Programming Assignment 1
Sorting Algorithms

CSCI 3412 001
Fall 2018
Kenneth Maguire

Part 1 - Introduction and Selection of Sorting Algorithms

For this assignment, I implemented 3 algorithms: one with runtime $O(n^2)$, one with runtime O(n lg n) and one with runtime O(n). The algorithms I selected are *Selection Sort* for $O(n^2)$, *Quick Sort* for O(n lg n), and *Counting Sort* for O(n). I initially implemented *Selection Sort* and *Quick Sort* in Python 3.7, but after struggling with the recursion limit in Python, and the fact that sorting 10^6 values couldn't complete after waiting 20 hours, I figured I should use a more efficient language, and switched to C++. My *Selection Sort* code searches for the minimum element in the vector, and then places that value at the beginning. My *Quick Sort* algorithm partitions the array into two sub-arrays, then recursively calls itself to partition and sort through the smaller arrays. I implemented the *Tail Recursive QuickSort* algorithm (from pg 188 in the Algorithms textbook) in place of the original *Quick Sort* because it removes one of the recursive calls by adding an "iterative control structure". My *Counting Sort* algorithm receives the maximum value that can be searched for in the input, counts the number of times (0-maxVal) shows up, then counts how many values less than or equal to any index in my array exist, and finally places each value into a new (sorted) array by taking the final value in the unsorted list,, finding it's spot in the new sorted array by how many values are less than or equal to that index, and then decrements the number of values less than or equal to that index (pg 196 from algorithms book illustrates this idea).

Problem Definition

The sorting problem is defined as:

Input: A sequence of n numbers <a1, a2, …, a(n)>

Output: A reordering <a1', a2', …, a(n)'> of the input sequence such that a1' <= a2' <= …. <= a(n)'

From this definition, we can find sorting algorithms that can take the input of a sequence of numbers and reorder them such that the values are in sorted order, using comparisons, divide-and-conquer, information about the set of number, or a combination of those options.

The problem asks for us to implement sorting algorithms, such that each one has a specific running time of either $O(n^2)$, O(n lg n) or O(n), then we'll need to show each algorithms' rate of growth by counting the number of comparisons and exchanges after n = (10^0 through 10^6) and plotting them. Then we'll compare their rate of growth given different set of 1000 integers. We can only do this for $O(n^2)$ and O(n lg n) since the *Counting Sort* algorithm doesn't use comparisons or exchanges to sort the input values.

Part 2 - Pseudo-Code

Selection Sort

The Selection Sort algorithm uses to iterators to find the minimum element in the array, and then place that element at the beginning. It starts by assuming the first value from A(j…n) is the minimum, then changes the index to any new minimum value if one or more are found up until A(n) is reached. If a new minimum value is found, then it swaps A[j] with the new minimum, A[min]. Otherwise it assumes the A[j] is the min and increments j.

Selection-Sort(**A**) (algorithm found @ **https://en.wikipedia.org/wiki/Selection_sort**)

```
1       j = 0
2       min = j
3       for j=0 to A.length-1
4               for i = j+1 to A.length
5                       if A[i] < A[min]
6                               min = i
7       if min != j
8               exchange A[j] with A[min]
```

This function returns the sorted array A after reaching A.length-1 in the outer loop.

Quick Sort (and Partition)

The Quick Sort algorithm uses the Partition algorithm to split the input sequence into lower and upper partitions, so that quick sort can recursively call itself and sort each partition. I implemented the Tail-Recursive algorithm because it helps prevent a recursion limit being hit due to a large stack which can happen in the initial Quicksort algorithm due to two recursive calls to quick sort.

Tail-Recursive-Quick-Sort(**A, p, r**) (algorithm found @ 188 in Algorithms textbook)

```
        while p < r
                q = Partition(A,p,r)
                Tail-Recursive-Quick-Sort(A, p, q-1)
                p = q + 1
```

Partition(**A,p,r**) (algorithm found @ 171 in Algorithms textbook)

```
        x = A[r] //the pivot point
        i = p - 1
        for  j = p  to r - 1
                if A[j] <= x
                        i = i + 1
                        exchange A[i] with A[j]
        exchange A[i+1] with A[r]
        return i + 1
```

The partition algorithm selects the highest index value of A as the pivot point, and partition the set A into lower and upper partitions, then quick sort is called recursively to sort the smaller subarray. After using partition to divide the array into subarrays, and calling quick sort recursively to sort the subarrays, the array A[p…r] is sorted.

Counting Sort

The Counting Sort Algorithm uses 3 (A,B,C) separate arrays to sort the input sequence. Array A is the original unsorted input sequence, B holds the sorted list and is only used in lines 8 through 10, and C is used for temporary working storage so that a count can be kept of each time a number occurs in the input sequence, and then organizes the input sequence such that the index can be found based on how many numbers less than or equal to a given index value appear.

Counting-Sort(**A,B,k)** (algorithm found @ pg 195 in Algorithms textbook)

```
1       let C[0..k] be a new array
2       for i = 0 to k
3               C[i] = 0
4       for j = 1 to A.length
5               C[A[j]] = C[A[j]] + 1
6       for i = 1 to k
7               C[i] = C[i] + C[i -1]
8       for j = A.length downto 1
9               B[C[A[j]]] = A[j]
10              C[A[j]] = C[A[j]] - 1
```

This function returns the sorted array B after completing lines 8 - 10.

Part 3 - Static Analysis

For these algorithms, they've been selected such

Part 4 - Implementation

All of the algorithms were coded in Python first, and then due to efficiency issues, I had to re-code them in C++. I've separated the algorithms into their own header files, and I have a main file that just receives the header for the algorithm it wants to process, calls a readfile() function that returns a vector containing the input sequence from the file, and then calls the function for the algorithm it wants to use to sort the input sequence.

SelectionSort.h

```
/*
Algorithm found at the following link https://en.wikipedia.org/wiki/Selection_sort, reimplemented from
SelectionSort.py
This program was compiled with c++ 14 standard
*/

void selectionSort(vector<int> &_unsortedList, int &_comp, int &_exch)
{
  int i = 0;
  int j = 0;
  int min = 0;




  for(j = 0; j < _unsortedList.size(); j++)
  {
     min = j;          //assume j is the smallest value
     for(i = j+1; i < _unsortedList.size(); i++)     //search array for smaller value than j
     {

        _comp += 1;
        if(_unsortedList[i] < _unsortedList[min])   //if smaller value found
        {
          min = i;                              //set new index for smaller value
        }
     }

     if(min != j)                               //if min isn't set to first value, then swap
     {
       _exch += 1;
       int tempInt;
       tempInt = _unsortedList[j];
       _unsortedList[j] = _unsortedList[min];
       _unsortedList[min] = tempInt;
     }
  }
}
```

QuickSort.h

```
#include "../../ReadFile.h"

//function declarations allow quicksort to call partition

int partitionQS(vector<int> &_partList, int _p, int _r, int &_comp, int &_exch);
void t_r_quickSort(vector<int> &_unsortedList, int _p, int _r, int &_comp, int &_exch);



void t_r_quickSort(vector<int> &_unsortedList, int _p, int _r, int &_comp, int &_exch)
{

  while(_p < _r)
  {
    int q = partitionQS(_unsortedList, _p, _r, _comp, _exch);
    t_r_quickSort(_unsortedList, _p, (q-1), _comp, _exch);      //recursive calls to t_r_quickSort
    _p = q + 1;                                                 //iterative method to mitigate too many recursive function calls

  }


}
int partitionQS(vector<int> &_partList, int _p, int _r, int &_comp, int &_exch)
{
    int pivot = _partList[_r];      //set pivot to last index in array
    int i = _p;                     //i will be top of i p through i
    for(int j=_p; j < _r; j++)      //in this loop, p through i are <= pivot, i through j are >= pivot, j through r are
unrestricted (source pg 173 Algorithms txt)
    {
        _comp += 1;
        if(_partList[j] <= pivot)   //if partList at j is <= pivot, add it to the lower partition, i++
        {
            _exch += 1;
            int tempInt1 = 0;
            tempInt1 = _partList[i];
            _partList[i] = _partList[j];
            _partList[j] = tempInt1;
            i = i+1;
        }
    }
    _exch += 1;
    int tempInt2 = 0;           //place pivot between the two partitions
    tempInt2 = _partList[i];    //
    _partList[i] = _partList[_r];
    _partList[_r] = tempInt2;
    return i;

}
```

CountingSort.h

```
#include "../../ReadFile.h"

void countingSort(vector<int> _unsortedList, vector<int> &_sortedOutput, int _maxVal)
{
   vector<int> auxArray(_maxVal);

   //assign 0 to all indexes 0 through maxVal
   for(int i = 0; i < _maxVal; i++)
   {
      auxArray[i] = 0;
   }
   //fill auxArray[i] with values equal to i
   for(int j = 1; j < _unsortedList.size(); j++)
   {


      auxArray[_unsortedList[j]] = auxArray[_unsortedList[j]] + 1;
   }
   //fill auxArray with the number of values less than or equal the indexes value
   for(int i = 1; i < _maxVal; i++)
   {

      auxArray[i] = auxArray[i] + auxArray[i-1];
   }
   //find the end array element value, fill the _sortedOutput with the value at the index specified (number of ints less
than or equal to the index val)
   for(int j=(_unsortedList.size()-1); j > 0; --j)
   {
      _sortedOutput[auxArray[_unsortedList[j]]] = _unsortedList[j];
      auxArray[_unsortedList[j]] = (auxArray[_unsortedList[j]] - 1);
   }

}
```

Main File (Quicksort.cpp in this example) (used for all algorithm functions with the include changed and one line changed to call the appropriate function, in this example the function is for quick sort"

```
#include "QuickSort.h"
int main()
{

                                //unsortedList that will read from file
  vector<int> unsortedList;
                                //count is used to test the funciton at each power of ten up to 10^6
  int count [7] = {1,10,100,1000,10000, 100000, 1000000};



//this loop
  for(int i = 0; i != 7; i++)
  {
                                          //fileNum calls appropriate file
                                           //comp is for comparisons returned from function
                                           //exch is for exchanges returned from function
      int fileNum = 1;
      int comp = 0;
      int exch = 0;
                                          //calls read from file function
      unsortedList = readFile(fileNum);
                     //prints the unsorted list to confirm the list is refreshed for each iteration of the
                     //outer for loop
      cout << "\n\n";
      cout << "The unsorted list is: ";
      for(int j = 0; j < unsortedList.size(); j++)
      {
        cout << unsortedList[j] << " ";
      }

                       //portionList contains a slice of the unsortedList, to be passed to the sorting
//algorithms
                     //then we can get comps and exch for each sorting algorithm at each power of
//10
      vector<int> portionList(count[i]);
      copy_n(unsortedList.begin(), count[i], portionList.begin());

                                //call sort function
      t_r_quickSort(portionList, 0, (portionList.size()-1), comp, exch);
                                  //example for quick sort

                                  //prints the sorted list to confirm that list is sorted
      cout << "\n\n";
      cout << "The sorted list is: " << endl;
      for(int j = 0; j < portionList.size(); j++)
      {
        cout << portionList[j] << endl;
      }
      cout << "\n\n";
                                   //this is used for later plotting and analysis
      cout << "\nThe size of the sorted list is: " << portionList.size() << endl;
      cout << "\nThe number of comparisons is: " << comp << endl;
      cout << "\nThe number of exchanges is: " << exch << endl;
      cout << "\n\n" << endl;
```

```
        cout << i << endl;
    }
        return 0;

}
```

ReadFile.h

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
using namespace std;

vector<int> readFile(int _fileNum)
{

    ifstream inFile;
    vector<int> dataSet(0);
    string datFiles[] = {"duplicate.txt", "nearly-sorted.txt", "nearly-unsorted.txt", "one-million-randoms.txt",
"shuffled.txt", "sorted.txt", "unsorted.txt"};
    string filePath = "data/" + datFiles[_fileNum];   //add strings together for folder path

    cout << filePath << endl;
    inFile.open(filePath);
    //test if file failed to open
    if(inFile.fail())
    {
      cout << "file failed to open" << endl;

    }

    inFile.ignore();
    string dummyString;
    getline(inFile, dummyString); //skip first 2 lines
  //  string dummyString;
    getline(inFile, dummyString);
    //read in file into vector until the end of the file is reached
    while(!inFile.eof())
    {

        int val = -1;
        inFile >> val;

        if(val != -1)
        {
          dataSet.push_back(val);
        }
        if(inFile.eof())
        {
          break;
        }
    }

    inFile.close();
    //close file
```

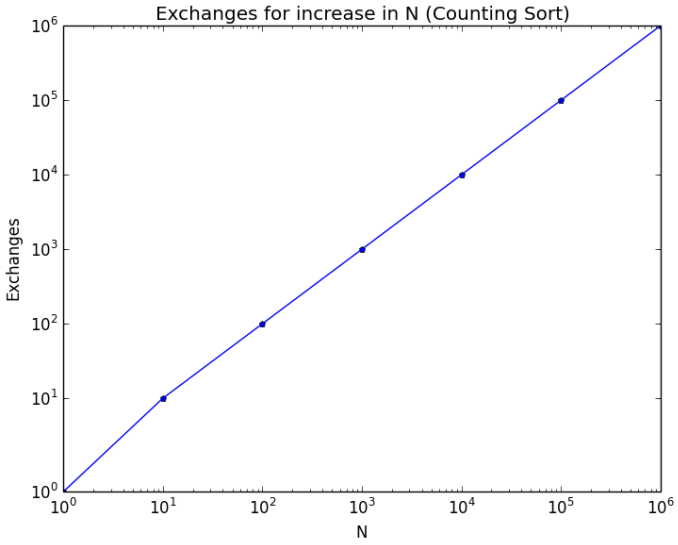
```
    cout << dataSet.size() << endl;
  /*  for(int i = 0; i < dataSet.size(); i++)
    {
        cout << dataSet[i] << endl;

    }

*/
    //return vector
    return dataSet;
}
```

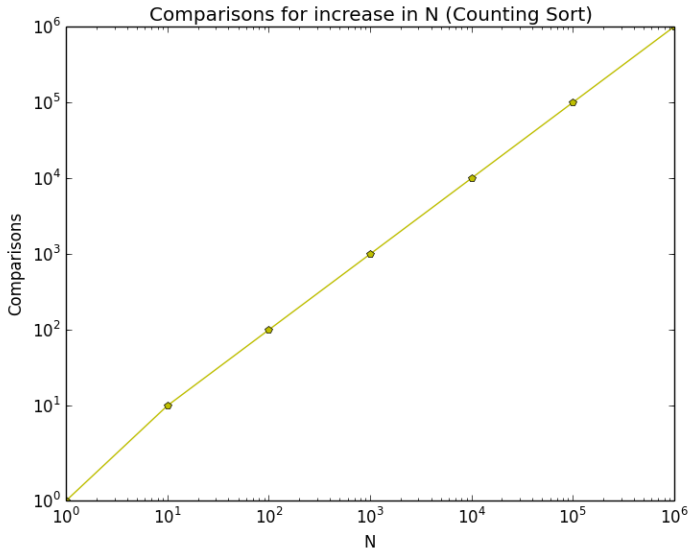
Part 5 - Analysis

Analysis of Growth

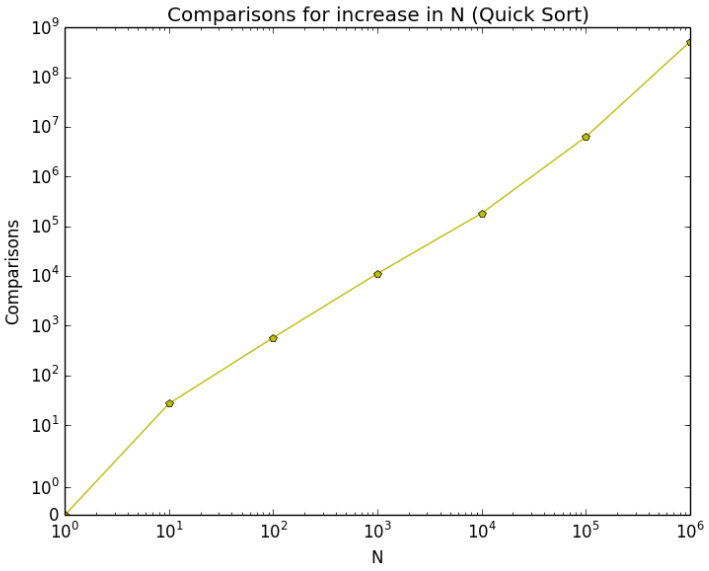
We will give the analysis of the Count Sort, Selection Sort, and Quick Sort for how many exchanges and comparisons each algorithm does given input sizes n = 10^0 through 10^6 using the one-million-randoms.txt file. I've plotted the functions using a log scales for both x and y to better represent the data. We can expect Count Sort (at O(n)) to perform the best, Quick Sort (at O(n lg n)) to perform slightly worse than Count Sort, and Selection Sort (at O(n^2) to perform the worst.

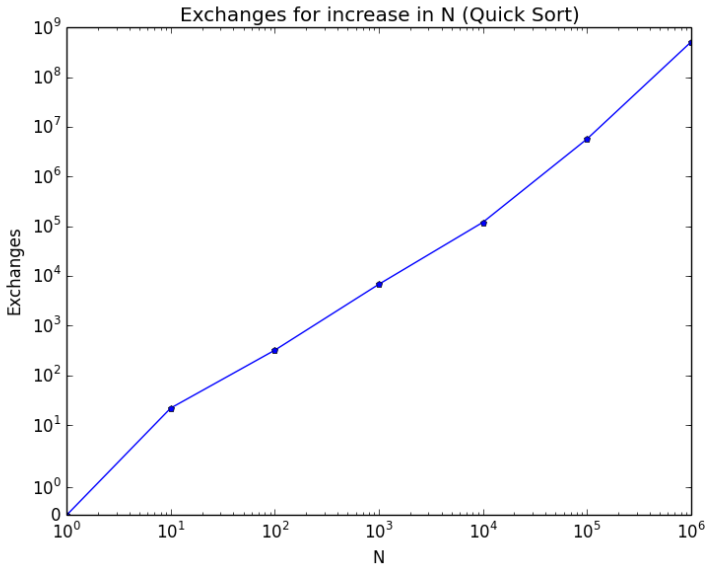
Below are the graphs of comparisons and exchanges for Counting Sort





Below are the graphs of comparisons and exchanges for Quick Sort





Sources:

Intro to Algorithms textbook by Cormen, Leiserson, Rivest, and Stein.

https://en.wikipedia.org/wiki/Selection_sort