CMP340 - HW6: Implementation
Quicksort Algorithm

Name: Angelo Sebastian
ID: b00087962

Name: Eesha Yasser Butt
ID: g00085661

Instructor: Professor Salam Dhou, Ph.D.
Due Date: November 27, 2022

Q1) Basic quicksort where the chosen pivot is the first element of an array.

/*note: The count is used for the experimental complexity. I counted the comparisons in finding the correct spot of the pivot (this contributes the most) and also the comparisons in filtering the left and right sides of the pivot. (this contributes less but I think is still relevant).*/

Code for Partition and Quicksort (also to print the arrays):

```
int partitionQ1(int a[], int left, int right, long& count)
{
	int pivot = a[left];
	int s = left;

	for (int i = left + 1; i <= right; i++) {
		if (a[i] < pivot) {
			s = s + 1;
			//swap(a[s], a[i]);
	}
		count++;
	}
	swap(a[s], a[left]);

	int i = left;
	int j = right;

	while (i<s and j>s) {
		while (a[i] <= pivot) i++;
		while (a[j] > pivot) j--;
		if (i<s and j>s) {
			swap(a[i++], a[j--]);
		}
		count++;
	}

	return s;
}

void quicksort(int a[], int left, int right, long& count)
{
	if (left < right) {
		int s = partitionQ1(a, left, right, count);
		quicksort(a, left, s - 1, count);
		quicksort(a, s + 1, right, count);
	}
}
```

```
void printArray(int a[], int size)
{
	for (int i = 0; i < size; i++) {
		cout << a[i] << "\t";
		if (i % 14 == 13) cout << endl;
	}
	cout << endl << endl;
}
```

Q2) Quicksort where the pivot is the median of three elements of an array (first, last, and middle).

Code for Partition and Quicksort (also to print the arrays and finding the pivot):

```
int indexOfMedian(int arr[], int l, int r) {
	int m = (l + r) / 2;
	int a = arr[l], b = arr[r], c = arr[m];

	if ((a <= b and b <= c) or (c <= b and b <= a))
		return r;
	else if ((b <= a and a <= c) or (c <= a and a <= b))
		return l;
	else
		return m;

}


int partitionQ2(int a[], int left, int right, long& count)
{
	int median = indexOfMedian(a, left, right);
	swap(a[left], a[median]);

	int pivot = a[left];
	int s = left;

	for (int i = left + 1; i <= right; i++) {
		if (a[i] < pivot) {
			s = s + 1;
			//swap(a[s], a[i]);
		}
		count++;
	}
```

```
	swap(a[s], a[left]);

	int i = left;
	int j = right;

	while (i<s and j>s) {
		while (a[i] <= pivot) i++;
		while (a[j] > pivot) j--;
		if (i<s and j>s) swap(a[i++], a[j--]);
		count++;
	}

	return s;
}

void quicksort(int a[], int left, int right, long& count)
{
	if (left < right) {
		int s = partitionQ2(a, left, right, count);
		quicksort(a, left, s - 1, count);
		quicksort(a, s + 1, right, count);
	}
}

void printArray(int a[], int size)
{
	for (int i = 0; i < size; i++) {
		cout << a[i] << "\t";
		if (i % 14 == 13) cout << endl;
	}
	cout << endl << endl;
}
```

Q3) Quicksort where the pivot is the median of three randomly chosen elements of an array.

Code for Partition and Quicksort (also to print the arrays and finding the pivot):

```
int indexOfMedian(int arr[], int left, int right) {

	int size = (right - left) + 1;
	int x = (rand() % size) + left, y = (rand() % size) + left, z = (rand() % size) + left;
```

```
	if (size >= 3) {
		while (y == x) y = (rand() % size) + left;
		while (z == x or z == y) z = (rand() % size) + left;
	}

	int a = arr[x], b = arr[y], c = arr[z];

	if ((a <= b and b <= c) or (c <= b and b <= a))
		return y;
	else if ((b <= a and a <= c) or (c <= a and a <= b))
		return x;
	else
		return z;

}


int partitionQ3(int a[], int left, int right, long& count)
{
	int median = indexOfMedian(a, left, right);
	swap(a[left], a[median]);

	int pivot = a[left];
	int s = left;

	for (int i = left + 1; i <= right; i++) {
		if (a[i] < pivot) {
			s = s + 1;
			//swap(a[s], a[i]);
	}
		count++;
	  }
	  swap(a[s], a[left]);

	  int i = left;
	  int j = right;

	  while (i<s and j>s) {
		while (a[i] <= pivot) i++;
		while (a[j] > pivot) j--;
		if (i<s and j>s) swap(a[i++], a[j--]);
		count++;
	  }

	  return s;
}

void quicksort(int a[], int left, int right, long& count)
```

```
{
	if (left < right) {
		int s = partitionQ3(a, left, right, count);
		quicksort(a, left, s - 1, count);
		quicksort(a, s + 1, right, count);
	}
}

void printArray(int a[], int size)
{
	for (int i = 0; i < size; i++) {
		cout << a[i] << "\t";
		if (i % 14 == 13) cout << endl;
	}
	cout << endl << endl;
}
```

Q4) Generate large arrays (of size $n = 10^5$) with different distributions reflecting the algorithms' best, average, and worst cases.
Fill in the following table and compare the algorithms' efficiency theoretically and experimentally. Analyze the results and state your implications.

Code for the Main(): /*note: We increased the memory allocation to the stack by increasing the stack reserve and commit sizes for the project file in order to accommodate the large arrays of size $n = 10^5$.*/

For Testing the Best and Average Cases:

```
int main() {

	srand(time(NULL) + rand());

	long count = 0;
	int size = 100000;
	int start = 0;
	int end = 99999;
	int* array = new int[size];

	for (int i = 0; i < size; i++) {
		array[i] = rand() % 30000;
	}

	cout << "\nBefore Quicksort:\n";
	printArray(array, size);
```

```
	quicksort(array, start, end, count);
	cout << "\nAfter Quicksort:\n";
	printArray(array, size);

	cout << "The total number of comparison operations are: " << count << endl;



	delete[] array;

	return 0;
}
```

/* note: The srand() is used to change the seed each time, and get different results per run. Of course, you know this, but just to explain our use of it. :)
Also, the RAND_MAX = 32767 is quite weird to me, I modded it with 30,000 which doesn't make any difference, but I would have liked to test the function with an unsorted array of unique elements. :(*/

For Testing the Worst Case:

```
int main() {

	srand(time(NULL) + rand());

	long count = 0;
	int size = 100000;
	int start = 0;
	int end = 99999;
	int* array = new int[size];

	for (int i = 0; i < size; i++) {
		array[i] = i+1;
	}

	cout << "\nBefore Quicksort:\n";
	printArray(array, size);

	quicksort(array, start, end, count);
	cout << "\nAfter Quicksort:\n";
	printArray(array, size);

	cout << "The total number of comparison operations are: " << count << endl;
```

```
        delete[] array;

        return 0;
}
```

/* note: It is basically the same code, just generating a sorted array instead (from 1 to the size). */

Theoretical Analysis (for all algorithms):

Best Case: When the pivot is partitioned to be exactly in the middle. This is Θ(nlog(n)).

100,000*log(100,000) = 1,660,964

Worst Case: When the array is already sorted. This is $\Theta(n^2)$.

0.5*100,000*100,000 = 5,000,000,000

/* note: It is specifically about $0.5n^2$ */

Average Case: When the array elements are randomly generated. This is Θ(nlog(n)).

1.39*100,000*log(100,000) = 2,308,740

/*note: It is specifically around 1.39*n*log(n) but just for the sake of simplicity, we compare the best and average cases together in experimental efficiency. */

	Quicksort in Q1	
	Theoretical Efficiency (expected)	**Experimental Efficiency** (average of 5 runs)
Best Case Complexity	1,660,964	2,581,731
Average Case Complexity	2,308,740	
Worst Case Complexity	5,000,000,000	4,999,950,000

Experimental Calculations (for Q1):

Best & Average Cases:

1st Run: 2,478,548
2nd Run: 2,649,201
3rd Run: 2,585,715
4th Run: 2,583,314
5th Run: 2,611,880

Total: 12,908,658
Average: 2,581,731

Worst Case:

The runs are all the same since I used a fixed sorted array for testing
where array[i] = i+1;
So it picks the smallest element (leftmost) as the pivot each time.

	Quicksort in Q2	
	Theoretical Efficiency (expected)	**Experimental Efficiency** (average of 5 runs)
Best Case Complexity	1,660,964	2,086,893
Average Case Complexity	2,308,740	
Worst Case Complexity	5,000,000,000	1,503,410

Experimental Calculations (for Q2):

Best & Average Cases:

1st Run: 2,054,521
2nd Run: 2,089,441
3rd Run: 2,137,681
4th Run: 2,114,756
5th Run: 2,038,067

Total: 10,434,466
Average: 2,086,893

Worst Case:

The runs are all the same since I used a fixed sorted array for testing

where array[i] = i+1;

This means that the median of the first, last, and median elements will always be the same. This means that the median of the given array is always the pivot.

	Quicksort in Q3	
	Theoretical Efficiency (expected)	**Experimental Efficiency** (average of 5 runs)
Best Case Complexity	1,660,964	2,044,724
Average Case Complexity	2,308,740	
Worst Case Complexity	10,000,000,000	1,813,463

<u>Experimental Calculations (for Q3):</u>

Best & Average Cases:

1st Run: 2,036,800
2nd Run: 2,069,003
3rd Run: 2,063,333
4th Run: 2,033,649
5th Run: 2,020,839

Total: 10,223,624
Average: 2,044,724

Worst Case:

1st Run: 1,769,685
2nd Run: 1,772,607
3rd Run: 1,812,455
4th Run: 1,953,502
5th Run: 1,759,067

Total: 9,067,316
Average: 1,813,463

Conclusions:

I tried to get the best and average experimental efficiencies similar to the theoretical when I chose which comparison operators to count. I could increase this count by counting the comparison when incrementing i or decrementing j in the partition function. I could decrease this by only counting the comparison to count the pivot's correct spot within the array. But ultimately, I just had to keep this choice consistent within the three partition styles.

To conclude, the main idea is that the **best case** would always be the same, since the pivot should be right down the middle each time. This type of array would be difficult to produce and so the experimental best case is grouped with the experimental average case. The **average cases**, from observations, are similar for Q2 and Q3 but better than the average case for Q1. This just shows that picking a median of three numbers can give you a better pivot than just picking one number and sticking with it. The main difference, however, is the worst case. The **worst case** is when the array is sorted, and so you have to partition the n elements into another n-1, then n-2, and so on. Which is far from O(logn) complexity, and is within $O(n^2)$ complexity. This is shown in Q1. As for Q2 and Q3, it eliminates the occurrence of this worst case since it picks the pivot as the median of three elements, so it will never be just the smallest element (leftmost of a sorted array).

Now, by observation, we can see that the worst cases of Q2 and Q3 are better than their respective average cases. This can be explained by this loop:

```
while (i<s and j>s) {
                while (a[i] <= pivot) i++;
                while (a[j] > pivot) j--;
                if (i<s and j>s) swap(a[i++], a[j--]);
                count++;
        }
```

Since I chose to count this loop, which is affecting mostly the average cases but not the worst cases, there is a small difference (within about 300,000 comparisons). This was eliminated from the worst case since in a sorted array, the left side of the pivot is already smaller than the pivot, and the right side is already greater than the pivot, so this loop is unnecessary. Basically if we omit this loop from the counting process of the average case, we get the same result as we would from the worst case runs. This is another approach to counting the experimental efficiency of the algorithm.

The worst case for Q2 is the lowest that you can get for the three pivot selections, since it will always choose the middle element as the pivot for the sorted array, and you will get very close to nlogn efficiency. In this case, it is even lower, but there are factors within the program that are unaccounted for in the counting process.

Overall. Q2 and Q3 eliminates the worst case that Q1 shows us and will consistently operate in a logarithmic complexity in any case.