COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 14: March 26

Lecturer: Dr. Andrew Hines *Scribes: Moriah Fiebiger - Clare Byrne*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

14.1 Outline - "Lets Sort This Out"

Before, we looked at a few sorting algorithms. In a lot of cases these were inefficient, although they gave us different ways that you can approach the same problem. Once we had created the basic algorithm, we could start adapting to improve; this had a significant effect on the performance of that algorithm.

In this class we looked at two different algorithms, slightly more complex in their implementation: Quick Sort and Merge Sort. In a number of cases, they would be more efficient. For certain types of data, these are generally 'better' algorithms; however, there is a time and a place for bubble sort!

14.1.1 Recap From Last Lecture

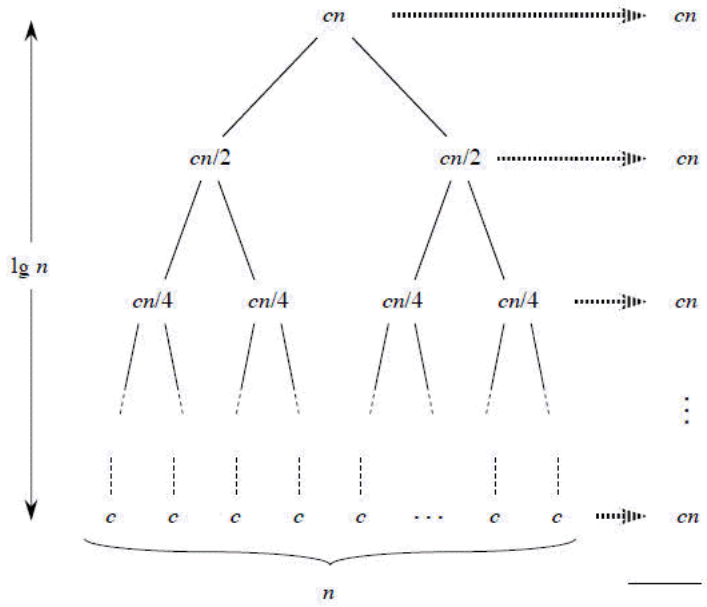
Before the break we looked at inefficient sorting algorithms

- Bubble: Repeats the algorithm for entire sequence. Each iteration compares two adjacent values and swaps them if they are in the wrong order.
- Insert: Repeats the algorithm for entire sequence. Each iteration searches through the entire sequence and moves the current value where they need to go. The list is adjusted after each iteration.
- Select: Repeats the algorithm for entire sequence. Like insert, at each iteration searches through the entire list, but searches for the smallest element on the right hand side of the current value and places it on the left hand side.

However, there are many ways to solve the same sorting problems It's good to adapt/optimize (refracture/make better) the sorting problems

14.2 Divide and Conquer

- Start with the name of the algorithm
- List inputs and outputs
- Be consistent in variable names and notation for choices, loops, data structures and assignment



A 'divide and conquer' algorithm is an algorithm that breaks its input down into smaller 'subproblems', which are all smaller instances of the original problem. The algorithm then solves the problem for the smallest instance of the problem, and then calls this solution on itself recursively.

The best-case time complexity for both Quick Sort and Merge Sort is O(nlogn). It's helpful to look at the time complexity for each step in a divide-and-conquer algorithm:

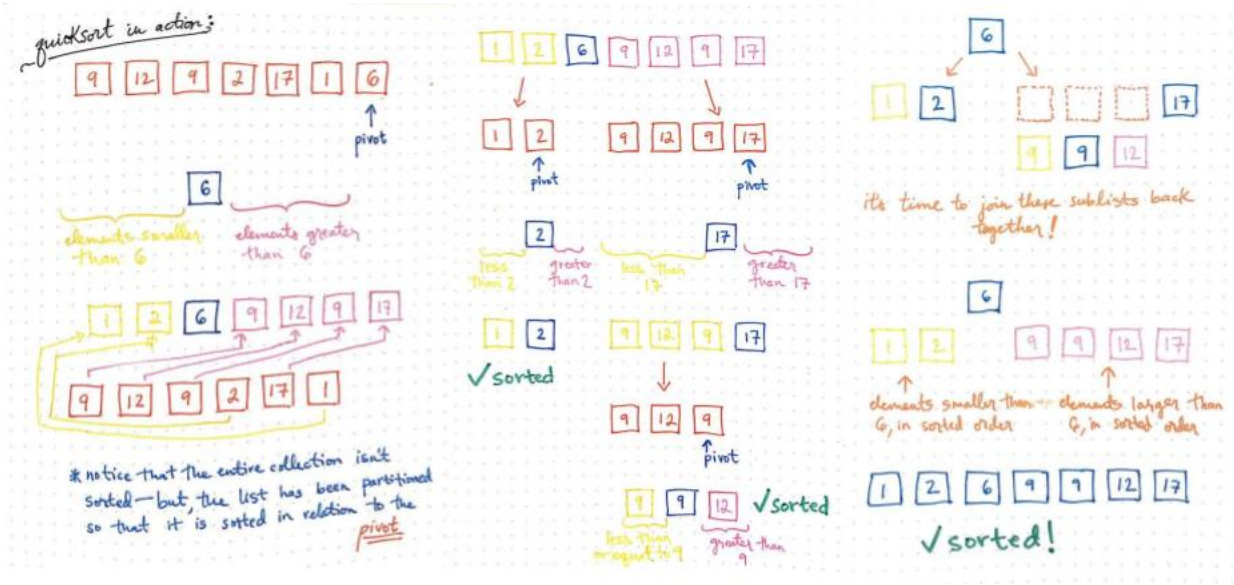
- The divide step gets the midpoint of each of the sub-arrays. This takes just O(1) time.
- The conquer step sweeps through the length of the subarray to recursively sort. There are n/2 elements to sort. The partition or merge step partitions/merges n elements, which takes O(n) time.
- If we look at the tree structure in the figure, we see that the height of the tree is (logn + 1). At each level of the tree, the level calls its method on each half of the array. The time complexity for this would be 2 * (n/2) = n.

Thus, we get n*(logn+1), which is O(nlogn) as a best case time complexity.

14.3 Quick-Sort

Quick sort is said to be the fastest sorting algorithm out there. The significant aspect to note here is that this is a recursive algorithm using the concept of, divide and conquer. The main identifier of this algorithm is choosing a pivot point and partitioning the array at the pivot point. The arrays recursively sort from this point; dividing the smaller values and large values on either side of the pivot.

But, why is quicksort so quick? The answer to this question is simple; quicksort does not require extra space to sort through the data. In other cases sorting algorithms will need an extra or temporary array to store the sorted items, whereas quicksort does not need this utilization.



14.3.1 Algorithm Break-down

- Takes an input (an unsorted array)
- The output is a sorted array
- pick a pivot (ideally in the middle - however can be any item within the array)
- partition elements in 2 groups around the pivot (L (less-than pivot) and G(greater-than pivot))
- repeat
- At the array will be broken up into many small groups. Now, concatenate the smaller groups into one big sorted array.

14.3.2 How it Works

In a worst-case scenarios, if we were to always pick the biggest or smallest value, then all values would end up in the 'less-than' column. This would give a running time of $O(n^2)$, which was just as bad as previous sorting algorithms. But with the normal average case – it branches out in an evenly distributed tree. The running time is O(nlogn).

This outcome could depend on what the data is like - how different kinds affect the performance of different algorithms.

14.4 Merge-Sort

Merge sort was invented in 1945 by John Von Neuman. In some ways, it's very similar to quick sort – we have two lists, or two halves of a list, and we join them together.

14.4.1 Algorithm Break-down

Merge sort is a "divide-and-conquer" algorithm that:

- takes a list of unsorted numbers as input
- divides its input into two halves
- defines a solution for sorting
- recursively calls this solution on each of the two halves
- merges the two halves together.

14.4.2 How it Works

The idea behind merge sort is that it's much quicker to sort two short lists and merge them, than to sort one large list. When we recursively break the list into halves, this can give us much shorter lists to sort and append.

For the other algorithms, we had to keep doing sweeps of the list to sort the numbers. In merge sort, instead of sweeping the full list, we break the list into smaller pieces. We build up our list, at each step, by taking the smallest remaining key from the two lists.

- Start with two sorted arrays:
- compare first two numbers
- choose the smallest
- put it into the output as the smallest.

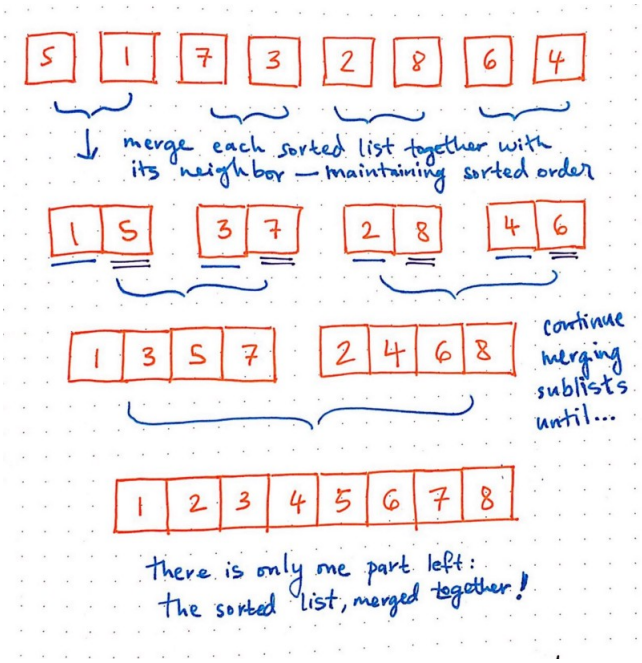
The image shows the list after it has been broken into halves recursively, until each 'half' is just one number.

To merge, the algorithm takes the first number in the first half, and compares it to the first number in the second half. The smallest number is added to a new list as the first number. In the second row of the figure, 1 and 3 are compared, and 1 is added to the new list as the first number.

The algorithm then takes the larger number, and compares it to the second number of the other half. In this example, 3 was the larger number; it is compared to the second number of the first list, which is 5. In this case, 3 is then added to the new list as the second number.

This process repeats until all numbers have been added to the new list; the new list then goes on to be a 'half' itself. In this example, the new list is [1,3,5,7]. This list goes on to be merge sorted with its second half, [2,4,6,8].

Input lists may not necessarily be the same size; you may end up with the first list having a bunch of elements left over at the end. Once the algorithm is finished going through the comparisons of all the arrays, it as to look at the leftover numbers, and add them in at the end of the sorted list.



14.4.3 How to write a Merge Sort algorithm?

There are two parts to a merge sort algorithm:

- Merge: merges the two halves together by sorting each item, and adding to the new result list.
- Main: split the array again and again, recursively, and call the merge function on each half recursively.

For an implementation in python, see the link in the Extra Reading section.

14.5 Considerations

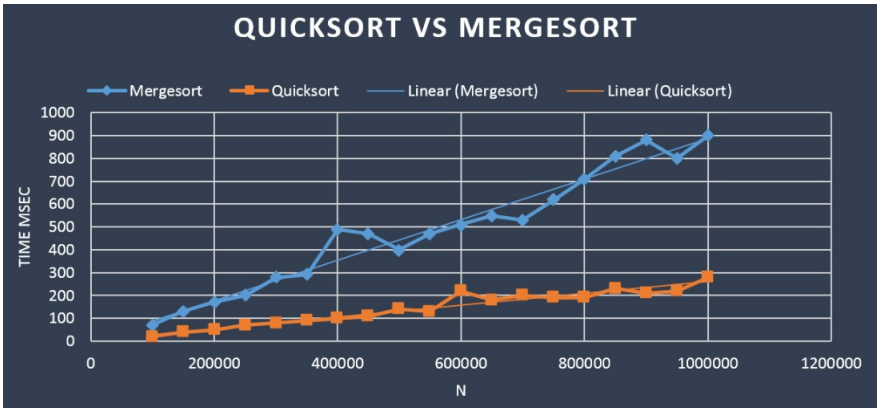
We have looked at five different ways of sorting things that are all very different approaches; all get us from an unsorted list to a sorted list

In terms of the average case, Merge Sort and Quick Sort have improved on $O(n^2)$ that we saw. It depends on the scenarios for inputs. Are the inputs likely to be the average case, or likely to be the worst case every time?

14.5.1 Time Complexity

As discussed in the divide-an-conquer algorithm section, the best-case time complexity for Merge Sort is O(nlogn).

As Merge Sort always divides a list in half, the worst-case time complexity is also O(nlogn).



14.6 Further Reading

Implementation in python: https://www.geeksforgeeks.org/merge-sort/

14.7 References

https://medium.com/15-minute-algorithms/merge-sort-in-15-minutes-or-less-abdf1f161480

$https://en.wikipedia.org/wiki/Bubble_sort$

https://softwareengineering.stackexchange.com/questions/297160/why-is-mergesort-olog-n

https://stackoverflow.com/questions/29927439/algorithms-how-do-divide-and-conquer-and-time-complexity-onlogn-relate

https://stackoverflow.com/questions/15799034/insertion-sort-vs-selection-sort

$https://en.wikipedia.org/wiki/Merge_sortcite_note-2$

https://medium.com/basecs/pivoting-to-understand-quicksort-part-1-75178dfb9313

https://medium.com/basecs/making-sense-of-merge-sort-part-1-49649a143478

https://stackoverflow.com/questions/29927439/algorithms-how-do-divide-and-conquer-and-time-complexity-onlogn-relate