Quiz 0

Answer Key

Answers other than the below may be possible.

Short sorts.

0.

2	4	1	5	3	6

1.

1	5	4	2	6	3

2.

2	5	4	1	6	3

Small problem.

3. Because `s` is of type `string`, it's actually the address of a `char` (i.e., a pointer of type `char*`). Even though the string to which `s` points is of length 4, the size of the pointer itself is apparently 8 on the (apparently 64-bit) machine on which the program has been executed.

Role reversal.

4. Hi, Rob. You've designed your loop to iterate so long as `i` does not equal `0`, but you've initialized `i` to `1`, after which you iteratively increment it by `1`. Only once `i` overflows and wraps back around to `0` will your loop terminate. But by then you'll have tried to print `argv[1]` through `argv[2147483647]`, almost all of which are invalid locations in memory if you're only running your program with one command-line argument, and so your program segfaults! Changing your condition to `i < argc` will fix the bug! <3

5. Hi, Jason. Right now your implementation of bubble sort is in $\Omega(n^2)$ because you have a for loop nested within a for loop, both of which iterate, no matter what, $n - 1$ times, which means $(n - 1)(n - 1) = n^2 - 2n + 1$ iterations in total, which is on the order of n^2. If you instead check whether you've actually swapped any values during an iteration of the outer loop, breaking out of it if not, you can reduce the lower bound on your algorithm's running time to $\Omega(n)$ since, in the best case, your algorithm's input will already be sorted, which you can ascertain with just $n - 1$ comparisons, which is on the order of n. <3

Dollar store.

6. Floating-point values are imprecise, and, by adding floating-point values (e.g., `0.01`) to floating-point values (e.g., `f`), that imprecision adds up. In this case, iteratively adding `0.01` to `0.00` never quite equals `1.00` (which, as a hard-coded constant, is technically of type `double` with even more bits of precision than a `float`), and so the loop iterates beyond `1.00`.

7.
```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 100; i++)
    {
        printf("$%.2f\n", (float) i / 100);
    }
}
```

CS50 Library 2.0.

8.
```
char* CopyString(char* s)
{
    char* t = malloc(sizeof(char) * (strlen(s) + 1));
    if (t == NULL)
    {
        return NULL;
    }
    for (int i = 0, n = strlen(s); i <= n; i++)
    {
        t[i] = s[i];
    }
    return t;
}
```

9.
```
#include <cs50.h>
#include <stdio.h>

void GetInts(int* a, int* b);

int main(void)
{
    int x, y;
    GetInts(&x, &y);
    printf("x is %i, y is %i\n", x, y);
}

void GetInts(int* a, int* b)
{
    *a = GetInt();
    *b = GetInt();
}
```

WTf.

10. `A`

11. `A`

12. `toupper`

Cost-benefit analysis.

13.

	benefit	**cost**
linked list instead of array	A linked list is not of a fixed size; it can grow or shrink as needed.	Linking a linked list's nodes together via pointers requires additional space. Linked lists do not allow random (constant-time) access, and operations tend to require linear time.
merge sort instead of insertion sort	Merge sort tends to be faster. Whereas merge sort is in $O(n \log n)$, insertion sort is in $O(n^2)$.	Merge sort requires more space (an additional array to store values while merging).
binary search instead of linear search	Binary search tends to be faster. Whereas binary search is in $O(\log n)$, linear search is in $O(n)$.	Binary search requires that its input be sorted, which might require more than linear time. Binary search also requires that its input be randomly accessible.

Searching and sorted.

14.
```
1  /**
2   * Searches sorted array of given length for value in O(log n)
3   * time.
4   */
5  bool search(int value, int* array, int length)
6  {
7      if (length == 0)
8      {
9          return false;
10     }
11     int middle = length / 2;
12     if (value < array[middle])
13     {
14         return search(value, array, middle);
15     }
16     else if (array[middle] < value)
17     {
18         return search(value, array + middle + 1, length - middle - 1);
19     }
20     else
21     {
22         return true;
23     }
24 }
```

15.
```
bool sorted(int array[], int length)
{
    for (int i = 0; i < length - 1; i++)
    {
        if (array[i] > array[i+1])
        {
            return false;
        }
    }
    return true;
}
```

Jack is back.

16. By using a queue, Jack would no longer wear the same clothes every day, as he would rotate through his clothes in FIFO (first in, first out) fashion rather than LIFO (last in, first out) fashion.

17.
```
void push(int n)
{
    if (s.size != CAPACITY)
    {
        s.numbers[s.size] = n;
        s.size++;
    }
}
```

18.
```
void enqueue(int n)
{
    if (q.size != CAPACITY)
    {
        q.numbers[(q.front + q.size) % CAPACITY] = n;
        q.size++;
    }
}
```

(Re)curses.

19. `123`

Bold claims.

20. Not true. Asymptotic notation like *O* disregards constant factors and lower-degree terms, so an algorithm that runs in $O(\log n)$ time might, in actuality, run in $k \log n$ time, where k is some very large constant. And $k \log n$ might very well be much larger than the actual running time of some $O(n^2)$ algorithm.

21. True. In the best case, the item for which linear search is searching might be the first item encountered, and so a lower bound on the running time of linear search is indeed some constant.

22. True. Selection sort naively iterates over its input, selecting the smallest item among n items, the smallest item among $n - 1$ items, the smallest item among $n - 2$ items, and so forth until no items are left. The implication is $n(n - 1)/2$ comparisons, which is on the order of n^2, and so a lower bound on the running time of selection sort is indeed $\Omega(n^2)$.

OIC.

23.
```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

bool gf(string code);

int main(void)
{
    if (gf(GetString()))
    {
        printf("gluten-free\n");
    }
    else
    {
        printf("not gluten-free\n");
    }
}

bool gf(string code)
{
    string prefix = "011206";
    for (int i = 0, n = strlen(prefix); i < n; i++)
    {
        if (code[i] != prefix[i])
        {
            return false;
        }
    }
    return true;
}
```

Similar but different.

24. A local variable exists only within the scope of a function (or within the scope of a block within a function), whereas a global variable, when properly declared, can be accessed by any function in a program.

25. Although both `NULL` and `'\0'` signify `0`, the former is a pointer (of type `void*`), typically used to indicate the absence of a valid address, and the latter is of type `char`, typically used to terminate a string.

26. Whereas the stack is a portion of memory in which functions' local variables and more are stored temporarily, the heap is a portion of memory that can be used to store values that survive functions' returns.

27. Whereas `#include <cs50.h>` is a preprocessor directive that tells a compiler to include within a file being compiled the contents of `cs50.h` (among which are declarations of functions and types), `-lcs50` is a command-line argument that tells `clang` to link a program being compiled with the CS50 Library's own object code (in which the library's functions are actually defined).