算法分析与设计 2024

浙江工业大学 2025.3.20

教师: 杜嘉欣 实验 2

实验 2

提交截止时间 2025.3.27。本次习题主要涉及分治算法。请用 LaTeX 编辑所有解答。所有问题请给出简洁的回答，任何冗余的回答可能会得低分。

姓名: xxx

学号: xxxxxxxx

题目 2-1. 高效求解幂函数

给定两个整数 x 和 n，其中 n 是非负整数，按照要求计算 x 的 n 次方，也就是 $pow(x, n)$。

(a) 给出一个简单遍历的算法求解该问题，要求时间复杂度为 $O(n)$。

解答：使用迭代方法计算 x^n，通过循环 n 次，每次将结果乘以 x。

```
double power(double x, int n) {
    double result = 1;
    for(int i = 1; i <= n; i++) {
        result = result * x;
    }
    return result;
}
```

时间复杂度为 $O(n)$，因为需要循环 n 次。

(b) 给出分治算法求解该问题的步骤，要求时间复杂度为 $O(\log n)$。

解答：使用分治策略快速计算幂函数：

1. 如果 $n = 0$，则返回 1

2. 递归计算 $y = x^{\lfloor n/2 \rfloor}$
3. 如果 n 是偶数，返回 y^2
4. 如果 n 是奇数，返回 $y^2 \times x$

该分治算法利用了指数运算的性质：$x^n = x^{n/2} \times x^{n/2}$（当 n 为偶数）或 $x^n = x^{n/2} \times x^{n/2} \times x$（当 n 为奇数）。

这样每次递归调用将问题规模缩小一半，因此时间复杂度为 $O(\log n)$。

(c) 给出分治算法求解的代码。

解答：以下是分治算法的 C++ 实现：

```cpp
double power(double x, int n) {
    // 基本情况
    if (n == 0) return 1;

    // 处理负指数
    if (n < 0) {
        return 1.0 / power(x, -n);
    }

    // 分治部分：计算 x^(n/2)
    double half = power(x, n / 2);

    // 根据 n 的奇偶性合并结果
    if (n % 2 == 0) {
        return half * half;
    } else {
        return half * half * x;
    }
}
```

题目 2-2. 字符串最长公共前缀

给定 n 个字符串，返回这些字符串最长的公共前缀。比如输入字符串序列是 technique, technician, technology, technical，那么应该返回 techni。

(a) 如果输入字符串序列中，最长的一个字符串长度为 m，描述一个时间复杂度为 $O(mn)$ 的算法求解该问题。

解答：一个简单的 $O(mn)$ 的算法是：

1. 取字符串数组中的第一个字符串作为初始的最长公共前缀（LCP）
2. 依次遍历剩余的 $n-1$ 个字符串，对于每个字符串：
   (a) 从头开始逐字符比较当前字符串与 LCP
   (b) 如果发现不匹配，就将 LCP 截断到当前位置
   (c) 如果当前字符串比 LCP 短，也要将 LCP 截断到当前字符串的长度

这个算法需要遍历 n 个字符串，对于每个字符串最多需要比较 m 个字符（即最长字符串的长度），因此总时间复杂度为 $O(mn)$。

(b) 描述一个时间复杂度为 $O(m \log n)$ 的算法求解该问题。

解答：使用分治法可以降低时间复杂度：

1. 将 n 个字符串分成两部分，每部分约有 $n/2$ 个字符串
2. 递归地计算左半部分的最长公共前缀 LCP_left
3. 递归地计算右半部分的最长公共前缀 LCP_right
4. 合并两个结果，即找到 LCP_left 和 LCP_right 的最长公共前缀

递归的基本情况是当只有一个字符串时，LCP 就是该字符串本身。

在每一层递归中，合并操作的时间复杂度为 $O(m)$，其中 m 是最长字符串的长度。递归树的高度为 $\log n$，因此总的时间复杂度为 $O(m \log n)$。

(c) 给出以上时间复杂度为 $O(mn)$ 算法的实现。

解答：以下是 $O(mn)$ 算法的 C++ 实现：

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;
```

```
5
6  string longestCommonPrefix(vector<string>& strs) {
7      // 空数组的情况
8      if (strs.empty()) return "";
9
10     // 以第一个字符串作为初始 LCP
11     string prefix = strs[0];
12
13     // 逐个比较剩余的字符串
14     for (int i = 1; i < strs.size(); i++) {
15         // 当 prefix 变为空或遍历完所有字符串时结束
16         if (prefix.empty()) break;
17
18         int j = 0;
19         // 逐字符比较并更新 prefix
20         while (j < prefix.length() && j < strs[i].length() &&
21                prefix[j] == strs[i][j]) {
22             j++;
23         }
24
25         // 截断 prefix 到匹配的长度
26         prefix = prefix.substr(0, j);
27     }
28
29     return prefix;
30 }
31
32 // 测试代码
33 int main() {
34     vector<string> example = {"technique", "technician",
35                               "technology", "technical"};
36     cout << "最长公共前缀: " << longestCommonPrefix(example) << endl;
37     return 0;
38 }
```

题目 2-3. 有序序列中缺失的最小元素

给定一个序列，其中的元素都是有序的非负整数，要求找出其中缺失的最小的那个元素。

输入:

0, 1, 2, 6, 9, 11, 15

输出:

3

输入:

1, 2, 3, 4, 6, 9, 11, 15

输出:

0

输入:

0, 1, 2, 3, 4, 5, 6

输出:

7

(a) 给出一个时间复杂度为 $O(n)$ 的算法求解该问题。

解答：线性扫描算法可以在 $O(n)$ 时间内解决此问题：

1. 初始化期望的下一个元素 expected = 0
2. 遍历序列中的每个元素 nums[i]
3. 如果 nums[i] 不等于 expected，则 expected 就是缺失的最小非负整数
4. 否则，更新 expected = nums[i] + 1
5. 如果遍历完整个序列还没有返回结果，则返回 expected

该算法的时间复杂度为 $O(n)$，因为只需要遍历一次数组。

(b) 给出一个时间复杂度为 $O(\log n)$ 的算法求解该问题。

解答：由于序列是有序的，我们可以使用二分查找的思想来设计 $O(\log n)$ 的算法：

1. 如果数组第一个元素不是 0，则缺失的最小元素是 0
2. 否则，使用二分查找：
   (a) 初始化左边界 left = 0，右边界 right = 数组长度 - 1
   (b) 当 left < right 时循环：
      i. 计算中间位置 mid = left + (right - left) / 2
      ii. 如果 nums[mid] == mid（即到目前为止没有缺失的元素），则缺失的最小元素在右半部分，left = mid + 1
      iii. 否则，缺失的最小元素在左半部分，right = mid
   (c) 当循环结束时，返回 left 作为最小缺失元素的位置

该算法利用了有序序列的特性，如果没有缺失元素，则序列中的每个元素等于其索引值。时间复杂度为 $O(\log n)$。

(c) 给出时间复杂度为 $O(\log n)$ 的算法实现代码。

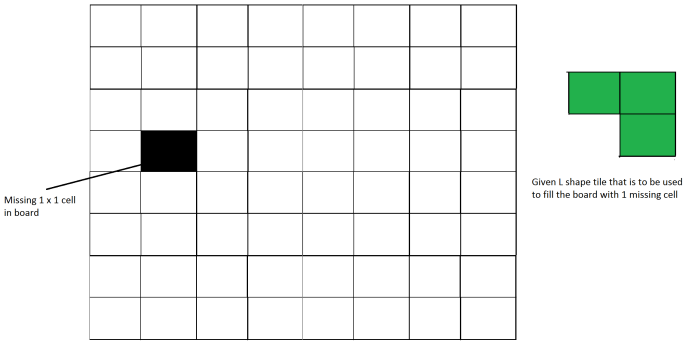
解答：二分查找算法的 C++ 实现：

```cpp
#include <iostream>
#include <vector>
using namespace std;

int findMissingMinimum(vector<int>& nums) {
    // 检查序列是否为空
    if (nums.empty()) return 0;

    // 检查第一个元素是否为0
    if (nums[0] != 0) return 0;

    // 检查是否只有一个元素
    if (nums.size() == 1) return 1;

    int left = 0;
    int right = nums.size() - 1;
```

```
17
18      // 如果最后一个元素等于其索引，说明缺失的是下一个数
19      if (nums[right] == right) return right + 1;
20
21      // 二分查找
22      while (left < right) {
23          int mid = left + (right - left) / 2;
24
25          // 如果中间元素等于其索引，说明缺失元素在右侧
26          if (nums[mid] == mid) {
27              left = mid + 1;
28          } else {
29              // 否则缺失元素在左侧
30              right = mid;
31          }
32      }
33
34      return left;
35  }
36
37  // 测试代码
38  int main() {
39      vector<int> test1 = {0, 1, 2, 6, 9, 11, 15};
40      vector<int> test2 = {1, 2, 3, 4, 6, 9, 11, 15};
41      vector<int> test3 = {0, 1, 2, 3, 4, 5, 6};
42
43      cout << "Test 1: " << findMissingMinimum(test1) << endl; // 应输出 3
44      cout << "Test 2: " << findMissingMinimum(test2) << endl; // 应输出 0
45      cout << "Test 3: " << findMissingMinimum(test3) << endl; // 应输出 7
46
47      return 0;
48  }
```

题目 2-4. 铺砖问题

地板长度 n 都是 2 的正整数幂，铺的都是唯一的同一种砖 (如图所示的 L 字形砖)，但地板上会有一水泥点已经被覆盖，无需再用铺砖 (图中黑的砖块，这个砖块的位置是随机的)。如果 $n = 2$，则存在 4 个方格，其中，除一个方格外，其余 3 个方格可被一 L 型条块覆盖；当 $n = 4$ 时，则存在 16 个方格，其中，除一个方格外，其余 15 个方格被 5 个 L 型条块覆盖。输入一个正整数 n，表示棋盘的大小是 $n \times n$ 的。输出一个被 L 型条块覆盖的 $n \times n$ 棋盘。该棋盘除一个方格外，其余各方格都被 L 型条块覆盖住。给出问题的分治求解过程、时间复杂度分析和实现代码。



解答:

分治求解过程:

1. **基本情况**：当 $n = 2$ 时，我们有一个 2×2 的棋盘，其中一个方格已被覆盖，剩余 3 个方格可以恰好用一个 L 型砖覆盖。
2. **分解问题**：对于 $n > 2$ 的情况，我们将 $n \times n$ 的棋盘分成四个 $n/2 \times n/2$ 的子棋盘。
3. **确定特殊方格**：在这四个子棋盘中，有一个子棋盘包含原始的特殊方格（已覆盖的水泥点）。对于其余三个子棋盘，我们需要人为创建一个特殊方格，使得每个子棋盘都有一个特殊方格。
4. **放置中心 L 型砖**：在棋盘的中心放置一个 L 型砖，使其覆盖四个子棋盘中相邻的三个方格，这三个方格就是我们在步骤 3 中创建的特殊方格。
5. **递归求解**：递归地解决四个 $n/2 \times n/2$ 子棋盘的铺砖问题，每个子棋盘都有一个特殊方格（一个是原始的，三个是我们创建的）。

时间复杂度分析：

设 $T(n)$ 表示解决大小为 $n \times n$ 的棋盘问题所需的时间。根据分治算法，我们有：$T(n) = 4 \times T(n/2) + O(1)$

其中 $O(1)$ 是放置中心 L 型砖所需的常数时间。

按照主定理，这个递归式的解是 $T(n) = O(n^2)$，因为总共有 n^2 个方格，我们需要放置 $(n^2 - 1)/3$ 个 L 型砖（每个 L 型砖覆盖 3 个方格）。

代码实现：

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

// 全局变量，存储棋盘和砖块的编号
vector<vector<int>> board;
int tileNumber = 1;

// 放置L型砖的函数
void placeTile(int row, int col, int specialRow, int specialCol, int size) {
    // 如果只剩下一个2x2的棋盘，直接放置L型砖
    if (size == 2) {
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                if (row + i != specialRow || col + j != specialCol) {
                    board[row + i][col + j] = tileNumber;
                }
            }
        }
        tileNumber++;
        return;
    }

    int halfSize = size / 2;
    int centerRow = row + halfSize - 1;
    int centerCol = col + halfSize - 1;

```

```
29      // 确定特殊方格在哪个象限
30      int quadrant;
31      if (specialRow <= centerRow && specialCol <= centerCol) {
32          quadrant = 0; // 左上角
33      } else if (specialRow <= centerRow && specialCol > centerCol) {
34          quadrant = 1; // 右上角
35      } else if (specialRow > centerRow && specialCol <= centerCol) {
36          quadrant = 2; // 左下角
37      } else {
38          quadrant = 3; // 右下角
39      }
40
41      // 放置中心L型砖
42      int currentTile = tileNumber++;
43
44      // 右上角
45      if (quadrant != 1) board[centerRow][centerCol + 1] = currentTile;
46      // 左下角
47      if (quadrant != 2) board[centerRow + 1][centerCol] = currentTile;
48      // 右下角
49      if (quadrant != 3) board[centerRow + 1][centerCol + 1] = currentTile;
50      // 左上角
51      if (quadrant != 0) board[centerRow][centerCol] = currentTile;
52
53      // 递归处理四个子棋盘
54
55      // 左上角子棋盘
56      int nextSpecialRow = (quadrant == 0) ? specialRow : centerRow;
57      int nextSpecialCol = (quadrant == 0) ? specialCol : centerCol;
58      placeTile(row, col, nextSpecialRow, nextSpecialCol, halfSize);
59
60      // 右上角子棋盘
61      nextSpecialRow = (quadrant == 1) ? specialRow : centerRow;
62      nextSpecialCol = (quadrant == 1) ? specialCol : centerCol + 1;
63      placeTile(row, col + halfSize, nextSpecialRow, nextSpecialCol, halfSize);
64
65      // 左下角子棋盘
66      nextSpecialRow = (quadrant == 2) ? specialRow : centerRow + 1;
```

```
67      nextSpecialCol = (quadrant == 2) ? specialCol : centerCol;
68      placeTile(row + halfSize, col, nextSpecialRow, nextSpecialCol, halfSize);
69
70      // 右下角子棋盘
71      nextSpecialRow = (quadrant == 3) ? specialRow : centerRow + 1;
72      nextSpecialCol = (quadrant == 3) ? specialCol : centerCol + 1;
73      placeTile(row + halfSize, col + halfSize,
74                nextSpecialRow, nextSpecialCol, halfSize);
75  }
76
77  // 打印棋盘的函数
78  void printBoard(int n) {
79      for (int i = 0; i < n; i++) {
80          for (int j = 0; j < n; j++) {
81              if (board[i][j] == -1) {
82                  cout << "S\t"; // 特殊方格
83              } else {
84                  cout << board[i][j] << "\t";
85              }
86          }
87          cout << endl;
88      }
89  }
90
91  int main() {
92      int n;
93      cout << "输入棋盘大小(必须是2的幂): ";
94      cin >> n;
95
96      // 初始化棋盘
97      board.resize(n, vector<int>(n, 0));
98
99      // 设置特殊方格的位置 (这里设为左上角, 可以根据需要修改)
100     int specialRow = 0, specialCol = 0;
101     cout << "输入特殊方格的位置(行 列): ";
102     cin >> specialRow >> specialCol;
103
104     // 标记特殊方格
```

```
105     board[specialRow][specialCol] = -1;
106 
107     // 解决铺砖问题
108     placeTile(0, 0, specialRow, specialCol, n);
109 
110     // 打印结果
111     printBoard(n);
112 
113     return 0;
114 }
```

题目 2-5. 序列逆序对数

给定一个序列 A，找出序列中逆序对的个数。逆序是当 $i < j$ 和 $A[i] > A[j]$ 同时成立，那么 $A[i]$ 与 $A[j]$ 构成逆序对。比如 $A = [1, 9, 6, 4, 5]$，该序列逆序对数为 5，分别为 $[9, 6], [9, 4], [9, 5], [6, 4], [6, 5]$。

(a) 给出一个时间复杂度为 $O(n^2)$ 的算法求解该问题。

解答：一个简单的 $O(n^2)$ 算法是使用双重循环直接枚举所有可能的逆序对：

1. 初始化逆序对计数器 count = 0
2. 对于每个位置 i（从 0 到 n-2）：
   (a) 对于每个位置 j（从 i+1 到 n-1）：
      i. 如果 A[i] > A[j]，则增加计数器 count++
3. 返回最终的计数值 count

这个算法的时间复杂度是 $O(n^2)$，因为我们需要检查 $\binom{n}{2} = \frac{n(n-1)}{2}$ 个可能的对。

(b) 给出一个时间复杂度为 $O(n \log n)$ 的算法求解该问题。

解答：使用归并排序的变体可以在 $O(n \log n)$ 时间内计算逆序对：

1. 实现一个修改版的归并排序，在合并两个已排序子数组时计算逆序对
2. 在归并排序过程中，当右子数组的元素被选择时，不会产生新的逆序对
3. 当左子数组的元素被选择时，右子数组中已经处理的所有元素都与当前左子数组元素构成逆序对

具体步骤如下：

1. 将数组分成两半
2. 递归统计左半部分的逆序对数量
3. 递归统计右半部分的逆序对数量
4. 在归并两个已排序数组的过程中，统计跨越左右两半部分的逆序对数量
5. 返回总的逆序对数量

由于归并排序的时间复杂度是 $O(n \log n)$，这个算法的总时间复杂度也是 $O(n \log n)$。

(c) 给出一个时间复杂度为 $O(n \log n)$ 算法的实现代码。

解答：基于归并排序的逆序对计数算法的 C++ 实现：

```cpp
#include <iostream>
#include <vector>
using namespace std;

// 合并两个已排序的子数组并计算逆序对
long long merge(vector<int>& nums, vector<int>& temp,
                int left, int mid, int right) {
    // i指向左子数组，j指向右子数组
    int i = left;
    int j = mid + 1;
    int k = left; // 指向临时数组的位置
    long long invCount = 0;

    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            // 左子数组元素小于等于右子数组元素，不构成逆序对
            temp[k++] = nums[i++];
        } else {
            // 左子数组元素大于右子数组元素，构成逆序对
            // 当前左子数组的元素及其后面的所有元素都与当前右子数组元素构成逆序对
            temp[k++] = nums[j++];
            invCount += (mid - i + 1);
        }
    }

    // 将剩余元素复制到临时数组
    while (i <= mid) {
        temp[k++] = nums[i++];
    }

    while (j <= right) {
        temp[k++] = nums[j++];
    }

    // 将临时数组中的元素复制回原数组
    for (i = left; i <= right; i++) {
        nums[i] = temp[i];
    }
```

```
39
40      return invCount;
41  }
42
43  // 归并排序并计算逆序对
44  long long mergeSort(vector<int>& nums, vector<int>& temp,
45                      int left, int right) {
46      long long invCount = 0;
47
48      if (left < right) {
49          int mid = left + (right - left) / 2;
50
51          // 计算左子数组中的逆序对
52          invCount += mergeSort(nums, temp, left, mid);
53
54          // 计算右子数组中的逆序对
55          invCount += mergeSort(nums, temp, mid + 1, right);
56
57          // 计算跨越左右子数组的逆序对
58          invCount += merge(nums, temp, left, mid, right);
59      }
60
61      return invCount;
62  }
63
64  // 计算逆序对数量的主函数
65  long long countInversions(vector<int>& nums) {
66      int n = nums.size();
67      vector<int> temp(n);
68      return mergeSort(nums, temp, 0, n - 1);
69  }
70
71  int main() {
72      vector<int> nums = {1, 9, 6, 4, 5};
73
74      cout << "序列: ";
75      for (int num : nums) {
76          cout << num << " ";
```

```
77      }
78      cout << endl;
79
80      long long inversions = countInversions(nums);
81      cout << "逆序对数量: " << inversions << endl;
82
83      return 0;
84  }
```