CS 008
Lecture notes
5/7/24

1 Sorting

1.1 Outline

- Why do we care about sorting?
- Search algorithms
- Sorting algorithms

2 Why do we care?

Sorting and searching algorithms are both really important because sorted arrays are a lot nicer to work with. You off load some of the work by sorting an array beforehand to create more efficient work.

There are tradeoffs for this (like everything in computer science) and this all depends on what's needed for the task.

When we look back at sets, we can see that the find function can be a lot more efficient if the set / array is sorted.

3 Search algorithms

Sequential search is when you go through every element in the array (or until you get to your desired element) to look for your element. However, a sequential search has a time complexity of O(n) and if you have a sorted array, you can shorten this search by using a binary search. A binary search has been mentioned before in earlier lectures and has a benefit of being faster than a sequential search but as mentioned before, everything has tradeoffs and a binary search can only be performed on a sorted array.

4 Sorting algorithms

Sorting algorithms are characterized by the following:

- Computational complexity
- Spatial complexity
- Destructiveness
- "In-place" (how much extra spatial complexity the algorithm uses)
- Stable or unstable

4.1 Stability

Stability in sorting algorithms imply that the order of "equal" items are in the same order as before. Equal-valued items are not exactly equal to each other and having a stable sorting algorithm means that the order before the sorting algorithm between equal value items are not changed after the sorting algorithm (which can be useful in some cases).

4.2 Sorting algorithms

As mentioned before, there are tradeoffs for everything and this is highlighted in sorting algorithms. Each have their pros and cons but there is no "best" sorting algorithm in general. Each sorting algorithm have their own strengths and weaknesses depending on the use case.

4.3 Permutation sort

As the name implies, this sorting algorithm uses permutations of the array itself. What this means is that this algorithm creates permutations (specific combinations of the array) and then checks if it's sorted. This algorithm is a bit of a joke and has a time complexity of $O(n! * n)$. This algorithm is not stable.

4.4 Selection sort

Selection sort is an algorithm that sorts an array one piece at a time and swaps values to create a sorted array. The time complexity for this is $O(n^2)$ and is stable depending on the implementation but is in-place.

4.5 Insertion sort

Insertion sort is an algorithm that sorts an array by sorting an initially small chunk of the array and then growing that chunk and sorting it until that

chunk becomes the entire array. The time complexity of this algorithm is $O(n^2)$ and stability depends on implementation and is in-place.

4.6 Merge sort

... IMPLEMENT LATER

4.7 Direct access array sorting

Sorting an array using a direct access is a lot simpler than the algorithms talked about but they do not handle duplicates well, which we call counting sort and is defined to be $O(n + U)$. However, we learned before that with direct access arrays, we could use a hash table to counter the issue of duplicate collisions. Using a hash table and a queue, we can still maintain stability with the sorted array. Which introduces us to Radix Sort.

4.8 Radix Sort

Radix Sort builds on counting sort. Using keys that are not just the values themselves but an ordered series of keys, we can use these series of keys to sort the array with counting sort. In general, the amount of times needed to do counting sort is $log_n(U)$