Syeda Kazmi-Shah

Algorithms

INSERTION SORT:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | 50 | 200 | 2000 | 3000 | 4000 | 5000 |
| Time | 0.000108003616333 | 0.00100803375244 | 0.137737035751 | 0.357388019562 | 0.475505828857 | 0.68875288963 |

\*n^2

MERGE SORT:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | 50 | 200 | 2000 | 3000 | 4000 | 5000 |
| Time | 0.00045084953 | 0.00089907646179 | 0.0208551883698 | 0.0539660453796 | 0.0539660453796 | 0.05937910079 |

\*nlog(n)

The two tables show the comparison between the two algorithms: merge sort, and insertion sort. While the input size is small, the operation to compute the algorithm is done rather quickly in insertion sort and beats the merge sort algorithm in efficiency. However, as the input size increases to a higher value, merge sort begins to handle larger inputs much better than the insertion sort algorithm and is much more efficient. In short, when the input size increases it eventually begins to take more time to compute for insertion sort, thus making it the slower algorithm.

**import** timeit  
**import** random  
  
**def** insertion\_sort(array):  
 **for** element **in** range(0,len(array)):  
 move\_left(array,element)  
  
**def** move\_left(array,index):  
 popValue = array.pop(index)  
 **while** index!=0 **and** popValue<array[index-1]:  
 index -= 1  
 array.insert(index,popValue)  
  
  
numbers = [4,7,3,8,1,12,9,14]  
**for** x **in** range(0,5000):  
 rand = random.randint(1,99999)  
 numbers.append(rand)  
  
  
start\_time = timeit.default\_timer()  
insertion\_sort(numbers)  
elapsed\_time = timeit. default\_timer() - start\_time  
**print**(**"Insertion Sort: "** + str(elapsed\_time))  
**print(**numbers**)**

**import** timeit  
**import** random  
  
**def** arrayCopy(src, srcPos, dest, destPos, length):  
 **for** i **in** range(length):  
 dest[i + destPos] = src[i + srcPos]  
  
  
  
**def** split\_Sort(array):  
 **if** len(array) <= 1: *#if already 1, you're done !  
 # print("The length reached 1, going back up")* **return** array  
 left = array[0:(len(array) / 2)]  
 *# print("Splitting " + (str(array)))* right = array[(len(array) / 2):len(array)]  
  
 split\_Sort(left)  
 split\_Sort(right)  
 merge\_Sort(left,right,array)  
 **return** array  
  
**def** merge\_Sort(left,right,array):  
 rcount = 0  
 lcount = 0  
 merge\_count = 0  
 **while** lcount < len(left) **and** rcount < len(right):  
 **if** left[lcount] > right[rcount]:  
 array[merge\_count] = right[rcount]  
 rcount += 1 *#go onto next element* **else**:  
 array[merge\_count] = left[lcount]  
 lcount += 1  
  
 merge\_count += 1  
 arrayCopy(left,lcount,array,merge\_count, len(left)-lcount)  
 arrayCopy(right,rcount, array, merge\_count, len(right) - rcount)  
  
*########################################*numbers = [7,4,3,8,1,12,9,14]  
  
**for** x **in** range(0,5000):  
 rand = random.randint(1,99999)  
 numbers.append(rand)  
  
start\_time = timeit.default\_timer()  
split\_Sort(numbers)  
elapsed\_time2 = timeit.default\_timer() - start\_time  
**print**(**"Merge Sort: "** + str(elapsed\_time2))  
**print**(numbers)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_