CSCI 2270 Name: _____
MIDTERM 1

Honor code.

Please sign: Copying other people's work and pretending it is mine is theft, and I know that. I promise not to cheat or to help someone else to cheat in this class.

_____

1a. Suppose my algorithm is cubic, $O(n^3)$, and I triple the size of its input. How much longer will it take to run?

Let m = 3n. The algorithm is $O(m^3) = O((3n)^3) = O(3^3n^3) = O(27n^3)$.

1b. Suppose my algorithm is logarithmic, $O(\log_2 n)$, and I quadruple the size of its input. How much longer will it take to run?

Let m = 4n. The algorithm is $O(\log_2 m) = O(\log_2 4n) = O(\log_2 4 + \log_2 n) = O(2 + \log_2 n)$

2. Why does the remove_node function:

```
void remove_node(node*& head_ptr, const int& target)
```

pass in the head ptr by reference? Give me a scenario where this is needed.

We need any changes that remove_node makes to the pointer head_ptr to stay permanent outside the code. For instance, any removal of the first node in the list requires us to update the head_ptr. To accomplish this, we add an & so we can pass in the head_ptr by reference. Any changes to the head_ptr's address will then be updated permanently when the function is done working.

3. The code below looks for a node containing target in a list, but will crash. Why? And how would you fix this problem?

```
node* cursor = head_ptr;

while (cursor->data != target && cursor != nullptr)

      cursor = cursor->next;

…
```

That while loop is the problem. C++ compares cursor->data to target before it checks if cursor == nullptr. If we reach the end of the list without finding the target, we'll try to get nullptr->data; this is like saying (*nullptr).data, and dereferencing nullptr is always going to be a bad idea.

To fix this, switch the 2 conditions. C++ can then decide that cursor's nullptr, and thus the whole while condition must be false, before it tries to grab the data at that nullptr.

4. How do I copy elements 7-12 of a 30-element array called **`alice`** over elements 9-14 of a 45-element array called **`bobo`**, using the copy command? You should not need any brackets [] in this answer.

Pointer arithmetic!

Remember that the first element is at index 0, so elements 7-12 are at alice[6] to alice[11].

Copy takes the index of the place to start copying from source, alice + 6,
the place to stop copying from source, alice + 12, which is one past alice[11]
and the place to start copying into destination, which is bobo + 8.

copy(alice + 6, alice + 12, bobo + 8);

5a. What can an array do faster than a linked list, and why?

Can be destroyed faster (one delete)
Can be walked faster (fewer dereferences)
Can access any element in the array in O(1) time (pointer arithmetic)
Can do binary search and find things quickly

5b. What can a linked list do faster than an array, and why?

It can add nodes between other nodes without needing to move things
It can thus resize in constant time

6. How can I tell if two arrays are shallow copies of each other when I am writing tests on my code?

Shallow copies are copies that aren't independent of each other, and usually these come from copying pointers. (Note that this question is not asking you if 2 unsorted arrays have the same numbers; both shallow and deep copies will look the same in terms of the contents.)

I could check if their addresses were the same (I sometimes call this the underpants test, but that's not a technical term). If the 2 arrays are crammed into the same memory address, they must be the same array.

Or I could change one array and check the other one. If the change appears in the array I didn't change, I must have a shallow copy.

7. For floating point numbers on computers, circle which of these values is most likely to be computationally *inaccurate*. There may be one answer, or several.

$0.75 = 2^{-1} + 2^{-2}$	0.325
$0.0625 = 2^{-4}$	$0.5 = 2^{-1}$
0.15	$0.125 = 2^{-3}$
$0.875 = 2^{-1} + 2^{-2} + 2^{-3}$	0.1

8a. Would this function

```
void bleh(int* stuff)
```

let us change the data in the array called stuff? Justify your answer.

Yes. We get the location of this array via the int* pointer, so we can change the data at the location given by that pointer, and it will stay changed after the function runs.

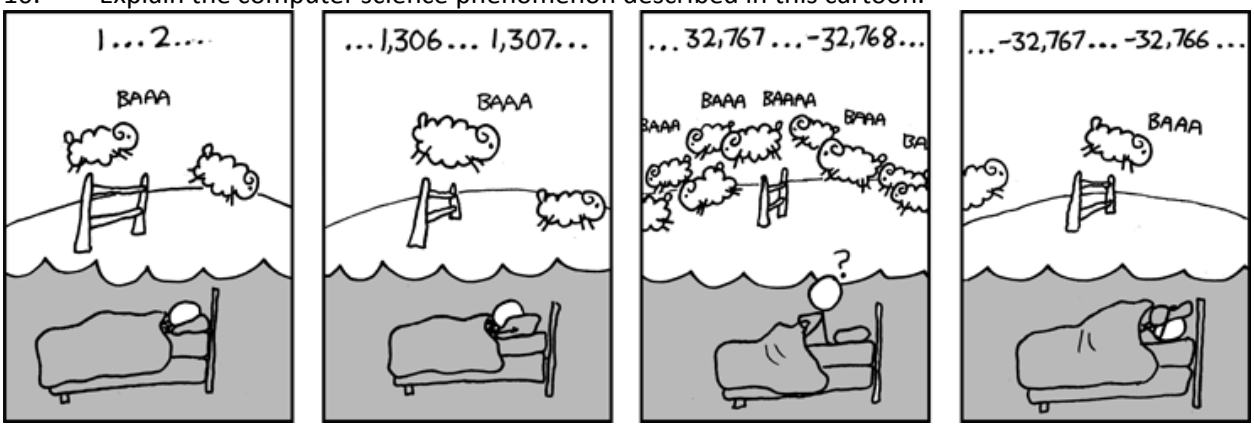
8b. Would that same function **`bleh`** be able to resize the stuff array? Justify your answer.

No. It could try, but any changes we make to the pointer stuff will not persist after the function ends. We'd need to pass the pointer to stuff in as a reference parameter (int*& stuff) to be able to change the array to a different size.

9. Why do you get a warning when assigning between ints and unsigned ints?

Unsigned integers go from a minimum of 0 to a maximum of $2^{32}-1$. Integers go from a minimum of -2^{31} to a maximum of $2^{31}-1$. If we assign an unsigned int > $2^{31}-1$ to an integer, it will roll over to a negative value. And if we assign a negative integer to an unsigned integer, it will roll over to a large positive value. So this assignment's only safe for numbers between 0 and $2^{31}-1$.

10. Explain the computer science phenomenon described in this cartoon.



We have a small (2-byte) integer counting sheep. At some point it reaches its maximum (2^{15} - 1) and then it rolls over to its minimum value, -2^{15}, at which point all the sheep (including 32768 who were never counted), all jump back over the fence. After this, the counter begins counting up again.