CSE 332: Data Structures and Parallelism

Sorting 5 Solutions

Different Sorts of Sorting

For each of the following sorting algorithms, indicate if they are in-place and stable. If the option is implementation-dependent, ensure your choice maintains the expected runtime. Justify your decision.

Solution:

(a) $\mathcal{O}(n^2)$

- **Insertion Sort**
  As we showed in lecture on Monday, Insertion Sort is in-place and stable.
- **Selection Sort**
  This is an in-place sort, and it can be implemented as a stable sort. An implementation that is stable requires strictly more operations than one that is not, but can still be implemented in $\mathcal{O}(n^2)$.

(b) $\mathcal{O}(nlogn)$

- **Merge Sort**
  This is usually implemented as a stable sort. To make this in-place actually has not been a trivial problem, and was an area of research for a few years back in the 1980s; there does now exist a stable, in-place implementation that still takes $\mathcal{O}(nlogn)$ time, albeit requiring more (though constant) extra space. That said, that method is not typically implemented because it's difficult to write.
- **Quicksort**
  This is in-place but not generally stable.
- **Heapsort**
  This is in-place as well, but recall that because heaps don't maintain FIFO, this is not stable.

Project Gutenberg has another task for you: They want to sort all their unique-length books by length.

(a) If the longest novel is (according to Wikipedia) 2,070,000 words, and you didn't care about space efficiency, how fast could you sort their books? Could you do it in faster than $\mathcal{O}(nlogn)$ time? How?

Solution:

If we don't care about space, and we have a maximum value for the number of words a book can have, then why not just create a 2,070,001-index array (so the largest index in it is 2,070,000) of strings, where you store a book title in the index corresponding to the length of that book. Now it's sorted by length. This takes $\mathcal{O}(\text{max length})$ time. If our max length is 2,070,000, this is *technically* $\mathcal{O}(1)$; but if we extrapolate our solution to handle any max length, where the max length is represented as n, then this takes $\mathcal{O}(n)$ time.
This solution is actually closely related to Counting Sort and Bucket Sort, two interesting sorting algorithms you may be interested in learning about on your own (though Bucket Sort will be covered in lecture on Friday).

(b) Let's now just consider the numbers involved in the previous situation. If we did the same method as in part (a) to just sort the numbers, in Java, roughly how much space would we end up using? Is there any way we could reduce it further? Even get it down below half a megabyte?

Solution:

We can first use the same idea we used in p1 to go from maps to sets: if we don't care about the strings anymore and are just sorting the integers, we can use an array of booleans, where if the ith index stores True, then i is in our list of integers we're sorting, else it isn't. This is still naturally sorted, and still takes n time if the maximum int in our list is n; but in Java, each boolean is 1 byte, so ignoring any array overhead, our solution is n bytes (well, technically $n+1$, but OBOBs aren't much of a concern here). If $n = 2,070,000$ as in the previous situation, this is more than half a kilobyte.
We can, however, notice that one byte isn't the smallest amount of information we can use to store True/False information. All we actually need to do that is a bit, where 1 indicates "True" and 0 "False". So what we do is, instead of an array, use a **Bit String** with $n+1$ bits (so the first bit represents the int 0). If the ith bit is 1, then i is in our list we want to sort, else it isn't.
This solution is also $\mathcal{O}(n)$, but it shrinks our space usage by 8, from about 2 megabytes to about 250 kilobytes! Of course, this (just like the previous solution) is only useful if the number of elements we're sorting is *close to* the value of the maximum element-if we have a 2 million bit string but only ten ints to sort, a bit string is not at all a space-efficient solution.
This Bit String idea is actually a real data structure, one for which Java has an API. It's called a **Bit Set**. Bit Sets are simply ways of representing lists of integers (so long as they have a maximum value) in as little space as possible (so long as the list is large compared to the maximum value). It provides $\mathcal{O}(1)$ find, insert, and delete, and $\mathcal{O}(n)$ time to sort int lists. It has a lot of limitations though, including that it requires the lists to be of ints, that you really need a large number of ints for it to be the most space-efficient solution, and also that it can't handle duplicates. Still, it's a pretty interesting data structure that you may be interested in exploring.
You will very likely not be tested on this material, but you may still find it useful to understand, in and beyond this course.