Marcellus Hunt

4/19/18

362 Project 2

Project Report

The objective of this project was to test and compare sorting algorithms and the input size they would be needed to sort an unsorted array and the time for it to go through an array that has already been sorted. The sorting algorithms that were used were insertions sort, merge sort, heap sort, and quick sort. Insertion Sort picked out an element from the list and it would go through the list to check if it was less than or greater than the next number. When the number is placed in its position, it goes to the next integer after that one to find its place. Merge sort divides the list in halves and sorts the halves made by cutting in halves again and sorting. At the end of the sorting it would merge the halves back in to a sorted list. For heap sort, the list would be built up like a tree and then when going through it. The largest number is the root and we move the root to the end and then continue to do that for the rest of the heap. It traverses this tree and rebuild the heap every time the root is removed. Quick sort picks an element as pivot, or something to do around, and breaks up the array around it. This puts numbers less than the pivot at the beginning of the array and the elements greater than pivot after it. Resulting in a sorted list.

(Insertion Sort= O(n^2) , presorted O(n)) (MergeSort = O(nlogn)) (HeapSort = O(nlogn) ) (Quick Sort = O(nlogn) )

Insertion sort takes the most time to sort when the data size gets higher.

Insertion sort ends up taking less time than the other sorting methods as the data size gets bigger because its time becomes O(n).

Actual execution time for each data size:

* Hundred
  + Insertion(unsorted) = .004s, sorted =.000s
  + Quick(unsorted) = .004s, sorted = .000s
  + Heap(unsorted) = .004s, sorted = .000s
  + Merge(unsorted) = .004, sorted = .000s
* Five-Hundred
  + Insertion(unsorted) = .004s, sorted =.001s
  + Quick(unsorted) = .005s, sorted = .000s
  + Heap(unsorted) = .005s, sorted = .000s
  + Merge(unsorted) = .005, sorted = .000s
* Thousand
  + Insertion(unsorted) = .006s, sorted =.00s
  + Quick(unsorted) = .004s, sorted = .003s
  + Heap(unsorted) = .004s, sorted = .003s
  + Merge(unsorted) = .004, sorted = .003s
* Two-Thousand
  + Insertion(unsorted) = .008s, sorted =.000s
  + Quick(unsorted) = .005s, sorted = .014s
  + Heap(unsorted) = .005s, sorted = .014s
  + Merge(unsorted) = .005, sorted = .014s
* Five-Thousand
  + Insertion(unsorted) = .026s, sorted =.000s
  + Quick(unsorted) = .006s, sorted = .040s
  + Heap(unsorted) = .006s, sorted = .040s
  + Merge(unsorted) = .006, sorted = .040s
* Eight-Thousand
  + Insertion(unsorted) = .060s, sorted =.000s
  + Quick(unsorted) = .006s, sorted = .174s
  + Heap(unsorted) = .006s, sorted = .174s
  + Merge(unsorted) = .006, sorted = .174s
* Ten-Thousand
  + Insertion(unsorted) = .102s, sorted =.010s
  + Quick(unsorted) = .006s, sorted = .234s
  + Heap(unsorted) = .006s, sorted = .234s
  + Merge(unsorted) = .006, sorted = .234s

The algorithm times with comparison to each other were similar to what happened when I ran the program. Insertion sort took the longest when it was not sorted as the data size got larger but when the array was already sorted then insertion sort was the faster of the other three algorithms. In the info above