Algorithms and Data Structures

Spring 2019

Assignment 5

Date: March 14, 2019

Taiyr Begeyev

Problem 5.1 Quick Sort with Partition Versions

Implement 3 versions of the Quicksort algorithm with 3 different versions of the partition algorithm:

(a) Lomoto partition as on the lecture slides (Lecture 8).

```
/*
----------            .--
\--     ---/ -----    |--|   ---.---. -------
  |     |    \--   \  |  | <    |  | \-   -- \
  |     |     / -- \- |  |  \---   |  |  | \/
  |----|     (----   / |--|  / ----|  |--|
                 \/         \/
*/
/**
 * @brief implementation of the Quick Sort Algorithm
 * @param arr[] array which is needed to be sorted
 * @param start - index we need to start sorting with
 * @param end - index we need to stop sorting at
*/

void quickSort(int arr[], int start, int end) {
    // base case
    if (start < end) {
        int q = partition(arr, start, end);
        quickSort(arr, start, q - 1);
        quickSort(arr, q + 1, end);
    }
}

/**
 *  @brief key to the Quick Sort Algorithm
 *
 *  Always selects an the front element as a pivot (our case)
 *  element around which to partition subarray A[start .. end]
 *  @param arr[] array which is needed to be sorted
 *  @param start - index we need to start sorting with
 *  @param end - index we need to stop sorting at
*/

int partition(int arr[], int start, int end) {
    // make the most left hand element a pivot
    int pivot = arr[start];
    int pivotIndex = start;

    for (int j = start + 1; j <= end; j++) {
        // iterate through all list starting from the
        // element next to the 'start'
        if (arr[j] < pivot) {
            pivotIndex++;
            swap(arr[j], arr[pivotIndex]);
        }
    }

    // after we located all elements which are <= than
    // the pivot put the pivot on its supposed place
    swap(arr[start], arr[pivotIndex]);
    return pivotIndex;
}
```

(b) Hoare partition

Reference: Wikipedia contributors. "Quicksort." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 1 Mar. 2019. Web. 14 Mar. 2019.

```
/**
 * @brief implementation of the Quick Sort Algorithm
 * @param arr[] array which is needed to be sorted
 * @param start - index we need to start sorting with
 * @param end - index we need to stop sorting at
*/

void quickSort(int arr[], int start, int end) {
    // base case
    if (start < end) {
        int q = partition(arr, start, end);
        quickSort(arr, start, q);
        quickSort(arr, q + 1, end);
    }
}

/**
 * @brief works by initializing two indexes that start
 * at two ends, the two indexes move toward each other
 * until an inversion is (A smaller value on left side
 * and greater value on right side) found. When an
 * inversion is found, two values are swapped and process
 * is repeated.
 *
*/

int partition(int arr[], int start, int end) {
    int pivot = arr[start];
    int i = start - 1; // initialize the left index
    int j = end + 1; // initialize the right index

    while(1) {
        do {
            // find leftmost element greater than or
            // equal to pivot
            i++;
        } while(arr[i] < pivot);

        do {
            // find rightmost element less than or
            // equal to pivot
            j--;
        } while(arr[j] > pivot);

        if (i >= j)
            // if two ponters met
            return j;

        swap(arr[i], arr[j]);
    }
}
```

(c) "Median-of-three" partition.

Reference: Quick Sort, COMP171, Spring 2018 http://read.pudn.com/downloads117/ebook/497420/8-comp171-qsort_6up.pdf

```
/**
 * @brief implementation of the Quick Sort Algorithm
 * @param arr[] array which is needed to be sorted
 * @param start - index we need to start sorting with
 * @param end - index we need to stop sorting at
*/

void quickSort(int arr[], int start, int end) {
    // base case
    if (start < end) {
        int pivot = median3(arr, start, end);
        // begin partitioning
        int i;
        i = partition(arr, start, end, pivot);
        quickSort(arr, start, i);
        quickSort(arr, i + 1, end);
    }
}

/**
 * @brief Median of three
 *
 * Compare three elements: the leftmost, rightmost
 * and center. Swap these elements if necessary so
 * that
 *      A[left] = smallest
 *      A[right] = largest
 *      A[center] = median of three
 * Pick A[center as the pivot]
 * Swap A[center] and A[right - 1] so that pivot is at
 * second last position
 *
*/
int median3(int arr[], int start, int end) {
    int center = (start + end) / 2;
    if (arr[center] < arr[start])
        swap(arr[start], arr[center]);
    if (arr[end] < arr[start])
        swap(arr[start], arr[end]);
    if (arr[end] < arr[center])
        swap(arr[center], arr[end]);

    // place a pivot at position end - 1
    swap(arr[center], arr[end - 1]);
    return center;
}

/**
 * @brief works by initializing two indexes that start
 * at two ends, the two indexes move toward each other
 * until an inversion is (A smaller value on left side
 * and greater value on right side) found. When an
 * inversion is found, two values are swapped and process
 * is repeated.
 *
*/
```

```
int partition(int arr[], int start, int end, int pivot) {
    pivot = arr[start];
    int i = start - 1; // initialize the left index
    int j = end + 1; // initialize the right index

    while(1) {
        do {
            // find leftmost element greater than or
            // equal to pivot
            i++;
        } while(arr[i] < pivot);

        do {
            // find rightmost element less than or
            // equal to pivot
            j--;
        } while(arr[j] > pivot);

        if (i >= j)
            // if two ponters met
            return j;

        swap(arr[i], arr[j]);
    }
    return arr[start];
}
```

(d) Measure the running times of the 3 Quicksort versions from above for the same 100000 randomly generated sequences of fixed length 1000, compute the average running times for each of the 3 versions and compare them. Explain the behaviour of the 3 versions and your observations.

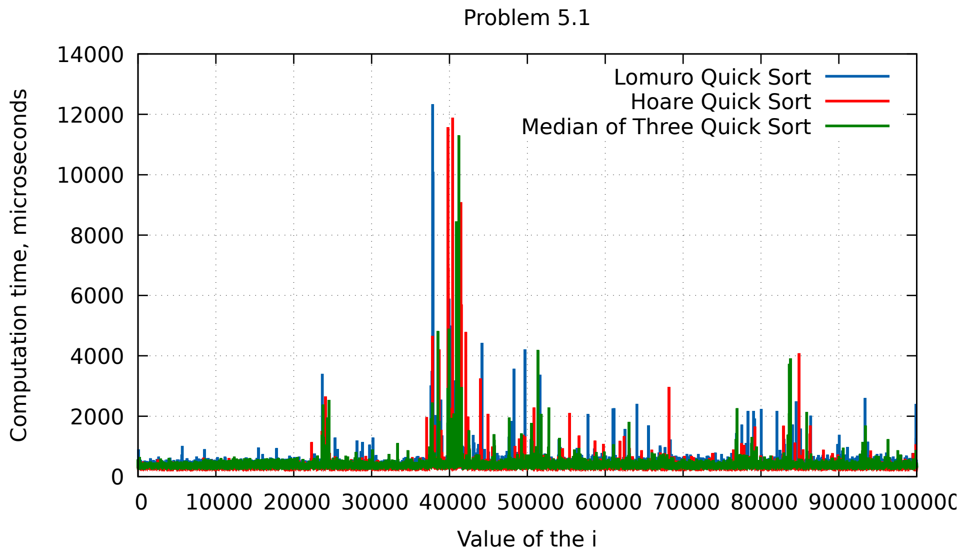
```
int main() {
    srand(static_cast<unsigned int>(time(NULL)));
    // OPEN FILE
    ofstream out;
    out.open("output.dat", ios::trunc | ios::out);
    if (!out.is_open()) {
        cerr << "Something went wrong. Coudln't open the file" << endl;
        exit(1);
    }

    vector<int> arr1;
    vector<int> arr2;
    vector<int> arr3;

    double average1 = 0, average2 = 0, average3 = 0;

    for (int i = 0; i < SEQUENCES; i++) {
        for (int j = 0; j < LENGTH; j++) {
            arr1.push_back(rand() % (LENGTH * 10) + 1);
        }
        out << i << " ";
        arr2 = arr1;
        arr3 = arr1;

        cout << "Initial array:" << endl;
        printArr(arr1);
        cout << "Sorted arrays: " << endl;

        // test Lomuro Quick Sort
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        LomutoQuickSort<int>(arr1, 0, LENGTH - 1);
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        double duration = duration_cast<microseconds>(t2 - t1).count();
        out << duration << " ";
        average1 += duration;
        printArr(arr1);

        // test Hoare Quick Sort
        t1 = high_resolution_clock::now();
        HoareQuickSort<int>(arr2, 0, LENGTH - 1);
        t2 = high_resolution_clock::now();
        duration = duration_cast<microseconds>(t2 - t1).count();
        out << duration << " ";
        average2 += duration;
        printArr(arr2);

        // test Median of Three
        t1 = high_resolution_clock::now();
        Median3QuickSort<int>(arr3, 0, LENGTH - 1);
        t2 = high_resolution_clock::now();
        duration = duration_cast<microseconds>(t2 - t1).count();
        out << duration << endl;
        average3 += duration;
        printArr(arr3);
        cout << endl;
        cout << endl;

        arr1.clear();
        arr2.clear();
        arr3.clear();
    }
    cout << endl;
    cout << "Average for Lomuro: " << average1 / SEQUENCES << endl;
    cout << "Average for Hoare: " << average2 / SEQUENCES << endl;
    cout << "Average for Median of Three: " << average3 / SEQUENCES << endl;

    out.close();
    return 0;
}
```

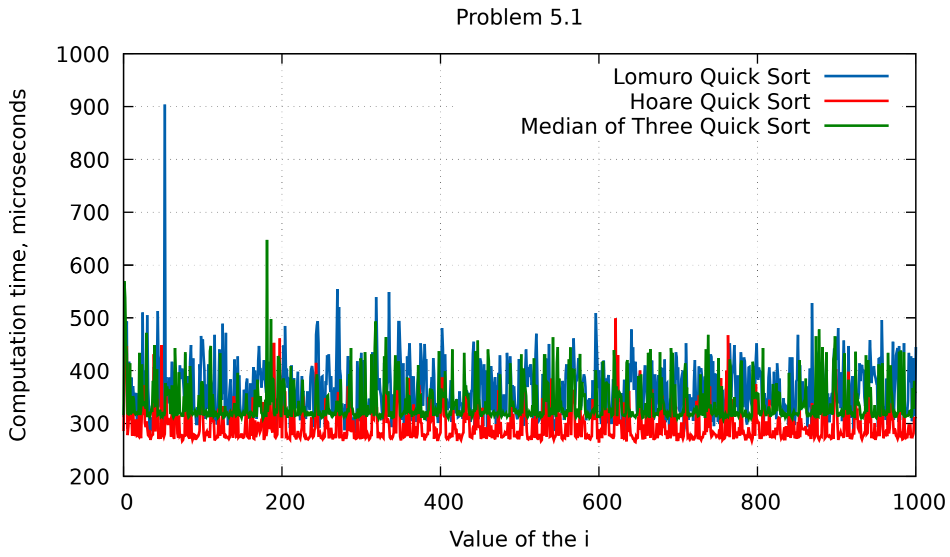
GnuPlot Script :

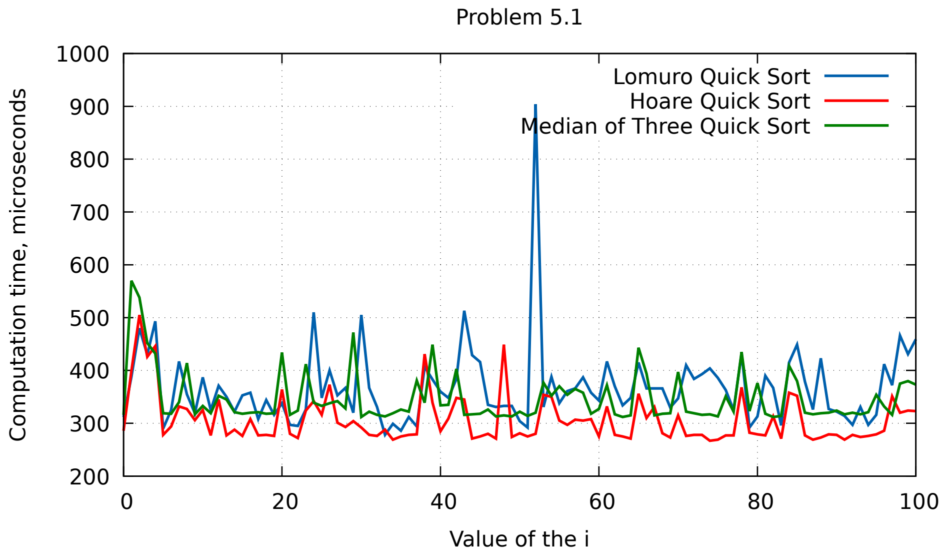
X axis in the graph stands for counter (iterator of the loop). All lists have the same length
Graph 1 shows computation times for all sequences from 1...100000
Graph 2 shows computation times for all sequences from 1...1000
Graph 3 shows computation times for all sequences from 1...100

```
set terminal pdf
set output "plot1.pdf"

# x and y labels
set xlabel "Value of the i"
set ylabel "Computation time, microseconds"
set title "Problem 5.1"

# Line styles
```

```
set style line 1 \
    linecolor rgb "#0060ad" \
    linetype 1 linewidth 2 \
    pointtype 7 pointsize 1.5

set style line 2 \
    linecolor rgb "#ff0000" \
    linetype 2 linewidth 2 \
    pointtype 7 pointsize 1.5

set style line 3 \
    linecolor rgb "#008000" \
    linetype 3 linewidth 2 \
    pointtype 7 pointsize 1.5

set grid

plot "output.dat" using 1:2 title "Lomuro␣Quick␣Sort"  with lines linestyle 1,\
     "output.dat" using 1:3 title "Hoare␣Quick␣Sort" with lines linestyle 2,\
     "output.dat" using 1:4 title "Median␣of␣Three␣Quick␣Sort" with lines linestyle 3

set output "plot2.pdf"

set xlabel "Value␣of␣the␣i"
set ylabel "Computation␣time,␣microseconds"
set title "Problem␣5.1"

plot "output.dat" every ::0::1000 using 1:2 title "Lomuro␣Quick␣Sort"
with lines linestyle 1,\
     "output.dat" every ::0::1000 using 1:3 title "Hoare␣Quick␣Sort" with lines lines
     "output.dat" every ::0::1000 using 1:4 title "Median␣of␣Three␣Quick␣Sort" with l

set output "plot3.pdf"

set xlabel "Value␣of␣the␣i"
set ylabel "Computation␣time,␣microseconds"
set title "Problem␣5.1"

plot "output.dat" every ::0::100 using 1:2 title "Lomuro␣Quick␣Sort"
with lines linestyle 1,\
     "output.dat" every ::0::100 using 1:3 title "Hoare␣Quick␣Sort" with lines linest
     "output.dat" every ::0::100 using 1:4 title "Median␣of␣Three␣Quick␣Sort" with li
```







The observation is that the most efficient method is Hoarte Quick Sort. It is the fastest, because it does three times fewer swaps on average and it creates efficient partitions even when all values are equal. The Median of Three method goes second and the last is the Lomuto Approach, which requires more comparisons.

Average for Lomuro: 370.849
Average for Hoare: 313.161
Average for Median of Three: 353.755

Problem 5.2 Modified QuickSort

(a) Implement a modified version of the Quicksort algorithm, where the sequence of elements is always split into three subsequences by simultaneously using the first two elements as pivots.

```
void ModifiedQuickSort(vector<int>& arr, int start, int end) {
    if(start < end){
        int lp, rp;
        partition(arr, start, end, lp, rp);
        ModifiedQuickSort(arr, start, lp - 1);
        ModifiedQuickSort(arr, lp + 1, rp - 1);
        ModifiedQuickSort(arr, rp + 1, end);
    }
}

void partition(vector<int> &arr, int start, int end, int &lp, int &rp) {
    // I swap the second element and the last element. Then I set the left pivot
    // to be the smaller element and right pivot to be the greater element.
    swap(arr[start + 1], arr[end]);
    if(arr[start] > arr[end]){
        swap(arr[start], arr[end]);
    }

    // Left index is start + 1 and right index is end - 1
    int pivot1 = arr[start];
    int pivot2 = arr[end];
    int idx = start + 1;
    int pivot1Index = start + 1;
    int pivot2Index = end - 1;

    while(idx <= pivot2Index) {

    // Put all elements smaller than the left(smaller) pivot to the left
    if(arr[idx] < pivot1) {
        swap(arr[idx], arr[pivot1Index]);
        pivot1Index++;
    }
    else if(arr[idx] >= pivot2) {
        // Put all elements greater than the right(greater) pivot to the right
        while(arr[pivot2Index] > pivot2 && idx < pivot2Index) pivot2Index--;

        swap(arr[idx], arr[pivot2Index]);
        pivot2Index--;
        // Check if the new swapped element is less than the left pivot. If so, take
        if(arr[idx] < pivot1){
            swap(arr[idx], arr[pivot1Index]);
            pivot1Index++;
        }
    }
    // All the elements between the left and right pivot go in the middle section
        idx++;
    }
    pivot1Index--;
    pivot2Index++;

    // swap pivots with their cursors(pivot1Index and pivot2Index)
    swap(arr[start], arr[pivot1Index]);
    swap(arr[end], arr[pivot2Index]);
    lp = pivot1Index;
    rp = pivot2Index;
}
```

(b) Determine and prove the best-case and worst-case running time for the modified Quicksort from subpoint (a)

Best Case
the partition splits the array evenly, which means $T(n) = 3T(n/3) + \Theta(n)$.
1. Extract $a, b, f(n)$

$$a = 3, b = 3, f(n) = \Theta(n)$$

2. Determine $n^{\log_b a}$

$$n^{\log_b a} = n^{log_3 3} = n$$

3. Compare $f(n)$ and $n^{\log_b a}$
Since $n = n$, then $n^{log_b a} = f(n)$
4. Determine the appropriate case and apply it
Thus case 2:

$$f(n) = \Theta(n) \implies$$
$$T(n) = \Theta(n \lg n)$$

Worst Case
The worst case occurs when is input is either sorted, reverse sorted, partition around min or max element or when one side has no elements

$$T(n-2) + \Theta(n) = \Theta(n^2)(ArithmeticSeries)$$

(c) Implement a modified version of the Randomized Quicksort algorithm, where the sequence of elements is always split into three subsequences by simultaneously using two random elements as pivots.

```
void randomizedPartition(vector<int> & arr, int start, int end, int &lp, int &rp) {
    int p1 = rand() % (end - start + 1) + start;
    int p2 = rand() % (end - start + 1) + start;
    while(p1 == p2){
        p1 = rand() % (end - start + 1) + start;
        p2 = rand() % (end - start +1 ) + start;
    }
    swap(arr[p1], arr[start]);
    swap(arr[p2], arr[start+1]);
    partition(arr, start, end, lp, rp);
}
```

Problem 5.3 Decision Tree

Write a different proof for $\lg n! = \Theta(n \lg n)$ without having to use Stirling's formula

First, we need to show that $\lg n!$ is less or equal to $n \lg n$

$$\begin{aligned} \lg n! &= \lg(1 \cdot 2 \cdot 3 ... \cdot n) \\ &= \lg 1 + \lg 2 + \lg 3 + ... + \lg n \\ &\leq \lg n + \lg n + \lg n + ... + \lg n \\ &= n \lg n \end{aligned}$$

Therefore, $\lg n! = O(n \lg n)$

Next, let's show that $\lg n!$ is greater or equal than a constant multiple of $n \lg n$.
We can just delete the first half of the sum:

$$\lg n! \geq \lg \frac{n}{2} + \lg(\frac{n}{2} + 1) + \lg(\frac{n}{2} + 2) + ... + \lg n$$

Replacing all remaining terms by the smallest one gives $\lg n! \geq \frac{n}{2} \lg \frac{n}{2}$

$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2} = \frac{n}{2}(\lg n - 1) = \frac{n}{2} \lg n - \frac{n}{2}$$

We want to show this is greater than a multiple of $n \lg n$. Since it is less than $\frac{n}{2} \lg n$, we need to pick a multiple less than one-half, for example one-quarter. That's why we need to show that $\frac{n}{2} \lg n - \frac{n}{2} \geq \frac{n}{4} \lg n$

$$\begin{aligned} \lg \frac{n}{2} &\geq 2 \\ \frac{1}{4} \lg n &\geq \frac{1}{2} \\ \frac{1}{4} n \lg n &\geq \frac{1}{2} n \\ \frac{1}{4} n \lg n - \frac{1}{2} n &\geq 0 \\ \frac{1}{2} n \lg n - \frac{1}{2} n &\geq \frac{1}{4} n \lg n \end{aligned}$$

Consequently,

$$\begin{aligned} \lg n! &\geq \frac{n}{2} \lg \frac{n}{2} \\ &= \frac{n}{2}(\lg n - 1) \\ &= \frac{n}{2} \lg n - \frac{n}{2} \\ &\geq \frac{n}{4} \lg n \\ &= \frac{1}{4} n \lg n \end{aligned}$$

Thus, $\lg n! = \Omega(n \lg n)$.
We can conclude that $\lg n! = \Theta(n \lg n)$

Reference:

1. Wikipedia contributors, 'Quicksort', Wikipedia, The Free Encyclopedia, 1 March 2019, 14:14 UTC, `https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=885665329` [accessed 14 March 2019]
2. Quick Sort, COMP171, Spring 2018 `http://read.pudn.com/downloads117/ebook/497420/8-comp171-qsort_6up.pdf`
3. `https://www.geeksforgeeks.org/dual-pivot-quicksort/`