CMPT 435 - Fall 2020 - Dr. Labouseur

Assignment Two – Sorting Algorithms
Applying Sorting Algorithms to a list of Strings

Sam Alcosser
Samuel.Alcosser1@Marist.edu

October 1, 2020

Contents

1 Introduction

This document describes four different methods for sorting items, and implementations of those methods, or algorithms, in C++. The four methods are known as insertion sort, selection sort, merge sort, and quick sort. Merge sort and quick sort are more efficient as they implement recursion within their sorting processes, as seen in the chart above. The reason for this will be discussed later on.

2 Ontology of the `Sort` class

The `Sort` class seen below contains all of the different sorts as static functions, with the idea that the main running function would then be able to have easy access to the methods. This class contains the sorting algorithms, helper functions such as `swap()` and `printList()`, and helper functions for the two recursive sorting algorithms.

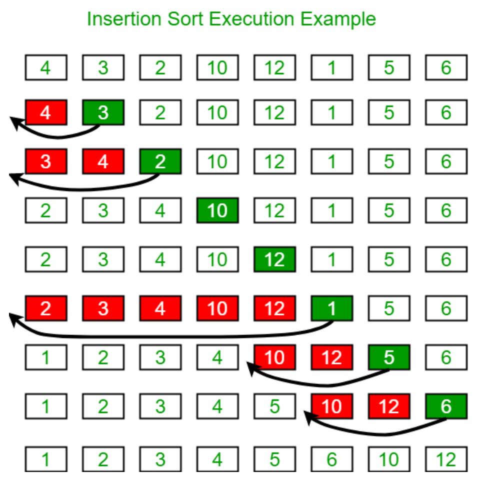
```cpp
#pragma once
#include <iostream>
#include <string>

class Sort {
public:


        static void swap(std::string &a, std::string &b);
        static void printList(std::string s[], int n);
        static int Insertion(std::string s[], int n);
        static int Selection(std::string s[], int n);
        static int subMSort(std::string s[], int l, int m, int r, int cnt);
        static void mergeSort(std::string s[], int lInd, int rInd, int &cnt);
        static void quickSort(std::string s[], int p, int r, int &cnt);
        static int Partition(std::string s[], int p, int r, int &cnt);

};
```

A Quick Note

To make sense of these algorithms, for each one I will start by giving an overview of the process including a diagram. After, I will show my implementation of the algorithm followed by a detailed description of each step with references to specific lines in the code. This is done to avoid the clutter of describing the algorithm itself and my implementation of said algorithm at the same time.

3 Insertion Sort

3.1 Understanding Insertion Sort



Insertion sort operates by stepping through each element of a list and comparing as it goes. Each current element will compare itself to each element that was before it in the order of the list, and swap. The site *Programiz.com* has a better description of the method.

> Insertion sort works similarly as we sort cards in our hand in a card game.
>
> We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.
>
> A similar approach is used by insertion sort.
>
> Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

–programiz.com

3.2 Implementation of Insertion Sort

Insertion sort is probably the simplest of the sorting algorithms out of the four. This does not mean however that it is the most efficient, as will be evident later on.

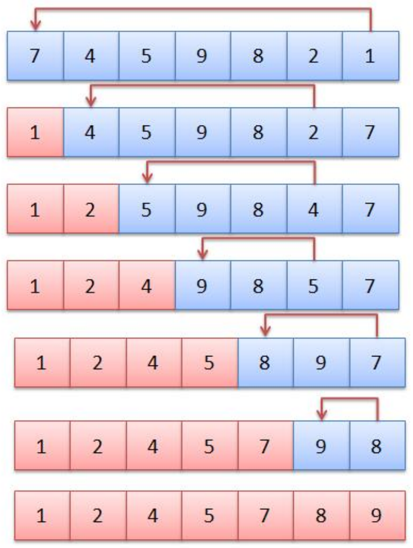
```
int Sort::Insertion(std::string s[], int n)
{
        int counter = 0;
        std::string last;
5          for (int i = 1; i < n; i++)
6          {
7                  last = s[i];
8                  int j = i - 1;
9                  while (j >= 0 && s[j] > last)
10                 {
11                         counter++;
12
13                         s[j + 1] = s[j];
14
15                         j = j - 1;
16                 }
17
18                 s[j + 1] = last;
19         }
20
21         //printList(s, n);                  //debugging
22         return counter;
23 }
```

Code Breakdown

1. For each index in the array `s[]` after index one(Line 5)
2. Compare the current element to the one before it (line 9)
3. If the current element is smaller than the one before it, keep comparing until the current element is bigger than the next one being compared. At that point, shift all elements that were previously compared up by one. (lines 9-16)
4. Place the current element in its place which was created by the movement of the other elements (line 18)
5. Continue the same process until the end of the array.

4 Selection Sort

4.1 Understanding Selection Sort



Selection sort is a sorting algorithm similar to Insertion sort. Although, due to its differences, it is usually slower and less efficient. The process is quite simple. For every index in the array, if there is any value further in the array that is smaller than the current index, swap the current index with the lowest of those values.

4.2 Implementation of Selection Sort

The simplicity of selection sort can be seen below, as the total number of lines of code is even less than insertion sort.

```
int Sort::Selection(std::string s[], int n)
{
        int low;
        int counter = 0;
        for (int i = 0; i < n; i++)
        {
                low = i;
                for (int j = i + 1; j < n; j++)
                {
                        counter++;

                        if (s[j] < s[low])
                        {
                                low = j;
                        }
16                  }
17                  Sort::swap(s[low], s[i]);
18          }
19          //printList(s, n);                //debugging
20          return counter;
21  }
```

Code Breakdown

1. for each element in the array `s[]` starting at index 0 (lines 5,6):
2. Start by assuming that this current index `i` is the lowest value by assigning the index number to the `low` variable (line 7).
3. For each element after `i`, compare it to element `s[low]` (lines 8-12)
4. Whenever a lower element is found, mark that as the new lowest index (line 14). Now, all elements after this new element will be compared to the new lowest value, until the true lowest value is found.
5. Once the inner iterator reaches the end of the array (line 8), swap the value of the current element `s[i]` with the newly found lowest element denoted by `s[low]` (line 17).

5 Merge Sort

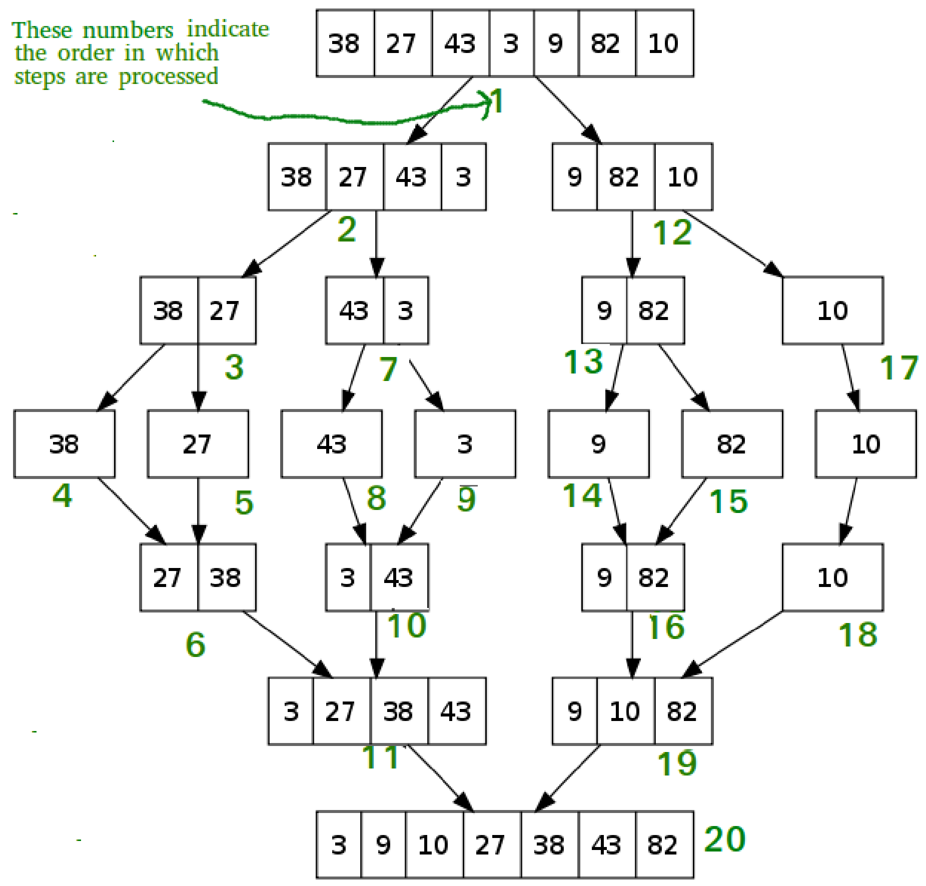
5.1 Understanding Merge Sort

Refresher on Recursion

Before we progress any further, we should quickly refresh oursleves on the concept of recursion. Simply put, when a function is recursive, this means that it calls itself.

```
1 def fact(n):
2         if n == 0:
3                 return 1
4         else:
5                 return n * fact(n-1)
```

The Python code above shows a simple use of recursion to calculate factorials. Since a factorial is just a number multiplied by each integer below it in succession, this can be expedited by recursion. Notice how on line 5 the function calls itself by passing `n-1` to itself. With this, the function can continue. The next part of recursion is its base case of 0. Without this, the program will never finish, and possibly will cause a stack overflow error. By having a base case, there is a breakpoint for the code where it will begin to move back up the chain of calculation, freeing up the stack as it progresses.



Upon first glance of the graphic above, one may think that merge sort is much more complex than the previous two algorithms discussed. This person would be correct. Although, it is easier to understand after it is broken into its parts. These parts being the splitting, and the sorting. To begin, we will discuss the splitting function, which also happens to be the recursive function.

5.2 Implementation of Merge Sort

```
void Sort::mergeSort(std::string s[], int lInd, int rInd, int &cnt)
{

        int m = 0;

        if (lInd < rInd)
        {
                m = lInd + (rInd - lInd) / 2;
                mergeSort(s, lInd, m, cnt);
                mergeSort(s, (m + 1), rInd, cnt);
11
12                  cnt += Sort::subMSort(s, lInd, m, rInd, cnt);
13          }
14          else
15          {
16                  //int n = sizeof(*s) / sizeof(s[0]);
17                  //printList(s, n);
18          }
19  }
```

Unforeseen Issues

Up until now, I have not acknowledged the lines of code dedicated to counting comparisons, because they are not integral to the function of the algorithms. In this sorting method though, it was a particular challenge. The reason being that this function is recursive, and therefore I could not simply run the function, count the comparisons as it goes, and then return it. Because of this, in this algorithm along with Quick Sort, I passed a reference to a count variable I called `cnt`. This way I could have the algorithms increment the counter while not having to worry about maintaining the variable within themselves. Now on to the description.

Code Breakdown

1. Check that there is actually enough range to sort by seeing if the left index is still before the right index (line 6)
2. set `m`, our midpoint variable, to `lInd + (rInd - lInd) / 2`, in order to account for odd numbers (line 8).
3. Call `mergeSort` on the first half (0-`m`) and the second half (`m+1` to end) (lines 9, 10)
4. Sort the array within the described bounds and add to the count (line 12).

If the process still doesn't make sense, focus on the left and right indexes that are being passed to `mergeSort` on lines 9 and 10.

```
mergeSort(s, lInd, m, cnt);
mergeSort(s, (m + 1), rInd, cnt);
```

Since this function will continue running until the left and right indexes match up, or when the bounds of the array given equal one index, these two lines will continue to take the given bounds and keep splitting them up into quarters, eighths, and so on until that base case is achieved. Now onto the sorting portion of the algorithm.

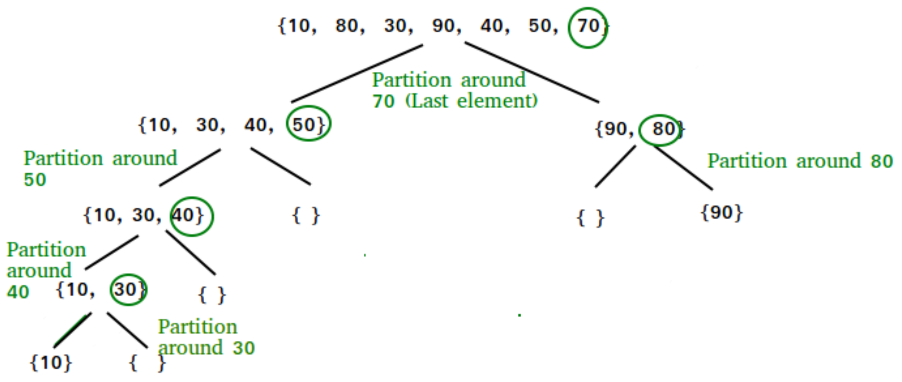
```
int Sort::subMSort(std::string s[], int l, int m, int r, int cnt)
{
        int counter = 0;
4           int len1 = m - l + 1;
5           int len2 = r - m;
6
7           std::string *left = new std::string[len1];
8           std::string *right = new std::string[len2];
9
10          for (int i = 0; i < len1; i++)
11          {
12                  left[i] = s[l + i];
13          }
14          for (int j = 0; j < len2; j++)
15          {
16                  right[j] = s[m + 1 + j];
17          }
18          int i = 0, j = 0;
19          int k = l;
20          while (i < len1 && j < len2)
21          {
22
23                  counter++;
24                  s[k++] = (left[i] < right[j]) ? left[i++] : right[j++];
25          }
26          while (i < len1)
27          {
28                  s[k++] = left[i++];
29          }
30          while (j < len2)
31          {
32                  s[k++] = right[j++];
33          }
34          return counter;
35  }
```

1. Allocate space for the left side of the given portion of the array with the correct size (lines 4, 7).
2. Allocate space for the right size of the given portion of the array with the correct size (lines 5, 8).
3. Put the first half of the items in the given portion of the array to the `left` array, and the second half to the `right` array (lines 10-17)
4. Set index counter `i` for the `left` array and `j` for the `right` array, both at 0. also set an index counter for the actual array as `k`, initiated with `l`, the left bound index for the array.
5. Compare both the `left` and `right` arrays by index starting 0, whichever value at the current index is larger, place that value at the next index in the main array `s[]`. Increment the index counter on the sub array that the value was pulled from, as well as the index counter for the whole main array. This way, the sorting can continue (line 24).

6. Continue comparing until the end of either sub array is reached (line 20).
7. At this point, append the remaining unsorted items after the last sorted index of `s[]` (lines 26-34).

6 Quick Sort

6.1 Understanding Quick Sort



Quick sort is even faster than merge sort. The reason being that much of the sorting is being done at the same time that the array is being broken up. This process will seem very similar to merge sort. Quick sort uses pivot values to choose where to compare values. The basic structure is to choose a pivot / partition value to start, and split at that point. Next, a new pivot value is chosen, and all values below the pivot value are put below the pivot value, and all values greater are put in front of the value. From here, a new pivot value is chosen and the process continues until each element is sorted in its place. The decision of what pivot value to use is widley discussed, although in my version I start with the last element.

6.2 Implementation of Quick Sort

Much like merge sort, quick sort calls itself recursivley on two halves of the given bounds. Although this time, the ”midpoint” value is determined by the partition function.

```
void Sort::quickSort(std::string s[], int p, int r, int &cnt)
{
        if (p < r)
        {
                int q = Sort::Partition(s, p, r, cnt);
                Sort::quickSort(s, p, (q - 1), cnt);
                Sort::quickSort(s, (q + 1), r, cnt);
8           }
9   }
```

1. Ensure that the given bounds account for more than one index (line 3)
2. Use the `Partition()` function to sort the given bounds and return a new midpoint. (line 5)
3. Call `quickSort()` on all elements from index 0 to `q` as defined by the `Partition` function.

Below is the definition of the `Partition` function. It is most likely the most difficult to understand. At a very high level, it is placing elements in front of or behind the pivot element (chosen as the last element in the unsorted array) depending on if the value of the element is higher or lower than the pivot. Also, it returns the index of the next partition index.

```
1   int Sort::Partition(std::string s[], int p, int r, int &cnt)
2   {
3           std::string x = s[r];
4           int i = p - 1;
5           for (int j = p; j < (r - 1); j++)
6           {
7                   if (s[j] <= x)
8                   {
9                           i++;
10                          cnt++;
11                          Sort::swap(s[i], s[j]);
12                  }else{
13                          cnt++;
14                  }
15          }
16          Sort::swap(s[(i + 1)], s[r]);
17          return ++i;
18  }
```

1. set the last element in the range as the pivot element `x` (line 3).
2. variable `i`, used to point to the last sorted element below the pivot is initialized to `(p - 1)`(line 4).
3. begin an for loop with an iterator called `j`. Initiate `j` with `p` and continue until `(r - 1)` (line 5).
4. If value `s[j] <= x`, then increment `i`, and swap the values `s[i]` and `s[j]` (lines 7-12)
5. After exiting the for loop, swap `s[(i + 1)]` and `s[r]`.(line 16)
6. Finally, `Partition` returns `i` after incrementing it one final time. With this, the `Partition` function returns the new pivot index (line 17).

7 Setup and Execution

Finally, Here is a brief description of the actual execution of these algorithms.

```
1  const int len = 666; //normal mode  finishes in ~15 - 35 ms
2
3  std::string arr[4][len];
4  //std::string arr[len];
5  void setup()
6  {
7  ...
8  ...
9  ...
10                                  arr[i][count++] = ln;
11 ...
12 ...
13 ...
14 }
```

I have excluded most of the `setup()` function, as it is a direct copy from the `AlanParse()` function. Briefly, it reads in the .txt file line by line to an array for use by the functions. The only difference this time is that the list needs to be reused. In the beginning, this was done by using a function called `reset()` to erase all of the content from the array each time. This was used between executions of the different algorithms.

Although When playing around with multithreading (which never was completed) I realized that I could not have just one array. The reason being that there would be memory issues if four separate threads were accessing the same memory. So, on line 10, it is clear to see that I made a multidimensional 4x666 array of strings. Each subarray for each algorithm. Although the dream of having multithreading, and testing on greater numbers of strings was never realized, this was a much cleaner organization of the data.

```
int main()
{
        setup();
        std::cout << "Insertion sort made " << std::to_string(Sort::Insertion(arr[0], len)) << " comparisons." <

        std::cout << "Selection sort made " << std::to_string(Sort::Selection(arr[1], len)) << " comparisons." <

        int cnt = 0;
        Sort::mergeSort(arr[2], 0, (len - 1), cnt);
        std::cout << "Merge sort made " << std::to_string(cnt) << " comparisons." << std::endl;

        int qCnt = 0;
        Sort::quickSort(arr[3], 1, len, qCnt);
        std::cout << "Quick sort made " << std::to_string(qCnt) << " comparisons." << std::endl;
        /* Sort::printList(arr[0], len); //test Insertion Sort
        Sort::printList(arr[1], len); //test Selection Sort
        Sort::printList(arr[2], len); //test Merge Sort
        Sort::printList(arr[3], len); //test Quick Sort */
}
```

The first to algorithms, insertion sort and selection sort, have nothing notable about their execution. They both are passed the array itself and the length of the array, and return an integer which is converted into a string to display. The execution of merge sort and quick sort are more complicated. Both are passed their array just as insertion and selection are, although these two get low and high bounds, as well as a count variable. As previously discussed, the bounds are very important to these sorts, as they are recursive. Without them, this process would be much more complicated.

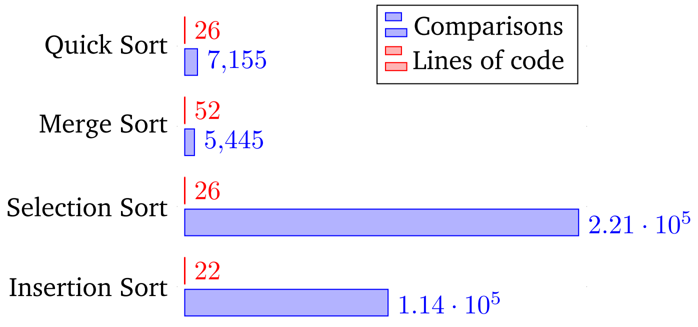
8 Conclusion: The Results

Finally, the results tell the real story. It is clear to see that the recursion helped to make Merge Sort and Quick Sort considerably faster. Although in the real world, this does not make much of a difference. On my personal laptop, a moderatley powerful machine, I clocked the whole program to run in just around 30 milliseconds.

Sort	Comparisons
Insertion Sort	114309
Selection Sort	221445
Merge Sort	5445
Quick Sort	7155

Another interesting metric is to compare the total lines of code per algorithm (in C++) compared to their number of comparisons. It should be noted that in this instance quick sort is less efficient than merge sort. This is truly down to the way that the pivot index is chosen. If the index was chosen at random, or some other way, the algorithm may be more efficient.

Complexity of Sorting Algorithms For 666 Items



Closing Thoughts

For someone not in the field of computer science, it may seem trivial to sort a list of numbers, or sentences, or anything else. Although to me, this was and still is one of the hardest things to grasp. Beyond that, this assignment began to get my wheels turning thinking about multi threading. I am sure that with more time, I would have had a working test case for sorting over 1,000,000 items per algorithm, each being on a separate core. This can wait until the next assignment though.

$\{$*S.A.*$\}$

9 Appendix

References

Insertion Sort Algorithm, Programiz.com, www.programiz.com/dsa/insertion-sort.

Insertion Sort, GeeksforGeeks, 25 July 2020, www.geeksforgeeks.org/insertion-sort/.

Merge Sort, GeeksForGeeks, 7 June 2020, www.geeksforgeeks.org/merge-sort/.

Quick Sort, GeeksforGeeks, 4 Sept. 2020, www.geeksforgeeks.org/quick-sort/?ref=lbp.

Selection Sort Algorithm, Top Sites, www.thetopsites.net/article/51681938.shtml.

Used for Graphics and quotes