Boston University, CS 330 Fall 2020, Midterm Practice Problems

How to Study for the Exam

Different people have different ways of studying for exams, so please incorporate this advice in the ways that work best for you! Regardless, of how you go about it, here's a good rule of thumb: you really understand a topic when you can *teach it to someone else* (ideally, without notes). Friends who are struggling with some concept make especially good practice buddies for this!

- For each of the **algorithms and data structures we've seen in class**
  - Explain the **input and output, and what correctness means**.
  - **Run the algorithm on examples**. You should have no doubt about how the algorithm will handle different types of inputs.
  - **Write pseudcode** for the algrithm: you should be able to write clean pseudocode for the algorithm in class from scratch without consulting your notes. That does not mean you should memorize the pseudocode—it means you should understand the algorithm well enough that you can regenerate pseudocode for it on your own.
  - State and explain the **running time** worst-case bound in terms of standard measures of input length (e.g. $m = |E|$ and $n = |V|$ for graphs).
  - **Write and explain the proof of correctness** (again, the goal is not to memorize the proof, but understand it well enough to teach it to someone else). Do you understand all the steps of the proof? Do you find it convincing?
  - **Code it up** and run it! If if you don't actually code it up, it should be clear to you how you would go about it. (A typical job interview question asks you to write code in your favorite language for some well-known algorithm.) For example, was it hard to write BFS code for the programming assignment? What were the obstacles? Would you be able to do the same for all the other algorithms we've discussed in class? If not, where is the first place you would get stuck? (That's makes for a great question in office hours.)
- For each of the **homework and lab problems**, make sure you understand the solution (at the level of the algorithms we saw in class). Exam problems that are very similar to some homework or lab problem, or to a practice problem.
- **Solve lots of problems!** Go through the practice problems below as well as the exercises in the book. Solve as many as you can. Work on them with friends!
- **Get help when there's something you don't understand!** Take advantage of office hours, Piazza, and the other students in the class to ask questions. No question is too silly or basic—if you are confused about something, you can bet that lots of other students are equally confused.

  Sometimes there will be topics where you're not even sure how to formulate the right question. If that's the case, come to office hours and we will try to help you. For any given algorithm or exercis, ask yourself where is the first place you are getting confused. Do you understand what the inputs and outputs are? Do you understand each line of the pseudocode? Etc.

About the questions below: Most of the questions below are challenging. We haven't put questions like "what would Dijsktra's algorithm output on this graph?" in the list, because you can and should generate questions like that for yourself, on your own. (Of course, if you're not sure of the answer, then you can ask for help.)

Questions marked with an asterisk (*) are more challenging.

Stable Matchings

1. KT, Chapter 1, Problems 1 to 8. Problems 3, 4, 5, 6 are useful for practicing transforming other problems into the stable matching problem, so you can run Gale-Shapley.
2. (*) Show that there are instances of the stable matching problem with exponentially-many stable matchings. Specifically, show a family of instances (that is, one for each value of n) where there are at least $2^{\lfloor * \rfloor n/2}$ stable matchings

Asymptotics and data structure basics

1. For each of the following statements, decide whether it is always true, never true, or sometimes true for nonnegative functions f and g. If it is always true or never true, give a proof (using the definitions of $O()$, $\Omega()$, $o()$, etc). If it is sometimes true, give one example for which it is true, and one for which it is false.

   (a) $f(n) = o(g(n))$ and $f(n) = \Omega(g(n))$

   (b) If $f(n) = O(g(n))$, then $2^{f(n)} = O\left(2^{g(n)}\right)$

   (c) If $f(n) = O(g(n))$, then $g(n) - f(n) = \Omega(g(n))$.

   (d) If $f(n) = o(g(n))$, then $g(n) - f(n) = \Omega(g(n))$.

   (e) $f(n) + g(n) = O(\max(f(n), g(n)))$

   (f) If $f(n) = O(g(n))$ and $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, then $\log(f(n)) = O(\log(g(n)))$.

   (g) (*) Either $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ or both.

2. Suppose we have k sorted lists, each with exactly n elements, that we would like to merge (so we end up with a sorted list of length kn) . Suppose that we proceed in the most straightforward way: we merge the first two lists (using the merge subroutine from mergeSort), then merge the result with the third list, then with the fourth list, and so on, merging with one of the original lists in each step. What is the total asymptotic running time of this procedure (in big-Theta notation)?

3. Consider the same problem as above: Suppose we have k sorted lists, each with exactly n elements, that we would like to merge (so we end up with a sorted list of length kn).

   This time, we do a k-way merge as follows: we maintain a binary min-heap of size k, containing the smallest elements from each of the input lists. Now, we proceed for k iterations: in each iteration, we extract the minimum element from the heap and add it to the output array. Then we delete the corresponding element from the input list where it came from, and insert the new smallest element from that list to the heap. What is the overall running time of this algorithm (in big-Theta notation)?

4. KT, Chapter 2, Problem 6 (sums of subarrays)

5. * KT, Chapter 2, Problem 8 (breaking glass bottles)

6. Recall the Priority Queue ADT allows for the following operations: Insert, Extract-Max, Change-Priority (from class). Suppose we implement a priority queue using a hash table, where Insert(priority,identifier) has the effect of adding the pair (identifier, priority), using identifier as the value to be hashed. How fast can the basic priority queue operations be implemented using this particular data structure, assuming the priority queue contains n elements?

   NB: Change-Priority(identifier, newPriority) should have the effect of removing a pair of the form (identifier, oldPriority) if one exists and replacing it with (identifier, newPriority).

7. Recall the Priority Queue ADT allows for the following operations: Insert, Extract-Max, Change-Priority (from class). Suppose we implement a priority queue using a sorted doubly linked list, where Insert(priority,identifier) has the effect of adding a node with contents (identifier, priority), while maintaining the invariant that the list is sorted by priority. How fast can the basic priority queue operations be implemented using this particular data structure, assuming the priority queue contains n elements?

8. Suppose we wish to use a priority queue inside another algorithm, where we know there will be n Insert operations, n^2 Change-Priority operations, and n Extract-Max operations.

   What is the total asymptotic expected running time of the Priority Queue operations when the PQ is implemented using...

   (a) a doubly linked list (as one question above)?
   (b) a hash table (as two questions above)?
   (c) a binary heap (as in the textbook)?

   (Use your answers to the previous questions.)

   Which data structure should we use to implement the priority queue in order to minimize the asymptotic expected running time?

9. Suppose we now have to implement a priority queue in a setting where we know the priorities always come from the set $\{1, ..., r\}$. We can consider yet another data structure: we maintain an array of r lists, where the i-th list contains all the elements currently in the priority queue that have priority i. How fast can the basic priority queue operations be implemented using this particular data structure, assuming the priority queue contains n elements?

10. Suppose that we want to implement the following data structure: the contents of the structure is a set of pairs *(name,quantity)* where *name* is a string with maximum length k (an integer constant) and *quantity* is a floating point number.

    The update operation is *Transaction(name,amount)* with the following meaning: if before the operation the set does *not* contain a pair *(name,something)* then a pair *(name,amount)* is inserted; if such pair *does* exist then it is replaced with *(name,something + amount)*.

    The data structure should support the query *Balance(name)*: if the set contains a pair *(name,something)* it returns *something*, otherwise it returns 0.

You can adapt one of the data structures that we have learned to implement a sequence of n updates and queries with as small expected running time as possible.

11. Suppose that we want to implement a data structure with status and operations as before, but with an additional query *High(name)* which returns the highest value that could have been returned by *Balance(name)* so far. E.g. in a sequence of operations *Transaction(Piotr,3), Transaction(Piotr, 4), Transaction(Piotr,-5), High(Piotr)*, the last operation returns 7. Again, you can adapt data structures that we have learned so far to implement a sequence of n operation with as small expected running time as possible.

12. (Finding a Needle in Binary Heap) We can insert elements into a heap in logarithmic time and find the smallest element among those in the min-heap in constant time. In addition, we can remove an element from it in logarithmic time, *if we know the the index of the element we want to remove.* The downside of using a heap is finding the location of an element, which can take linear time.

    However, finding an element in data structures is a required functionality in many applications and its efficient execution is often desired.

    Design a data structure that offers *find* operation in constant time, in addition to supporting *insertion*, *removal*, and *finding the minimum element* in logarithmic time.

    Hint: Use a hash table next *and* a min-heap. Use your hash table to find the location of elements inside the min-heap, and use the min-heap to find the smallest element, both in constant time. You need to keep them in sync for operations that change the content of your data structure.

13. Binary search trees are dictionaries that store $(key, value)$ pairs, and support extra operations efficiently. Suppose we have a binary search tree with n items in it, and suppose the tree currently has height h. For each of the following operations, give the asymptotic running time for the best implementation you can think of. (Even better, write pseudocode for the operation.)

    Example: for the first operation (node with minimum key), the answer would be $O(h)$.

    (a) Given a tree T return a pointer to the node with the smallest key.

    (b) Given a tree T and a number z, find the node whose key is largest among all those with value less than z.

    (c) Given a tree T and a pointer to a node x, return a pointer to a node with the next largest key (that is, the next key after $x.key$ in the sorted order).

    (d) Given a tree T, a node x and a positive integer k, find the nodes with the next k largest keys after $x.key$. [*Hint:* There is a straightforward way to show you can do this in time $O(kh)$. However, there is a simple algorithm that runs in time $O(h + k)$. Can you see why?]

14. **(Combining indexes.)** Suppose you manage a data set of n student records, each of which has a unique ID, a name, a date of birth, a GPA, as well as more detailed information (a complete transcript, etc). Your data structure should allow efficient insertion and deletion. Furthermore, it should store only 1 copy of the detailed information for each student.

How could you organize a data structure that would allow you to search for students by either name or student ID or date of birth in $O(1)$ time? What if you wanted to be able to quickly find all the records with birth dates in a particular range?

Data structures - adjacency lists and matrices

1. **graph statistics.** You are given a undirected/directed graph $G(V, E)$ with $|V| = n$ nodes and $|E| = m$ columns. How many operations does it take to compute the following values if G is stored in an adjacency matrix or list? Computing the degree/indegree/outdegree of one particular node. Finding the number of nodes with degree/indegree/outdegree k. Finding the set of neighbors/ incoming or outgoing neighbors of a particular node.

2. **graph square.** The *square* of a graph $G(V, E)$ is the graph $G^2(V, E^2)$, where the set of nodes is the same, and there is an edge between two nodes u and v iff they are connected by either an edge or a path of length two. How long does it take to compute G^2 for G in an adjacency list data structure? (Fun fact: In an adjacency matrix you can do this in fact by a simple matrix multiplication!)

3. **degree of neighbors.** Given a graph $G(V, E)$ in an adjacency list data structure find the array `neighbor_sum` of length n in linear time, where `neighbor_sum`$[u]$ holds the sum of the degrees of u's neighbors.

Directed graphs.

1. **Semi connected graphs.** A directed graph $G(V, E)$ is *semi connected* if for any pair of nodes u and v there exist either a directed pathf rom u to v or from v to u. Find an efficient algorithm that takes G as input and decides whether it is semi connected.

2. **Topological order.** Let $G(V, E)$ be a directed graph and let $v_1 \preceq v_2 \preceq \ldots \preceq v_n$ be a topological order of its nodes. Show that for any two consecutive nodes v_i and v_{i+1} in this order, if they are not connected by an edge, then we can swap them, and the resulting order is still a valid topological order.

3. *Is the topological order unique?* Give an algorithm that takes a DAG as input, and returns "True" if the graph has only one topological ordering (and "False" otherwise). Your algorithm should run in $O(m + n)$ time.

Greedy algorithms

1. **Latest start time first.** You want to solve the interval scheduling problem from class; given n jobs, each job j with a starting time s_j and finish time f_j, find a maximum size subset of jobs, such that they don't overlap. For this problem we discussed the Earliest-Finish-Time-First algorithm. Let S_E be the schedule output by this algorithm. Now, consider a different algorithm to solve this problem; we order jobs in *decreasing order* of starting times.

That is, we process the job, with the latest starting time first. Does the Latest-Starting-Time-First algorithm output an optimal schedule? Let this schedule be S_L. Are S_E and S_L always/never/sometimes the same?

2. **Job scheduling on two machines.** You are given n jobs and *two* machines. Each job has a given start time s_i and finish time f_i. Find a feasible schedule of maximum number of jobs assigned to one of two machines, so that no two jobs overlap.

3. **Greedy Stays Ahead** KT Chapter 4, Problems 4, 6, 7, 16, 17.

4. **Exchange Arguments** KT Chapter 4, Problem 12

5. **(Duality)** KT, Chapter 4, problem 14 (security checks on sensitive processes). Notice that there is a connection to the "interval scheduling" problem from lectures.

Greedy graph algorithms

1. **Shortest paths with vertex costs** Suppose you have a directed graph with nonnegative edge lengths representing travel times. Furthermore, suppose there is a travel time for passing through each vertex v (denoted t_v). Give an algorithm that computes shortest travel times from a source vertex s to all other vertices, where the travel time along a path is sum of the lengths of all the edges on the path, plus the sume of the travel times of all the intermediate vertices. [*Hint:* Modify Dijkstra.]

2. **Shortest paths with evolving edge costs** KT, Chapter 4, Problem 18.

3. **Unique MST.** Let G be an undirected weighted graph. Show that if each edge weight is unique then there is only one minimum spanning tree in G.

4. **Modifiying MSTs** Suppose you have already computed the MST T for an undirected, weighted graph G. For each of the following modifications to the graph, show how you can modify T to get the new MST T':

   (a) A new edge e with weight w_e is added to the graph

   (b) An existing edge e is *removed* from the graph [this problem was on Homework 7.]

   (c) The weight of an existing edge e is changed. (You might want to handle the cases where it is increased or decreased separately.)

5. **Modifiying shortest path trees.** Suppose G is a graph with nonnegative edge weights, and that we have already run Dijkstra's algorithm to compute a tree of shortest paths from a particular vertex s. Show (by giving a family of examples) that adding one edge to the graph can cause lots of edges in the shortest path tree to change (for example, at least $n/2$ edges). [Contrast this with the case of MSTs, where adding or removing an edge in the graph causes at most one edge in the MST to change.]

6. **MST and shortest paths.** Let $G(V, E)$ be a directed weighted graph and let $s \in V$ be a specific node. The shortest paths tree (SPT) rooted at s is a tree that contains a shortest paths from s to every other node. (For example, the tree returned by Dijkstra's algorithm.)

(a) Give an example of a weighted graph G where the SPT is not an MST.

(b) Show that in any directed weighted graph any SPT and MST have at least one edge in common. (*hint: the common edge is adjacent to* S).

7. **MSTs and bottlnecks** KT Chapter 4, Problems 19, 20.

8. **When is reverse-delete a good idea?** KT Chapter 4, Problem 21.

9. **Reductions to MST** KT, Chapter 4, Problem 28

Divide-And-Conquer

1. **Finding local minima** KT, Chapter 5, Problems 6 and 7.

2. **Closest number.** Given a *sorted* array A of integers of length n and a number k, find the value in A that is closest to k. Devise your algorithm to query $O(\log n)$ values in A.

3. **Counting Inversions** KT, Chapter 5, Problem 2 (it will help to read the example in Section 5.3 first).

4. (*)**Median element of two arrays.** There are two *sorted* arrays A_1 and A_2, with length n and m respectively. You do not have access to the arrays but you can query the value at specific indices. That is, for any integer k you can query the value of $A_1[k]$ or $A_2[k]$. Find an efficient algorithm that identifies, with as few queries as possible, the median value in the combined set of $n+m$ numbers stored in the two arrays. (if $n+m$ is odd, then the median is the $\lfloor \frac{n+m}{2} \rfloor + 1$th value, if $n+m$ is even, then the median is the $\frac{n+m}{2}$nd value.)

   Generalized version: instead of finding the median, find the kth element among the numbers.

5. (*) **2-dimensional binary search.** n^2 numbers are stored in an $n \times n$ matrix where every row and every column is sorted in increasing order. Note, that this is *not* the same as having a 1-dimensional sorted array of length n^2. Here is an example of such a matrix:

$$\begin{pmatrix} 5 & 7 & 10 & 15 \\ 6 & 11 & 12 & 17 \\ 8 & 13 & 16 & 22 \\ 14 & 19 & 21 & 23 \end{pmatrix}$$

   Find a divide-and-conquer style algorithm to efficiently search in this matrix in time $O(n)$. (Warm-up: get an algorithm that runs in time $O(n \log n)$ by sorting each row separately.)