CS20B

5-23-2019

What about Primitive vs. Object data types?

Learning algorithms with primitive data types to explore their uses.

**Questions:**

***How will things change for the sorting algorithms when we are working with a minimal data structure (e.g. Lists, i.e. not graphs/trees)***

isSorted() method – iterating array to check the true (sorted) or false (unsorted), used as an assertion for the class

Questions:

***Is O(N) operation on an assertion or test case for checking the sorted status a list/array itself optimal or could that be more efficiently done with something O(logN)***

Swap – basic method for assisting a more efficient sorting algorithm

static public void swap(int index1, int index2)

NOTE: swap doesn’t care about values only needs the indexes to make a swap.

Questions:

***All the algorithms made use of swap which using index to make the comparison, but what if the data structure, e.g. List not Array is without an index. Would we need to 1) assign an index or 2) make a more complex swap method for this data structure?***

Final class before review

Topic: Algorithms for sorting searching

Goals:

* Don’t have to memorize the algorithms
  + Knowing the idea/concept (properties) (May help to be visualizing sorting)
  + Knowing the drawbacks of each one.
* Efficiency
  + Worst case and best case for algorithms
  + How much processing in order to get data completely sorted. partially sorted, etc.

Why sort?

Sorting is done in order to organize and handle the processing of other methods in the data structure, e.g. Sorting is a prelude for the searching method.

How to measure the performance with efficiency?

The number of sort operations, e.g. counting how many performed should help to understand the time and space efficiencies, often time space efficiency is sacrificed for time efficiency (especially given the advanced technologies that allow for it).

Bubblesort algorithm

Traversing the array either from LtoR or RtoL and processing items as we meet them. It is very slow goes back through list many times over. O(N-squared).

It works by comparing adjacent elements in the list/array to each other and swapping.

RtoL: if values(i) < values(i - 1)

LtoR: if values(i) > values(i+1)

**Questions:**

***Why/where to use it? My guess: to simplify the coding of sorting. Or when using it to knowingly sort just a smaller portion of a large set of data.***

***Is it ever useful to call a sort from inside a sort, e.g. Wrapping bubblesort inside of another sorting algorithm?***

Selection sort

Space – wasted if using a second list or array

But with some planning and algorithm no space is wasted and using a relatively efficient code.

For loop that calls the swap on current, minIndex function \*\*\*recall swap always uses indexes

minIndex function is called N-1 times , proof for O(N-squared) is a series of n-1 + n-2 + n – 3

Shorter Bubble sort

Same idea, but helper function checks if the array is sorted. So it can reduce number of operations.

Going over example of starting at the bottom and going up the list and the smallest one ends up on top

Still poor O(N-squared) for complete sorting. But improvement over selection sort for partial sort.

If it is almost sorted then bubble sort can be improved further with Shorter Bubble sort

Insertion sort

Two functions the simple for loop that increments the counter and calls a insert function with proper index. The values that are sorted from the start of the list and unsorted at the end of the list. Current is the boundary of sorted/unsorted.

NOTE: Better in worst/best case than bubble sort because of the number of checks required.

MergeSort

Algorithm:

1) split array, 2) process two halves separately, 3) merge

Splitting array is done the same way as with binary search (NOTE: there is no new space made when splitting an array, and everything is done within the data structure list, array, etc. of values).

MergeSort its a recursive algorithm to handle sorting in two halves.

The sorting in each half is just like done in a binary tree. Where logN is the number of levels, but in the sort there are N operations per level. Big O of NlogN is the best possible efficiency for sorting!

Merge at the end is a lengthy amount of code to piece the two pieces back together.

Works better on almost sorted data.

QuickSort

Another fast sorting algorithm with O(NlogN) efficiency. Split value used to assist the algorithm also makes it difficult to analyze the efficiency. Works better on random data.

Heap Sort

Heap is very efficient. Intermediate structure to sorting. Heap rearranges in logarithmic time.

Heap QuickSort MergeSort

Best Case

Average Case

Worst Case

**Also look into the Java impementation of these various sorting algorithms available.**

*Note about Writing Good Tests for testing the sort methods*

Thorough testing vary the size of the array, vary the order of the array, random, reverse, almost sorted, all identical elements.