Sample Final Exam Questions

May 19

These are sample questions of the type you might find on the final exam. You should feel free to consult course materials as you work on these problems. You will also be able to use course materials on the final exam, as you were on the mid-term.

For each of the questions, a solution should briefly describe the appropriate data structures (if any), implementation, and/or algorithm(s), indicate how you would use them to design an efficient solution, and include a time and space complexity analysis. Specifically,

- Give Big-O time and space bounds for your solution. If your solution specifies multiple steps (e.g., you first populate a `Vector` of `String`s by reading in the contents a text file, then you search through the list elements one at a time), describe the Big-O time and space bounds for each step and the describe the total overall running time. You should focus on the worst case bounds.
- When describing a data structure, give sufficient implementation details so that your answer is unambiguous (e.g., do not simply say "graph"; instead specify "a graph using an adjacency matrix"; do not simply say "a priority queue," say a "min heap"; and do not simply say "a list", say a "doubly linked list" or "a singly linked list that has a head and a tail pointer" if those details are important).

Some solutions might benefit from using more than one data structure. Feel free to utilize multiple data structures in your solution if you find it helpful, but keep in mind that the goal is to optimize performance. When in doubt, prioritize time efficiency over space efficiency.

The quality of your responses will be measured with respect to correctness, efficiency (Big-O time complexity), and clarity of exposition.

No response to these sample questions should require the writing of Java code.

1. (*0 points*) ..... Removing Duplicate Elements ("small")

You are presented with an array of n arbitrary integers (i.e., you do not know the minimum value, the maximum value, or the distribution of the integers). You want to determine whether this array contains any duplicate values. Describe an efficient method that returns `true` if there are any duplicates, and `false` otherwise.

Solution:

An efficient example solution: Use a `Hashtable` or, even more simply, a `HashSet`).

The idea is to add the integers in the array to the `HashSet`, but to check before adding to determine whether the integer has already been added. If it has been, return `true` because you've found a duplicate. If not, add it to the `HashSet` and continue processing the remaining elements of the array. If we use a `HashSet` *and if we assume that* `get` *and* `put` *are* $O(1)$ *operations*, then the worst-case performance to check for duplicates is $O(n)$, since `get` and `put` are called at most once for each element of the array. The big-O space requirement is also $O(n)$, since the `HashSet` uses space proportional to the size of the array.

A less efficient example solution: Sort the array, using either mergesort or heapsort, each of which takes $O(n \log n)$ time. Then scan through the elements of the array to see whether any two consecutive elements are the same. Note: If there are any duplicates, there must be a pair of duplicates that occur in consecutive locations in the array. This step takes $O(n)$ time for a total runtime of $O(n + n \log n) = O(n \log n)$. No additional space is needed beyond the array, if heapsort is used (beyond a few constant-sized local variables for swapping elements, loop indices, etc–which we mention here but you wouldn't need to note in your exam solutions).

Third best solution: Use two nested `for` loops: For each element, compare it to every other element. If you find a match, return true. The best case and worst case big-O performance is $O(n^2)$.

Other correct but less efficient solutions will also receive partial credit.

2. (*0 points*) ..... Finding k Smallest Elements ("medium")

You are given an unordered Vector $data[]$ of n Comparable items (where n is large), and you are also given a positive integer k (where $k < n$). Your goal is to return the k smallest items from the `Vector` (from smallest to largest). Describe the most efficient algorithm you can for solving this problem under the following two scenarios:

(a) The value of k is a constant (that is, k is not a function of n).

(b) The value of k is approximately $\sqrt{n}$.

Note: This two-part problem would most likely be presented as two separate problems on the exam, but because of the effect of the constraints on the parameter k on the ordering of the different solutions, we thought it made sense to frame this as a single question.

Solution:

(a) **An efficient example solution:** For each i from 0 to $k-1$, find the smallest item in the vector in the slice $data[i..n-1]$ and swap it with the item i. Then return $data[i]$. Since finding the smallest item in a slice of the Vector requires a linear scan of that slice, the time complexity for performing the k scans is $O(k \times n) = O(n)$. The space complexity is also $O(n)$.

An equally efficient example solution: Produce a minHeap from the Vector in $O(n)$ time (using Bottom-up Heapify) and then call removeMin on the minHeap k times. The k calls to removeMin take $O(k \log n)$ time. The time complexity of the algorithm is therefore $O(n + k \log n) = O(n)$, since k is a constant, and the space needed is also $O(n)$.

A less efficient example solution: Sort the vector (in $O(n \log n)$ time as in the previous problem) and return the elements in positions $0, \ldots, k-1$ of the vector. Since an element of the vector can be accessed in $O(1)$ time, the time complexity is $O(n \log n + k) = O(n \log n)$, since k is a constant. The space needed is still $O(n)$.

Third best solution: Sort the vector using selection/insertion/bubble sort (requiring $O(n^2)$ time) and return the elements in positions $0, \ldots, k-1$ of the vector. The time complexity is $O(n^2 + k) = O(n^2)$ and the space needed is still $O(n)$.

(b) **An efficient example solution:** Produce a Heap from the Vector in $O(n)$ time (using Bottom-up Heapify) and then call removeMin on the Heap k times. The k calls to removeMin take $O(kn)$ time. The time complexity of the algorithm is therefore $O(n + k \log n) = O(n + \sqrt{n} \log n) = O(n)$ and the space needed is $O(n)$.

A less efficient example solution: Sort the vector and return the elements in positions $0, \ldots, k-1$ of the vector. The time complexity is $O(n \log n + k) = O(n \log n + \sqrt{n}) = O(n \log n)$ if you use mergesort or heapsort and the space needed is still $O(n)$.

Third best solution: For each i from 0 to $k-1$, find the smallest item in the vector in the slice $data[i..n-1]$ and swap it with the item i. Then return $data[i]$. The time complexity is $O(k \times n) = O(n^{3/2})$. The space complexity is $O(n)$.

Other correct but less efficient solutions will also receive partial credit.

3. (0 points) Flight Scheduling ("large")

When searching for airline reservations, Travelocity has a "Shortest Duration" option that will list options (including multi-stop flights) based on total duration (including the waits between flights). One feature missing from this option is the ability to indicate how early a passenger can arrive at their initial departure airport. This question asks how you would provide this option.

Specifically, you are given a large file of flight data of the form

```
FlightName FromAirport ToAirport DepartureTime ArrivalTime FlightDuration
```

for example

```
AA6183 ALB ORD 06:00 07:35 2:35
```

For simplicity, we assume that

- every flight runs every day
- all times are '24-hour' time
- all departure and arrival times are local to the departure and arrival airports

Note that including the duration of the Albany to Chicago flight above is useful because merely subtracting departure from arrival times wouldn't give the correct answer if the airports are in different time zones.

Your goal is to design a data structure for storing the information about all flights in the file that will support the following operation

- `Vector<String> bestFlightSchedule(String startAt, String finishAt, Time: arrivingAt)`

which produces an itinerary of the sequence of flights that will get the passenger from airport `startAt` to airport `finishAt` as quickly as possible given that they will show up at `startAt` at time `arrivingAt`.

You should assume that this method will be used frequently with different parameters.

You can assume that information about a flight is stored in a `FlightInfo` object that has accessor functions for all of the flight information. You can also assume that times are stored in a Time object that allows reasonable time computations (like addition and subtraction of time periods).

Describe an efficient implementation of a data structure that supports the efficient computation of the `bestFlightSchedule` method. A complete correct answer should include

- Descriptions of any data structures your structure might use
- A justification of why your implementation should be preferred to other alternatives
- The space complexity of your structure and the time complexities of each method

Solution:

An efficient example solution:

We will use a graph data structure, specifically the `GraphListDirected` structure from the `structure5` package.

- Each vertex will represent an airport, storing the airport name as its label
- Each edge $e = (fromA, toA)$ will represent the set of all flights from airport $fromA$ to airport toA. The edge label for edge e will store a vector of `FlightInfo` objects for each flight from $fromA$ to toA, sorted by departure time (early to late).

We assume that the edge label supports a method `bestFlight(Time t)` which returns the `FlightInfo` for the flight from $fromA$ to toA that has the earliest arrival time to toA of all flights from $fromA$ to toA whose departure time is no earlier than time t.

The method `bestFlightSchedule(startAt, finishAt, arrivingA)` now works by running Dijkstra's algorithm for the single source shortest paths problem, starting at vertex `startAt`, but with two modifications:

- Instead of the Priority Queue storing the entire edge label of an edge, it just stores the `FlightInfo` object for the flight returned by `bestFlight()`.
- The priority of the edge (flight) is the sum of the time spent in airport $fromA$ since arriving there before taking the flight returned by `bestFlight()` added to the duration of the flight returned from `bestFlight()`.

The tree of paths built by Dijkstra's algorithm will now have shortest paths from `startAt` to every airport A reachable from `startAt` by some sequence of flights, where the path length now reflects the shortest amount of time required to get to airport A given that the the traveller arrived at `startAt` at time arrivingAt.

The space required for this algorithm is proportional to the number of flights in the file, assuming that the `GraphListDirected` implementation of the graph is used. If the file contains F flights then the space required is $O(|V| + |F|)$, since each edge is storing multiple flights but each flight is only stored in a single edge.

The time required by the original version of Dijkstra's algorithm is $O(|V| + |E| \log |E|)$. The only impact of storing multiple flights at each edge is that before adding an "edge" (tt object) to the priority queue, the `bestFlight()` method must be run. This method takes time proportional to the number of flights stored at that edge, so, over the entire run of the algorithm, the time spent adding to the priority queue will be no worse than $O(|F| \log |E|)$ (there is at most 1 flight in the priority queue for each edge, but figuring out which flight to add requires looking at all flights associated with that edge).

Thus our method takes time $O(|V|+|F| \log |E|)$, which is, at worst, $O(|V|+|F| \log |F|)$. In fact, since the method will only visit vertices reachable from V, we can simplify the runtime to $O(|F| \log |F|)$.