Programming Interview Prep

Institute for Computational and Mathematical Engineering

Stanford University

Andreas Santucci

Fall 2020

Contents

1 A Note on Complexity and Leveling

Interviewing for a Fixed Level When new grads enter the job market, they are all considered for a level that is based on their degree, e.g. level L for an M.S. and $L+1$ for a PhD. It doesn't matter how proficient you are, given your credentials you will be interviewing for a new grad position. This means that the interview questions will be scoped accordingly. Remember that in a 45 minute interview, you can expect a warmup question to be solvable in under 15 minutes and a more challenging question which lends itself to open-ended follow up questions for the remainder of the technical interview. It is not likely that you will have time to ask your interviewer questions at the end of the interview, because they will likely be continually assessing your ability by probing with increasingly difficult questions. This means that it's OK if you don't finish all the interview questions; don't let it hurt your morale and focus on the the next interview!

Showing Leadership It is still tough to earn your worth and obtain the job offer. How can you showcase your understanding of programming and quantitative reasoning?

- Ask questions about the problem statement.
  - Clarify ambiguities that the interviewer may have intentionally left open to interpretation.
  - Determine what constraints you are working under; sometimes the solution can be made simpler if you have permission to use certain methods or tools.
- Demonstrate knowledge of tradeoffs.
  - Each problem likely has multiple solutions. Can you spot them and mention their relative merits during the interview? This can showcase your broad knowledge. It's enough to mention various tradeoffs before proceeding with an an approach that is tied for efficiency.
- Explain what you're thinking, always!
  - One of the biggest mistakes interviewers make is solving problems silently; this comes across as a hard pass in that people don't want to work with peers who don't explain their solutions eloquently.
  - Make sure to explain what you're doing at each step, out loud, and consistently pause to ask the interviewer if they have any questions on why you're taking the approach proposed.

2 Intermediate Programming Questions

2.1 Balanced String Split

Balanced strings are those who have equal quantity of 'L' and 'R' characters.

Given a balanced string s split it in the maximum amount of balanced strings.

Return the maximum amount of splitted balanced strings.

```cpp
int balancedStringSplit(string s) {
    int balance {};
    int num_splits {};
    for (char c : s) {
        c == 'L' ? ++balance : --balance;
        if (balance == 0) num_splits++;
    }
    return num_splits;
}
```

Idea behind the algorithm The trick is to maintain a 'balance' variable which we will increment by unit value when we run into an 'L' and we will decrement by unit value when we run into an 'R'. Each time the balance var "hits" zero, we know that we can split off a piece of the string into a `balancedSubString`. One *key argument to make* is that the remainder is balanced by virtue of the original input being balanced.

Complexity It's easy to see that our algorithm requires $\mathcal{O}(1)$ storage and $\mathcal{O}(n)$ work.

2.2 Intersection of Three Arrays

Find the intersection of three input arrays.

```cpp
vector<int> arraysIntersection(vector<int>& arr1, vector<int>& arr2, vector<int>& arr3) {
    map<int, int> counts;
    for (const auto& val : arr1) counts[val] += 1;
    for (const auto& val : arr2) counts[val] += 1;
    for (const auto& val : arr3) counts[val] += 1;
    vector<int> result;
    for (const auto& kv : counts)
        if (kv.second == 3) result.push_back(kv.first);
    // We don't need to sort, since we're already using an ordered map.
    return result;
}
```

Idea behind the algorithm Create a map which counts how many times we see a particular value *across* all three input arrays. In the end, the values which are associated with a count of three correspond to being in all three input arrays.

Follow Up It's trivial to find the intersection of k arrays using the same technique.

Analysis Each insertion costs logarithmic time with respect to the size of the *ordered* map. If n is the size of the largest input vector, then we require $O(n \log n)$ work to populate our ordered map. Populating the output container only requires a linear traversal. Therefore, the total work is given by $O(n \log n)$.

2.3 Two-Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice. Don't worry about possible overflow.

(Brute-Force) Solution It's almost always worth it to at least mention a brute force solution before proceding to an efficient implementation.

```
vector<int> twoSum(vector<int>& nums, int target) {
    for (unsigned i = 0; i < nums.length() - 1; ++i)
        for (unsigned j = i + 1; j < nums.length(); ++j)
            if (nums[i] + nums[j] == target)
                return {i, j};
    return {-1, -1};
}
```

The brute force solution requires $\mathcal{O}(n^2)$ work. It's also possible to come up with a **linear time algorithm**, i.e. we can find the indices in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ storage.

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> m;
    m[nums[0]] = 0;
    for (int i = 1; i < nums.size(); ++i) {
        // find is O(1) average complexity for unordered map.
        // Search contains a first (key) and second (val) element.
        auto search = m.find(target - nums[i]);
        if (search != m.end())
            return {search->second, i};
        m[nums[i]] = i;
    }
    return {-1, -1};
  }
```

Idea behind the linear time algorithm Addition is commutative, i.e. $a + b = b + a$. Therefore, it doesn't matter what order we consider our inputs. For each element in the input, we can compute `target` $- x$ and see if this candidate value is contained in our container. If it is, it means we've already come across an element which, when added to the current element x that we're iterating over, adds to our `target` value.

Complexity analysis We make a linear traversal over the input. For each element, we perform a subtraction in constant time and then use a `find` operation on an *unordered* map to determine if the two-sum achieves the target value in constant time. The `if`-statement and the final insertion into the unordered map are both $\mathcal{O}(1)$ as well, and so the entire algorithm is $\mathcal{O}(n)$.

Extension: three sum A natural follow-up question is to come up with an algorithm for finding a three-sum that achieves a target value. This can be an exercise for the reader.

2.4 Counting Complete Tree Nodes

There are a myriad of tree traversal algorithms that are fun to cover in interviews. We present a *very* simple one here.

Problem Given a complete binary tree, count the number of nodes. In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h.

Example

```
Input:
    1
   / \
  2   3
 / \  /
4  5 6
```

Correct output: 6.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
int countNodes(TreeNode* root) {
    if (!root) return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

Complexity Analysis Our algorithm touches each node once and performs a simple arithmetic computation with each invocation. We therefore have a linear traversal (with respect to the number of nodes in the input graph).

2.5 Majority Element

Given an array of size n, find the majority element. The majority element is the element that appears more than n/2 times. You may assume that the array is non-empty and the majority element always exists in the array.

```
Example 1:
Input: [3,2,3]
Output: 3

Example 2:
Input: [2,2,1,1,1,2,2]
Output: 2
```

Sorting as a means to an end One simple approach we can take is to use a sort as a means to an end. Since the majority element appears *more* than $\lfloor \frac{n}{2} \rfloor$ times, after sorting we can simply inspect the middle element.

```
int majorityElement(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    return nums[nums.size() / 2];
}
```

For any reasonable `sort` implementation we can say this is an $\mathcal{O}(n \log n)$ algorithm.

Divide and Conquer Next we explore a divide and conquer approach. **Hint:** suppose we are told the majority element for the left half and the right half of the array. If these values coincide, we've found our majority element and we're done. Otherwise, iterative over the sub-input in linear time and see which of the two occurs more often, that's the majority element. The base case is when there's only one element in the array, then trivially it itself is the majority element.

Intuition for why linear scan is necessary To see why we need the linear traversal, think about an input where an element occurs all but one time in the left half of the array, and the right half consists of it appearing only a couple times (with some other element appearing a "majority" number of times) then we clearly need to figure out which of the two candidate majority elements is the right one.

```
int majorityElement(vector<int>& nums) { return majorityElement(nums, 0, nums.size() - 1); }
```

We then define a recursive work-horse.

```
int majorityElement(const vector<int>& nums, int l, int r) {
    if (l == r)
        return nums[l];
    int mid = (r-l)/2 + l;
    int l_max = majorityElement(nums, l, mid);
    int r_max = majorityElement(nums, mid+1, r);
    if (l_max == r_max)             // If they agree, we're done.
        return l_max;
    int l_count{}, r_count{};       // Else count which occurs more often.
    for (int k = l; k <= r; k++) {  // Iterate over the _sub_-array.
        if (nums[k] == l_max) l_count++;
        if (nums[k] == r_max) r_count++;
    }
    return l_count > r_count ? l_max : r_max;
}
```

Complexity analysis Recurrence relation for work required:

$$T(N) = 2 * T(N/2) + \mathcal{O}(N) = \mathcal{O}(NlogN).$$

You can use Master's theorem or unroll the recurrence relation by hand to arrive at the asymptotic complexity.

3 Intermediate SQL

3.1 With Clauses and Sub-Queries

A `with` clause let's you name a *sub-query*. I.e. it's possible to write a SQL program with more than one `SELECT ... FROM ... WHERE ...`. Suppose you have the following two tables, the first is maps customer id's to transaction amounts, and it's called `transactions`:

```
     id |    amt |
--------+--------+
    901 |   8.00 |
    901 |   9.00 |
    902 |   1.00 |
    902 |   2.00 |
```

and the second maps customer id's to names, called `customer_info`:

```
     id |   name |
--------+--------+
    901 |  Andre |
    902 |   Nick |
```

What if we want to calculate the average transaction amount by customer, and then display the result alongside their name? We can use a `with` clause.

```
WITH avg_amt AS (
  SELECT
    id
    , AVG(amt) AS avg_transaction_price
  FROM
    transactions
  GROUP BY 1
) SELECT
    name
    , avg_transaction_price AS amt
  FROM
    avg_amt JOIN customer_info ON avg_amt.id = customer_info.id;
```

The result would look like:

```
   name |    amt |
--------+--------+
  Andre |   8.50 |
   Nick |   1.50 |
```

3.2 Window Expressions

An aggregation function takes in a sequence of observations and collapses it into a single number. What if we want to take in a sequence of values and output another sequence of values, e.g. a cumulative sum? We can use a window function for this.

Suppose we have a race tournament where there are multiple stages of racing, and we'd like to calculate (ex-post) each participant's final ranking at each stage of the race, based on all previous rounds. The easy way to do this would be to generate a column that describes the cumulative time spent up until the given stage. Suppose our data looks like `races`:

```
      id |  trial |   time
--------+--------+-------
       A |      0 |   0:30
       A |      1 |   0:10
       A |      2 |   0:05
     ... |    ... |
       B |      0 |   0:28
       B |      1 |   0:07
       B |      2 |   0:05
```

We already know how we could calculate the *total* time spent across all races for a given participant-id using a group-by clause, e.g. we could rank the participants by their total times via `SELECT id, SUM(time) AS final_time FROM races GROUP BY 1 ORDER BY 2;`. But what about the cumulative sum of times? We need a function that takes in a window of n observations and returns another sequence of n observations, rather than a typical aggregation method which collapses n observations into a scalar value...

```
SELECT
  id
  , trial
  , SUM(time) OVER(PARTITION BY id ORDER BY trial) AS cum_time
FROM
  races;
```

This would return something like:

```
      id |  trial | cum_time
--------+--------+---------
       A |      0 |     0:30
       A |      1 |     0:40
       A |      2 |     0:45
     ... |    ... |      ...
       B |      0 |     0:28
       B |      1 |     0:35
       B |      2 |     0:40
```

3.3 Applied Problem: Ranking Scores

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no ”holes” between ranks. Given a `Scores` table:

```
+----+-------+      +-------+------+
| Id | Score |      | score | Rank |
+----+-------+      +-------+------+
| 1  | 3.50  |      | 4.00  |  1   |
| 2  | 3.65  |      | 4.00  |  1   |
| 3  | 4.00  |  --> | 3.85  |  2   |
| 4  | 3.85  |      | 3.65  |  3   |
| 5  | 4.00  |      | 3.65  |  3   |
| 6  | 3.65  |      | 3.50  |  4   |
+----+-------+      +-------+------+
```

Solution The problem can be solved by using both sub-queries and a window clause.

```
WITH distinct_scores AS (
  SELECT DISTINCT
    Score AS score
  FROM
    Scores
), ranked_scores AS (
    SELECT
      score
      , ROW_NUMBER() OVER(ORDER BY score DESC) AS ‘rank‘
    FROM distinct_scores
)
SELECT
  score
  , ‘rank‘
FROM
  Scores JOIN ranked_scores USING(score)
ORDER BY 1 DESC;
```

Explanation of query We first query for the distinct scores, and in this way we can later get a rank ordering without having to worry about duplicate scores getting assigned different ranks. In the next step, we actually rank our scores in descending order by utilizing a window function. Lastly, we join our original scores with our crafted ranks and order the result.

References

[1] LeetCode Accessed Fall 2020