CSC14111 – Introduction to Design and Analysis of Algorithms

Take-home assignment

CLC 2024

21KHMT – 21127135 – Diep Huu Phuc

Q1 (20pts) InsertionSort

"Consider the Insertion sort algorithm described by the following pseudo-code. The basic operation in this case is the comparison $a[j] > v$.

```
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, best, or average case, if there is any, provide the time complexity expression and the corresponding Big-O (or Big-θ or Big-Ω) asymptotic notation."

Best case

The array is already sorted. Every element v is greater than, or equal to, all $a[j]$, and the condition $a[j] > v$ will fail instantly, resulting in only 1 comparison for each i loop. The outer loop happens $n - 1$ times since i goes from 2 to n so,

$$T_{best}(n) = n - 1$$

Time complexity in the best case is $\Omega(n)$. Furthermore, with the already sorted array and a linear number of operations being performed, we can even confidently state it as $\theta(n)$.

Worst case

The array is sorted in reverse order. Every element v has to be compared with all previously sorted elements $a[j]$, since v will always be smaller than every $a[j]$. Therefore,

- 1st iteration ($i = 2$): 1 comparison $a[j] > v$, although no assignment (shift).
- 2nd iteration ($i = 3$): 2 comparisons.
- …
- $n - 1^{th}$ iteration ($i = n$): $n - 1$ comparisons. Since we start at $i = 2$, the for loop iterates $n - 1$ times.

$$T_{worst}(n) = 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2}$$

Time complexity in the worst case is $O(n^2)$, or to be more precise, $\theta(n^2)$.

Average case

The array has no particular order. On average, each element is expected to be less than half the elements before it, which requires a shift by about that half. Thus, for each v, i.e., $a[i]$, the average number of comparisons is $\frac{i}{2}$.

$$T_{avg}(n) = \sum_{i=2}^{n} \frac{i}{2} = \frac{1}{2} \times \frac{(n-1)n}{2} = \frac{(n-1)n}{4}$$

Despite the average number of comparisons being roughly half of the worst case's, time complexity in the average case is still $O(n^2)$, or $\theta(n^2)$.

Q2 (20pts) Mergesort

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

a. Consider the comparison to be the algorithm's basic operation.
b. Consider the data movement to be the algorithm's basic operation."

Let us use a code snippet provided during the lecture.

```
merge(a[1 .. n], low, mid, high) {
    i = low;   j = mid + 1;
    k = low;
    while (i ≤ mid) && (j ≤ high)
        if (a[i] ≤ a[j])      buf[k++] = a[i++];
        else                  buf[k++] = a[j++];
    if (i > mid)  buf[k .. high] = a[j .. high];
    else          buf[k .. high] = a[i .. mid];
    a[low .. high] = buf[low .. high];
}
```

```
mergeSort(a[1 .. n], low, high) {
    if (low < high) {
        mid = ⌊ (low + high) / 2⌋;
        mergeSort(a, low, mid);
        mergeSort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}
mergeSort(a, 1, n);
```

In case **a**, the comparison step occurs primarily during the merge phase when elements from the two halves are compared, it should be $a[i] \leq a[j]$.

As for **b**, data movement can be inferred as the act of copying elements during merging, from subarrays to the new array – buf[] in this case.

a. Comparison is the basic operation

Best case

$$T(n) = \begin{cases} 0, & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \frac{n}{2}, & n > 1 \end{cases}$$

The $2T\left(\frac{n}{2}\right)$ comes from the recursive division of the array into two equal halves of size $\frac{n}{2}$. And, the $\frac{n}{2}$ term represents the total number of comparisons made when merging, in the best case, we only need to compare $\frac{n}{2}$ times to combine two subarrays into a sorted array of size n.

For simplicity, let us assume $n = 2^k$, so the recurrence relation becomes,

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{2} \\ &= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{4}\right] + \frac{n}{2} = 4T\left(\frac{n}{4}\right) + \frac{n}{2} + \frac{n}{2} \\ &= 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{8}\right] + \frac{n}{2} + \frac{n}{2} = 8T\left(\frac{n}{8}\right) + 3 \times \frac{n}{2} \end{aligned}$$

$$
\begin{aligned}
&= \cdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + k \times \frac{n}{2} \\
&= n \times T(1) + \log(n) \times \frac{n}{2}, with\ n = 2^k \rightarrow k = \log(n)
\end{aligned}
$$

Thus, the time complexity for the best case is $\theta(n \times \log(n))$.

Worst case

$$
T(n) = \begin{cases} 0, & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n - 1, & n > 1 \end{cases}
$$

Similar to the **Best case**, but here we may have to compare every element across the two halves before completing the merge, e.g., the last element might only get placed after all other comparisons are made. For that, merging could require up to $n - 1$ comparisons.

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n - 1 \\
&= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2} - 1\right] + n - 1 = 4T\left(\frac{n}{4}\right) + 2n - 3 \\
&= 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right] + 2n - 3 = 8T\left(\frac{n}{8}\right) + 3n - 7 \\
&= \cdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + k \times n - c, with\ c \in \mathbb{N} \\
&= n \times T(1) + \log(n) \times n - c, with\ n = 2^k \rightarrow k = \log(n)
\end{aligned}
$$

Thus, the time complexity for the worst case is $\theta(n \times \log(n))$.

b. Data movement is the basic operation

$$
T(n) = \begin{cases} 0, & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n, & n > 1 \end{cases}
$$

Just like the previous two cases of **Comparison**, except during the merging process, we need to move or assign each element from the two subarrays into the final merged array of size n. In **Data movement**'s both best and worst case, there are exactly n elements that need to be moved resulting in the additional term of n.

Notice that, for the expansion, we will get the exact same result as in the **Worst case of Comparison**, but without the c constant in the end. Thus, the time complexity for this is also $\theta(n \times \log(n))$.

Q3 (20pts) Josephus

"Answer the following question about Josephus problem.

a. Define the Josephus problem and give the related formula.
b. Prove the following formula by induction: $J(2k+i) = 2i+1, \forall i \in [2k-1]$.
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 $J(9) = J(10012) = 112 = 3$."

a. Problem definition and related formula

The Josephus problem is a theoretical puzzle in which a group of people standing in circle waiting to be eliminated subsequently. For n people numbered from 1 to n foming the circle, starting from the 1st position, every k-th person is eliminated until only one remains. The goal is to determine the position of the last one standing.

With a standard Josephus problem, we set $k = 2$, and the position $J(n)$ of the last remaining person can be described recursively as,

$$J(n) = \begin{cases} 1, & n = 1 \\ [J(n-1)+1] \bmod n + 1, & n > 1 \end{cases}$$

b. Proof by induction

Assuming the provided formula is of a **standard** Josephus problem, let us take $k = 3$ and check for $i = 0$ and $i = 1$.

- $J(6) = 5$, and $J(2 \times 3 + 0) = 2 \times 0 + 1 = 1$, which is incorrect.
- $J(7) = 7$, and $J(2 \times 3 + 1) = 2 \times 1 + 1 = 3$, which is also incorrect.

Furthermore, notice that in c.'s description, all the bases are displayed incorrectly (e.g., $J(1102)$ should have been $J(110_2)$, etc.). It is fair to conclude that there is an improper formatting leading to an invalid formula. Therefore, I'm going to proceed with proving the correct standard Josephus formula, which is,

$$J(2^k + i) = 2i + 1, \forall i \in [0, 2^k - 1]$$

For the **Base case ($k = 0$)**, $n = 2^0 + i = 1,\ i \in [0, 2^0 - 1]$, meaning there is only one person, and $J(1) = 1$. This satisfies $J(2^0 + 0) = 2 \times 0 + 1 = 1$, so the base case holds.

Now for **Induction**, assume the formula holds for some k, i.e.,

$$J(2^k + i) = 2i + 1, \forall i \in [0, 2^k - 1] \qquad (\boldsymbol{b1})$$

We will prove that the formula holds for $k + 1$. In other words,

$$J(2^{k+1} + i) = 2i + 1, \forall i \in [0, 2^{k+1} - 1]$$

Consider the problem for $n = 2^{k+1} + i, i \in [0, 2^{k+1} - 1]$, it can be split into two cases based on whether n is within the group's first half (the first 2^k people) or in the second half.

1. **First half**, when $i \in [0, 2^k - 1]$, this is the same as $(\boldsymbol{b1})$ so $J(2^k + i) = 2i + 1$, and the formula still holds.
2. **Second half**, when $i \in [2^k, 2^{k+1} - 1]$, the survivor's position can be considered after eliminating all the people in the first half. That is, through symmetry, the survivors in the second half can be mapped back to the first by subtracting 2^k. Let's define this as $j = i - 2^k$, then $j \in [0, 2^k - 1]$, and $J(2^k + j) = 2j + 1$. Mapping back to the original position $i = 2^k + j$, we get,

   $$J(2^{k+1} + i) = 2 \times (2^k + j) + 1 = 2i + 1$$

   Thus, the formula also holds for the second half.

By induction, we've shown that $J(2^k + i) = 2i + 1$ holds for $k + 1$, assuming it holds for k. As the base case holds and the inductive step is valid, the formula is true for all k and $\forall i \in [0, 2^k - 1]$.

c. Proof $J(n)$ can be obtained by a 1-bit cyclic left shift of n itself.

The formula for $J(n)$, particularly when n is written as $n = 2^k + i$ where $i < 2^k$, is,

$$J(n) = J(2^k + i) = 2i + 1, \forall i \in [0, 2^k - 1]$$

n can be expressed in binary in the form of $10 \ldots 0$, with k zeros, followed by the binary representation of i. After applying a 1-bit cyclic left shift on n, the MSB, i.e., the first 1, moves to the end, the remaining bits shift left. Simply put, it is the same as $2 \times i$ (shift left) then adding 1 (move MSB to the end).

For the known formula $J(n) = 2i + 1$, notice it is the identical process as just have been described with the 1-bit cyclic left shift of n. Thus, we have successfully proved that $J(n)$ can be obtained by a 1-bit cyclic left shift of n itself.

For illustration, the 1-bit cyclic left shift of $n = 9 = 1001_2$,

- Shift left: $1001_2 \rightarrow 0010_2 = 2 \times 1$.
- Move MSB to the end: $0010_2 \rightarrow 0011_2 = 2 \times 1 + 1 = 3$.

And, using the formula $J(9) = J(2^3 + 1) = 2 \times 1 + 1 = 3$.

Q4 (20pts) Sudoku

> "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.
>
> Note: Choose a problem that is simple and clear, as a more complex problem might necessitate the use of additional techniques. Furthermore, the problem does not appear in the lecture."

Problem chosen: Sudoku Solver.

- This problem is simple enough, and does not appear in any lecture. Although it did turn up once in a test (midterm I think), it's reasonable to not count that as "lecture."

Given a partially filled 9×9 Sudoku board ($2D$ grid of $N = 9$), complete it so that each row, column, and 3×3 sub-grid contains every digit from 1 to 9 exactly once.

Although in implementation, the two might resemble each other, notice that,

- For **Exhaustive search**, we make sure every possible solution is generated without any pruning, regardless of invalid paths. That is, only when a whole grid is achieved do we check if it satisfies all Sudoku constraints. This is extremely computationally expensive so consider inputting a grid with few blank cells when testing.
- Unlike said approach, in **Backtracking**, after placing a number, we immediately check whether it leads to a valid solution. If not, the placement is undone (backtracked) and another number is tried. By pruning invalid paths early, this avoids unnecessary searches.

Brute-force (Exhaustive search)

```
bool isValidSudoku(int grid[N][N]) {
    for (int row = 0; row < N; row++) { // Check all rows and columns
        bool rowFlags[N] = {0}, colFlags[N] = {0};

        for (int col = 0; col < N; col++) {
            int numR = grid[row][col] - 1, numC = grid[col][row] - 1;

            if (numR != 0) { // Check rows
                if (rowFlags[numR]) return false;
                rowFlags[numR] = true;
            }
```

```
            if (numC != 0) { // Check columns
                if (colFlags[numC]) return false;
                colFlags[numC] = true;
            }
        }
    }

    // Check all 3x3 sub-grids
    for (int stRow = 0; stRow < N; stRow += 3)
        for (int stCol = 0; stCol < N; stCol += 3) {
            bool boxFlags[N + 1] = {0};

            for (int i = 0; i < 3; i++)
                for (int j = 0; j < 3; j++) {
                    int num = grid[stRow + i][stCol + j];
                    if (num == 0) continue;

                    if (boxFlags[num]) return false;
                    boxFlags[num] = true;
                }
        }

    return true;
}

bool solveSudokuBF(int grid[N][N], int idx) {
    // After a whole grid is generated, check if it's valid
    if (idx == N * N) return isValidSudoku(grid);

    int row = idx / N, col = idx % N;

    // If this cell is not empty, move to the next one
    if (grid[row][col] != 0) return solveSudokuBF(grid, idx + 1);

    for (int num = 1; num <= 9; num++) {
        grid[row][col] = num;
        if (solveSudokuBF(grid, idx + 1)) return true;
    }

    // If no number leads to a valid grid
    grid[row][col] = 0;
    return false;
}
```

Backtracking

```
bool isSafe(int grid[N][N], int row, int col, int num) {
    for (int i = 0; i < N; i++) // Check rows and columns
        if (grid[row][i] == num || grid[i][col] == num) return false;

    // Check 3x3 sub-grids
    int stRow = row - row % 3, stCol = col - col % 3;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grid[i + stRow][j + stCol] == num) return false;

    return true;
}

bool solveSudokuBTrack(int grid[N][N]) {
    // Optional: Check if the input is even valid
    if (!isValidSudoku(grid)) return false;

    for (int row = 0; row < N; row++)
        for (int col = 0; col < N; col++) {
            if (grid[row][col] != 0) continue;

            for (int num = 1; num <= 9; num++) {
                if (!isSafe(grid, row, col, num)) continue;

                grid[row][col] = num;
                if (solveSudokuBTrack(grid)) return true;
                grid[row][col] = 0;
            }

            return false;
        }

    return true;
}
```

Q5 (20pts) Median

"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 $O(logk)$ 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 $xl = yl = 0, xr = m - 1, yr = n - 1$.

Note: Assume that the median is the element at index $[l + 2r]$, where l and r are indices of the leftmost and rightmost elements of the (sub)array being considered, respectively. In addition, these two arrays contain no duplicate values."

The idea is to apply Binary Search on the smaller array (out of x and y) so it can be used as an anchor for partitioning. If we choose the partition point for x to be i, then for y it will be $j = \frac{m+n+1}{2} - i$ (the $+1$ is added so it can support both odd and even sizes nicely).

When a valid partition, i.e., when all elements on the left are less, or equal, to all on the right, is found, we can then derive the median based on the left's max(es) and right's min(s). Otherwise, we shift the partition point left, or right, along the smaller array.

Since Binary Search is used on the smaller array, the runtime of $O(\log(\min(m, n))$ is satisfied.

```
int findMedian(int x[], int y[], int xl, int xr, int yl, int yr) {
    static const int INF_MIN = std::numeric_limits<int>::min(),
        INF_MAX = std::numeric_limits<int>::max();

    int m = xr + 1, n = yr + 1;
    // Target the smaller array for binary search.
    if (m > n) return findMedian(y, x, yl, yr, xl, xr);

    for (int left = 0, right = m; left <= right;) {
        int i = (left + right) / 2,
            j = (m + n + 1) / 2 - i;

        // Edge cases: If there is nothing on the left, or the right side.
        int maxXl = (i == 0) ? INF_MIN : x[xl + i - 1],
            minXr = (i == m) ? INF_MAX : x[xl + i];

        int maxYl = (j == 0) ? INF_MIN : y[yl + j - 1],
```

```
            minYr = (j == n) ? INF_MAX : y[yl + j];

        // If all left elements aren't greater than the right ones, then
        // if the merged size is odd, the median is the max of left side,
        // otherwise it's the avg of max left and min right.
        if (maxXl <= minYr && maxYl <= minXr)
            if ((m + n) % 2 != 0) return std::max(maxXl, maxYl);
            else return (std::max(maxXl, maxYl) + std::min(minXr, minYr)) / 2;
        else if (maxXl > minYr) right = i - 1; // <- Move left.
        else left = i + 1; // -> Move right.
    }

    return INF_MIN; // There might be problem if reachable.
}
```