

Practice Quiz 1 - Answer Key

CS50 — Fall 2011

Prepared by: Doug Lloyd '09

November 14, 2011

```
1. bool isEven(int x);

2. bool isEven(int x) {
    return (x % 2) ? false : true;
}

3. bool isEven(int *px) {
    return (*px & 1) ? true : false;
}

4. #include <cs50.h>
   #include <stdio.h>

   int main(int argc, char **argv) {
       printf("Please input an integer: ");
       int x = GetInt();
       do {
           if(x & 1)
               printf("%d is odd.\n", x);
           else
               printf("%d is even.\n", x);
           printf("Please input an integer: \n");
           int x = GetInt();
       } while(x != 0);
       return 0;
   }
```

```
5. #include <cs50.h>
    #include <stdio.h>
    #include <ctype.h>
```

```
    int main(int argc, char **argv) {
        if(argc != 2)
            return 1;
        int len = strlen(argv[1]);
        for(int i = 0; i < len; i++) {
            if(!isdigit(argv[1][i]))
                return 2;
        }
        int x = atoi(argv[1]);
        if(x & 1)
            printf("%d is odd.\n", x);
        else
            printf("%d is even.\n", x);
        return 0;
    }
```

6. The result is 16 and x is 10 here.

7. Selection sort involves swapping the lowest remaining unsorted value with the value in the position immediately to the right of the highest sorted value. Insertion sort involves shifting the entire unsorted portion of the array to the right for the purpose of inserting the lowest remaining unsorted value into the position immediately to the right of the highest sorted value.

8. Given key, array, and arraySize:

```
    Let first = 0 and last = arraySize-1;
    while first <= last:
        let middle = average(first, last);
        if array[middle] < key:
            last = middle-1;
        else if array[middle] > key:
            first = middle+1;
        else return true;
    return false;
```

9. 01001101

10. 00000010

11. Either 3 or 4, depending on implementation.

12. It simply prints the argument string passed to `mystery()`.

13. $O(n)$

14. It prints the argument string passed to `newMystery()` backwards.

15. Each pass of cocktail sort is really equivalent to two passes of bubble sort. Though in practice cocktail sort is faster as it stops the problem of “turtles”, in the worst case, such as an array like [7,6,5,4,3,2,1], it will still be making n^2 swaps.

```

16. int main(int argc, char *argv[]) {
    int count = 0;
    double f, sum = 0.00;
    while(f != SENTINEL) {
        sum += f;
        count++;
    }
    double avg = f / count;
    printf("%f is the average of the %d values you entered.", avg, count);
    return 0;
}

```

17. 5 2 1 6 7 3 4 8

18. 2 1 5 6 3 4 7 8

19. 1 2 5 3 4 6 7 8

20. Infinite loops are loops that never terminate. Here are two:

```

for(;;);
while(1);

```

21. Yes, it will terminate. Eventually we will have so many copies of `anotherLevel()` frames running that we will run out of memory, and we will crash.

22. logarithmic time

23. polynomial time

24. linearithmic (loglinear) time

25. `ptr = &i`

26. 2

27. `double *arr = (double *)malloc(40 * sizeof(double));`

28. k^n , where n is the length of the keyword

- 29.
- `#define` should not have a semicolon
 - First for loop goes outside the bounds of the array
 - `j` is undeclared
 - No semicolon after the `malloc()`
 - The second for loop uses `i` where `j` is likely intended

```

30. typedef struct _cue {
    int enrolledStudents;
    char *prof;
    float cueRating;
} cue;

```

Declaring an instance of this structure would be as simple as `cue cs50;`

31. `x.prof`

32. `px->enrolledStudents` or `(*px).enrolledStudents`

33. 68

34. A quick format will only delete the tables that describe where the data is, as an effort to save time. A complete format, however, will write over all of the data with random 1s and 0s, so the old data will be sufficiently removed, and you will not need to worry about the eBay buyer finding old credit card numbers (which they might have been able to do by scanning the disk, much in the same way you scanned the .raw file in PS5).

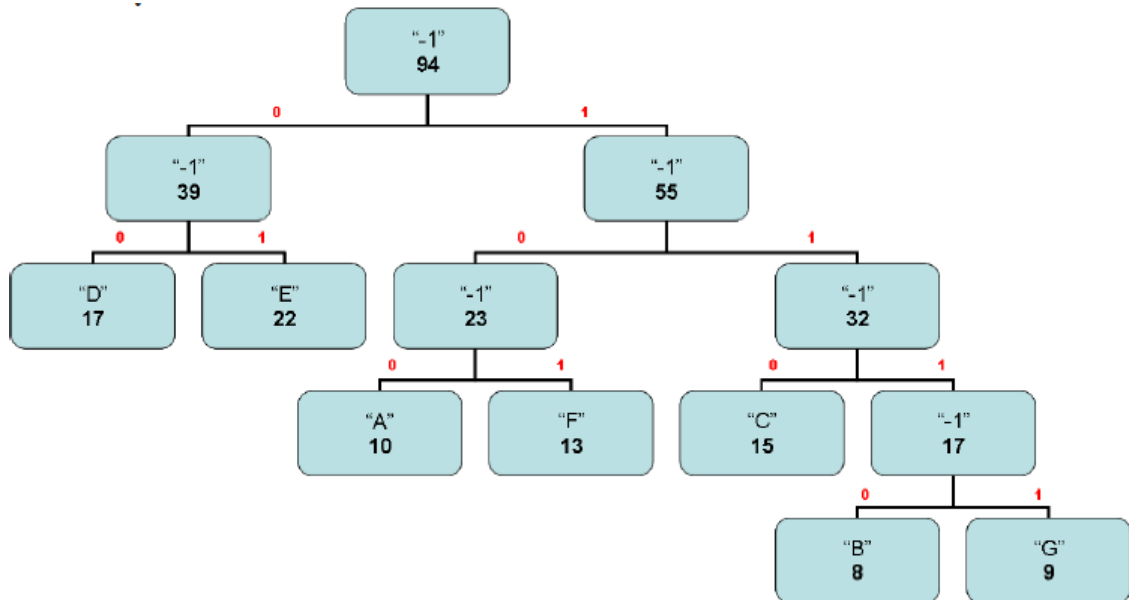
35. `FILE *infile = fopen("input.txt", "r");`

36. 5 coins: 1x150, 1x70, 3x2

37. 4 coins: 1x150, 1x50, 1x25, 1x1

```
38. char *strncpy(char *dest, const char *src, int size) {
    if(dest == NULL)
        dest = (char *) malloc(sizeof(char) * (size+1));
    int len = strlen(src);
    for(int i = 0; i < len && i < size; i++)
        dest[i] = src[i];
    if(len >= size)
        return dest;
    for(int i = len; i < size; i++)
        dest[i] = '\0';
    return dest;
}
```

39. The tree appears below:



And the encoding for each character is:

A	100
B	1110
C	110
D	00
E	01
F	101
G	1111

40. The prefix property states that no encoding of a character may be a prefix to the encoding of another character. For example, an encoding where A = 1 and B = 10 is not immediately decodable, since the "1" in "10" could be a part of B, or a stand-alone A.

41. We have direct and convenient lookup with an associative array. However, we cannot iterate numerically through an associative array using a for loop to print out the full contents.

```

42. void delete(node *ele) {
    if(ele->prev == NULL)
        head = ele->next;
    else
        ele->prev->next = ele->next;
    if(ele->next == NULL)
        last = ele->prev;
    else
        ele->next->prev = ele->prev;
    free(ele);
}
  
```

- ```

43. int treeSize(node *root) {
 if(root == NULL)
 return 0;
 else
 return treeSize(root->left) + 1 + treeSize(node->right);
}

```
44. If we insert onto the end of a chain, as opposed to the beginning of one, it would take  $O(n)$  steps to insert.
45. Optimization of hash tables, such as by altering its size or by using different hash functions, can yield dramatically different results.
- ```

46. int size(node *hashtable[]) {
    int i, size = 0;
    node *start;
    for(int i = 0; i < LENGTH; i++) {
        start = hashtable[i];
        while(start != NULL) {
            size++;
            start = start->next;
        }
    }
    return size;
}

```
47. Open the files in each call for reading and writing, respectively.
48. FILE pointers
49. We are reading up to 1,000 characters at a time from the source file, and writing it to the destination file.
50. Now, we want to read from the destination file, as opposed to writing to it. Since our original handle was used to write, we need to close and reopen to get a handle that instead reads.
- ```

51. void delete_list(node *head) {
 if(head == NULL)
 return;
 else {
 delete_list(head->next);
 free(head);
 }
 return;
}

```

52. 

```
node *reverseList(node *head) {
 node *next, *current, *result = NULL;
 current = head;
 while(current != NULL) {
 next = current->next;
 current->next = result;
 result = current;
 current = next;
 }
 return result;
}
```
53. The error is in the segment that checks for insertion in the middle. The lines `predptr->next = newptr;` and `newptr->next = predptr->next;` need to be switched. Otherwise we are setting `newptr->next` to `newptr`!
54. The function should instead read as follows:
- ```
void printTree(node *root) {
    if(node == NULL)
        return;
    printTree(node->left);
    printf("%d ", node->data);
    printTree(node->right);
}
```
55. It takes up a lot of memory to make a table that large...and most of it will be empty anyway!
56. Because it doesn't take any variables, instead it can take a "type", as an argument, and so it is properly classified as an operator.
57. Everywhere I'm using the dot operator, I should be using the arrow operator!
58. There is a swap counter that stops the algorithm if equal to 0.
59. It doesn't check to see if swaps have been made.
60. It inserts onto the end of the list.
61. It inserts onto the front of the list.
62. It has a memory leak! `second` and `third` never get freed.
63.

```
hashtable[0]:
hashtable[1]:
hashtable[2]: peach
hashtable[3]:
hashtable[4]:
hashtable[5]:
hashtable[6]: banana
hashtable[7]: kiwi->apple->Cherry
hashtable[8]: lemon
hashtable[9]: clementine
```
64. ``
65. (c)

66. `chmod` changes the “permissions” on a file, which control who can read, write, and execute a given file.
67. `chmod 651 file.php`
68. `-rw-r-x--x`
69. `SELECT password FROM users WHERE username='malan'`
70. `INSERT INTO users (username, password, fullname) VALUES ('mtucker', 'l33t', 'Mike Tucker')`
71. `UPDATE users SET password='n3rd' WHERE username='malan'`
72. `SELECT * FROM users ORDER BY username DESC`
73. `SELECT COUNT(*) FROM users`
74. `DELETE FROM users WHERE username='cs50stud'`
75.

```
<? $result = mysql_query("SELECT * FROM users"); ?>
<tr><td>username</td><td>password</td><td>fullname</td></tr>
<? while ($row = mysql_fetch_array($result)) { ?>
<tr>
    <td><? print($row["username"]) ?></td>
    <td><? print($row["password"]) ?></td>
    <td><? print($row["fullname"]) ?></td>
</tr>
<? } ?>
```
76. Two functions that, together, work a single recursive unit.
77. Best-case runtime.
78. A language that does not need to be prepared into an object file before execution.
79. Languages that do not have explicit type declarations.
80. A computer science principle whereby we do not need to understand the inner workings of some code to be able to integrate with it.
81. A function that is called when a certain “event” occurs, such as clicking a button, and that is associated with that button.
82. Like a queue, the first object inserted is the first object to be removed from the list.
83. A special case of a binary tree where the left subtree contains only nodes with keys less than the node’s key, and where the right subtree contains only nodes with keys greater than the node’s key.
84. A type of function that remembers certain aspects of the circumstances in which it was created. It is special in that it has free variables that are bound in the lexical environment.
85. Data that remains after a function or program has stopped running.
86.

```
<html>
<script type=text/javascript>
    function turnGreen()
    {
        document.getElementById(colorflip).style.color = green;
    }
</script>
```



```

<body>
  <div id=colorflip>Turn me green!</div>
  <input type=button value="Do it!" onclick=turnGreen(); />
</body>
</html>

```

87. <head>

88. \$username = \$_POST["username"];

89. The type of the password field should be "password", so as to obscure the input with asterisks.

90. Passwords are passed by \$_GET, which is inherently weak. Also, the strings are not escaped to prevent a SQL injection attack.

91. head

```

{
  background: purple;
  color: yellow;
  font-family: times;
  font-size: 30pt;
}
body
{
  background: green;
  color: red;
  border: orange 16px dotted;
  font-family: verdana;
  font-size: 20pt;
}

```

92. Each is an associative array storing user-inputted variables.

93. The DOM appears on the next page. Note that some elements have been omitted due to space constraints and for the sake of readability.

