Mahyar Vahabi

10/14/2021

Asgn 3: DESIGN Document

Purpose:

The purpose of this lab is to implement sorting algorithms such as; Insertion Sort, Shell Sort, Heapsort, and recursive Quicksort based on the provided Python pseudocode in C. These sorting functions will appropriately, but differently sort the elements in a list/array. We then create a test harness that takes the user input and applies it to the program's designated output. In addition we take data about each sort and its performance. The statistics include the number of moves and the number of comparisons made while sorting the elements in an array. We are required to use a set to track which command-line options are specified when your program is run.

Pseudocode:

- **Insertion sort:**

insertionSort(array, size)
  first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort

- **Heap sort:**

build_heap(array, size, largest)
  Find largest among root and children:
  Left side = 1 before ; right = 1 after middle
heap_sort(array, size)
  heap_build(arr, size)
  for (element in between first and last element)
    swap(first element with the new element)
    build_heap(array, element, 0)

- **Quick sort:**

partition(array, leftmostIndex, rightmostIndex)

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex + 1

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex - 1)

quickSort(array, pivotIndex, rightmostIndex)

- **Shell sort:**

gaps(size**)**

for terms in $log(3 + 2size) / log(3)$

save and return floor value of $s^{term} - 1/2$.

shell_sort(array, size)

for gap in gaps(size)

for steps in between gap and size

j also equals steps

Save temporary value of the array[step]

while j is less than or equal to gap and also temporary value is less than the value j - gap in the array:

Array[j] equals array[j - gap]

j decremented by gap

Array[j] = temporary value

Sorting()
Initialize all used variables: seed, size, elements etc.
Initialize stats to take in arrays later on.
While (taking in arguments) {
print all command lines options
Make switch and case statements for each command line option and their purpose }
. allocate memory for when we run the program
Set srandom with seed as an argument
Make a bit wise and operation for the random generated numbers to take in only 30/32 bits of their numbers.
If (each sorting algorithms are a member set of set s) {
Print all necessary output values
}
Deallocate the memory that was allocated earlier	

In each 4 sorting files, there are specific areas that take a few steps to sort the elements of arrays, which is tracked through our handy dandy Stats structure. Wherever we are comparing the elements in an array to check if it is bigger or smaller than the other one, or when we swap their positions, as well as moving them to their respective position, it adds those amounts of steps to “stats.moves” and “stats.comparisons”. These two items will return the number of moves and comparisons made in each sorting algorithm.