FUNDAMENTALS OF COMPUTER ENGINEERING

AKSHATA KUMBLE

HOMEWORK - 2

Q1. a>

Randomized Quicksort

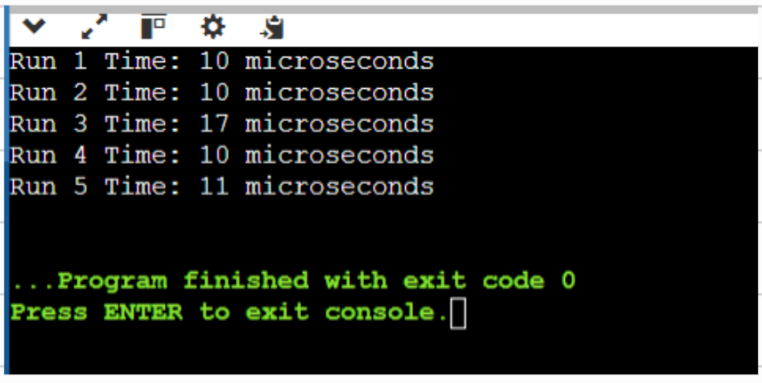
main.cpp

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

using namespace std;
using namespace std::chrono;

// swap two elements
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// partition
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

main.cpp

```cpp
// randomized partition function
int randomizedPartition(vector<int>& arr, int low, int high) {
    int randomPivot = low + rand() % (high - low + 1);
    swap(arr[randomPivot], arr[high]);
    return partition(arr, low, high);
}

// randomized quicksort
void randomizedQuicksort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = randomizedPartition(arr, low, high);
        randomizedQuicksort(arr, low, pi - 1);
        randomizedQuicksort(arr, pi + 1, high);
    }
}

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

    srand(time(0));

    for (int i = 0; i < 5; i++) {
        vector<int> arr = A;

        auto start = high_resolution_clock::now();
        randomizedQuicksort(arr, 0, arr.size() - 1);
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(stop - start);
        cout << "Run " << i + 1 << " Time: " << duration.count() << " microseconds" << endl;
    }
    return 0;
}
```

Output



Code Explaination:

The partion function implements the partitioning logic for the array. It selects the last element as the pivot and arranges all elements smaller than the pivot to its left and all elements larger to its right.

The randomizedPartition function picks a random index within a subarray swaps it with the last element and then proceeds with the usual partitioning

The randomizedQuicksort function recursively applies the partitioning and sorting to the left and right parts of the array.

The randomized quicksort algorithm ran 5 times on the array $A = \{1, 2, 3, \ldots, 100\}$ and the corresponding running times were reported:

1. 10 microseconds
2. 10 microseconds
3. 17 microseconds
4. 10 microseconds
5. 11 microseconds

$O(n \log n)$ is the time complexity for randomized quicksort.

Q1. b>

Revised Quicksort

main.cpp

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

using namespace std;
using namespace std::chrono;

// swap two elements
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
// 2-way partitioning
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
// partition function
int randomizedPartition(vector<int>& arr, int low, int high) {
    int randomPivot = low + rand() % (high - low + 1);
    swap(arr[randomPivot], arr[high]);
    return partition(arr, low, high);
}
// 2-way quicksort
void randomizedQuicksort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = randomizedPartition(arr, low, high);
        randomizedQuicksort(arr, low, pi - 1);
        randomizedQuicksort(arr, pi + 1, high);
    }
}
```

2 way partition is inefficient with repeated elements

```cpp
// 3-way partitioning
void threeWayPartition(vector<int>& arr, int low, int high, int& i, int& j) {
    i = low - 1, j = high;
    int p = low, pivot = arr[high];

    while (p < j) {
        if (arr[p] < pivot) {
            i++;
            swap(arr[i], arr[p]);
            p++;
        } else if (arr[p] > pivot) {
            j--;
            swap(arr[j], arr[p]);
        } else {
            p++;
        }
    }
    swap(arr[j], arr[high]);
}

// Randomized quicksort with 3-way partitioning
void randomizedQuicksort3Way(vector<int>& arr, int low, int high) {
    if (low < high) {
        int i, j;
        threeWayPartition(arr, low, high, i, j);
        randomizedQuicksort3Way(arr, low, i);
        randomizedQuicksort3Way(arr, j + 1, high);
    }
}
```

3 way partitioning is better at handling repeated elements more efficiently grouping the together leading to improved performance

```cpp
int main() {
    vector<int> A = {4, 2, 2, 8, 7, 7, 7, 5, 4, 3, 3};

    srand(time(0));

    cout << "Original Algorithm (2-way partitioning):" << endl;
    for (int i = 0; i < 5; i++) {
        vector<int> arr = A;  // Copy of original array for each run
        auto start = high_resolution_clock::now();
        randomizedQuicksort(arr, 0, arr.size() - 1);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop - start);
        cout << "Run " << i + 1 << " Time: " << duration.count() << " nanoseconds" << endl;
    }


    cout << "\nRevised Algorithm (3-way partitioning):" << endl;
    for (int i = 0; i < 5; i++) {
        vector<int> arr = A;  // Copy of original array for each run
        auto start = high_resolution_clock::now();
        randomizedQuicksort3Way(arr, 0, arr.size() - 1);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop - start);
        cout << "Run " << i + 1 << " Time: " << duration.count() << " nanoseconds" << endl;
    }

    return 0;
}
```

```
Original Algorithm (2-way partitioning):
Run 1 Time: 1340 nanoseconds
Run 2 Time: 1217 nanoseconds
Run 3 Time: 1297 nanoseconds
Run 4 Time: 991 nanoseconds
Run 5 Time: 910 nanoseconds

Revised Algorithm (3-way partitioning):
Run 1 Time: 774 nanoseconds
Run 2 Time: 602 nanoseconds
Run 3 Time: 537 nanoseconds
Run 4 Time: 412 nanoseconds
Run 5 Time: 438 nanoseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

eg2.

```cpp
int main() {
    vector<int> A = {1, 1, 7,7 ,7, 2, 50,6, 6, 4, 2, 2, 8, 7, 7, 7, 55, 4, 30, 3};
```

```
Original Algorithm (2-way partitioning):
Run 1 Time: 2371 nanoseconds
Run 2 Time: 1740 nanoseconds
Run 3 Time: 1471 nanoseconds
Run 4 Time: 2180 nanoseconds
Run 5 Time: 1640 nanoseconds

Revised Algorithm (3-way partitioning):
Run 1 Time: 2041 nanoseconds
Run 2 Time: 1703 nanoseconds
Run 3 Time: 1430 nanoseconds
Run 4 Time: 1424 nanoseconds
Run 5 Time: 1346 nanoseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

eg3

```cpp
int main() {
    vector<int> A = {1, 1, 7, 7, 7, 1, 1, 1, 1, 1, 1, 1, 1};
```

```
Original Algorithm (2-way partitioning):
Run 1 Time: 1690 nanoseconds
Run 2 Time: 1390 nanoseconds
Run 3 Time: 1300 nanoseconds
Run 4 Time: 1350 nanoseconds
Run 5 Time: 1530 nanoseconds

Revised Algorithm (3-way partitioning):
Run 1 Time: 490 nanoseconds
Run 2 Time: 430 nanoseconds
Run 3 Time: 340 nanoseconds
Run 4 Time: 440 nanoseconds
Run 5 Time: 380 nanoseconds
```

To fix the poor performance of the original Quicksort with repeated elements, we can use a 3-way partitioning scheme. In the original quicksort the partition function splits the array into two parts based on a single pivot. However when the input array contains many repeated elements, it can lead to unbalanced partitions resulting in poor performance $O(n^2)$

In 3 way partitioning the array is divided into 3 parts :

1. Elements less than the pivot. 2. Elements equal to the pivot. 3. Elements greater than the pivot.

This approach ensures that all repeated elements are grouped together, reducing redundant comparisons.

Q2. Heapsort

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;
using namespace std::chrono;

// swap two elements
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// function to heapify
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// function to perform heapsort
void heapsort(vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i])
        heapify(arr, i, 0);
    }
}

// function to generate and print a random permutation
void generateRandomPermutation(vector<int>& A) {
    // using std::random_shuffle to generate a random per
    random_shuffle(A.begin(), A.end());

    //random permutation
    cout << "Random Permutation of Array A: ";
    for (int i = 0; i < A.size(); i++) {
        cout << A[i] << " ";
    }
    cout << endl;
}

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

    generateRandomPermutation(A);
```

```
    auto start = high_resolution_clock::now();
    heapsort(A);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);

    cout << "Sorted Array: ";
    for (int i = 0; i < A.size(); i++) {
        cout << A[i] << " ";
    }
    cout << endl;

    cout << "Time taken by heapsort: " << duration.count() << " microseconds" << endl;

    return 0;
}
```

Output

```
Random Permutation of Array A: 36 55 15 23 1 12 24 6 34 54 39 47 73 86 60 17 88 20 74 41 50 66 42 19 79 43 98 8 65 48 51 4 76 78 28 82 95 75 46 40 97 10 22 52 63 18 69 91 59 81 37 77
 70 31 93 25 61 21 14 35 45 11 5 96 84 72 94 83 67 26 38 68 32 100 99 16 13 64 49 85 87 80 56 3 9 92 53 33 90 29 30 58 62 2 27 7 57 71 89 44
Sorted Array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Time taken by heapsort: 16 microseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

This program generates a random permutation of the array and sorts it using heapsort.

Heapify ensures that the subtree rooted at index i follows the max heap property

Build max heap converts the array into a max heap.

Heapsort repeatedly extracts the largest element from the heap and places it at the end of the array, shrinking the heap after each extraction.

$O(n \log n)$ is the time complexity.

Q3. Counting Sort

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// counting sort function
void countingSort(vector<int>& arr) {
    int maxElement = *std::max_element(arr.begin(), arr.end());
    int minElement = *std::min_element(arr.begin(), arr.end());

    int range = maxElement - minElement + 1;

    vector<int> count(range, 0);

    vector<int> output(arr.size());

    // storing count of each element
    for (int i = 0; i < arr.size(); i++) {
        count[arr[i] - minElement]++;
    }

    // stores the position of that element in the output array
    for (int i = 1; i < count.size(); i++) {
        count[i] += count[i - 1];
    }

    for (int i = arr.size() - 1; i >= 0; i--) {
        output[count[arr[i] - minElement] - 1] = arr[i];
        count[arr[i] - minElement]--;
    }

    for (int i = 0; i < arr.size(); i++) {
        arr[i] = output[i];
    }
}

int main() {
    vector<int> A = {20, 18, 5, 7, 16, 10, 9, 3, 12, 14, 0};
    cout << "Original Array: ";
    for (int i = 0; i < A.size(); i++) {
        cout << A[i] << " ";
    }
    cout << endl;

    countingSort(A);

    cout << "Sorted Array: ";
    for (int i = 0; i < A.size(); i++) {
        cout << A[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```
Original Array: 20 18 5 7 16 10 9 3 12 14 0
Sorted Array: 0 3 5 7 9 10 12 14 16 18 20
```

Output ↑

Counting sort is an efficient algorithm for sorting an array when the range of possible values in the input is known and small. It works by counting the occurences of each unique value in the array and using that information to place each element in its correct position

- We first find the maximum (20) and minimum (0) elements in the array to determine the range. In this case the range is 21 (from 0 to 20)
- Build a count array to count the frequency of each element. The size of the count array is equal to the range of input values.
- The count array is modified so that each element at index 'i' contains the number of elements less than or equal to 'i' in the original array.
- Using the cummulative count, the algorithm places each element of the input array in its correct position in the output array.

Time Complexity: $O(n + K)$

where n is the number of elements in the input array and K is the range of the input values.

Q4. Radix Sort

main.cpp

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

// function to get maximum value in an array
int getMax(const vector<int>& arr) {
    return *max_element(arr.begin(), arr.end());
}

// function to perform counting sort
void countingSortByDigit(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);
    vector<int> count(10, 0);

    // count the occurrences of each digit
    for (int i = 0; i < n; i++) {
        int digit = (arr[i] / exp) % 10;
        count[digit]++;
    }

    // modify the count array to store the actual positions of digits in the output array
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // build the output array
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }
```

main.cpp

```cpp

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

// main function to perform radix sort
void radixSort(vector<int>& arr) {
    int maxElement = getMax(arr);

    for (int exp = 1; maxElement / exp > 0; exp *= 10) {
        countingSortByDigit(arr, exp);
    }
}

int main() {
    // input array
    vector<int> A = {329, 457, 657, 839, 436, 720, 353};
    cout << "Original Array: ";
    for (int i = 0; i < A.size(); i++) {
        cout << A[i] << " ";
    }
    cout << endl;
    radixSort(A);

    // print sorted array
    cout << "Sorted Array: ";
    for (int i = 0; i < A.size(); i++) {
        cout << A[i] << " ";
    }
    cout << endl;

    return 0;
}

```

Output

```
Original Array: 329 457 657 839 436 720 353
Sorted Array: 329 353 436 457 657 720 839
```

Radix Sort is a non comparative integer sorting algorithm that processes digits of numbers from the least significant digit to the most significant digit. It uses counting sort as a subroutine to sort the digits.

- Extract each digit of the numbers starting from the LSD (1's place)
- Using counting sort, sort the array based on each digit.
- Repeat the process for each digit (10's place then 100's place)

Time Complexity.

$O(d * (n+K))$, where:

n: no. of elements

d: no. of digits in the largest number

K: range of the digit values. (10)