Lec 12&13 Recursion, recursive sorting and Recursion with Backtracking

这份课件主要涵盖了**递归（Recursion）**、**递归排序（Recursive Sorting）**和**带回溯的递归（Recursion with Backtracking）**三个主题，并通过具体的例子（如阶乘函数、快速排序和N皇后问题）进行了深入说明。

1. 递归的定义和基本概念

核心概念

- **递归函数**：递归是一种通过函数自身调用来解决问题的方法。它将一个大问题分解为更小的子问题，通过解决子问题逐步构建出最终的答案。
- **与数学中的递推函数类似**：例如，阶乘函数 $n!$ 可以定义为：

$$n! = n \cdot (n-1)! \tag{1}$$

  并设定基本情况 $0! = 1$。
- **与迭代的比较**：递归和迭代（循环）都可以解决重复性任务，但递归通常更简洁、直观，尤其在处理分治问题时。
- **基本要求**：递归函数必须包含至少一个**基本情况（Base Case）**，即在特定条件下停止递归调用，直接返回结果，以避免无限递归。

示例：阶乘函数

- **C语言实现**：

```c
int Factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

2. 阶乘函数的执行过程

递归调用栈

- 当调用 `Factorial(3)` 时，函数会依次调用 `Factorial(2)`、`Factorial(1)`、`Factorial(0)`。
- 每次递归调用都在**运行时栈（Run-time Stack）**上创建一个新的栈帧（Stack Frame），用于存储局部变量（如 `n`）、参数和返回地址。
- 到达基本情况 `n == 0` 时，返回 `1`，然后逐层回溯，计算最终结果。

3. 运行时栈的管理

运行时栈的结构

- **全局数据区**：存储全局变量。
- **运行时栈**：存储局部变量、函数参数和返回地址。
- 关键寄存器：
  - **R4（全局指针）**：指向第一个全局变量。
  - **R5（帧指针）**：指向当前函数的栈帧。
  - **R6（栈指针）**：指向运行时栈的顶部。

栈帧管理

- 每次递归调用时，分配新的栈帧，保存当前函数的状态（如参数 n 和返回地址）。
- 函数返回时，栈帧被弹出，恢复调用者的状态。

4. 阶乘函数的LC-3实现

LC-3汇编代码

- 主程序：
  - 初始化栈指针 `R6`。
  - 将参数 `n = 3` 压入栈。
  - 调用 `FACTORIAL` 子程序。
  - 从栈中弹出返回值并存储。
- `FACTORIAL` 子程序：
  - 保存调用者的返回地址（R7）和帧指针（R5）。
  - 检查 `n` 是否为 0：
    - 如果是，返回 `1`。
    - 如果不是，递归调用 `FACTORIAL(n-1)`，然后将结果与 n 相乘。

关键操作

- **压栈和出栈**：管理参数和局部变量。
- **JSR 和 RET**：实现函数调用和返回。
- **帧指针 R5**：定位当前栈帧中的变量和参数。

5. 递归排序算法

整体思路：快速排序

快速排序的核心思想是通过**分治法（Divide and Conquer）**将数组排序：

1. **选择基准值（Pivot）**：从数组中挑选一个元素作为基准（这里选择最后一个元素 arr[high]）。
2. **划分（Partition）**：将数组重新排列，使得基准值左边的元素都小于等于它，右边的元素都大于它。

3. **递归处理子数组**：对基准值左右两侧的子数组递归执行相同的操作，直到子数组长度为1或0（有序）。

- 关键函数：
  - **partition**：选择 pivot 并重新排列数组。

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

  - 参数：
    - `arr[]`：待排序的数组。
    - `low`：当前子数组的起始索引。
    - `high`：当前子数组的结束索引。
  - **返回值**：基准值最终所在位置的索引。

工作原理

`partition` 函数的任务是将数组划分为两部分：小于等于基准值的部分和大于基准值的部分，并将基准值放到正确的位置。

1. **初始化**：
   - `int pivot = arr[high]`：选择子数组的最后一个元素作为基准值。
   - `int i = low - 1`：`i` 表示小于等于基准值的区域的右边界，初始时为空（因此设为 `low-1`）
2. **划分过程**：
   - `for (int j = low; j < high; j++)`：从 `low` 到 `high - 1` 遍历子数组（不包括 high，因为它是基准值）
   - `if (arr[j] <= pivot)`：如果当前元素 `arr[j]`小于等于基准值：
     - `i++`：扩展小于等于基准值的区域。
     - `swap(arr[i], arr[j])`：将 arr[j] 交换到小于等于基准值的区域中。

   这个过程的目的是将所有小于等于 `pivot` 的元素移动到数组的左侧，而大于 `pivot` 的元素自然被推到右侧。
3. **放置基准值**：
   - 循环结束后，`i` 是小于等于基准值的最后一个元素的索引。
   - `swap(arr[i + 1], arr[high])`：将基准值 `arr[high]` 交换到 `i + 1` 位置，此时：

        - `i + 1` 左边的元素都小于等于 `pivot`。
        - `i + 1` 右边的元素都大于 `pivot`。
    - `return i + 1`：返回基准值的新位置。

示例

假设 `arr = [4, 3, 7, 1, 5], low = 0, high = 4`：

1. `pivot = arr[4] = 5`
2. `i = -1`，`j` 从 0 到 3：
    - `j = 0`：`arr[0] = 4 <= 5`，`i = 0`，`swap(arr[0], arr[0])`，数组仍为 `[4, 3, 7, 1, 5]`。
    - `j = 1`：`arr[1] = 3 <= 5`，`i = 1`，`swap(arr[1], arr[1])`，数组仍为 `[4, 3, 7, 1, 5]`。
    - `j = 2`：`arr[2] = 7 > 5`，无交换。
    - `j = 3`：`arr[3] = 1 <= 5`，`i = 2`，`swap(arr[2], arr[3])`，数组变为 `[4, 3, 1, 7, 5]`。
3. `swap(arr[2 + 1], arr[4])`：交换 `arr[3]` 和 `arr[4]`，数组变为 `[4, 3, 1, 5, 7]`。
4. 返回 `i + 1 = 3`，即 `pivot` 现在在索引 `3`。

结果：`[4, 3, 1, 5, 7]`，`5` 左边的元素 `[4, 3, 1]` 都小于等于 `5`，右边的 `[7]` 大于 `5`。

- **quicksort**：递归调用 partition 和自身。

```
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
```

- 参数同上

工作原理

`quicksort` 函数通过递归调用实现整个数组的排序。

1. 基本情况：
    - `if (low < high)`：如果子数组的长度大于 1，需要排序；否则（长度为 0 或 1），已有序，直接返回。
2. 递归过程：
    - `int pi = partition(arr, low, high)`：调用 `partition` 函数，将子数组划分为两部分，并返回基准值的位置 `pi`。
    - `quicksort(arr, low, pi - 1)`：对基准值左边的子数组递归排序。
    - `quicksort(arr, pi + 1, high)`：对基准值右边的子数组递归排序。

时间复杂度

- **平均情况**：$O(n \log n)$，每次划分接近均匀时。
- **最坏情况**：$O(n^2)$，当数组已排序或逆序，且每次选择的 pivot 是最大或最小值。
- **空间复杂度**：$O(\log n)$，递归调用栈的深度。

6. 带回溯的递归（PAGE38 至 PAGE45）

N皇后问题

- **问题描述**：在一个 N×N 的棋盘上放置 N 个皇后，使得它们互不攻击（不在同一行、同一列或同一对角线上）。
- **回溯法**：通过尝试放置皇后并在遇到冲突时撤销选择（回溯）来寻找解。

关键函数

- `Solve(board, row)`：尝试在第 row 行放置皇后。
  - 如果 `row >= N`，表示所有皇后已放置，返回 `true`。
  - 否则，遍历每一列：
    - 如果在 `(row, col)` 放置皇后是安全的（通过 `isSafe` 检查），则放置皇后并递归调用 `Solve(board, row + 1)`。
    - 如果递归返回 `true`，则找到解；否则，撤销放置（回溯），尝试下一列。
- **isSafe(board, row, col)**：检查在 `(row, col)` 放置皇后是否安全，需检查列和对角线。
- **printSolution(board)**：打印棋盘上的皇后位置。

回溯过程

1. **决策**：尝试在当前行的某列放置皇后。
2. **递归**：如果放置成功，递归到下一行。
3. **回溯**：如果某行无法放置皇后，回退到上一行，尝试其他列。
4. **基本情况**：所有行都放置了皇后，返回 `true`。

主函数

```
#include <stdio.h>
#define N 4
#define true 1
#define false 0

int main() {
    int board[N][N] = {0};  // 初始化棋盘为全0
    if (Solve(board, 0)) {
        printSolution(board);  // 找到解后打印
    } else {
        printf("No solution exists\n");
```

```
    }
    return 0;
}
```

Solve函数

```
int Solve(int board[N][N], int row) {
    if (row >= N)  // 基本情况：所有皇后已放置
        return true;
    for (int j = 0; j < N; j++) {
        if (isSafe(board, row, j)) {
            board[row][j] = 1;  // 放置皇后
            if (Solve(board, row + 1))  // 递归下一行
                return true;
            board[row][j] = 0;  // 回溯，撤销放置
        }
    }
    return false;  // 无解
}
```

isSafe函数

```
int isSafe(int board[N][N], int row, int col) {
    int i;
    // 检查列
    for (i = 0; i < row; i++)
        if (board[i][col])
            return false;
    // 检查左上对角线
    for (i = 0; i < row && col - (row - i) >= 0; i++)
        if (board[i][col - (row - i)])
            return false;
    // 检查右上对角线
    for (i = 0; i < row && col + (row - i) < N; i++)
        if (board[i][col + (row - i)])
            return false;
    return true;
}
```

printSolution函数

```
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf(" %d ", board[i][j]);
        }
        printf("\n");
    }
}
```