Hacking a Google Interview – Handout 2

Course Description

Instructors: Bill Jacobs and Curtis Fonger
Time: January 12 – 15, 5:00 – 6:30 PM in 32-124
Website: http://courses.csail.mit.edu/iap/interview

Classic Question #4: Reversing the words in a string

Write a function to reverse the order of words in a string in place.

Answer: Reverse the string by swapping the first character with the last character, the second character with the second-to-last character, and so on. Then, go through the string looking for spaces, so that you find where each of the words is. Reverse each of the words you encounter by again swapping the first character with the last character, the second character with the second-to-last character, and so on.

Sorting

Often, as part of a solution to a question, you will need to sort a collection of elements. The most important thing to remember about sorting is that it takes O(n log n) time. (That is, the fastest sorting algorithm for arbitrary data takes O(n log n) time.)

Merge Sort:

Merge sort is a recursive way to sort an array. First, you divide the array in half and recursively sort each half of the array. Then, you combine the two halves into a sorted array. So a merge sort function would look something like this:

```
int[] mergeSort(int[] array) {
    if (array.length <= 1)
        return array;
    int middle = array.length / 2;
    int firstHalf = mergeSort(array[0..middle - 1]);
    int secondHalf = mergeSort(
                        array[middle..array.length - 1]);
    return merge(firstHalf, secondHalf);
}
```

The algorithm relies on the fact that one can quickly combine two sorted arrays into a single sorted array. One can do so by keeping two pointers into the two sorted

arrays. One repeatedly adds the smaller of the two numbers pointed to to the new array and advances the pointer.

Quicksort:

Quicksort is another sorting algorithm. It takes O(n^2) time in the worst case and O(n log n) expected time.

To sort an array using quicksort, one first selects a random element of the array to be the "pivot". One then divides the array into two groups: a group of elements that are less than the pivot and a group of elements that are greater than the pivot. After this, there will be an array consisting of elements less than the pivot, followed by the pivot, followed by elements greater than the pivot. Then, one recursively sorts the portion of the array before the pivot and the portion of the array after the pivot. A quicksort function would look like this:

```
void quicksort(int[] array, int startIndex, int endIndex) {
    if (startIndex >= endIndex) {
        // Base case (array segment has 1 or 0 elements
    } else {
        int pivotIndex = partition(array,
                                   startIndex,
                                   endIndex);
        quicksort(array, startIndex, pivotIndex - 1);
        quicksort(array, pivotIndex + 1, endIndex);
    }
}
```

Quicksort is typically very fast in practice, but remember that it has O(n^2) worst-case running time, so be sure to mention another sorting algorithm, such as merge sort, if you need guaranteed O(n log n) running time.

Order Statistics:

Sometimes, an interviewer will ask you to describe an algorithm to identify the kth smallest element in an array of n elements. To do this, you select a random pivot and partition the array as you would in the quicksort algorithm. Then, based on the index of the pivot element, you know which half of the array the desired element lies in. For example, say k = 15 and n = 30, and after you select your pivot and partition the array, the first half has 10 elements (the half before the pivot). You know that the desired element is the 4th smallest element in the larger half. To identify the element, you partition the second half of the array and continue recursively. The reason that this is not O(n log n) is that the recursive partition call is only on one half of the array, so the expected running time is n + (n/2) + (n/4) + (n/8) + ... = O(n).

Note that finding the median of an array is a special case of this where k = n / 2. This is a very important point, as an interviewer will often ask you to find a way to get the median of an array of numbers.

Question: Nearest Neighbor

Say you have an array containing information regarding n people. Each person is described using a string (their name) and a number (their position along a number line). Each person has three friends, which are the three people whose number is nearest their own. Describe an algorithm to identify each person's three friends.

Good answer: Sort the array in ascending order of the people's number. For each person, check the three people immediately before and after them. Their three friends will be among these six people. This algorithm takes O(n log n) time, since sorting the people takes that much time.

Linked Lists

A linked list is a basic data structure. Each node in a linked list contains an element and a pointer to the next node in the linked list. The last node has a "null" pointer to indicate that there is no next node. A list may also be doubly linked, in which case each node also has a pointer to the previous node. It takes constant (O(1)) time to add a node to or remove a node from a linked list (if you already have a pointer to that node). It takes O(n) time to look up an element in a linked list if you don't already have a pointer to that node.

Classic Question #5: Cycle in a Linked List

How can one determine whether a singly linked list has a cycle?

Good answer: Keep track of two pointers in the linked list, and start them at the beginning of the linked list. At each iteration of the algorithm, advance the first pointer by one node and the second pointer by two nodes. If the two pointers are ever the same (other than at the beginning of the algorithm), then there is a cycle. If a pointer ever reaches the end of the linked list before the pointers are the same, then there is no cycle. Actually, the pointers need not move one and two nodes at a time; it is only necessary that the pointers move at different rates. This takes O(n) time. This is a tricky answer that interviewers really like for some reason.

Okay answer: For every node you encounter while going through the list one by one, put a pointer to that node into a O(1)-lookup time data structure, such as a hash set. Then, when you encounter a new node, see if a pointer to that node already exists in your hash set. This should take O(n) time, but also takes O(n) space.

Okay answer: Go through the elements of the list. "Mark" each node that you reach. If you reach a marked node before reaching the end, the list has a cycle; otherwise, it does not. This also takes O(n) time.

Note that this question is technically ill-posed. An ordinary linked list will have no cycles. What they actually mean is for you to determine whether you can reach a cycle from a node in a graph consisting of nodes that have at most one outgoing edge.

Stacks and Queues

An interviewer will probably expect you to know what queues and stacks are. Queues are abstract data types. A queue is just like a line of people at an amusement park. A queue typically has two operations: enqueue and dequeue. Enqueueing an element adds it to the queue. Dequeueing an element removes and returns the element that was added least recently. A queue is said to be FIFO (first-in, first-out).

A stack is another abstract data type with two common operations: push and pop. Pushing an element adds it to the stack. Popping an element removes and returns the element that was added most recently. A stack is said to be LIFO (last-in, first-out). A stack operates like a stack of cafeteria trays.

Hash Tables

A hash table is used to associate keys with values, so that each key is associated with one or zero values. Each key should be able to compute a "hash" function, which takes some or all of its information and digests it into a single integer. The hash table consists of an array of hash buckets. To add a key-value pair to a hash table, one computes the key's hash code and uses it to decide the hash bucket in which the mapping belongs. For example, if the hash value is 53 and there are 8 hash buckets, one might use the mod function to decide to put the mapping in bucket 53 mod 8, which is bucket 5. To lookup the value for a given key, one computes the bucket in which the key would reside and checks whether the key is there; if so, one can return the value stored in that bucket. To remove the mapping for a given key, one likewise locates the key's mapping and removes it from the appropriate bucket. Note that the hash function is generally decided on in advance.

A problem arises when two keys hash to the same bucket. This event is called a "collision". There are several ways to deal with this. One way is to store a linked list of key-value pairs for each bucket.

Insertion, removal, and lookup take expected O(1) time, provided that the hash function is sufficiently "random". In the worst-case, each key hashes to the same bucket, so each operation takes O(n) time. In practice, it is common to assume constant time.

Hash tables can often be used as smaller components of answers to questions. In our experience, some interviewers like hash tables and some don't. That is, some interviewers will allow you to assume constant time, while others will not. If you want to use a hash table, we recommend subtly trying to figure out which category your interviewer belongs to. You might, for example, say something like, "Well, I could used a hash table, but that would have bad worst-case performance." The interviewer might then indicate that he'll allow you to use a hash table.

Classic Question #6: Data structure for anagrams

Given an English word in the form of a string, how can you quickly find all valid anagrams for that string (all valid rearrangements of the letters that form valid English words)? You are allowed to pre-compute whatever you want to and store whatever you optionally pre-compute on disk.

Answer: We want to use a hash table! If your interviewer really hates hash tables (which they sometimes do for some reason), you can use a tree instead. But let's assume you can use a hash table. Then for the pre-computing step, go through each word in the dictionary, sort the letters of the word in alphabetical order (so "hacking" would become "acghikn") and add the sorted letters as a key in the table and the original word as one of the values in a list of values for that key. For example, the entry for "opst" would be the list ["opts", "post", "stop", "pots", "tops", "spot"]. Then, whenever you get a string, you simply sort the letters of the string and look up the value in the hash table. The running time is O(n log n) for sorting the string (which is relatively small) and approximately O(1) for the lookup in the hash table.

There are several other possible answers to this question, but we feel that the answer above is considered an optimal solution.

Question: Factorial Zeros

Without using a calculator, how many zeros are at the end of "100!"? (that's 100*99*98*...*3*2*1)

Answer: What you don't want to do is start multiplying it all out! The trick is remembering that the number of zeros at the end of a number is equal to the number of times "10" (or "2*5") appears when you factor the number. Therefore think about the prime factorization of 100! and how many 2s and 5s there are. There are a bunch more 2s than 5s, so the number of 5s is also the number of 10s in the factorization. There is one 5 for every factor of 5 in our factorial multiplication (1*2*...*5*...*10*...*15*...) and an extra 5 for 25, 50, 75, and 100. Therefore we have 20+4 = 24 zeros at the end of 100!.