Omer Seyfeddin Koc
EECE 7205: Fundamentals of Computer Engineering
October 12, 2023

Homework 2

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. Quicksort and Randomized Quicksort:

1) Write codes for partition subroutine, quicksort, and randomized quicksort. You may need `rand()` to generate random numbers. Run the randomized quicksort 5 times for input array $A = \{1, 2, 3, \ldots, 99, 100\}$ and report the 5 running times. **(1 pt.)**
Solution: The C++ program for the relevant question is as follows:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// Partition subroutine
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    // Divide the array based on the pivot value
    for (int j = low; j <= high - 1; ++j) {
        if (arr[j] < pivot) {
            ++i;
            swap(arr[i], arr[j]);
        }
    }

    // Place the pivot element in its correct position
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Quicksort algorithm
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        // Recursive calls for the partitions before and after the pivot element
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

// Randomized partition subroutine
int randomizedPartition(int arr[], int low, int high) {
    srand(time(0));

```

```
    // Randomly select a pivot index and swap it with the last element
    int random = low + rand() % (high - low + 1);
    swap(arr[random], arr[high]);
    return partition(arr, low, high);
}

// Randomized quicksort algorithm
void randomizedQuicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = randomizedPartition(arr, low, high);

        // Recursive calls for the partitions before and after the pivot element
        randomizedQuicksort(arr, low, pi - 1);
        randomizedQuicksort(arr, pi + 1, high);
    }
}

int main() {
    int A[100];
    for (int i = 0; i < 100; ++i) {
        A[i] = i + 1;
    }

    // Display the initial array
    cout << "Initial Array: ";
    for (int i = 0; i < 100; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;

    // Run randomized quicksort 5 times and report the running times and sorted
    arrays
    for (int i = 0; i < 5; ++i) {
        clock_t start = clock();

        // Perform randomized quicksort
        randomizedQuicksort(A, 0, 99);
        clock_t end = clock();
        double elapsedTime = double(end - start) / CLOCKS_PER_SEC;

        // Display the running time of the current iteration
        cout << "Running Time " << i + 1 << ": " << elapsedTime << " seconds" <<
    endl;

    }

    return 0;
}
```

The provided C++ code implements the randomized quicksort algorithm on a randomly generated array of 100 elements. Reporting 5 running times are:

- **Running Time 1:** 6.1×10^{-5} seconds
- **Running Time 2:** 5×10^{-5} seconds
- **Running Time 3:** 5.1×10^{-5} seconds
- **Running Time 4:** 5.9×10^{-5} seconds
- **Running Time 5:** 8.2×10^{-5} seconds

2) The quicksort algorithms taught may not work well with repeated elements. Revise your codes to solve the problem. Briefly describe your revisions first. Report results on a couple of example inputs that have repeated elements.**(1 pt.)**

Solution: The initial quicksort algorithm encountered limitations when sorting arrays with repeated elements. To address this issue, the code was revised to implement a **three-way partitioning**. This modification ensures efficient sorting even when the input contains repeated elements. The revised quicksort algorithm was applied to two example inputs containing repeated elements. The results demonstrated that the three-way partitioning method effectively handled the repeated elements, leading to efficient and accurate sorting. The C++ program for the relevant question is as follows:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// Three-way Partitioning Algorithm
void threeWayPartition(int arr[], int low, int high, int &i, int &j) {
    // Base case: If the subarray has 1 or 0 elements, it is already sorted
    if (high - low <= 1) {
        // If the last element is smaller than the first, swap them
        if (arr[high] < arr[low])
            swap(arr[high], arr[low]);

        // Set i to the last element of the left partition, and j to the first element of the right partition
        i = low;
        j = high;
        return;
    }

    // Initialize mid as the first index of the subarray and pivot as the last element of the subarray
    int mid = low;
    int pivot = arr[high];

    // Traverse the subarray from left to right
    while (mid <= high) {
        // If the current element is smaller than the pivot, move it to the left partition
        if (arr[mid] < pivot)
            swap(arr[low++], arr[mid++]);
        // If the current element is equal to the pivot, move to the next element
        else if (arr[mid] == pivot)
            mid++;
        // If the current element is larger than the pivot, move it to the right partition
        else if (arr[mid] > pivot)
            swap(arr[mid], arr[high--]);
    }

    // Set i to the last element of the left partition, and j to the first element of the right partition
    i = low - 1;
    j = mid;
}

// Quicksort Algorithm with Three-way Partitioning
```

```
44 void quicksort(int arr[], int low, int high) {
45     // Base case: If the subarray has 1 or 0 elements, it is already sorted
46     if (low >= high) {
47         return;
48     }
49 
50     int i, j;
51     // Perform Three-way Partitioning and get the indices of the partitions
52     threeWayPartition(arr, low, high, i, j);
53 
54     // Recursively sort the left partition (elements smaller than the pivot)
55     // The left partition is from index 'low' to 'i'
56     quicksort(arr, low, i);
57 
58     // Recursively sort the right partition (elements greater than the pivot)
59     // The right partition is from index 'j' to 'high'
60     quicksort(arr, j, high);
61 }
62 
63 int main() {
64     // Examples
65     int A1[18] = {3, 3, 3, 3, 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 3, 3, 3, 3};
66     int A2[18] = {9, 5, 2, 9, 9, 7, 8, 9, 5, 5, 5, 1, 5, 6, 2, 4, 8, 2};
67     int A3[18] = {4, 2, 3, 5, 1, 1, 1, 6, 9, 9, 9, 9, 9, 6, 7, 8, 3, 2};
68 
69     // Display Example Input 1
70     cout << "Example Input 1: ";
71     for (int i = 0; i < 18; ++i) {
72         cout << A1[i] << " ";
73     }
74     cout << endl;
75 
76     // Sort Example Input 1 using Three-way Quicksort and display sorted Example
       Input 1
77     quicksort(A1, 0, 17);
78 
79     cout << "Sorted Output 1: ";
80     for (int i = 0; i < 18; ++i) {
81         cout << A1[i] << " ";
82     }
83     cout << endl << endl;
84 
85     // Display Example Input 2
86     cout << "Example Input 2: ";
87     for (int i = 0; i < 18; ++i) {
88         cout << A2[i] << " ";
89     }
90     cout << endl;
91 
92     // Sort Example Input 2 using Three-way Quicksort and display sorted Example
       Input 2
93     quicksort(A2, 0, 17);
94 
95     cout << "Sorted Output 2: ";
96     for (int i = 0; i < 18; ++i) {
97         cout << A2[i] << " ";
98     }
99     cout << endl << endl;
100 
101      // Display Example Input 3
```

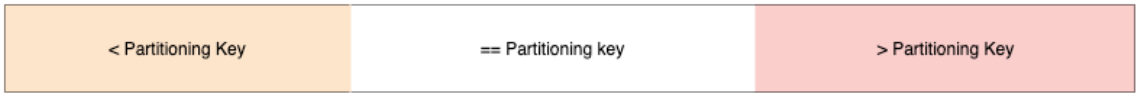
```
    cout << "Example Input 3: ";
    for (int i = 0; i < 18; ++i) {
        cout << A3[i] << " ";
    }
    cout << endl;

    // Sort Example Input 3 using Three-way Quicksort and display sorted Example
    Input 3
    quicksort(A3, 0, 17);

    cout << "Sorted Output 3: ";
    for (int i = 0; i < 18; ++i) {
        cout << A3[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Code output as follows:
Example Input 1: 3 3 3 3 3 1 4 1 5 9 2 6 5 3 3 3 3 3
Sorted Output 1: 1 1 2 3 3 3 3 3 3 3 3 3 3 4 5 5 6 9

Example Input 2: 9 5 2 9 9 7 8 9 5 5 5 1 5 6 2 4 8 2
Sorted Output 2: 1 2 2 2 4 5 5 5 5 5 6 7 8 8 9 9 9 9

Example Input 3: 4 2 3 5 1 1 1 6 9 9 9 9 9 6 7 8 3 2
Sorted Output 3: 1 1 1 2 2 3 3 4 5 6 6 7 8 9 9 9 9 9

In comparison to the first code, the new code was enhanced by incorporating a three-way partitioning scheme, allowing for efficient handling of repeated elements during the sorting process. The initial version's standard partitioning method did not address elements equal to the pivot separately, potentially leading to inefficiencies, especially with repeated elements. The updated code divides the array into three distinct regions: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot.



This three-way partitioning strategy ensures that repeated elements are managed effectively, resulting in a more efficient sorting process.

Question 2. Heapsort: Write codes for heapsort. The input array is a random permutation of $A = \{1, 2, 3, , 99, 100\}$. You should write codes to generate and print the random permutation first.**(1pt.)**

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

```
#include <iostream>
#include <algorithm>
#include <random>

using namespace std;

// Heapify Function
void heapify(int arr[], int N, int i)
{
    int largest = i; // Initialize the largest as the root
```

```
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If the left child is larger than the root
    if (left < N && arr[left] > arr[largest])
        largest = left;

    // If the right child is larger than the largest so far
    if (right < N && arr[right] > arr[largest])
        largest = right;

    // If the largest is not the root
    if (largest != i)
    {
        // Swap the root with the largest element
        swap(arr[i], arr[largest]);
        // Apply heapify
        heapify(arr, N, largest);
    }
}

// Function to perform heapsort
void heapSort(int arr[], int N)
{
    // Build a heap (rearrange the array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract elements from the heap
    for (int i = N - 1; i > 0; i--)
    {
        // Move the current root to the end
        swap(arr[0], arr[i]);

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

int main()
{
    // Generate a random permutation of A={1, 2, 3, ..., 99, 100}
    int A[100];
    for (int i = 0; i < 100; ++i)
    {
        A[i] = i + 1;
    }

    // Shuffle the array to get a random permutation
    random_device rd;
    default_random_engine rng(rd());
    shuffle(begin(A), end(A), rng);

    // Print the random permutation
    cout << "Random Permutation: ";
    for (int i = 0; i < 100; ++i)
    {
        cout << A[i] << " ";
    }
    cout << endl;
```

```
71
72      // Perform heap sort on the random permutation
73      heapSort(A, 100);
74
75      // Print the sorted array
76      cout << "Sorted Array using Heap Sort: ";
77      for (int i = 0; i < 100; ++i)
78      {
79          cout << A[i] << " ";
80      }
81      cout << endl;
82
83      return 0;
84  }
```

Code output as follows:
Input Array: 100, 7, 4, 56, 59, ..., 28, 55, 33, 38, 58.
Sorted Array using Heap Sort: 1, 2, 3, 4, 5, ..., 95, 96, 97, 98, 99, 100

This code demonstrates the Heap Sort algorithm. It first generates a random permutation of numbers from 1 to 100. Then, it shuffles the array to create a random order. The program prints this random permutation, applies the Heap Sort algorithm to sort the array in ascending order, and finally displays the sorted array.

Question 3. Counting Sort: Write codes for counting sort. The input array is $A = \{20, 18, 5, 7, 16, 10, 9, 3, 12, 14, 0\}$.**(1pt.)**

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

```
1  #include <iostream>
2
3  void countingSort(int A[], int n) {
4      // Find the maximum value in array A to determine the range (k)
5      int k = A[0];
6      for (int i = 1; i < n; ++i) {
7          if (A[i] > k) {
8              k = A[i];
9          }
10     }
11
12     // Create a count array and initialize with zeros
13     int count[k + 1] = {0};
14
15     // Count the occurrences of each element in the input array A
16     for (int j = 0; j < n; ++j) {
17         count[A[j]] = count[A[j]] + 1;
18     }
19
20     // Update count array to store the actual position of elements in the output array
21     for (int i = 1; i <= k; ++i) {
22         count[i] = count[i] + count[i - 1];
23     }
24
25     int B[n]; // Temporary array to store the sorted output
26
27     // Build the sorted output array using count array
28     for (int j = n - 1; j >= 0; --j) {
29         B[count[A[j]] - 1] = A[j];
30         count[A[j]] = count[A[j]] - 1;
```

```
    }

    // Copy the sorted array back to the input array A
    for (int i = 0; i < n; ++i) {
        A[i] = B[i];
    }
}

int main() {
    // Initial input array
    int A[] = {20, 18, 5, 7, 16, 10, 9, 3, 12, 14, 0};
    int n = sizeof(A) / sizeof(A[0]);

    // Print the initial array
    std::cout << "Initial Array: ";
    for (int i = 0; i < n; ++i) {
        std::cout << A[i] << " ";
    }
    std::cout << std::endl;

    // Perform Counting Sort
    countingSort(A, n);

    // Print the sorted array
    std::cout << "Sorted Array using Counting Sort: ";
    for (int i = 0; i < n; ++i) {
        std::cout << A[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Code output as follows:
Initial Array: 20 18 5 7 16 10 9 3 12 14 0
Sorted Array using Counting Sort: 0 3 5 7 9 10 12 14 16 18 20

This code implements the Counting Sort algorithm to sort a given input array. It first finds the maximum value in the input array and uses it to create a counting array. By counting the occurrences of each element, it determines their positions in the sorted output array. The sorted array is then printed to the console, displaying the input array in sorted order.

Question 4. Radix Sort: Write codes for radix sort: use counting sort for decimal digits from the low order to high order. The input array is $A = \{329, 457, 657, 839, 436, 720, 353\}$.**(1pt.)**

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

```
#include <iostream>

// Function to find the maximum number to determine the number of digits
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; ++i) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

```cpp

// Counting Sort function to sort the elements based on significant digit at exp
void countingSort(int arr[], int n, int exp) {
    const int BASE = 10;
    int output[n];
    int count[BASE] = {0};

    // Count the occurrences of digits at the current significant digit position
    for (int i = 0; i < n; ++i) {
        count[(arr[i] / exp) % BASE]++;
    }

    // Update count array to store the actual position of elements in output array
    for (int i = 1; i < BASE; ++i) {
        count[i] += count[i - 1];
    }

    // Build the output array using count array and update count array
    for (int i = n - 1; i >= 0; --i) {
        output[count[(arr[i] / exp) % BASE] - 1] = arr[i];
        count[(arr[i] / exp) % BASE]--;
    }

    // Copy the output array back to the original array
    for (int i = 0; i < n; ++i) {
        arr[i] = output[i];
    }
}

// Radix Sort function to sort the input array using Counting Sort for each digit
void radixSort(int arr[], int n) {
    int max = getMax(arr, n);

    // Perform counting sort for every digit, starting from the least significant digit
    to the most significant digit
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

int main() {
    // Input array
    int A[] = {329, 457, 657, 839, 436, 720, 353};
    int n = sizeof(A) / sizeof(A[0]);

    // Print the initial array
    std::cout << "Initial Array: ";
    for (int i = 0; i < n; ++i) {
        std::cout << A[i] << " ";
    }
    std::cout << std::endl;

    // Perform Radix Sort
    radixSort(A, n);

    // Output the sorted array
    std::cout << "Sorted Array using Radix Sort: ";
    for (int i = 0; i < n; ++i) {
        std::cout << A[i] << " ";
    }
```

```
72      std::cout << std::endl;
73
74      return 0;
75  }
```

Code output as follows:
Initial Array: 329 457 657 839 436 720 353
Sorted Array using Radix Sort: 329 353 436 457 657 720 839

This code implements the Radix Sort algorithm to sort an input array of integers. It first finds the maximum number to determine the number of digits in the maximum value. The algorithm then applies Counting Sort for each digit, starting from the least significant digit to the most significant digit. Counting Sort is utilized to sort the elements based on the current significant digit position. After sorting each digit, the original array is updated. Finally, the program prints the initial unsorted array, performs Radix Sort, and displays the sorted array.