COP 4531 – Assignment 2

James Bach

I have implemented a C++11 standard program to analyze the asymptotic analysis of various sorting functions. I used the linprog machines to compile with optimization flags. I used random generation of integer data and the <chrono> library for timestamps to determine the time in milliseconds how long it would take to sort with insertion sort which is a Θ(n^2 ) algorithm and merge sort which can Θ(n log n) algorithm. The user can interact with the input to get randomized data and set certain values of k. The specified run commands are available to run the program:

‘algorithm <sorttype> <datafile.txt>’

In order to hold the specified data, I created a class called unit, with member data of a character array of the specified size and an integer to hold the sorting number. The sorts mentioned below sort by this integer value.

The insertion sort implementation goes the length of n, with each element it starts right behind it and traversing back to the beginning and trickling down swapping until it is sorted on the left of the start location of the first loop. This is a Θ (n^2) algorithm because potentially for every element n it visits every element of n again.

The merge sort implementation recursively breaks the entire array up into two parts of size beginning to middle and middle + 1 to end. This happens recursively until it is a single element, which by definition is sorted. Then it merges each pair of broken up parts all the way back up into a sorted array. The merge function compares each side and figures out which next element is next in order and places it in a temporary array that is built sorted. If one of the two sides to be merged runs out, the remaining of the other side is placed in the temporary array. The temporary array then gets placed back into the original array in order.

The counting sort algorithm finds the largest value of k when the initial data is read. A temporary array with the ability to hold integers and strings (using the special class) is created (vector is used) of the size of k and all the integer values are set to 0. Then it loops through the data and on each value, the algorithm increments the matching key of the temporary array. In other words the value of the first array is the index of the second array and the value of the second array is incremented by one indicated that there is exists +1 of this k. After that loop, a final loop loops through the temporary array starting at 0 and for each value in the key it places that key in the value of the return array a decrements the value in that key until 0 then moves on throughout the loop. The return array is started at index 0 and each element placed will increment the index for the next placement until the array is full.

Experiment 1:

|  |  |  |
| --- | --- | --- |
|  | Θ(n log n) algorithm | Θ(n^2 ) algorithm |
| n=10 | .002346ms | .001278ms |
| n=100 | .01987ms | .037796ms |
| n=1,000 | .255323ms | 4.23654ms |
| n=10,000 | 4.228667ms | 443.013ms |
| n=100,000 | 55.981ms | 44640.6ms |
| n=1,000,000 | 777.048ms | 884712.61855ms |

The results indicated that for low values of n, the sorting function to use is trivial. However, the merge sort Θ(n log n) algorithm is grossly more efficient when n becomes large and the Θ(n 2 ) algorithm because radically ineffective.

In order to check where k is limited by a value that is not integer max like before, I created a subsection in my program where you can select a maximum k value to use for sorting. I used n = 100 and n = 10,000 for a set of tests where k is the values in the leftmost column of the charts below.

Experiment 2:

|  |  |  |  |
| --- | --- | --- | --- |
| n (≤ 100) | Θ(n + k) algorithm | Θ(n log n) algorithm | Θ(n^2 ) algorithm |
| k = 2 | .013922ms | .017299ms | .02087ms |
| k = 5 | .018464ms | .018246ms | .033683ms |
| k = 10 | .020178ms | .018528ms | .038914ms |
| k = 200 | .038902ms | .019464ms | .036633ms |
| k = 500 | .054598ms | .024013ms | .039812ms |
| k = 5000 | .432097ms | .018757ms | .041695ms |

|  |  |  |  |
| --- | --- | --- | --- |
| n (≥ 10, 000) | Θ(n + k) algorithm | Θ(n log n) algorithm | Θ(n^2 ) algorithm |
| k = 2 | .838071ms | 3.69888ms | 223.222ms |
| k = 5 | .848263ms | 3.8309ms | 352.955ms |
| k = 10 | .746175ms | 4.10341ms | 404.529ms |
| k = 200 | .955964ms | 4.23785ms | 437.256ms |
| k = 500 | 1.35428ms | 4.27826ms | 448.911ms |
| k = 5000 | 2.34154ms | 4.315ms | 447.767ms |

The trend by both graphs when k gets larger is that it takes longer to sort. There are always potential cases where the elements in the array are sorted faster or slower based on their positions in the array. A reverse sorted array may sort slower than a partially sorted array. This is based on how many times it causes certain expressions to run. Additionally, shown by the different values of n, the data on the chart shows that the ability of a linear algorithm such as Θ(n + k) of counting sort drastically outperforms the merge sort and insertion sort when n becomes larger.

Bonus:

The sorting algorithm scans the values coming in from a file and determine the largest value of k in process. If k is not excessively large (k < 100000), it will use counting sort. The reason of this is overhead from allocation of memory and setting data for a huge value of k is considerably time consuming, and in this case my trials indicated the tradeoff happens here since we are using more than just a simple integer array for counting sort. Otherwise, if memory allocation was never an issue, counting sort would win every time. However, if the value of k is too large, a quick sort will sort the data if n is > 30, otherwise it will use insertion sort. I chose quicksort because the constant for which it sorts as a Θ(n log n) algorithm is lower. Additionally, I made the modification that when the parts that are being sorted are < 30 it chooses to use insertion sort. The usage of insertion sort is that insertion sort can be considerably more efficient on lower values of n than other sorting such as quick sort or merge sort. The quick sort recursively splits in parts and randomly picks a pivot. Then it starts at each end and picks a bigger value on the left of the pivot and a smaller value on the right of the pivot, then swaps. This continues moving from the ends towards the pivot until the moving points intersect. This will guarantee the pivot is in the correct place at the end and then each recursion continues, placing the values in the correct place.