IT5003 Oct-Dec 2023

Data Structures and Algorithms

Tutorial+Lab 02
Sorting

Document is last modified on: October 2, 2023

1 Introduction and Objective

Today, we will discuss various Sorting algorithms.

You are encouraged to review e-Lecture of `https://visualgo.net/en/sorting?slide=1` to the last slide 19-5, but skipping Slide 16 to 16-2 (Radix Sort) to save a bit of time (Radix Sort and a few other details of other sorting algorithms are discussed in harder algorithm courses).

2 Questions

Sorting, Applications

Q1). At `https://visualgo.net/en/sorting?slide=1-2`, Steven mentions a few (not exhaustive) array (also known as list) applications that become easier/more efficient if the underlying array is sorted. Now, we will quickly discuss application *a few* of these application 1-7 in algorithmic level only.

Note to TAs: Explaining all 7 applications will consume lots of time. Just pick 3-4 that are more relevant for the current semester, i.e., more directly applicable to the ongoing PS tasks.

Consider discussing binary search first (application 1 of sorted array (or list)), the easiest: check middle, go left or right or stop.

Then for application 2, min/max/k-th smallest item are A[0], A[n-1], and A[k-1], respectively, if array A is sorted. Discuss 0-based versus 1-based indexing for k. What is the special name for the n/2-th smallest (or largest) item? (ans: median, if n is odd, then the median is a single number, otherwise there are two medians)

For application 3, we can detect uniqueness of sorted array A by checking if A[i] and adjacent neighbor A[i+1] are different for i in [0..N-2]. Note that to delete such duplicates, we better use a separate array to maintain $O(N)$ speed as using the remove(i) method of ListArray if A[i] and A[i+1] are the same will be costly ($O(N^2)$, think about it) if all items are actually the same.

For application 4, we can first find the position i of v using binary search (application 1), then we go left from i to find lowerbound l (first occurrence A[l] == v) and go right from i to find upperbound u (last occurrence where A[u] == v) and the answer is u-l+1. This is $O(\log n + k)$ where k is the length of the answer (but if $k = O(n)$, then this is $O(n)$ again). However, we can also modify binary search to directly find lowerbound l and upperbound u and this is $O(2 * \log n) = O(\log n)$. Counting sort style answer is also possible but that requires $O(n)$ scan and a big memory (not feasible if the range of the numbers is big).

For application 5, assuming both arrays A (of size n) and B (of size m) are sorted, assuming $n < m$, we can go through each element v of A one by one and use **binary** search to see if the same element v exists in B. For set intersection, we add v into answer if we find v in B too. Overall $O(n \log m)$. For set union, we go through A and B and use the same technique above but don't add the duplicate, so $O(n \log m + m \log n)$. But we can also use the two pointers solution like merge sort 'merge' operation to have $O(n + m)$ complexity. PS: There is also a good hashing-based solution but we have not reached that data structure yet at this point of time.

For application 6, target pair (2-SUM) problem, we can go through each element x of A and use binary search if $y = z - x$ exists or not. This is $O(n \log n)$. However, as A is already sorted, we can use two pointers approach (front and back) to solve this in $O(n)$. PS: There is also a good hashing-based solution but we have not reached that data structure yet at this point of time.

For application 7, after sorting, we can use two binary searches like in application 4, but this time we find the index of A that is right after the last occurrence of value that is at most hi and subtract that with the first index of A that is at least lo. This is $O(\log n)$. PS 1: Sometime later throughout this module, we will also learn some applications where greedy algorithms 'emerge' after we 'sort the initially chaotic input data'.

PS 2: Introduce Python Array bisection algorithm,
`https://docs.python.org/3/library/bisect.html`, e.g., (`bisect_left(A, v)`) and `bisect_right`.

Sorting, Mini Experiment

Q2). Please use the 'Exploration Mode' of `https://visualgo.net/en/sorting` to complete the following table. You can use 'Create' menu to create input array of various types. You will need to

fully understand the individual strengths and weaknesses of each sorting algorithms discussed in class in order to be able to complete this mini experiment properly.

For example, on random input, Optimized (Opt) Bubble Sort that stops as soon as there is no more swap inside the inner loop runs in $O(N^2)$ but if we are given an non-decreasing numbers as input, that Optimized Bubble Sort can terminate very fast, i.e., in $O(N)$.

Note that N-d and N-i means Non-decreasing (increasing or equal) and Non-increasing (decreasing or equal), respectively.

Note that Many Duplicates include All Equal test cases.

Note also that the term 'Nearly sorted' can have multiple definitions and we will discuss this in class.

Input type →		Sorted		Nearly Sorted		
↓ Algorithm	Random	N-d	N-i	N-d	N-i	Many Duplicates
(Opt) Bubble Sort	$O(N^2)$	$O(N)$				
(Min) Selection Sort					$O(N^2)$	
Insertion Sort		$O(N)$				
Merge Sort				$O(N \log N)$		
Quick Sort		$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					$O(N \log N)$
Counting Sort					$O(N)$	

Things to be discussed:

- The differences between sorted in ascending order (not the fully correct term if there is at least one duplicate elements in the input) versus sorted in non-decreasing order (a more general term and works if there are duplicate elements in the input). Similarly for descending versus non-increasing.
- There are multiple ways to define 'nearly sorted' (and also 'many duplicates'). A few possible definitions:
  - An almost sorted array with 'a few' pairs in wrong position (the one used in VisuAlgo, create(A), Nearly sorted option – and the current model answer for the answer table below that can make Optimized Bubble Sort that stops as soon as there is no more swap in the inner loop and Insertion Sort runs in $O(kN)$. If k is reasonably low and much smaller than N ($k < N$), most of the time we will treat this as $O(N)$). Note that we can compute k in $O(N^2)$ time by counting the number of Selection Sort swaps (there is another way to compute k using an $O(N \log N)$ algorithm),
  - An array with reasonably low inversion count. For example, this test case $A = [2, 3, 4, 5, 6, 7, 8, ..., N, 1]$ only has N inversions to move 1 (starting at the very back) to the front and many will say this is actually 'Nearly Sorted' with just one element in the wrong position. But Optimized Bubble Sort will run in $O(N^2)$ due to the need to wait until the very last pass to finally put 1 at the front. Insertion Sort still runs in $O(N)$ on this test case,

  - An array with many duplicates but 'looks nearly sorted'. For example, some people will say this test case $A = [2, 2, 2, 2, ..., 2, 2, 1, 1, ..., 1, 1, 1, 1]$ is 'nearly sorted' in a glance. However, it actually has lots of inversions and will still run in $O(N^2)$ for (Opt)imized Bubble Sort and Insertion Sort,
  - An array so that each element that is not in sorted position is just (small) k step away from its final sorted position. For this one, both Optimized Bubble Sort and Insertion Sort runs in $O(kN)$. On a reasonably low k, this will be treated as $O(N)$.

- If the input is in reverse sorted (non-increasing order) and the default behavior of the sorting algorithm is to sort by non-decreasing order, the sorting algorithm will usually do the most work (not just in terms of comparison, but also number of swaps).
- Non-randomized quick sort can be very slow on (nearly) sorted input and how randomization helps a lot here (details to be deferred until a more advanced algorithm *analysis* course like CS3230). You can make a remark that the $O(N \log N)$ time complexity of the Randomized Quick Sort algorithm is 'in-expectation'.
- How Selection Sort, Merge Sort, and Counting Sort (but Counting Sort only works if the range of integer is small) are not affected by input type at all (all these algorithms do exactly the same number of steps on all kinds of input)

Input type →		Sorted		Nearly Sorted		
↓ Algorithm	Random	N-d	N-i	N-d	N-i	Many Duplicates
(Opt) Bubble Sort	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^2)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^2)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
(Rand) Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Hands-on 2

TA will run the second half of this session with a few to do list:

- PS1 Quick Debrief,
- Do a sample speed run of VisuAlgo online quiz that are applicable so far, e.g., `https://visualgo.net/training?diff=Medium&n=5&tl=5&module=sorting`.
- Finally, live solve a chosen Kattis problems involving sorting.

Do PS1 quick debrief according to the context of your lab group.

Do VisuAlgo Online Quiz sample run in medium setting.

`https://nus.kattis.com/problems/judging`, technically a set intersection problem (application no 5 of Q1 :).

Sort DOMjudge list and Kattis list then use Merge sort's merge operation and count == items :).

Problem Set 2

We will end the tutorial with high level discussion of PS2 A+B.

For PS2 A (/chartingprogress), you are a given 'sorting' problem on a 2D-grid, but it is in column-major order and also in a 'new' input method (multiple test cases separated by blank lines)... The first step is to be able to read the input properly. Study `https://github.com/stevenhalim/cpbook-code/blob/master/ch1/IO.py` for using `sys.stdin` for reading the entire input first and then create the 2D-grid test-case per test-case (with that blank line (len of that line is 0) as separator). Once we have the 2D-grid, can we... turn a column-major matrix into a row-major matrix (to simplify the sorting process?). Hint: transpose the matrix...

For PS2 B (/massivecardgame), both N and Q is up to $100K$ (10^5), so a naïve $O(Q * N)$ algorithm will go up to 10^{10}, too slow. This PS is about sorting, so what if we sort N cards first upfront. This is $O(N \log N)$. Now what can be easier after we sort the N cards? From here, it is quite close to the solution.