Quiz 1 – Review Packet

Computer Science 50 – Fall 2010

Prepared by Doug Lloyd '09

Week of November 15, 2010

---

1 Helpful Study Hints

Here are a few study tips for preparing for the quiz:

- Read the Quiz 0 review section notes thoroughly! This quiz is cumulative, and you will be expected to know everything that you were supposed to know for Quiz 0! Although the quiz will prioritize material not yet tested, questions may assume an understanding of previously tested concepts. Therefore, you should review questions you missed on the first quiz to ensure those same concepts don't trip you up again!
- Practice tracing through code, and asking yourself why things work the way they do. If you've been doing the hacker problem sets, make sure you understand the concepts presented in the standard problem sets!
- Practice with pointers on your computer. Write a program called pointers.c and compile it for a bunch of different use cases of pointers. For example, practice passing objects by value vs. by reference into a function call, practice using * to dereference pointers, and & to get the address of objects. A good exercise would be to write a swap function using pointers.
- Look over the makefiles that you have used and written for your early assignments. Make sure you have some idea of how they work and what they do.
- Review all the lecture notes, section notes and scribe notes. Find and make note of tricky syntax.
- The quiz is closed-book, but you're allowed to bring "one two-sided page (8.5" × 11") of notes, typed or written." Use that page to jot down details you're worried you might forget (e.g., the format of a for loop in C). To be clear, it's not terribly important to have such details memorized at this point (after all, you can always look such up in the real world). But why waste time on the quiz trying to remember little things like that.
- Be sure to go over any "how-computers-work"-related questions that were asked in the problem sets, like those you were asked to answer in the questions.txt portion of your earlier assignments. You do not need to know these things in depth, but you should be able to answer short questions on them.
- Be familiar with the guest lectures. These are fair game, too!
- Practice writing some XHTML and PHP. Start with some simple stuff, and build up the complexity a little bit.
- Make some SQL tables and practice querying them from the interpreter.

- Know all of the HTML tags that we went over in lecture, and try to recall the bugs you frequently encountered when you were writing your own webpage in HTML (forgetting tags, forgetting quotes, etc.).
- Review thoroughly the scribe notes, lecture videos, and code examples for JavaScript. The best way to get your head around the material is to really immerse yourself in it!
- Do all the practice questions we've written for you, both for this quiz review and the last one!

2 C Topics

2.1 Functions

- Declarations: Make sure that you know how do declare and call functions. Remember that a function header always has the following format:

  ```
  <return type> <name> (<arg1 type> <arg1 name>, ...)
  ```

  So, for example, the declaration of a sort function might look something like:

  ```
  bool sort(int array [], int size)
  ```

- Arguments: What happens to the arguments that you "pass" to a function? Are they copied? What can you do to ensure that an argument is not copied? We refer to this as passing by _____?
- Functional Decomposition: Functional decomposition means breaking down a program into its (logically) smallest parts. There are two main reasons that we do this:
  - Reuse of Code: Consider writing code to sort an array of integers. It would be annoying if you had to write this code every time you wanted to sort something! A much more sensible approach is to decompose your program by writing a sort function that takes, as an argument, the array to be sorted. In this way, any time you want to sort an array, you can just pass it to your sort function, as easy as that!
  - Readability: Functional decomposition makes code much more legible for other people reading it!

2.2 Scope and Shadowing

The scope of a variable is analogous to "where it can be seen" in your program. Thus, variables defined within a particular function are only visible and accessible inside of that function, after the variable declaration. Similarly, a variable defined within a for loop or a while loop will not be accessible anywhere except after where you declared the variable inside of that loop!

To generalize, each block of code defines a new scope, so that any variable declared in that block is accessible (after the declaration) inside of that block (including any more nested blocks within that block). You can think of blocks as the sections of code enclosed in curly braces.

Variable shadowing (a term we may not have used before) means that you have two variables with the same name, declared in overlapping scopes (normally from nested

blocks). We say that the inner scope declaration shadows the outer declaration because any use of that variable name in the inner scope (after the inner scope declaration) will refer only to the declaration in the inner scope. There is no way to "reach out" into the outer scope in C...so do be careful when reusing variable names.

2.3 Search Algorithms

So far, we have studied two major searching algorithms for an array. They are linear search and binary search.

2.3.1 Linear Search

The pseudocode for linear search is as follows:

```
linear(array, element_sought):
      for each element in array
            if (element equals element_sought)
                  return true
      return false
```

Linear search runs in linear time, O(n). Remember that when quoting computational complexity, we always quote the worst case scenario. In the worst case of linear search, we have to compare each of the n elements of the array to what we are looking for. What is the computational complexity of this algorithm in the best case?

2.3.2 Binary Search

Binary search is more efficient, running in O(log n). However, it only works on arrays that have been sorted! A practice exercise asks you to draw up some pseudocode for the binary search algorithm.

Why does this algorithm run in O(log n)? Try thinking about it as $O(\log_2 n)$ instead. Is it clearer? The algorithm eliminates half of the remaining array each pass through. And you can't do that more than $\log_2 n$ times, because $n/(2^{\log_2 n}) = 1$, and a singleton set can't be further divided. Think of this as a similar algorithm to the counting algorithm presented near the beginning of the course. Half of the remaining people in the class are counted with each pass, and the rest sit down. Thus a class containing x students will take only $\log_2 x$ passes, rounded up to the nearest integer, to count.

2.4 Asymptotic Notation

If you didn't understand what was going on in the previous section when discussing O(n), then this is where you want to focus some attention! O(n) is how we represent time complexity, also known as computational complexity. When we say this, what we are getting at is trying to figure out how long it will take (how much of our computational resources it will take) to solve a given problem. Some of the most common notations are as follows:

- O(1) - constant time - Given a data input of any size, the computation will always take the exact same amount of time. The following function runs in constant time, because no matter how large array is, the same answer is always returned:

```
int func(int array[]) {
     return 2;
}
```

- O(n) - linear time - Given a data input of any size, the computation will run in an amount of time proportional to the size of the list, linearly. This means that given a data input of size k, adding one element to the data input makes the computation take one unit of time longer to run. An example of this is the linear search algorithm, outlined in Section 2.3.1.
- O(log n) - logarithmic time - Given a data input of size n, the computation runs in log2 n. Examples of this include the binary search algorithm, outlined above in Section 2.3.2, as well as the counting algorithm discussed in Week 4.
- $O(n^2)$ - quadratic time - Given a data input of any size, the computation will run in an amount of time proportional to the size of the list, squared. This means that given a data input of size k, adding one element to the data input makes the computation take $(k + 1)^2 - k^2 = (2k + 1)$ units of time longer to run. Examples of this include bubble sort and selection sort.

Some of the other useful time complexities to know are supralinear time, O(n log n) [an example of this is merge sort]; polynomial time, $O(n^c)$; exponential time, $O(c^n)$; and factorial time, O(n!). All of these are with respect to a data input of size n.

2.5 Recursion

A very important topic to remember for this quiz is the idea of recursion. Remember that an algorithm is recursive if it (1) breaks a problem down into smaller versions of itself, and (2) has one or more base cases to which we know the solution. The practical result of this is a function that calls itself.

A good example to demonstrate recursion is any algorithm that computes a summation or a factorial. Look at the following code that gets the sum of all the integers from 1 to max using a recursive approach:

```
int sum_recursive(int max) {
     if(max < 1) {
          printf("Illegal argument. Exiting.\n");
          return 0;
     }
     if(max == 1)
          return 1;
     else
          return n + sum_recursive(max – 1);
}
```

A practice example exercise asks you to convert this back to an iterative algorithm. Remember that the two components of a recursive algorithm are (1) the recursive process and (2) the base case. We can see from the algorithm the base case: it is when max is equal to 1; that is when the algorithm (the for loop) stops doing work and moves on. Intuitively, we know that this equates to the sum of all the numbers from 1

to 1, which we know to be 1. This is the simplest case of the algorithm. The recursive process comes from noting that, for example:

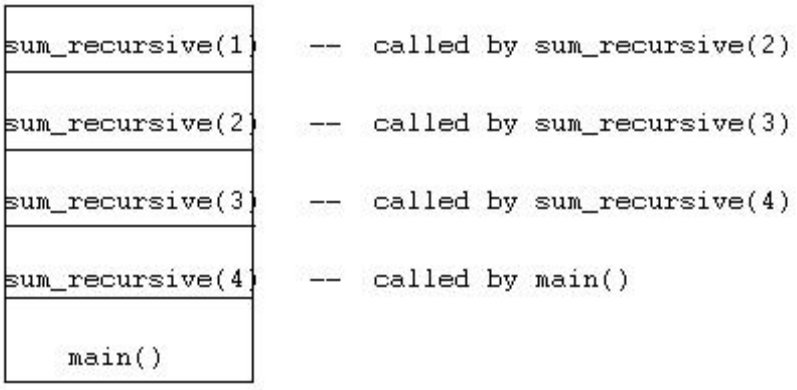
```
sum_iterative(4) = 4 + 3 + 2 + 1
sum_iterative(3) =     3 + 2 + 1
sum_iterative(2) =         2 + 1
```

or...

```
sum_iterative(2) =         2 + 1
sum_iterative(3) =     3 + sum_iterative(2)
sum_iterative(4) = 4 + sum_iterative(3)
```

2.6 The Stack

Another important thing to consider when discussing recursion is the stack. Think of the stack as a pile of function "frames". The last function you called has a frame on top of the stack, while the first function you called has a frame on the bottom of the stack. Applying this to the sum recursive() function, above: If we call sum recursive(4) from main(), what does the stack look like just before the base case returns its value?



Why is this the case?

2.7 Sorting

Again, we have focused extensively on some sorting algorithms thus far. Three in particular are going to be important for you to know: bubble sort, selection sort, and merge sort. Take a look at the section notes that describe these algorithms in detail. Some of the take home points are these:

- Bubble sort and selection sort run, in the worst case, in O(n2). Read through the algorithms that these sorts use, and make sure you understand why this is the time complexity for each.
- Merge sort runs in O(n log n). Note that merge sort is a naturally recursive algorithm. Make sure you understand why we describe it as a recursive algorithm, and why the time complexity is as it is.

There are other sorts that you may want to know or use on the Quiz. These include insertion sort, quicksort, cocktail sort (aka bidirectional bubble sort), etc. Knowing the

time complexity and general algorithms behind these will be sufficient preparation for you.

2.8 Pointers

If you remember nothing else about pointers, remember this: Pointers are Addresses! Pointers are variables, just like any other variables, but instead of holding (for example) a character, they hold an address.

- **Syntax**
  Remember that you declare pointers by inserting an asterisk (*) in front of the variable name.

  ```
  int * iptr ;
  int * iptr1 , var2 ;
  ```

  Note the in the previous example, the first statement declares one pointer, and the second statement declares one pointer and a normal variable. Just because I put the asterisk near int does NOT mean that all variables defined on that line are integer pointers. Remember that pointers store addresses of variables rather than the data directly. If you have an address, you can use the * operator to get to the value. To find the address of a variable, use the & operator. See below for an example.

  ```
  int var;
  int * iptr ;
  iptr = &var ;
  * iptr = 4; /* this will set the var = 4 */
  iptr = 4; /* will compile, but not what you want */
  ```

  Remember that a pointer can only hold the address of a certain type of variable, specified when you declared it. In the above example, iptr can only hold the address of an integer, and not any other type.
- **Passing by Reference**
  Before pointers, we could only pass arguments to function by value. This meant that when we called a function foo with an argument x, the value of x would be copied to a new local variable in foo. No matter what happened to this local variable, it did not change x in the calling function. In other words, we passed only the value of x to foo, but not x itself.

  With pointers, we can now give functions access to the original variable. We do this by passing by reference. We pass the address of the original x (&x) to foo, and foo stores it in a pointer. Now, when foo dereferences that pointer, it grabs a hold of x. Why would we ever want to do this? When x is very large, like a huge array or a big structure, it takes up a lot of memory to keep making new copies of it in each new function. It is much more efficient to have one copy of it, and just tell all the functions where to find it!
- **Arrays and Pointers**
  Arrays and pointers are very interrelated. In memory, an array is a contiguous block of memory. An array of 10 integers is, in memory, a block of (10 * sizeof(int)) bytes. As the array name is implicitly converted into a pointer to the first element of the array, the array name can be thought of as a pointer to the

beginning of this block. This means we can treat the array name as the address of the array, and assign it to a pointer:

```
int arr [10] = {0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9};
int *ptr;
ptr = arr; /* ptr holds address at beginning of array */
```

The indexing operators ('[' and ']') are interrelated with pointer arithmetic. The following boolean expressions all evaluate to TRUE:

```
( ptr[3] == arr[3] )
( *(ptr + 3) == arr[3] )
( arr + 5 == &(ptr[5]) )
( *(++ptr) == arr[1] )
```

2.9 Dynamic Memory

malloc is a function, declared in stdlib.h, that returns memory allocated off the heap. When we declare a pointer, it is uninitialized - it does not point to any memory that our program owns. If we already have the memory that we want our pointer to point to, then we can use the & operator to get the address. However, if we don't have that memory yet, we use malloc to get it dynamically. If there was an error in allocating the memory, malloc returns NULL, so we always must check for this return value before proceeding. This is useful if, for example, we want an array, but we don't know how big it should be until the program is already running.

malloc allocates memory from the heap. The heap is the other main section of memory for programs, along with the stack. The heap is a large pool of memory that we may dip into. While the stack is a tightly policed city park, where local variables and function stack frames are created and wiped out according to strict rules, the heap is a country pasture that is unsupervised. Once you grab some memory from the heap, it sticks around until you release it - you can keep it forever, as long as your program is running. This is great, because as long as you keep the address of your memory in the heap, you can keep using it no matter what function you are in. The downside is you must keep track of its address. When you are done with memory on the heap, you must free it with the function free. It is a serious error to not free all the memory you allocate.

All memory allocated with malloc must be freed with free.

2.10 User-Defined Types

2.10.1 *typedef*

The keyword typedef allows the naming or renaming of types in C. Its syntax is:

```
typedef <old type> <new name>;
```

2.10.2 *enum*

Enums are types of variables that can hold one of an enumerated list of possible values. When you enumerate the possible values, they are assigned integer equivalents, starting with 0 for the first one, 1 for the second one, and so on.

```
enum quad_house_t {CURRIER, PFOHO, CABOT);
enum quad_house_t johns_house = CABOT;
```

2.10.3 struct

Structs are variable types that combine more than one variable.

```
struct student_t {
      char *name;
      int year;
      int age;
};
```

If you have a pointer to a struct, then you may use the -> operator to access member variables. You may also dereference the pointer and then use the . (dot) operator. For instance, the following piece of code accesses members of the struct in three different ways.

```
struct student_t student 1;

int main(int argc, char *argv[]) {
      struct student_t *studentptr = &student1;
      student1.name = "Bobby";
      studentptr->year = 1;
      (*studentptr).age = 19;
      return 0;
}
```

2.11 File I/O

C programs can interact with other files using the file input/output functions defined in stdio.h. An excellent resource for information about all the functions in stdio.h is online, at http://www.cplusplus.com/reference/clibrary/cstdio/.

Within our programs, we interact with files using a special variable type, a FILE* (file pointer). We get one of these by calling fopen, which returns a FILE*. Two key file I/O functions are fread and fwrite, which behave a bit differently than functions we are used to. The return type of fread is size_t. So where does it put the data that it read from the file? It puts it in a location that we specify when calling the function, by passing it a pointer. Take a look at fread's function header:

```
size_t fread (void *ptr, size_t size,
              size_t count, FILE * stream );
```

We have to pass in the address of the location for fread. We pass this in as the first argument, called ptr. The return value is the number of bytes read.

2.12 Digital Forensics

In Problem Set 5, we learned about what happens to files when they are "deleted" from a hard drive. Remember, the reference to them in the drive's file system is erased,

but the data is not. This allows someone to go back to that drive, and read all the data, as we did in the problem set.

How did we recognize different files, when there was no directory structure? We looked at the file headers, which contain metadata about the file, rather than the actual file data itself.

Remember the different ways to truly erase a hard drive. We could physically destroy it, or degauss it, which are the only totally secure ways of data deletion. There are methods that don't render the hard drive useless, but none of these are 100% safe. Make sure you understand why just overwriting data doesn't prevent recovery of the old data.

2.13 Algorithms of Composition

A common problem in computer science is that of finding the best way to make something out of the smallest number of parts possible. A good example of this is the "making change" problem: Given that I have unlimited supplies of coins in the following denominations: 2c, 5c, 10c, and 25c, what is the smallest number of coins needed for me to make up x cents?

- **The Greedy Algorithm** employs the following rule: *Pick the largest denomination possible (still smaller than or equal to x). Subtract this value from x; let this value be the new x. Repeat until x = 0.*

For example, say that x = 70c. If I were to use a greedy algorithm, I would pick the following coins in this order:

  - 25c (now we need 45c)
  - 25c (now we need 20c)
  - 10c (now we need 10c)
  - 10c (now we're done!)

- **The Exhaustive Search** fixes some of the problems of the greedy algorithm. Consider what would happen if we were trying to make 28c instead of 70c. We would get to the point where we had 1c left, and no way to make up that 1c! It is also important to remember that even if a greedy algorithm *does* yield a solution, it might not be the optimal solution. To find the optimal solution, we must employ an exhaustive search.

  The idea behind an exhaustive search is that we find all possible combinations of denominations that make up our total goal. We then compare these, and use the one with the fewest units as our optimal solution.

For example, with denominations 2c, 5c, 10c, and 25c, I can make up 9c as follows:

  - {5c, 2c, 2c}
  - {5c, 1c, 1c, 2c}
  - {5c, 1c, 1c, 1c, 1c}

Clearly the optimal solution is solution 1. In this case, this is also the solution that a greedy algorithm would have returned.

2.14 Linked Lists

Be sure to consider answers to all of the following questions when studying.

- What is our motivation for the use of linked lists? How do linked lists compare to arrays? How do they differ? What is one advantage provided by linked lists that is not provided by arrays? In terms of memory, how do arrays and linked lists differ?

- How do we represent singly-linked lists in C? What are the general algorithms for inserting, deleting, and finding elements in a singly-linked list?
- What is the time required to do an insertion or deletion? What is the time required to perform a lookup? How do these differ from arrays?
- How is the struct definition for a doubly-linked list different from that of a singly-linked list? When is it advantageous to use a doubly-linked list instead of a singly-linked list?

2.15 Binary Trees

Past section notes will prove a valuable resource for you when studying this topic.

- How do we go about implementing a binary tree in C? What is its structure definition? How do we declare a new tree?
- What distinguishes a binary search tree from ordinary binary trees? How do we insert elements into a binary search tree? How do we search the binary search tree? What is the runtime of this search algorithm?
- What are the two distinguishing properties of heaps? How do you heapify a non-heap? What is the runtime of heapification and why? How do we use heaps to implement heap sort? What is the runtime of heap sort and why?

2.16 Hash Tables

- Although hash tables, when implemented with separate chaining, can have asymptotically the same lookup time as linked lists searched linearly, in practice they are usually much more efficient. Why might that be?
- How do we typically represent a hash table in C? How would you declare an instance of a hash table?
- Why are collisions a problem with hash tables but not other data structures we have studied? What are the two ways to deal with collisions in a hash table?
- What is the complexity of the lookup operation? What is the complexity of the insert and delete operations?

2.17 Miscellaneous C Topics

If you feel confident with all of the above topics, and the big topics in the web programming section, then feel free to dive into these:

- Huffman Coding
- Bitwise Operators
- The C Preprocessor
- Compilers

3 Web Programming

We **strongly** encourage you to review the Section Notes for weeks nine and ten, as they contain comprehensive detail into JavaScript and PHP/SQL. Also, review walkthroughs and the scribe notes, as they will be very useful resources when you are brushing up on this material.

3.1 TCP/IP

TCP/IP stands for Transmission Control Protocol/Internet Protocol. It is a set of rules that define how most electronic devices, namely computers, should be connected to the Internet, and how data should be transferred between electronic devices. TCP and IP refer to different protocols which handle different aspects of this communication.

Take home points:

- Every computer has an IP address that uniquely identifies it on the Internet
- This unique address lets you send data (in the form of IP packets) across the Internet
- Routers perform the physical task of figuring out where to send your data to get it to the correct IP address
- IP is connection-less in that it doesn't specify the route that data will take
- TCP handles packets once they arrive at a computer, determining what application they go to.

3.2 HTML and XHTML

Here are some of the important things to remember:

- Basic components of page structure: <html>, <head>, <body> tags
- Basic tags: Links, images, headings
- Tables
- Forms
- GET vs. POST
- Differences between XHTML and HTML

3.3 CSS

Cascading Style Sheets provide a way to override basic XHTML tags

- Basic syntax
    - Selectors
    - Declarations
    - Arguments to
- CSS Classes
- Embedding CSS in XHTML

3.4 PHP

- What advantages does PHP have over plain XHTML?
- $_GET
- $_POST
- $_SESSION
- Connecting to SQL databases

3.5 SQL

SQL (Structured Query Language) provides an interface to server-side databases which you can create.

- SELECT
  - FROM
  - WHERE
  - ORDER BY
- UPDATE
- INSERT
- DELETE

3.6 JavaScript

- Variables and lack of type
- Arrays
  - Differences between C arrays and JavaScript arrays
- +, ==, ===
- Determining the type of a variable
- switch
- for…in, and syntactic differences between that and a for loop in C
- Functions
  - What does it mean to be a first-class object?
  - Event handlers
- Objects
  - Properties
  - Methods
  - Constructors
- The Document Object Model (DOM)
  - Properties of DOM
  - Methods of DOM

Good luck!