Week6

Studying Sorting through an Algorithmic Lens

In this section, we will study sorting as an algorithmic problem, addressing general issues about sorting algorithms.

Lower Bound for Sorting

Can we sort any faster than O(n log n) time?



[Explanation is in the book]

Linear-Time Sorting: Bucket-Sort and Radix-Sort

In the previous section, we showed that Ω(n log n) time is necessary, in the worst case, to sort an n-element sequence with a comparison-based sorting algorithm. Are there other kinds of sorting algorithms that can be designed to run asymptotically faster than O(n log n) time? Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted. Example:

- sorting integers from a known range,

- sorting character strings.

Bucket-Sort

Consider a sequence S of n entries whose keys are integers in the range [0,N−1], for some integer N ≥ 2, and suppose that S should be sorted according to the keys of the entries. In this case, it is possible to sort S in O(n+N) time.

[Check (both theory and implementation) online, preferably on udemy.]

[Check other than array implementations as well.]

Time Complexity:

Runs in O(n + N) time and uses O(n + N) space.

n : sequence size

N : range of values.

An important property of the bucket-sort algorithm is that it works correctly even if there are many different elements with the same key.

Stable Sorting

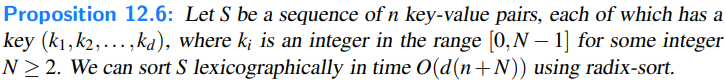
**Stable**: example: if we have the characters a-c-b-b, after sorting: a-b-b-c, but the two b characters did not change place with each other.

In bucket-sort stability is guaranteed as long as we ensure that all sequences act as queues, with elements processed and removed from the front of a sequence and inserted at the back. That is, when initially placing elements of S into buckets, we should process S from front to back, and add each element to the end of its bucket. Subsequently, when transferring elements from the buckets back to S, we should process each B[i] from front to back, with those elements added to the end of S.

Radix-Sort

[Check (both theory and implementation) online, preferably on udemy.]

[Check other than array implementations as well.]



Comparing Sorting Algorithms

Briefly:

O(n2) : insertion-sort and selection-sort

O(n log n) : heap-sort, merge-sort, and quick-sort[Read about it below.]

O(n) [Only for certain types of keys] : bucket-sort and radix-sort

Selection-Sort

Pretty bad in every case

Insertion-Sort

If implemented well, the running time of insertion-sort is O(n + m), where m is the number of inversions (that is, the number of pairs of elements out of order). Thus, insertion-sort is an excellent algorithm for sorting small sequences (say, less than 50 elements), because insertion-sort is simple to program, and small sequences necessarily have few inversions. Also, insertion-sort is quite effective for sorting sequences that are already “almost” sorted.

Apart from these special contexts, insertion-sort is also pretty bad.

Heap-Sort

The heap-sort runs in O(n log n) time in the worst case, which is optimal for comparison-based sorting methods. Heap-sort can easily be made to execute in-place, and is a natural choice on small- and medium-sized sequences, when input data can fit into main memory. However, heap-sort tends to be outperformed by both quick-sort and merge-sort on larger sequences. A standard heap-sort does not provide a stable sort, because of the swapping of elements.

Quick-Sort

Although its O(n2)-time worst-case performance makes quick-sort susceptible in real-time applications where we must make guarantees on the time needed to complete a sorting operation, we expect its performance to be O(nlog n) time, and experimental studies have shown that it outperforms both heap-sort and merge-sort on many tests. Quick-sort does not naturally provide a stable sort, due to the swapping of elements during the partitioning step.

Merge-Sort

Merge-sort runs in O(nlog n) time in the worst case. It is quite difficult to make merge-sort run in-place for arrays, and without that optimization the extra overhead of allocate a temporary array, and copying between the arrays is less attractive than in-place implementations of heap-sort and quick-sort for sequences that can fit entirely in a computer’s main memory.

Even so, merge-sort is an excellent algorithm for situations where the input is stratified across various levels of the computer’s memory hierarchy (e.g., cache, main memory, external memory). In these contexts, the way that merge-sort processes runs of data in long merge streams makes the best use of all the data brought as a block into a level of memory, thereby reducing the total number of memory transfers.

[Note: **Tim-Sort**: is a hybrid approach that is essentially a bottom-up merge-sort that takes advantage of initial runs in the data while using insertion-sort to build additional runs.]

Bucket-Sort and Radix-Sort

Finally, if an application involves sorting entries with small integer keys, character strings, or d-tuples of keys from a discrete range, then bucket-sort or radix-sort is an excellent choice, for it runs in O(d(n+N)) time, where [0,N −1] is the range of integer keys (and d = 1 for bucket sort). Thus, if d(n+N) is significantly “below” the nlog n function, then this sorting method should run faster than even quick-sort, heap-sort, or merge-sort.

Selection

There are a number of applications in which we are interested in identifying a single element in terms of its rank relative to the sorted order of the entire set. Eg. Finding minimum or maximum element, median, etc.

In this section, we discuss the general order-statistic problem of selecting the kth smallest element from an unsorted collection of n comparable elements. This is known as the **selection problem**.

Prune-and-Search [prune : ritkit]

We can indeed solve the selection problem in O(n) time for any value of k. The design pattern we use is known as **prune-and-search**.

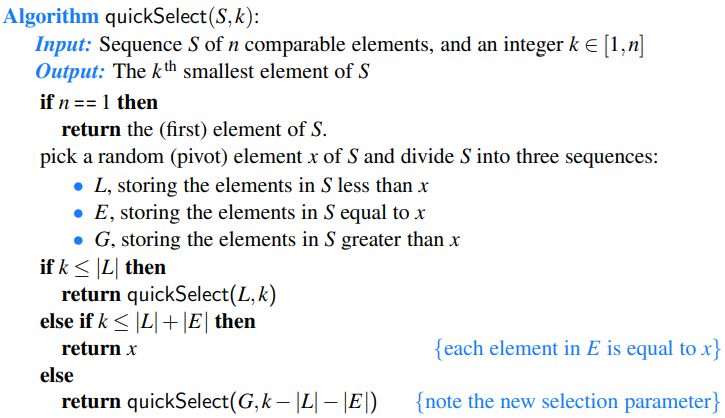
In applying this design pattern, we solve a given problem that is defined on a collection of n objects by pruning away a fraction of the n objects and recursively solving the smaller problem. When we have finally reduced the problem to one defined on a constant-sized collection of objects, we then solve the problem using some brute-force method. Returning back from all the recursive calls completes the construction. In some cases, we can avoid using recursion, in which case we simply iterate the prune-and-search reduction step until we can apply a brute-force method and stop. Incidentally, the binary search method described earlier [in Week1] is an example of the prune-and-search design pattern.

Randomized Quick-Select

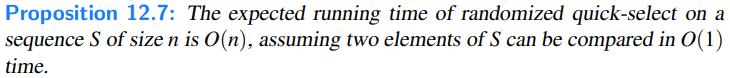
In applying the prune-and-search pattern to finding the k th smallest element in an unordered sequence of n elements, we describe a simple and practical algorithm, known as **randomized quick-select**. This algorithm runs in O(n) **expected** time. We note though that randomized quick-select runs in O(n2) time in the worst case.

Suppose we are given an unsorted sequence S of n comparable elements together with an integer k ∈ [1,n]. At a high level, the quick-select algorithm for finding the kth smallest element in S is similar to the randomized quick-sort algorithm described earlier. We pick a “pivot” element from S at random and use this to subdivide S into three subsequences L, E, and G, storing the elements of S less than, equal to, and greater than the pivot, respectively. In the prune step, we determine which of these subsets contains the desired element, based on the value of k and the sizes of those subsets. We then recur on the appropriate subset, noting that the desired element’s rank in the subset may differ from its rank in the full set.

Pseudocode:



Analyzing Randomized Quick-Select



[Explanation with recurrence equation is in the book.]

Maps