CSP 复赛复习 - 基础算法(3)

6. 搜索算法

深度优先搜索 (DFS)

核心思想：一条路走到底，走不通再回溯

特点

- 递归实现
- 使用栈结构
- 适合求解所有解的问题

时间复杂度：$O(b^d)$，其中 b 为分支因子，d 为深度

DFS 模板

```cpp
bool visited[N]; // 访问标记数组

void dfs(int u) {
    visited[u] = true; // 标记已访问

    // 处理当前节点
    cout << u << " ";

    // 遍历所有邻居
    for (int v : neighbors[u]) {
        if (!visited[v]) {
            dfs(v); // 递归访问
        }
    }
}
```

DFS 示例：全排列问题

```cpp
int n;
int path[N];    // 当前路径
bool used[N];   // 标记数组

void dfs(int depth) {
    // 递归终止条件
    if (depth == n) {
        for (int i = 0; i < n; i++) {
            cout << path[i] << " ";
        }
        cout << endl;
        return;
    }

    // 遍历所有选择
    for (int i = 1; i <= n; i++) {
        if (!used[i]) {
            used[i] = true;
            path[depth] = i;
            dfs(depth + 1);  // 递归下一层
            used[i] = false; // 回溯
        }
    }
}

int main() {
    cin >> n;
    dfs(0);
    return 0;
}
```

广度优先搜索 (BFS)

核心思想：层层扩展，先访问距离近的节点

特点

- 队列实现
- 保证找到最短路径
- 适合求解最短路径问题

时间复杂度：$O(V+E)$

BFS 模板

```cpp
#include <queue>
using namespace std;

bool visited[N];
int dist[N]; // 距离数组

void bfs(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;
    dist[start] = 0;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        // 处理当前节点
        cout << u << " ";

        // 遍历邻居
        for (int v : neighbors[u]) {
            if (!visited[v]) {
                visited[v] = true;
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

BFS 示例：迷宫最短路径

```cpp
#include <queue>
#include <cstring>
using namespace std;

const int N = 100;
char maze[N][N];
bool vis[N][N];
int dist[N][N];
int dx[4] = {0, 1, 0, -1};
int dy[4] = {1, 0, -1, 0};

struct Point {
    int x, y;
};
```

```cpp
int bfs(int sx, int sy, int ex, int ey, int n, int m) {
    queue<Point> q;
    q.push({sx, sy});
    vis[sx][sy] = true;
    dist[sx][sy] = 0;

    while (!q.empty()) {
        Point p = q.front();
        q.pop();

        if (p.x == ex && p.y == ey) {
            return dist[p.x][p.y];
        }

        for (int i = 0; i < 4; i++) {
            int nx = p.x + dx[i];
            int ny = p.y + dy[i];

            if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
                !vis[nx][ny] && maze[nx][ny] != '#') {
                vis[nx][ny] = true;
                dist[nx][ny] = dist[p.x][p.y] + 1;
                q.push({nx, ny});
            }
        }
    }
    return -1; // 不可达
}
```

7. 图论算法

深度优先遍历

应用场景：连通分量、环检测、拓扑排序

```cpp
vector<int> graph[N]; // 邻接表
bool visited[N];

void dfs_traverse(int u) {
    visited[u] = true;
    cout << u << " ";

    for (int v : graph[u]) {
        if (!visited[v]) {
            dfs_traverse(v);
        }
    }
}

// 遍历整个图
void dfs_graph(int n) {
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) {
            dfs_traverse(i);
        }
    }
}
```

广度优先遍历

应用场景： 最短路径、层次遍历

```
void bfs_traverse(int start, int n) {
    bool visited[N] = {false};
    queue<int> q;

    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cout << u << " ";

        for (int v : graph[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

泛洪算法 (Flood Fill)

应用场景：连通区域标记、图像处理

```cpp
int grid[N][N];
bool visited[N][N];
int dx[4] = {0, 1, 0, -1};
int dy[4] = {1, 0, -1, 0};

// DFS 实现 Flood Fill
void flood_fill_dfs(int x, int y, int color, int n, int m) {
    if (x < 0 || x >= n || y < 0 || y >= m) return;
    if (visited[x][y] || grid[x][y] != color) return;

    visited[x][y] = true;
    // 处理当前单元格

    for (int i = 0; i < 4; i++) {
        flood_fill_dfs(x + dx[i], y + dy[i], color, n, m);
    }
}
```

```cpp
// BFS 实现 Flood Fill
void flood_fill_bfs(int sx, int sy, int color, int n, int m) {
    queue<pair<int, int>> q;
    q.push({sx, sy});
    visited[sx][sy] = true;

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
                !visited[nx][ny] && grid[nx][ny] == color) {
                visited[nx][ny] = true;
                q.push({nx, ny});
            }
        }
    }
}
```

8. 动态规划

动态规划基本思路

核心思想：将复杂问题分解为子问题，避免重复计算

三要素

1. 最优子结构
2. 重叠子问题
3. 状态转移方程

解题步骤

1. 定义状态
2. 确定状态转移方程
3. 确定边界条件
4. 计算顺序

简单一维动态规划

斐波那契数列

```
int dp[N]; // dp[i] 表示第 i 个斐波那契数

int fibonacci(int n) {
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}
```

爬楼梯问题

```
// 每次可以爬 1 或 2 个台阶，求到第 n 阶的方法数
int climbStairs(int n) {
    if (n <= 2) return n;

    int dp[N];
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}
```

线性 DP：最长上升子序列 (LIS)

问题：求序列中最长的严格递增子序列长度

状态定义：$dp[i]$ 表示以 $a[i]$ 结尾的最长上升子序列长度

状态转移：

$$dp[i] = \max_{j<i \text{ 且 } a[j]<a[i]} \{dp[j]\} + 1$$

```
int LIS(int a[], int n) {
    int dp[N], ans = 0;

    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

时间复杂度：$O(n^2)$

线性 DP：最长公共子序列 (LCS)

问题：求两个序列的最长公共子序列长度

状态定义：$dp[i][j]$ 表示 $a[0..i-1]$ 和 $b[0..j-1]$ 的 LCS 长度

状态转移：

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & \text{if } a[i-1]=b[j-1] \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise} \end{cases}$$

```
int LCS(char a[], char b[], int n, int m) {
    int dp[N][N] = {0};

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i-1] == b[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[n][m];
}
```

时间复杂度： $O(n \times m)$

线性 DP：最长公共子串

问题：求两个序列的最长公共连续子串长度

状态定义：$dp[i][j]$ 表示以 $a[i-1]$ 和 $b[j-1]$ 结尾的最长公共子串长度

状态转移：

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & \text{if } a[i-1]=b[j-1] \\ 0, & \text{otherwise} \end{cases}$$

```
int LCSubstring(char a[], char b[], int n, int m) {
    int dp[N][N] = {0};
    int ans = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i-1] == b[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
                ans = max(ans, dp[i][j]);
            } else {
                dp[i][j] = 0;
            }
        }
    }
    return ans;
}
```

时间复杂度： $O(n \times m)$

线性 DP：编辑距离

问题：将字符串 a 转换为字符串 b 所需的最少操作次数

操作：插入、删除、替换

状态定义：$dp[i][j]$ 表示 $a[0..i-1]$ 转换为 $b[0..j-1]$ 的编辑距离

状态转移：

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1, & \text{删除} \\ dp[i][j-1] + 1, & \text{插入} \\ dp[i-1][j-1] + (a[i-1] \neq b[j-1]), & \text{替换} \end{cases}$$

```
int editDistance(char a[], char b[], int n, int m) {
    int dp[N][N];

    // 初始化
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i-1] == b[j-1]) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = min(min(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1]) + 1;
            }
        }
    }
    return dp[n][m];
}
```

时间复杂度： $O(n \times m)$

简单背包类型动态规划

0-1 背包问题

```
int weight[N];  // 物品重量
int value[N];   // 物品价值
int dp[N][M];   // dp[i][j] 前 i 个物品，容量为 j 的最大价值

int knapsack(int n, int capacity) {
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= capacity; j++) {
            if (j < weight[i]) {
                dp[i][j] = dp[i-1][j];
            } else {
                dp[i][j] = max(dp[i-1][j],
                               dp[i-1][j - weight[i]] + value[i]);
            }
        }
    }
    return dp[n][capacity];
}
```

0-1 背包空间优化

```
int dp[M]; // 一维数组优化

int knapsack_optimized(int n, int capacity) {
    for (int i = 1; i <= n; i++) {
        for (int j = capacity; j >= weight[i]; j--) {
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

完全背包问题

```
// 每个物品可以选无限次
int complete_knapsack(int n, int capacity) {
    for (int i = 1; i <= n; i++) {
        for (int j = weight[i]; j <= capacity; j++) {
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

简单区间类型动态规划

区间 DP 模板

```
int dp[N][N]; // dp[i][j] 表示区间 [i,j] 的最优值

void interval_dp(int n) {
    // 初始化长度为 1 的区间
    for (int i = 1; i <= n; i++)
        dp[i][i] = initial_value;

    // 按区间长度递增
    for (int len = 2; len <= n; len++) {
        for (int i = 1; i + len - 1 <= n; i++) { // 枚举左端点
            int j = i + len - 1; // 计算右端点

            // 枚举分割点
            for (int k = i; k < j; k++)
                dp[i][j] = max(dp[i][j], dp[i][k] + dp[k+1][j] + cost);
        }
    }
}
```

矩阵连乘问题

```
int matrix_chain(int p[], int n) {
    int dp[N][N];
    // 初始化
    for (int i = 1; i <= n; i++)
        dp[i][i] = 0;

    for (int len = 2; len <= n; len++) {
        for (int i = 1; i <= n - len + 1; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;

            for (int k = i; k < j; k++) {
                int cost = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j];
                dp[i][j] = min(dp[i][j], cost);
            }
        }
    }

    return dp[1][n];
}
```

算法复杂度总结

算法类型	时间复杂度	空间复杂度	适用场景
DFS	$O(b^d)$	$O(d)$	所有解、连通性
BFS	$O(V + E)$	$O(V)$	最短路径、层次遍历
Flood Fill	$O(n \times m)$	$O(n \times m)$	连通区域
一维 DP	$O(n)$	$O(n)$	线性序列问题
背包 DP	$O(n \times W)$	$O(W)$	组合优化
区间 DP	$O(n^3)$	$O(n^2)$	区间最值

复习要点

1. 掌握 DFS 和 BFS 的适用场景
2. 熟练编写 Flood Fill 算法
3. 理解动态规划的状态定义
4. 掌握背包问题的变体
5. 注意边界条件的处理

掌握搜索与动态规划，解决复杂问题！