CS2040S Semester 1 2023/2024

Data Structures and Algorithms

Tutorial+Lab 02

Sorting, ADT, List

For Week 04

Document is last modified on: September 6, 2023

MODAL ANSWER IS FOR OUR CLASS ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

1 Introduction and Objective

In this tutorial, we will discuss various Sorting algorithms and the concept of Abstract Data Type (ADT) that will appear several times throughout this course. We discuss List ADT today.

You are encouraged to review e-Lecture of `https://visualgo.net/en/sorting?slide=1` (to the last slide 19-5; if you read all these slides under login (all three sectors of the e-Slides are fully read), you will get a 'one star achievement' from the VisuAlgo system) and `https://visualgo.net/en/list?slide=1` (to the last slide 9-6; also aim to get another 'one star' here) for this tutorial.

2 Tutorial 02 Questions

Sorting

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 Java `Collections.binarySearch` algorithm.

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 test cases 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	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Finding k-th Smallest Element in an Unsorted Array (Selection Algorithm)

Q3). We will revisit the concept of QuickSort's partitioning algorithm (please review `https://visualgo.net/en/sorting?slide=12-2` to slide 12-7) and combine it with a binary search-like algorithm on an unsorted array to find the k-th smallest element in the array *efficiently*. In this tutorial,

we will spend some time discussing the details. Your job before attending this tutorial is to read this algorithm from the Internet: `http://en.wikipedia.org/wiki/Quickselect`, or if you have Competitive Programming 4 (or earlier) book, read about it in Section 2.3.4 (the earlier part of Order Statistics Tree).

Ideas:

1. We need a partition function (`Randomized_Partition`) to partition a given range [start, end] of the array about the given pivot, where all values < array[pivot] are placed before the pivot and all values > array[pivot] are placed after the pivot (equal elements can be thrown to either side as with the latest patch of VisuAlgo Quick Sort partition routine). The final position of the pivot, PivotIndex, is returned.

2. Next, we employ a standard binary-search-like method (a Divide and Conquer paradigm) to search the array. We start by using a random pivot element and calling the partition function. The position returned by the partition function indicates the final correct rank of the pivot element. If the rank is $k-1$ (assuming that k is 1-based), we return the pivot element. If the rank is $> k-1$, pass the range [left, PivotIndex-1] into the algorithm and repeat. If the rank is $< k-1$, pass the range [PivotIndex+1, right] into the algorithm and repeat.

3. The time complexity is *expected* $O(n)$. The detailed proof is beyond CS2040/C/S (your CS3230 lecturer may re-use this example).

To be discussed by tutors:
1. What is the best case for this Partition algorithm (regardless it is randomized or not)?
2. As with QuickSort, what happen to this QuickSelect algorithm if we do NOT randomize the pivot of partition?
3. How to convert the problem of finding k-th smallest element to finding k-th largest element?

Algorithm 1 QuickSelect(A, l, r, k)

```
if l == r then
    return A[l]
end if
PivotIndex = Randomized_Partition(A, l, r)      ▷ The pivot is randomly selected between l and r
if PivotIndex == k-1 then                                              ▷ Assume that k is 1-based
    return A[PivotIndex]
else if PivotIndex > k-1 then
    return QuickSelect(A, l, PivotIndex-1, k)                  ▷ We have to search on the left side
else
    return QuickSelect(A, PivotIndex+1, r, k)                  ▷ We have to search on the right side
end if
```

Abstract Data Type (ADT)

Q4). Please self-read List ADT and its common operations: `https://visualgo.net/en/list?slide=2-1`. Now compare it with the sample (fixed-size array) implementation discussed back in last Tutorial Tut01 and if we replace that fixed-size array with Java ArrayList implementation instead. What are the concepts of ADT that you have understood by now? We will discuss List ADT in more details in Lecture when we discuss 3 related ADTs: Stack, Queue, Deque.

To be discussed/mentioned:

1. ADT is usually implemented in OOP fashion (C++, Python, or Java class), encapsulating the technical details and only exposing the (simpler) interface (see below)
2. ADT specifies common operations on that abstract data type and does not care about the actual underlying data structure used (so far we have implemented List ADT using fixed-size array in Tut01 and then here we show that resizeable array/vector can also do the same; later we will use Linked List for this list ADT), thus the code that uses the ADT should remain working even if the underlying implementation of the ADT is changed; However, if we use the correct/best data structure, we can implement the ADT in the most efficient/versatile manner and this is what we want to learn in CS2040/C/S
3. ADT specification varies but at least the common operations are rather universal (constructor specification, search for an item, insert a new item, and remove existing item)

Hands-on 2

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

Note to TAs: It is well-known from past AYs that the tutorial side of tut02 will be much longer than 1 hour (but still, try not to over-discuss Q1+Q2). Thus, calibrate time by doing less on the Hands-on 2 section, i.e., by skipping sorting Online Quiz and/or only showing parts of the chosen hands-on task (no need to go full score).

- 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 another chosen Kattis problem involving sorting.

Kattis /basicprogramming2, is an implementation task of what we have discussed in Q1 earlier. It was previously used as graded PS but it has been too heavily overused. Try to get as many points as possible, applying what you have learned earlier in this tutorial/lab. To save time, focus on task $t = 1$ (the $x + y = 7777$) and purposely show the important binary-search related solution.

Problem Set 2

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

The first thing that TA has to clear is that for /universityzoning, we do *not* need any fancy algorithm (BFS :O) to compute the shortest distance between two cells in a grid because a simple $O(1)$ Manhattan distance is sufficient. You can tell students that last AY (2020/2021), NUS used this very unpopular zoning system...

Then, the high level explanation of each subtask:

1. Just print 1 (both sample case 1 and 2 have the same output... free 2 out of 100 marks)
2. $R = 1$ means 1 dimensional array. $G = F$ means all faculties must comply. $T = E$ means all students must comply. We can use brute force/complete search for this.
3. Same as Subtask 2, but R is not necessarily 1, so we are back to 2D array. As long as we use Manhattan distance to compute distances, this is not a big issue.
4. Same as Subtask 3, but G may be smaller than F, so we need to 'greedily' choose G easiest faculties to comply. We can sort these faculty values and pick G smallest... (this is a very major hint).
5. No other constraints, so T may be smaller than E, we need to 'greedily' choose T easiest students to comply. We need to use sort twice.

For PS2 B (/jobbyte), try to fully understand the problem first by using a naïve $O(N^2)$ algorithm... We have discussed this earlier. The answer is related to how Selection Sort algorithm works. For an $O(N \log N)$ solution (or $O(N)$, depending on how your implementation), this becomes a classic (mathematical) problem that was tested in CS2040C midterm test S1 AY2022/23 one year ago. Since students are allowed to use ChatGPT/the Internet, try asking it for a hint.