TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TPHCM

KHOA CÔNG NGHỆ THÔNG TIN

CSC14111 - Introduction to the Design and Analysis of Algorithms

Take-Home Assignment

Instructors:

- Lê Phúc Lữ
- Nguyễn Ngọc Thảo
- Bùi Huy Thông

Class: 21KHMT

Executing Student:

21127329 - Châu Tấn Kiệt

Table of Contents:

Q1: Consider the Insertion sort algorithm described by the following pseudo-code. The basic operation in this case is the comparison.

```
InsertionSort(a[1 .. n]) {
    for (i = 2; i ≤ n; i++) {
        v = a[i];
        j = i − 1;
        while (j ≥ 1) && (a[j] > v) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = v;
    }
}
```

For each case—worst case, best case, or average case, if there is any, provide the time complexity expression and the corresponding Big-O (or Big- or Big-) asymptotic notation.

Answer:

Worst case: $O(n^2)$ - occurs when the array is sorted in reverse order. a[j] > v is always true until the very end of the array, making the while loop run i - 1 times for each i. The number of comparisons is 1+ 2 + 3 + ... + (n -2) + (n-1) = n(n-1) / 2. Thus the time complexity in this case is $O(n^2)$

Best case: O(n) - The array is already sorted, the algorithm only needs to iterate through the array once, and a[j] > v is always false in the while loop, meaning the inner loop doesn't execute, and the algorithm just assigns a[j + 1] = v. Thus, the time complexity in this scenario is linear.

Average case: $O(n^2)$ - occurs when the elements are in random order. On average, the while loop will execute half the number of times it would in the worst case. Thus the time complexity in this case is $O(n^2)$

Q2: Analyze the complexity of Mergesort algorithm (average case is not required) without using the Master Theorem in two cases.

Answer:

a) Considering comparison as the basic operation:

Let T(n) be the number of comparisons for an input of size n. In this case, I'll be analyzing the worst case.

- Divide step: The array is divided into two halves in each recursive call. This step has no comparisons.
- Conquer step: The algorithm makes two recursive calls, each half of the input size.
- Merge step: In the worst case, we compare each element of one half with each element of the other half.

The recurrence relation for Mergesort in this case is: T(n) = T(n/2) + T(n/2) + n – 1. (n – 1) is the number of comparisons in the worst case.

$T(n) = 2T(n/2) + n - 1$
$= 2(2T(n/4) + n/2 - 1) + n – 1$
$= 4T(n/4) + 2n - 2 + n - 1$
$= 4T(n/4) + 3n - 3$
$= 4(2T(n/8) + n/4 - 1) + 3n - 3$
$= 8T(n/8) + n + 3n - 3$
$= 8T(n/8) + 4n – 3$
$\Rightarrow T(n) = 2^iT(n/2^i) + i*n - (2^i -1)$

This stops when $n/2^i = 1$ or $i = \log_2 n$

$T(n) = nT(1) + n\log_2 n - (n - 1)$

With T(1) = 0, we get: $T(n) = n\log_2 n - n + 1$

So the time complexity is O(nlogn)

b) Considering data movement as the basic operation:

Let T(n) be the number of comparisons for an input of size n. In this case, I'll be analyzing the worst case.

- Divide step: If we implement the divide step by creating new arrays, we move n elements in each recursive call.
- Conquer step: This involves recursive calls, but no data movement.

- Merge step: In worst case, we move each element once.

The recurrence relation in this case is: $T(n) = T(n/2) + T(n/2) + n + n = 2T(n/2) + 2n$. $2n$ is the number of data movements.

$T(n) = 2T(n/2) + 2n$
$= 2(2T(n/4) + n) + 2n$
$= 4T(n/4) + 4n$
$= 4(2T(n/8) + n/2) + 4n$
$= 8T(n/8) + 6n$
$\Rightarrow T(n) = 2^iT(n/2^i) + 2in$

This stops when $n/2^i = 1$ or $i = \log_2 n$

$T(n) = nT(1) + 2n\log_2 n$

With $T(1) = 0$, we get: $T(n) = n + n\log_2 n$

So the time complexity is O(nlogn)

Q3: Answer the following question about Josephus problem.

a) Define the Josephus problem and give the related formula.

The Josephus problem is a theoretical problem related to a circular arrangement of n people, numbered from 1 to n. Starting from the first person, every kth person is eliminated until only one person remains. The goal is to find the position of the survivor.

For the special case where k = 2 (every second person is eliminated), the formula for the survivor's position J(n) is:

$J(n) = 2(n - 2^{\log_2 n}) + 1$

b) Prove the following formula by induction: $J(2k + i) = 2i + 1, \forall i \in [2k - 1]$

Base case: k = 0 => i = 0

$J(2^0 + 0) = J(1) = 2 \times 0 + 1 = 1$

Assuming k >= 0:

Consider $J(2^{k+1} + i)$ where $0 \leq i < 2^{k+1}$

- If i is even, i = 2j where $0 \leq j < 2^k$:
  $J(2^{k+1} + 2j) = 2J(2^k + j) - 1 = 2(2j + 1) - 1 = 4j + 1 = 2(2j) + 1 = 2i + 1$
- If i is odd, i = 2j + 1 where $0 \leq j < 2^k$:
  $J(2^{k+1} + 2j + 1) = 2J(2^k + j) + 1 = 2(2j + 1) + 1 = 4j + 3 = 2(2j + 1) + 1 = 2i + 1$

So the formula is proven: $J(2k + i) = 2i + 1, \forall i \in [2k - 1]$

c) Prove that $J(n)$ can be obtained by a 1-bit cyclic shift left of n itself. For example, $J(6) = J(1102) = 1012 = 5$ and $J(9) = J(10012) = 112 = 3$.

Let $m = \log_2 n$ and $b_m = 1$
Let $n = (b_m \ldots b_2 b_1 b_0)_2$.

$J(n) = 2(n - 2^{\log_2 n}) + 1 = 2((b_m \ldots b_2 b_1 b_0)_2 - (1000\ldots0)_2) + 1$
$= 2(0b_m \ldots b_2 b_1 b_0)_2 + 1$

$= (b_m \ldots b_2 b_1 b_0 0)_2 + 1$

$= (b_m \ldots b_2 b_1 1)_2$

Conclusion: J(n) can be obtained by a 1-bit cyclic shift left of n itself.

Q4: Present a problem that can be solved using both the Brute-force (Exhaustive search) technique and Backtracking search technique. Write two C++ code snippets: one demonstrating the Exhaustive search method and another illustrating the Backtracking search method to solve the problem.

Answer:

In this exercise, I'm choosing the Sudoku solving problem.

Helper function: Check if the row, column and the 3x3 box is valid or not.

```
bool isValid(vector<vector<int>>& board, int row, int col, int num) {
    for (int x = 0; x < 9; x++)
        if (board[row][x] == num)
            return false;

    for (int x = 0; x < 9; x++)
        if (board[x][col] == num)
            return false;

    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (board[i + startRow][j + startCol] == num)
                return false;

    return true;
}
```

Brute-force approach:

```
bool solveSudokuBruteForce(vector<vector<int>>& board) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            if (board[row][col] == 0) {
                for (int num = 1; num <= 9; num++) {
                    if (isValid(board, row, col, num)) {
                        board[row][col] = num;
                        if (solveSudoku(board))
                            return true;
                        board[row][col] = 0;
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

Backtracking approach:

```
bool findEmptyCell(vector<vector<int>>& board, int& row, int& col) {
    for (row = 0; row < 9; ++row)
        for (col = 0; col < 9; ++col)
            if (board[row][col] == 0)
                return true;
    return false;
}

bool solveSudoku(vector<vector<int>>& board) {
    int row, col;

    if (!findEmptyCell(board, row, col))
        return true;

    for (int num = 1; num <= 9; ++num) {
        if (isValid(board, row, col, num)) {
            // Make tentative assignment
            board[row][col] = num;

            if (solveSudoku(board))
                return true;

            board[row][col] = 0;
        }
    }

    return false;
}
```

Q5: Given two sorted arrays x and y of size m and n respectively, write a C/C++ function (and auxiliary functions, if needed) to find the median of these arrays.

The overall runtime complexity should be $O(\log k)$ where $k = min\{m,n\}$.

The prototype of the function is: int findMedian(int x[], int y[], int xl, int xr, int yl, int yr);

The initial values of the last four parameters are $xl = yl = 0, xr = m - 1, yr = n - 1$.

Answer:

```
int findMedian(int x[], int y[], int xl, int xr, int yl, int yr) {
    if (xr < xl) return findMedian(y, x, yl, yr, xl, xr);

    int xmid = (xl + xr) / 2;
    int ymid = (yl + yr + 1) / 2 - (xmid - xl + 1);

    int xleft = (xmid == xl) ? INT_MIN : x[xmid - 1];
    int xright = (xmid == xr) ? INT_MAX : x[xmid];
    int yleft = (ymid == yl) ? INT_MIN : y[ymid - 1];
    int yright = (ymid == yr) ? INT_MAX : y[ymid];

    if (xleft <= yright && yleft <= xright) {
        return max(xleft, yleft);
    } else if (xleft > yright) {
        return findMedian(x, y, xl, xmid - 1, ymid, yr);
    } else {
        return findMedian(x, y, xmid + 1, xr, yl, ymid - 1);
    }
}
```

References:

1. Sudoku solving algorithms - Wikipedia
2. Josephus problem - Wikipedia
3. Mathematics of Sudoku - Wikipedia
4. Insertion Sort Algorithm - GeeksforGeeks
5. Merge sort - Wikipedia
6. Master theorem (analysis of algorithms) - Wikipedia
7. Lecture materials