Tran, Van

10/14/15

Write up Data Structures

Element choice:

I chose for my elements to be organized in strings because I can get the length of the strings without counting them. The overhead of using strings compared to char\* is probably smaller than counting each character in each word.

Data Structure choice:

I chose a vector for my wrapper around the string words. Vectors deals with all the memory management and is essentially a dynamically allocated array that manages memory for you.

However the BIGGEST part of choosing a vector is that it counts the number of elements inside of it for you already, allowing you to see the N before you sort. This is huge, because

one can now choose to run code based on how large or small N is to optimize the code.

Selection sort:

Selection sort is the base case of sorting that we start with for the purpose of this project. The selection sort goes through the array and on each pass places the unsorted element with the smallest value into its correct place, swapping the values in the array. The code for this algorithm is as follows.

for (int i = 0; i < asize; i++){

smallest = i;

for(int a = i+1; a < asize; a++){

if(array[a] < array[smallest]){

smallest = array[a];

}

}

temp = array[i];

array[i] = array[a];

array[a] = temp;

}

On each pass through the array the array gets smaller by 1. Meaning the number of comparisons is equal to:

n + n-1 + … + (n-(n-1)) = N\*(n-1)/2

The number of adds are equal to:

n + n + n-1 + .. + (n-(n-1) = n + n\*(n-1)/2

The max number of swaps is equal to:

n

The number of times variables are assigned a value are equal to:

1 + n + n – 1 + n – 1 = 2n -1

So the total number of operations is equal to:

N\*(n-1)/2 + n + n\*(n-1/2) + n + 2n -1 = n\*(n-1) + 3n - 1 = **n^2 + 2n - 1**

With the average being **n^2 + 3n/2 - 1** and the smallest being **n^2 + n - 1**.

The asymptotic performance of the sort is always O (n^2).

We can make small adjustments to the algorithm such as instead of incrementing a, and then resetting a to i + 1, we can decrement a until i + 1 finding the maximum. This idea is basically to make the selection sort bidirectional, which would decrease the number of times variables are assigned a value by n-1 times. A nice improvement, but it does not change the asymptotic performance. In my implementation of this sort I had to make an extra one or two comparisons every time, because if the word lengths were equal, then I had to compare them alphabetically.

Although selection sort is O(n^2), it is widely used when the swapping of the data is costly, because every swap puts an element into its correct spot.

Insertion Sort:

I tried to improve the selection sort by instead of going through the array each time for a swap, swap each time I see that an element is smaller than another.

This difference is called an insertion sort. Insertion sort is similar to selection sort, but iterates through the array, swapping elements if it sees an element is smaller to its adjacent one. This is advantageous, because it makes it an extremely fast sort if the elements are almost sorted, and in many cases outperforms O(nlgn) algorithms. However, if the data has an even distribution, then insertion sort becomes worse than selection sort. Although there will be the same amount of comparisons, selection sort will outperform insertion sort due to the smaller amount of swaps needed to sort the array. However, in practical cases, data is almost never random, and data is often small, but in large quantity. Therefore insertion sort almost always outperforms selection sort. In my implementation of this sort I had to make an extra one or two comparisons every time, because if the word lengths were equal, then I had to compare them alphabetically.

We can also make the insertion sort bidirectional similar to the optimization of the selection sort mentioned above. Although it does not improve the asymptotic performance of the algorithm, it does reduce the number of reassignments of variables by n-1.

Heap Sort:

Instead of trying to change the way I compare and swap elements, I can organize the data in a way that makes the sort more efficient. This improvement of implementing a selection sort on a heap is called heapsort. First we create a heap, which is a data structure in which a number’s parent is larger than it and each parent can only have two children. This creates an organization in which a parent can be accessed by using (x-1)/2 where x is a child index and a child can be similarly accessed by x\*2+1 or 2 where x is a parent index. After we create the heap, we then use the heap to sort, by swapping the last element with the end of the heap. We then recreate the heap, but the last element is now is in its sorted position. This is repeated until the whole array is sorted. As you can see, heap sort has the same concept as selection sort. They both have a sorted and unsorted portion where you put the next smallest or largest element into the sorted portion. However, heap sort uses the properties and organization of a heap to do that.

Quick Sort:

Instead of organizing data into another data structure, I can take a different approach and look at the properties of numbers and how they organize themselves when compared. Quick sort is takes advantage of the fact that if you put all the larger numbers above a selected element and all the smaller numbers below a selected element, then that element will be in the correct spot. In the best case where the median element is always selected, the sort will be O(nlgn), as the data set will be bisected every iteration of the sort n­k= n/2­k where k is the number of iterations. However, finding the median is very costly, so in implementations of quicksort the median is estimated. In my implementation I select the middle element of the data set as a pivot. In my implementation of this sort I had to make an extra one or two comparisons every time, because if the word lengths were equal, then I had to compare them alphabetically.

Radix Sort MSD:

This is my final implemented algorithm. Instead of comparing elements to one another to find out which element is larger, radix sort organizes the information by putting digits into buckets recursively until each element is in its own bucket. I used the size of my strings for the first two significant digits and then the content of my strings for the remaining “significant digits”. Because of the large constant that comes with MSD radix sort, if N is small, then I can simply choose to use insertion sort, or any other sort that performs well on small N size.

For some odd reason my quicksort was always faster so I went ahead and used it.

QUICKSORT:

SELETION SORT:

INSERTION SORT: