Abdullah Tahir

21L-5419

Assignment # 2

Design and analysis with Divide and Conquer

Problem 1. Comparison of Sorting Algorithms

(1) Implement

i. Merge Sort: One with division in two parts each of size (n/2) already implemented in Assignment 1

Solution:

```
void Merge(int arr[], int low, int mid, int high)
{
    int n_a = mid - low + 1;
    int n_b = high - mid;
    int* arr_a = new int[n_a];
    int* arr_b = new int[n_b];
    for (int i = 0; i &lt; n_a; i++)
    {
            arr_a[i] = arr[low + i];
    }
    for (int i = 0; i &lt; n_b; i++)
    {
            arr_b[i] = arr[mid + 1 + i];
    }
    int i = 0;
    int j = 0;
    int k = low;
    while (i &lt; n_a &amp;amp;&amp;amp; j &lt; n_b)
    {
            if (arr_a[i] &lt;= arr_b[j])
            {
                    arr[k] = arr_a[i];
                    i++;
            }
            else
            {
                    arr[k] = arr_b[j];
                    j++;
            }
            k++;
    }
    while (i &lt; n_a)
    {
            arr[k] = arr_a[i];
            i++;
            k++;
```

```
    }
    while (j &lt; n_b)
    {
            arr[k] = arr_b[j];
            j++;
            k++;
    }
    delete[] arr_a;
    delete[] arr_b;

}
void mergeSort(int arr[], int low, int high)
{
    if (low &lt; high)
    {
            int mid = low + (high - low) / 2;
            mergeSort(arr, low, mid);
            mergeSort(arr, mid + 1, high);
            Merge(arr, low, mid, high);
    }
}
```

ii. Quick Sort: In place (Pivot as random element)

Solution:

```
int Partition(int A[], int p, int q)
{
   int pivot = p+(rand() % (q-p+1));
   swap(A[p], A[pivot]);
   int x = A[p];
   int i = p - 1;
   for (int j = i + 1; j <= q; j++)
    {
      if (A[j] <= x)
              {
         i++;
         swap(A[i], A[j]);
      }
   }
   swap(A[i], A[p]);
   return i;
}

void QuickSort(int A[], int p, int q)
{
   if (p < q)
    {
      int m = Partition(A, p, q);
      QuickSort(A, p, m - 1);
```

```
        QuickSort(A, m + 1, q);
    }
 }
```

iii. Quick Sort: In place (Pivot as Median of three elements randomly chosen)

Solution:

```
int Partition(int A[], int p, int q)
{
   int l = p+(rand() % (q-p+1));
   int m = p+(rand() % (q-p+1));
   int h = p+(rand() % (q-p+1));
   int pivot = (l+m+h)/3;
   swap(A[p], A[pivot]);
   int x = A[p];
   int i = p - 1;
   for (int j = i + 1; j <= q; j++)
      {
      if (A[j] <= x)
               {
         i++;
         swap(A[i], A[j]);
      }
   }
   swap(A[i], A[p]);
   return i;
}

void QuickSort(int A[], int p, int q)
{
   if (p < q)
      {
      int m = Partition(A, p, q);
      QuickSort(A, p, m - 1);
      QuickSort(A, m + 1, q);
   }
}
```

(2) Compare three algorithms for an array of size 100, to size 100,000 as given in following table. a. Generate a random array, A, of n integers.

b. Make three identical copies of A: A1, A2 and A3.

c. Execute and compute their execution times T1, T2 and T3. Each entry should show the running time in milliseconds. What time difference do you see in each case?

n	Merge Sort (n/2) T1	Quick Sort Random Pivot T2	Quick Sort Median of three T3

100	1.3748 ms	0.2507 ms	0.3224 ms
1,000	11.0413 ms	8.7789 ms	4.4228 ms
10,000	112.81 ms	80.8078 ms	100.888 ms
100,000	1225.84 ms	2924.37 ms	3055.88 ms

Problem 2. Radix Sort

In this problem, you will implement the Radix sort for sorting of the strings containing first and last names of individuals in lexicographic Alphabetical order. The order input will be provided by user (A-Z) or (Z-A). The radix sort will use the count sort to sort the individual columns of alphabets.

Solution:

```
		void countSort(string A[], int n, int pos, bool ascending)
{
	string* B = new string[n];
	const int count_size = 128;
	int count[count_size] = { 0 };

	for (int i = 0; i < n; i++)
	{
		int charCode = A[i][pos];
		int index;

		if (ascending)
		{
			index = charCode;
		}
		else
		{
			index = count_size - 1 - charCode;
		}

		count[index]++;
	}

	for (int i = 1; i < count_size; i++)
	{
		count[i] += count[i - 1];
	}

	for (int i = n - 1; i >= 0; i--)
	{
		int charCode = A[i][pos];
		int index;
```

```
                if (ascending)
                {
                        index = charCode;
                }
                else
                {
                        index = count_size - 1 - charCode;
                }

                B[count[index] - 1] = A[i];
                count[index]--;
        }

        for (int i = 0; i < n; i++)
        {
                A[i] = B[i];
        }
}

void radixSort(string A[], int n, bool ascending)
{
        int Max = 0;
        for (int i = 0; i < n; i++)
        {
                Max = max(Max, (int)A[i].length());
        }

        for (int pos = Max - 1; pos >= 0; pos--)
        {
                countSort(A, n, pos, ascending);
        }
}
```

Problem 3. Design and Analysis

(1) The kth Smallest Element in a Sorted array is a number greater than or equal to exactly the k-1 elements of the array. The problem is to find the kth smallest element from an unsorted array.

For Example, if input array A = {4, 12, 5, 20, 8, 13} and k = 3 then kth smallest is 8.

i. Design a brute force algorithm to find and return the kth smallest element from an unsorted array of integers. What is the time complexity of this algorithm in terms of Big-O?

Solution:

```
int findKthSmallest(int arr[], int n, int k)
{
	for (int i = 0; i < k; i++)
	{
		for (int j = 0; j < n - i - 1; j++)
		{
			if (arr[j] > arr[j + 1])
			{
				int temp = arr[j];
				arr[j] = arr[j + 1];
				arr[j + 1] = temp;
			}
		}
	}

	return arr[k - 1];
}
```

ii. Design an algorithm by modifying the quick sort algorithm to find and return the kth smallest element from an unsorted array of integers in O(n). What is the recurrence of this algorithm and solve the recurrence to find the Big-O.

Solution:

```
int Partition(int A[], int p, int q)
{
int x = A[p];
int i = p;
for (int j = i + 1; j <= q; j++)
{
	if (A[j] <= x)
	{
		i++;
		swap(A[i], A[j]);
	}
}
swap(A[i], A[p]);
return i;
}

int QuickSelect(int A[], int p, int q, int k)
{
if (p == q)
{
	return A[p];
}

int m = Partition(A, p, q);
```

```
int length = m - p + 1;

if (k == length)
{
        return A[m];
}
else if (k < length)
{
        return QuickSelect(A, p, m - 1, k);
}
else
{
        return QuickSelect(A, m + 1, q, k - length);
}
}

int FindKthSmallest(int arr[], int n, int k)
{
 srand(time(NULL));
 return QuickSelect(arr, 0, n - 1, k);
}
```

iii. Design an algorithm by modifying the counting-sort algorithm to find and return the kth smallest element from an unsorted array of integers in O(n).

Solution:

```
int findKthSmallest(int arr[], int n, int k, int kth)
{
        int *count = new int[k + 1];
        int *temp = new int[n];

        for (int i = 0; i < n; i++)
        {
                count[arr[i]]++;
        }
        for (int i = 1; i <= k; i++)
        {
                count[i] += count[i - 1];
        }
        for (int i = n - 1; i >= 0; i--)
        {
                temp[--count[arr[i]]] = arr[i];
        }
        for (int i = 0; i <= n; i++)
        {
                arr[i] = temp[i];
        }

        delete[] count;
```

```
        delete[] temp;

        return arr[kth - 1];
    }
```

(2) Selection sort is an unstable sorting algorithm, how can you convert it into a stable version in O(n) extra time?

Solution:

```
void Sort(int arr[], int n)

{

    int i, j, min_idx;

    for (i = 0; i < n - 1; i++)

    {

        min_idx = i;

        for (j = i + 1; j < n; j++)

        {

            if (arr[j] < arr[min_idx])

            {

                min_idx = j;

            }

        }

        if (min_idx != i)

        {

            int temp = arr[min_idx];

            for (j = min_idx; j > i; j--)

            {

                arr[j] = arr[j - 1];

            }

            arr[i] = temp;
```

```
        }


    }

}
```

i. Provide the pseudo code with brief explanation of your modification.

Solution:

```
ModifiedSelectionSort(arr, n)

    for i from 0 to n-1
        min_idx = i

        for j from i+1 to n-1
            if arr[j] < arr[min_idx]
                min_idx = j

        if min_idx is not equal to i
            temp = arr[min_idx]

            for j <- min_idx to i, j--
                arr[j] = arr[j-1]

            arr[i] = temp
```

Explanation: The working in second **if condition** is modified. In this portion the array elements are moved to their next index starting loop in reverse order. The loop works every time from i to min_idx. If the min_idx is greater than i then this loop will work else it will not work. After moving i to min_idx elements to their next index the arr[min_idx] is placed at ith index. For each iteration this process repeats and finally the array is sorted and our algorithm becomes stable.

ii. Perform the selection sort with your proposed modification on the following array A of seven elements (integer-key value pairs).

(10, a)	(5, a)	(4, a)	(10, b)	(3, a)	(4, b)	(1, a)

Solution:

Initial Array: 10a 5a 4a 10b 3a 4b 1a

i=0, min_idx=0:
Array: 1a 10a 5a 4a 10b 3a 4b
i=1, min_idx=6:

i=1, min_idx=1:
Array: 1a 3a 10a 5a 4a 10b 4b

i=2, min_idx=5:

i=2, min_idx=2:
Array: 1a 3a 4a 10a 5a 10b 4b
i=3, min_idx=4:

i=3, min_idx=3:
Array: 1a 3a 4a 4b 10a 5a 10b
i=4, min_idx=6:

i=4, min_idx=4:
Array: 1a 3a 4a 4b 5a 10a 10b
i=5, min_idx=5:

For i = 5 and i = 6 no changes will be made because i will always equal to min_idx in these iterations and also the arr[i] < arr[min_ind] will also be false in these iterations so no more changes

Array after sorting: 1a 3a 4a 4b 5a 10a 10b