CSE 373: Data Structures & Algorithms Interviews and Problem Solving

Riley Porter

Winter 2017

Slides adapted from Kevin Quinn

Course Logistics

- HW6 due Friday

- Final review session next Monday (Exam resources updated on the website after class today)

Today's Outline

- What even is a technical interview?
- How do you break down problems and what do they want you to demonstrate
- Practice for technical interviews

Getting Hired

1. Apply Online or at Career Fair
   - work on your resume, put projects you liked, relevant classes you took, programming languages you know, relevant work experience
2. Phone Interview / Phone Screen
   - can be with recruiter, usually is just short technical engineering interview
   - can be screen shared coding, get a headset or headphones with mic
   - stand up! smile! be personable, it does matter
3. Onsite Technical Interviews
   - can be all day
   - between 2-5 interviews mostly generic tech interviews
   - can sometimes have 1 or 2 system design or design or test interviews

Technical Interview Breakdown

1. Introduction (5-10 minutes):
   - basic background
   - projects you've worked on
   - what you're passionate about and what you want to work on
2. Coding Question(s) (30 minutes - infinity):
   - could be smaller and several, or one large one with many levels
   - problem description and conversation clarifying all parameters
   - constraints, goals, use case, etc. (summarize it)
   - code it, normally on a whiteboard
   - test it if you have time
3. Questions for them (2-5 minutes):
   - how does the company work, how easy is it to move teams, what's the culture like, are they happy, would they choose differently now, are they excited about what they're working on, what is the structure within the company / project

Tips for Coding Part of Interview

- Ask questions (this is a two way street)
- Verbalize everything. All the things you're thinking
- Draw pictures before writing code, make sure you've designed your data structures and how they interact
- Write pseudocode bulleted list of steps you're going to take before code
- Analyze runtime and space complexity of your design before coding, maybe iterate on your design, ask input from your interviewer. Ask if it's okay to start with your first design if that's all you can think of, you can improve it if you have time
- Get to testing, or at least think of test cases as you go and verbalize them
- Practice writing on a whiteboard, practice writing code out as you talk / think
- (when done) Analyze if your solution passes the test cases you thought of, run through it to see if you forgot anything, re-analyze the runtime and space complexity

When solving: tools at your disposal

Over the past 8 weeks we have developed a broad knowledge of data structures and algorithms (I hope):

- Stacks and Queues
  - Various implementation (Stack/List)
- PriorityQueues
  - BinaryHeap
- Dictionaries (Maps)
  - HashMap, TreeMap
- Trees
  - Balanced (AVL), BST
- Union Find
  - Uptrees
- Graphs
  - Directed/Undirected, Acyclic/Cyclic, Weighted
  - Dijkstra's, BFS, DFS
  - Topological Sort
  - Minimum Spanning Trees
- Sorting

When solving: everything is a trade-off

- Very rarely is there a "perfect" solution in the real world.
  - Often must prioritize things like:
    - space vs. time
    - simplicity vs. robustness

- Understanding the ins and outs of each structure allows you to make informed design decisions that balance these trade-offs.

When solving: don't reinvent the wheel

- More often than not, the problem you are trying to solve is not entirely unique
  - Usually it is possible to simplify a problem down to a few core principles
    - Important operations
    - Space/time constraints
- Once you have found an appropriate analog, allow the well-thought out design to assist you
  - Example: AVL trees handle balancing for you
  - Example: Hash tables will handle collisions for you

When solving: sometimes simple is best

- In this class, we have lived and died by the asymptotic runtime, however this is not always the case
  - Sometimes simple and readable code is more important
  - Sometimes you know very distinct things about your input
    - Sorting input that is almost entirely sorted
    - Dictionary of elements that have nearly identical keys
- It can be more important to get a complete solution or have a complete discussion, it depends, ask your interviewer

Question 1:

```
Given a value ‘x’ and an array of integers,
determine whether two of the numbers add up to ‘x’:
```

Questions you should have asked me:

1) Is the array in any particular order?
2) Should I consider the case where adding two large numbers could cause an overflow?
3) Is space a factor, in other words, can I use an additional structure(s)?
4) Is this method going to be called frequently with different/the same value of ‘x’?
5) About how many values should I expect to see in the array, or is that unspecified?
6) Will ‘x’ always be a positive value? What about the values in the array?
7) Can I assume the array won’t always be empty, what if its null?

Why these questions matter!

1) Is the array in any particular order?

If the array is already sorted, then this question becomes a lot easier, and can be done in O(n) time.

2) Should I consider the case where adding two large numbers could cause an overflow?

If the integers are very large, I should use something other than 'ints' to store my results, such as double or longs, or else I could get inconsistent results.

3) Is space a factor, in other words, can I use an additional structure(s)?

If space is not a factor, then it might be better to leave the original array alone, and instead sort the array in a separate structure. Or even use a BST representation.

Why these questions matter!

1) Is this method going to be called frequently with different/the same value of 'x'?

This is a *great* question. If the client will be calling this frequently, it might make more sense to store a copy of the sorted array to prevent needing to re-sort it every time. This could drastically speed-up frequent calls. This process is called **memoization.**

2) About how many values should I expect to see in the array, or is that unspecified?

Often times, it is safe to assume that there could be any range of values (or in our case, asymptotically very many). However, this is not always the case. Our solution to this problem may be different if we knew that there were always exactly 12 values in our array.

Question 1.5

Given an array of integers, return a new array of the same values without any duplicates

Question 1.5

Given an array of integers, return a new array of the same values without any duplicates

```
create set, s
for each value, x in input_array:
    add x to s
create new array, result
for each value, x in s:
    add x to result
return result
```

Question 2:

Given an array that contains the values 1 through ‘n’ two times each, find the one number that is contained only 1 time.

Question 2:

Given an array that contains the values 1 through ‘n’ two times each, find the one number that is contained only 1 time.

```
create map from strings->ints, map
for each value, x in input_array:
    if !map.contains(x):
        map.put(x, 0)
    map.put(x, map.get(x) + 1)

for each key in map, key:
   if map.get(key) == 1:
       return key
```

Question 3:

Given a list of integers, find the highest value obtainable by concatenating them together.

For example: given [9, 918, 917], result = 9918917

For example: given [1, 112, 113], result = 1131121

Question 3:

Given a list of integers, find the highest value obtainable by concatenating them together.

For example: given [9, 918, 917], result = 9918917

For example: given [1, 112, 113], result = 1131121

-Convert all numbers to strings
-Sort numbers based on largest first number, break ties by moving on to next digit if its greater than the previous

Question 4:

Given a very large file of integers (more than you can store in memory), return a list of the largest 100 numbers in the file

Question 4:

Given a very large file of integers (more than you can store in memory), return a list of the largest 100 numbers in the file

```
Create min-heap, h
Add first 100 values to h
while there are remaining numbers:
    x = next number
    if x > h.getMin():
        h.deleteMin()
        h.add(x)

create new list, l
while h.isEmpty():
    l.add(h.deleteMin())
return l
```

Question 5

Given an unsorted array of values, find the 2nd biggest value in the array.

(Harder alternative)

Find the k'th biggest value in the array

Question 5

Given an unsorted array of values, find the 2nd biggest value in the array.

```
sort input_array
return input_array[len - 2]
```

```
max = -infinity
2nd_max = -infinity
for each value, v in input_array:
    if v > max:
        2nd_max = max
        max = v
return 2nd_max
```

```
max-heap h = heapify(input_array)
h.removeMax()
Return h.removeMax()
```

Question 6

Given a list of strings, write a method that returns the frequency of the word with the highest frequency.

(Harder version)

Given a list of strings, write a method that returns a sorted list of words based on frequency

Question 6

Given a list of strings, write a method that returns the frequency of the word with the highest frequency.

```
max = 0
map from string->int, map
for each string, s:
    if !map.contains(s):
        map.put(s,0)
    map.put(s, map.get(s) + 1)
    if map.get(s) > max:
        max = 0
```

Question 7:

Given an array of strings that are each sorted lexicographically, determine the order of characters in the given alphabet.
For example, given the english alphabet, the ordering is: “a,b,c,d,e,f . . . x,y,x”.

Your output should be the lexicographic order of only the characters that were found in the input strings.

For example: input = [xyz, yk, zk, xm, my], then the output would be [x,m,y,z,k]

Today's Takeaways

- Interviewing takes practice
  - actually practice, interview for companies you don't care about first
- Breathe, it's supposed to be fun
  - It's a conversation between you and the interviewer
  - Sometimes you don't click, that's not your fault.
- Remember all of your tools
  - ask questions, pseudocode, draw out solutions, talk through your thought process, use extra storage if it makes it faster, think about sorting if that is useful