
    vector<ll> x_coords;
    for (int i = 0; i < N; ++i) {
        ll x1, y1, x2, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        events.emplace_back(x1, x2, y2, 1);
        events.emplace_back(x1, x2, y1, -1);
        x_coords.push_back(x1);
        x_coords.push_back(x2);
    }
    // 排序事件
    sort(events.begin(), events.end());
    // 离散化 x 坐标
    sort(x_coords.begin(), x_coords.end());
    // 去重，unique将重复元素移动到向量末尾并返回第一个重复元素的迭代器
    x_coords.erase(unique(x_coords.begin(), x_coords.end()), x_coords.end());
    // 创建差分数组
    vector<int> diff(x_coords.size(), 0);

    ll prev_y = events[0].y, total_area = 0;
    for (const auto& event : events) {
        ll current_y = event.y;
        if (current_y != prev_y) {
            ll height = current_y - prev_y, width = 0;
            // 使用cover变量还原覆盖矩形数目
            for (int i = 0, cover = diff[i]; i < diff.size() - 1; ++i, cover += diff[i])
{
                if (cover > 0) width += x_coords[i + 1] - x_coords[i];
            }
            total_area += height * width;
            prev_y = current_y;
        }
        // 更新差分数组
        int idx1 = lower_bound(x_coords.begin(), x_coords.end(), event.x1) -
x_coords.begin();
        int idx2 = lower_bound(x_coords.begin(), x_coords.end(), event.x2) -
x_coords.begin();
        diff[idx1] += event.type;
        diff[idx2] -= event.type;
```

```
    }
    cout << total_area << endl;
    return 0;
}
```

为什么使用 `lower_bound` 不使用 `find`

`lower_bound` 搜索第一个大于等于该元素的位置，的时间复杂度为$O(\log n)$，而 `find` 搜索第一个相等元素的位置，时间复杂度为$O(n)$。对于有序数组，二分查找的效率更高。