Hacking a Google Interview – Handout 1

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 #1: Coin Puzzle

You have 8 coins which are all the same weight, except for one which is slightly heavier than the others (you don't know which coin is heavier). You also have an old-style balance, which allows you to weigh two piles of coins to see which one is heavier (or if they are of equal weight). What is the fewest number of weighings that you can make which will tell you which coin is the heavier one?

Good answer: Weigh 3 coins against 3 coins. If one of the groups is heavier, weigh one of its coins against another one of its coins; this allows you to identify the heavy coin. If the two groups balance, weigh the two leftover coins.

Not-so-good answer: Weigh 4 coins against 4 coins. Discard the lighter coins, and weigh 2 coins against 2 coins. Discard the lighter coins and weigh the remaining two coins.

Interview tips

When asked a question, open a dialog with the interviewer. Let them know what you are thinking. You might, for example, suggest a slow or partial solution (let them know that the solution is not ideal), mention some observations about the problem, or say any ideas you have that might lead to a solution. Often, interviewers will give hints if you appear to be stuck.

Often, you will be asked to write a program during an interview. For some reason, interviewers usually have people write programs on a blackboard or on a sheet of paper rather than on a computer. It is good to get practice with writing code on the board in order to be prepared for this.

Here is a list of "do's" and "don't's" when doing a programming interview:

Do's

- Ask for clarification on a problem if you didn't understand something or if there is any ambiguity

- Let the interviewer know what you are thinking
- Suggest multiple approaches to the problem
- Bounce ideas off the interviewer (such as ideas for data structures or algorithms)
- If you get stuck, don't be afraid to let them know and politely ask for a hint

Don't's

- Never give up! This says nothing good about your problem solving skills.
- Don't just sit in silence while thinking. The interviewer has limited time to find out as much as possible about you, and not talking with them tells them nothing, except that you can sit there silently.
- If you already know the answer, don't just blurt it out! They will suspect that you already knew the answer and didn't tell them you've seen the question before. At least pretend to be thinking though the problem before you give the answer!

Big O Notation

Big O notation is a way that programmers use to determine how the running speed of an algorithm is affected as the input size is increased. We say that an algorithm is O(n) if increasing the input size results in a linear increase in running time. For example, if we have an algorithm that takes an array of integers and increments each integer by 1, that algorithm will take twice as long to run on an array of size 200 than on an array of size 100.

Now let's look at an algorithm of running time O(n^2). Consider the following Java code:

```
boolean hasDuplicate(int[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array.length; j++) {
            if (array[i] == array[j] && i != j) {
                return true;
            }
        }
    }
    return false;
}
```

This algorithm takes in an array of integers and compares each integer to every other integer, returning true if two integers are equal, otherwise returning false. This array takes O(n^2) running time because each element has to be compared with n elements (where n is the length of the array). Therefore, if we double the input size, we quadruple the running time.

There is also a more formal definition of big O notation, but we prefer the intuitive approach for the purposes of programming interviews.

Question: Searching through an array

Given a sorted array of integers, how can you find the location of a particular integer x?

Good answer: Use binary search. Compare the number in the middle of the array with x. If it is equal, we are done. If the number is greater, we know to look in the second half of the array. If it is smaller, we know to look in the first half. We can repeat the search on the appropriate half of the array by comparing the middle element of that array with x, once again narrowing our search by a factor of 2. We repeat this process until we find x. This algorithm takes O(log n) time.

Not-so-good answer: Go through each number in order and compare it to x. This algorithm takes O(n) time.

Parallelism

Threads and processes:

A computer will often appear to be doing many things simultaneously, such as checking for new e-mail messages, saving a Word document, and loading a website. Each program is a separate "process". Each process has one or more "threads." If a process has several threads, they appear to run simultaneously. For example, an e-mail client may have one thread that checks for new e-mail messages and one thread for the GUI so that it can show a button being pressed. In fact, only one thread is being run at any given time. The processor switches between threads so quickly that they appear to be running simultaneously.

Multiple threads in a single process have access to the same memory. By contrast, multiple processes have separate regions of memory and can only communicate by special mechanisms. The processor loads and saves a separate set of registers for each thread.

Remember, each process has one or more threads, and the processor switches between threads.

Mutexes and semaphores:

A mutex is like a lock. Mutexes are used in parallel programming to ensure that only one thread can access a shared resource at a time. For example, say one thread is modifying an array. When it has gotten halfway through the array, the processor switches to another thread. If we were not using mutexes, the thread might try to modify the array as well, which is probably not what we want.

To prevent this, we could use a mutex. Conceptually, a mutex is an integer that starts at 1. Whenever a thread needs to alter the array, it "locks" the mutex. This causes the thread to wait until the number is positive and then decreases it by one. When the thread is done modifying the array, it "unlocks" the mutex, causing the number to increase by 1. If we are sure to lock the mutex before modifying the array and to unlock it when we are done, then we know that no two threads will modify the array at the same time.

Semaphores are more general than mutexes. They differ only in that a semaphore's integer may start at a number greater than 1. The number at which a semaphore starts is the number of threads that may access the resource at once. Semaphores support "wait" and "signal" operations, which are analogous to the "lock" and "unlock" operations of mutexes.

Synchronized methods (in Java):

Another favorite question of interviewers is, "What is a synchronized method in Java?" Each object in Java has its own mutex. Whenever a synchronized method is called, the mutex is locked. When the method is finished, the mutex is unlocked. This ensures that only one synchronized method is called at a time on a given object.

Deadlock:

Deadlock is a problem that sometimes arises in parallel programming. It is typified by the following, which is supposedly a law that came before the Kansas legislature:

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Strange as this sounds, a similar situation can occur when using mutexes. Say we have two threads running the following code:

Thread 1:

```
acquire(lock1);
acquire(lock2);
[do stuff]
release(lock1);
release(lock2);
```

Thread 2:

```
acquire(lock2);
acquire(lock1);
[do stuff]
```

release(lock2);
release(lock1);

Suppose that thread 1 is executed to just after the first statement. Then, the processor switches to thread 2 and executes both statements. Then, the processor switches back to thread 1 and executes the second statement. In this situation, thread 1 will be waiting for thread 2 to release lock1, and thread 2 will be waiting for thread 1 to release lock2. Both threads will be stuck indefinitely. This is called deadlock.

Classic Question #2: Preventing Deadlock

How can we ensure that deadlock does not occur?

Answer: There are many possible answers to this problem, but the answer the interviewer will be looking for is this: we can prevent deadlock if we assign an order to our locks and require that locks always be acquired in order. For example, if a thread needs to acquire locks 1, 5, and 2, it must acquire lock 1, followed by lock 2, followed by lock 5. That way we prevent one thread trying to acquire lock 1 then lock 2, and another thread trying to acquire lock 2 then lock 1, which could cause deadlock. (Note that this approach is not used very often in practice.)

Some Other Topics

What is polymorphism?

Interviewers love to ask people this question point-blank, and there are several possible answers. For a full discussion of all the types of polymorphism, we recommend looking at its Wikipedia page. However, we believe that a good answer to this question is that polymorphism is the ability of one method to have different behavior depending on the type of object it is being called on or the type of object being passed as a parameter. For example, if we defined our own "MyInteger" class and wanted to define an "add" method for it (to add that integer with another number), we would want the following code to work:

```
MyInteger int1 = new MyInteger(5);
MyInteger int2 = new MyInteger(7);
MyFloat float1 = new MyFloat(3.14);
MyDouble doub1 = new MyDouble(2.71);
print(int1.add(int2));
print(int1.add(float1));
print(int1.add(doub1));
```

In this example, calling "add" will result in different behavior depending on the type of the input.

What is a virtual function/method? (in C++)

Out of all the possible questions interviewers could ask about C++, this one seems to be a strange favorite. A method's being "virtual" simply describes its behavior when working with superclasses and subclasses. Assume class B is a subclass of class A. Also assume both classes A and B have a method "bar()". Let's say we have the following code in C++:

```
A *foo = new B();
foo->bar();
```

If the method "bar()" is declared to be virtual, then when we call foo->bar(), the method found in class B will be run. This is how Java always handles methods and it's usually what we want to happen. However, if the method bar() is not declared to be virtual, then this code will run the method found in class A when we call foo->bar().

Classic Question #3: A to I

Write a function to convert a string into an integer. (This function is called A to I (or atoi()) because we are converting an ASCII string into an integer.)

Good answer: Go through the string from beginning to end. If the first character is a negative sign, remember this fact. Keep a running total, which starts at 0. Each time you reach a new digit, multiply the total by 10 and add the new digit. When you reach the end, return the current total, or, if there was a negative sign, the inverse of the number.

Okay answer: Another approach is to go through the string from end to beginning, again keeping a running total. Also, remember a number x representing which digit you are currently on; x is initially 1. For each character, add the current digit times x to the running total, and multiply x by 10. When you reach the beginning, return the current total, or, if there was a negative sign, the inverse of the number.

Note: The interviewer is likely to ask you about the limitations of your approach. You should mention that it only works if the string consists of an optional negative sign followed by digits. Also, mention that if the number is too big, the result will be incorrect due to overflow.