Omer Seyfeddin Koc
EECE 7205: Fundamentals of Computer Engineering
September 23, 2023

Homework 1

Submission instruction:

- Submit one single pdf file for this homework including both coding problems and analysis problems.
- For coding problems, copy and paste your codes. Report your results.
- For analysis problems, either type or hand-write and scan.

Question 1. Coding: write programs of insertion sort, and mergesort. Find the input size n, that mergesort starts to beat insertion sort in terms of the worst-case running time. You can use clock_t function (or other time function for higher precision) to obtain running time. You need to set your input such that it results in the worst-case running time. Report running time of each algorithm for representative input sizes n. And show how you find the n that mergesort starts to beat insertion sort. (2pt.)

Solution: The C++ program for the relevant question is as follows:

```cpp
#include <iostream>

using namespace std;

// Insertion Sort Algorithm
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i]; // Copy the i-th element of the array to the 'key' variable.
        j = i - 1; // Start moving backward from the previous element.

        // Shift elements that are greater than 'key' to the right.
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // Move the element one position to the right.
            j = j - 1; // Move to the previous element.
        }
        arr[j + 1] = key; // Place 'key' in its correct position (sorted order).
    }
}

// Merge Sort Algorithm
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Calculate size of left subarray.
    int n2 = right - mid;    // Calculate size of right subarray.

    int Left[n1], Right[n2]; // Create temporary arrays for subarrays.

    // Copy data from the original array to left and right subarrays.
    for (int i = 0; i < n1; i++)
        Left[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        Right[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left; // Initialize indices for merging.

    // Merge left and right subarrays back into the original array.
    while (i < n1 && j < n2) {
```

```
        if (Left[i] <= Right[j]) {
            arr[k] = Left[i]; i++; // Place smaller element from left or right subarray
        } else {
            arr[k] = Right[j]; j++;
        }
        k++; // Move to the next position in the original array.
    }

    // Copy any remaining elements in left or right subarrays to the original array.
    while (i < n1) {
        arr[k] = Left[i]; i++; k++;
    }

    while (j < n2) {
        arr[k] = Right[j]; j++; k++;
    }
}


void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        // Calculate the middle point to divide the array into two subarrays.
        int mid = left + (right - left) / 2;

        // Recursively call mergeSort to sort the left subarray.
        mergeSort(arr, left, mid);

        // Recursively call mergeSort to sort the right subarray.
        mergeSort(arr, mid + 1, right);

        // Merge the two sorted subarrays.
        merge(arr, left, mid, right);
    }
}


int main() {
    const int maxN = 2500; // Set the maximum value of n
    int arr[maxN];

    cout << "n\tInsertion Sort (time)\tMerge Sort (time)\n";

    for (int n = 1; n <= maxN; n *= 4) {
        // Generate a worst-case scenario input array
        for (int i = 0; i < n; i++) {
            arr[i] = n - i;
        }

        // Measure insertion sort time
        clock_t insertionStart = clock();
        insertionSort(arr, n);
        clock_t insertionStop = clock();
        double insertionDuration = (double)(insertionStop - insertionStart) /
    CLOCKS_PER_SEC;

        // Reset the array
        for (int i = 0; i < n; i++) {
            arr[i] = n - i;
        }

```

```
97          // Measure merge sort time
98          clock_t mergeStart = clock();
99          mergeSort(arr, 0, n - 1);
100         clock_t mergeStop = clock();
101         double mergeDuration = (double)(mergeStop - mergeStart) / CLOCKS_PER_SEC;
102
103         cout << n << "\t" << insertionDuration << "\t\t\t" << mergeDuration << endl;
104     }
105
106     return 0;
107 }
```

I selected values for n as 1, 4, 16, 64, 256, and 1024. For each of these values of n, the time values for insertion sort and merge sort are as follows in the table below.

n	Insertion Sort (time)	Merge Sort (time)
1	0.000001	0.000001
4	0.000001	0.000001
16	0.000002	0.000002
64	0.000009	0.000005
256	0.000092	0.000023
1024	0.001388	0.000059

TABLE 1. Execution Times for Insertion Sort and Merge Sort

To find the value of n where Merge Sort starts to beat Insertion Sort, we can visually inspect the table. It appears that Merge Sort becomes more efficient than Insertion Sort when n is around 64 or 256. Beyond these values, Merge Sort consistently outperforms Insertion Sort in terms of execution time.

Question 2. You are given with an array 10, 5, 7, 9, 8, 3. Show the arrangement of the array for each iteration during insertion sort. You are given with the same array. Show the arrangement of the array for each iteration of the Partition subroutine of quicksort and the result of Partition subroutine. (1pt.)

Solution: For Insertion Sort, given an initial array, the arrays formed after each iteration are as follows:

Starting Array: [10] [5] [7] [9] [8] [3]

Iteration 1: [5] [10] [7] [9] [8] [3]

Iteration 2: [5] [7] [10] [9] [8] [3]

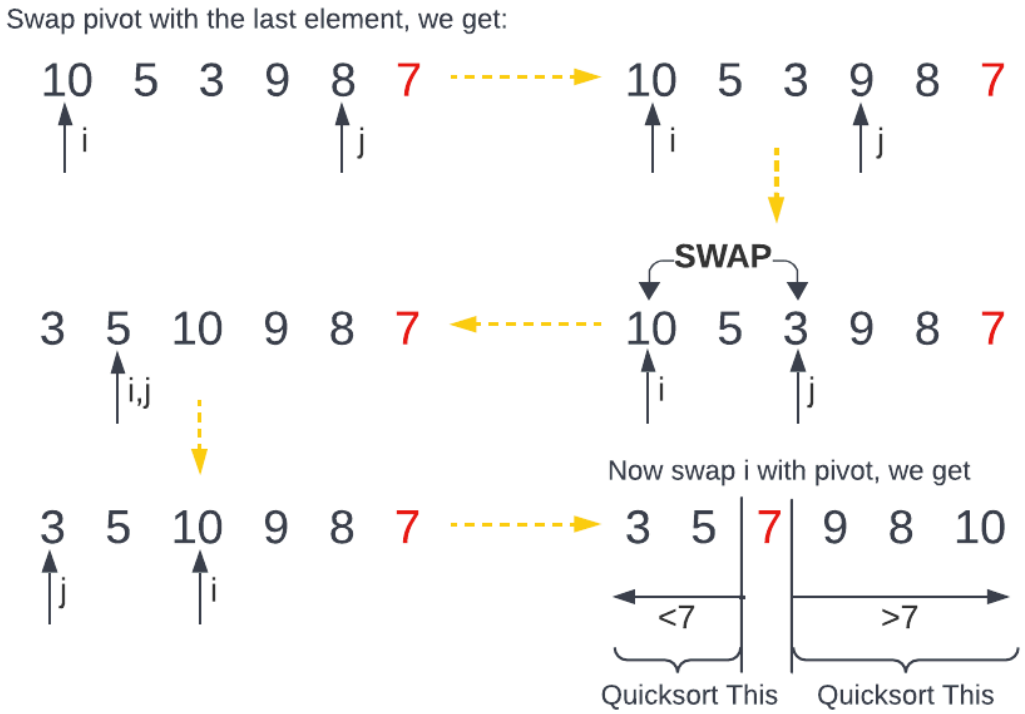
Iteration 3: [5] [7] [9] [10] [8] [3]

Iteration 4: [5] [7] [8] [9] [10] [3]

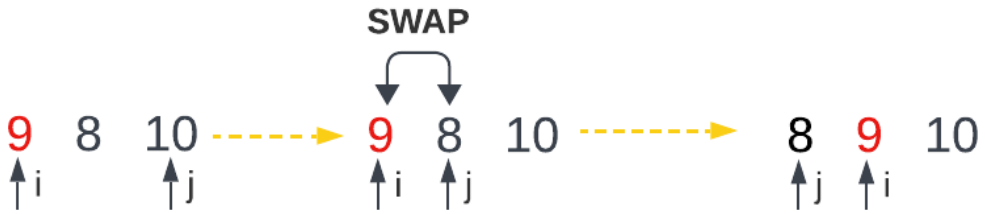
Iteration 5: [3] [5] [7] [8] [9] [10]

For each 'j' element we examine in the **for loop** of the Insertion Sort, highlighted in green. Additionally, each element that we check within the **while loop** and move one position to its right is highlighted in orange. As a result of all these operations, our final array is 3 5 7 8 9 10.

For Partitioning, **7** is chosen as the **pivot** of the array.



An array with with two elements is pretty easy to sort, too - check if the first elements is smaller than the second, and of it isn't, swap them. Therefore, **Left Array** is **3 5**. Let's check the right array. **9** is chosen as the **pivot** of the right array.



Finally, the sorted left and right arrays were merged. **The final array is 3 5 7 8 9 10.**

Question 3. True or False? (1pt.)

$$n+3 \in \Omega(n)$$
$$n+3 \in O\left(n^2\right)$$
$$n+3 \in \Theta\left(n^2\right)$$
$$2^{n+1} \in O(n+1)$$
$$2^{n+1} \in \Theta\left(2^n\right)$$

Solution: The answer to each question is as follows, in sequential order.

1. $f(n) = n + 3 \in \Omega(n)$:
$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.

$$f(n) = n + 3,\ g(n) = n$$
$$0 \leq cn \leq n + 3$$
if we assume $c = 1$, then $n_0 = 2$.
for $\boldsymbol{c = 1}$ and $\boldsymbol{n_0 = 2}$ **proved.**

Therefore, $\boldsymbol{n + 3 \in \Omega(n)}$ is **TRUE.**

2. $f(n) = n + 3 \in O(n^2)$:
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

$$f(n) = n + 3,\ g(n) = n^2$$
$$0 \leq n + 3 \leq cn^2$$
$$\text{if we assume } c = 3, \text{ then } n_0 = 1.$$
$$\textbf{for } \boldsymbol{c = 3} \text{ and } \boldsymbol{n_0 = 1} \textbf{ proved.}$$

Therefore, $\boldsymbol{n + 3 \in O(n^2)}$ **is TRUE.**

3. $f(n) = n + 3 \in \Theta(n^2)$:
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.

$$f(n) = n + 3,\ g(n) = n^2$$
$$0 \leq c_1 n^2 \leq n + 3 \leq c_2 n^2$$

It's not true. Because; for large value of n, n^2 grows much faster than n. Therefore there is no way to find constants value that make true for all n≥n_0. Therefore, $\boldsymbol{n + 3 \in \Theta(n^2)}$ **is FALSE.**

4. $f(n) = 2^{n+1} \in O(n + 1)$:
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

$$f(n) = 2^{n+1},\ g(n) = n + 1$$
$$0 \leq 2^{n+1} \leq c(n + 1)$$

It's not true. Because; In this context, 2^{n+1} grows exponentially with n, while $n + 1$ grows linearly. In Big O notation, exponential growth dominates linear growth. Therefore, $\boldsymbol{2^{n+1} \in O(n + 1)}$ **is FALSE.**

5. $f(n) = 2^{n+1} \in \Theta(2^n)$:
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.

$$f(n) = 2^{n+1},\ g(n) = 2^n$$
$$0 \leq c_1 2^n \leq 2^{n+1} \leq c_2 2^n$$
$$0 \leq c_1 2^n \leq 2 \cdot 2^n \leq c_2 2^n$$
$$\text{if we assume } c_1 = \frac{1}{2},\ c_2 = 4 \text{ then } n_0 = 1.$$
$$\textbf{for } \boldsymbol{c_1 = \frac{1}{2}},\ \boldsymbol{c_2 = 4} \text{ and } \boldsymbol{n_0 = 1} \textbf{ proved.}$$

Therefore, $\boldsymbol{2^{n+1} \in \Theta(2^n)}$ **is TRUE.**

Question 4. Using the master method, determine $T(n)$ for the following recurrance. (1pt.)

$$T(n) = 8T\left(\frac{n}{2}\right) + n$$
$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$
$$T(n) = 8T\left(\frac{n}{2}\right) + n^3$$
$$T(n) = 8T\left(\frac{n}{2}\right) + n^4$$

Solution: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:
1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

1. $\boldsymbol{T(n) = 8T\left(\frac{n}{2}\right) + n}$: We have $\boldsymbol{a = 8}$, $\boldsymbol{b = 2}$, and $\boldsymbol{f(n) = n}$.

$$\begin{aligned} \log_b a &= \log_2(8) = 3 \\ n^{\log_b(a)} &= n^3 \\ f(n) &= n \\ n^3 &> f(n) \end{aligned}$$

Since n^3 is larger than $f(n) = n$, we are in **Case 1** to solve this problem.

$$\begin{aligned} T(n) &= \Theta(n^{\log_b(a)}) \\ T(n) &= \Theta(n^3) \end{aligned}$$

2. $\boldsymbol{T(n) = 8T\left(\frac{n}{2}\right) + n^2}$: We have $\boldsymbol{a = 8}$, $\boldsymbol{b = 2}$, and $\boldsymbol{f(n) = n^2}$.

$$\begin{aligned} \log_b a &= \log_2(8) = 3 \\ n^{\log_b(a)} &= n^3 \\ f(n) &= n^2 \\ n^3 &> f(n) \end{aligned}$$

Since n^3 is larger than $f(n) = n^2$, we are in **Case 1** to solve this problem.

$$\begin{aligned} T(n) &= \Theta(n^{\log_b(a)}) \\ T(n) &= \Theta(n^3) \end{aligned}$$

3. $\boldsymbol{T(n) = 8T\left(\frac{n}{2}\right) + n^3}$: We have $\boldsymbol{a = 8}$, $\boldsymbol{b = 2}$, and $\boldsymbol{f(n) = n^3}$.

$$\begin{aligned} \log_b a &= \log_2(8) = 3 \\ n^{\log_b(a)} &= n^3 \\ f(n) &= n^3 \\ n^3 &= f(n) \end{aligned}$$

Since n^3 is equal to $f(n) = n^3$, we are in **Case 2** to solve this problem.

$$\begin{aligned} T(n) &= \Theta\left(n^{\log_b a} \lg n\right) \\ T(n) &= \Theta\left(n^3 \lg n\right) \end{aligned}$$

4. $\boldsymbol{T(n) = 8T\left(\frac{n}{2}\right) + n^4}$: We have $\boldsymbol{a = 8}$, $\boldsymbol{b = 2}$, and $\boldsymbol{f(n) = n^4}$.

$$\begin{aligned} \log_b a &= \log_2(8) = 3 \\ n^{\log_b(a)} &= n^3 \\ f(n) &= n^4 \\ n^3 &< f(n) \end{aligned}$$

Since n^3 is smaller than $f(n) = n^4$, we are in **Case 3** to solve this problem. For Case 3, we need to **check the regularity condition**: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n:

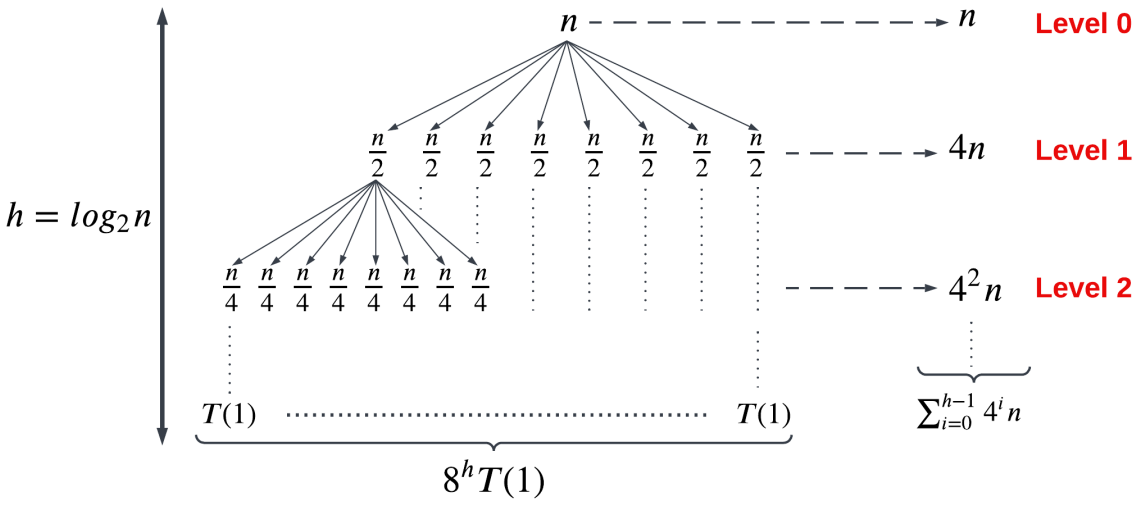
$$\begin{aligned} 8f(n/2) &\leq cn^4 \\ 8(\frac{n}{2})^4 &\leq cn^4 \\ \frac{n^4}{2} &\leq cn^4 \\ \frac{1}{2} &\leq c \end{aligned}$$

Since we found c ($c \geq \frac{1}{2}$) such that the regularity conditions holds, we can apply **Case 3** of the Master Theorem.

$$\begin{aligned} T(n) &= \Theta(f(n)) \\ T(n) &= \Theta\left(n^4\right) \end{aligned}$$

Question 5. Draw recursion tree for $T(n) = 8T\left(\frac{n}{2}\right) + n$. And prove the obtained $T(n)$ by substitution method. (Extra 1pt.)

Solution: The recursion tree starts with the root representing the original problem ($T(n)$). At each level, **8 recursive calls** are made, each with a **problem size of** $\frac{n}{2}$. This recursive branching continues until reaching the base case, $\mathbf{T(1)}$, where a constant value is obtained.



Find **T(n)** from tree:

$$T(n) = 8^h T(1) + \sum_{i=0}^{h-1} 4^i n$$
$$h = \log_2(n)$$
$$T(n) = n^3 T(1) + n \sum_{i=0}^{h-1} 4^i$$
$$T(n) = n^3 T(1) + n[1 + 4 + 4^2 + ... + 4^{h-1}]$$

Increasing Geometric Series with $r = 4$, and $a = 1$:

$$S = a + ar + ar^2 + ... + ar^{k-1} = \frac{a(r^k - 1)}{r - 1}$$

Since $r = 4$ and $a = 1$:

$$n[1 + 4 + 4^2 + ... + 4^{h-1}] = \frac{1(4^h - 1)}{4 - 1}$$
$$= \frac{1}{3}(n^2 - 1)$$

Therefore,

$$T(n) = n^3 T(1) + n[\frac{(n^2 - 1)}{3}]$$
$$T(n) = n^3 T(1) + \frac{1}{3}(n^3 - n)$$
$$T(n) = \theta(n^3)$$

Prove by **substitution method:** Assume T(n)≤cn^3 for all n≥n$_0$, where c and n_0 are positive constants.

$$T(n) = 8T(\frac{n}{2}) + n$$
$$\leq 8c(\frac{n}{2})^3 + n$$
$$\leq cn^3 + n$$
$$cn^3 + n \leq cn^3 \rightarrow \text{ FAIL}$$

With this we cannot prove our assumption in it's exact form. Now, let us assume T(n)≤c$_1 n^3 - c_2 n$ for n ≥n$_0$, where c_1, c_2 and n_0 are positive constants.

$$T(n) = 8T(\frac{n}{2}) + n$$
$$\leq 8c_1(\frac{n}{2})^3 - 8c_2\frac{n}{2} + n$$
$$\leq c_1 n^3 - 4c_2 n + n$$
$$\leq c_1 n^3 - c_2 n - [3c_2 n - n]$$

Therefore,

$$c_1 n^3 - c_2 n - [3c_2 n - n] \leq c_1 n^3 - c_2 n$$
$$0 \leq 3c_2 n - n$$

If we set $n_0 = 1$, our hypothesis works for any $\frac{1}{3} \leq c_2$. **So, $T(n) = \theta(n^3)$ proved.**