Instructions:

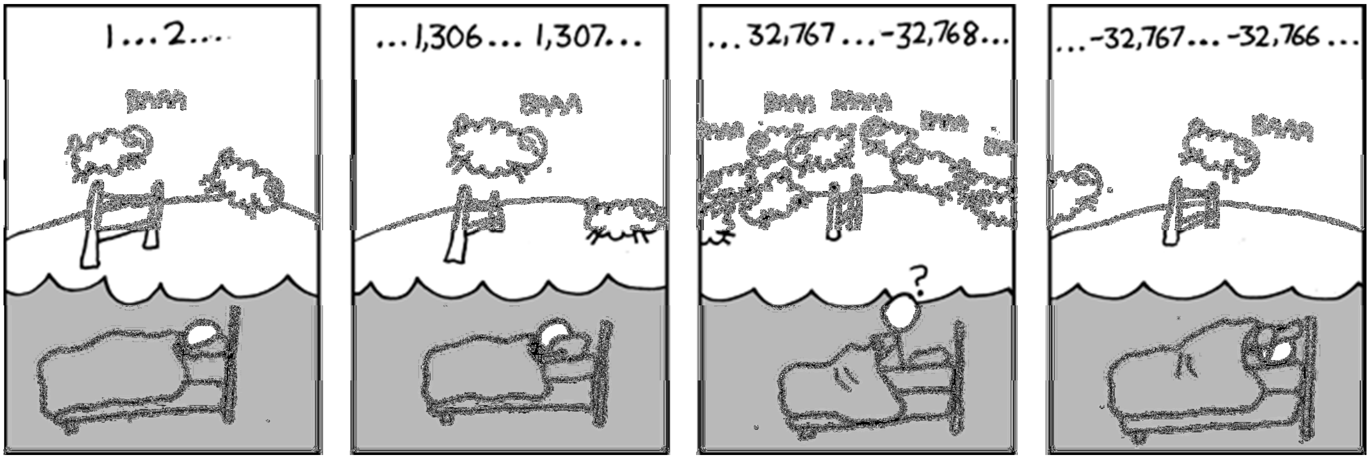
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 7 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name ______**Solution**______________________________
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

____
signed


1... 2...
BAAA
...1,306... 1,307...
BAAA
...32,767...-32,768...
BAAA
BAAA
BAAA
BAAA
BAAA
?
...-32,767...-32,766...
BAAA

xkcd.com

1. [18 points] Perform the indicated conversions. You must show work in order to receive credit!

a) Convert from unsigned form to base-10: `1010 0101`

This is a straight base-2 representation, so it's 2^0 + 2^2 + 2^5 + 2^7 = 1 + 4 + 32 + 128 = 165.

b) Convert from 8-bit 2's complement (i.e., signed) form to base-10: `1001 0111`

This is a negative value (high-order bit is 1). The corresponding positive value is 0110 1001 (flip the bits and add 1).

That represents 2^0 + 2^3 + 2^5 + 2^6 = 1 + 8 + 32 + 64 = 105.

So the original bit-sequence is equivalent to the base-10 representation -105.

c) Convert from base-10 to 8-bit unsigned binary: `29`

Successively divide by 2 and record remainders:

```
2  29
   14   1
    7   0
    3   1
    1   1
    0   1
```

So, in base-2 this is represented as 11101.

2. Recall that C supports both signed and unsigned integer types. For example, in C we have the signed type `int16_t` and the unsigned type `uint16_t`.

a) [8 points] What is the difference between the way an `int16_t` value and a `uint16_t` value would be represented in memory? Be precise.

Signed integer values, like `int16_t`, are represented in 2's complement.

Unsigned integer values, like `uint16_t`, are represented in pure base-2.

Both types would be represented using 16 bits.

b) [8 points] Briefly describe a programming situation in which having the `uint16_t` type available would lead to a better solution than if you only had the signed integer types available, and explain why.

The key difference is that `uint16_t` variables can store substantially larger values than `int16_t` variables; both types have the same size range, but the maximum for unsigned types is larger.

So, any scenario in which we need to represent only non-negative values, but the largest ones will exceed the maximum for an `int16_t` type, would illustrate the point.

One example would be in storing scores for a game (if the maximum likely score was no larger than 65000 or so).

Another example would be in numbering records for a small club or school; negative values would be irrelevant, and the upper limit for the `uint16_t` might be sufficient to handle records for a considerable length of time.

3. [10 points] Circle the strings below that would match the following regular expression:

```
(abc)+.[0-9]?(z|x|f)
```

a) `abcKt` [✓ b) `abcH8z`] c) `%6x` [✓ d) `abc.2x`] [✓ e) `abc12f`]

---

4. [14 points] Formulate a `grep` regular expression that will match a small subset of possible web addresses. The regular expression should match all, and only, those strings described below. A web address must satisfy the following conditions:

- The web address may or may not start with "http://"
- Immediately after the "http://" the string may or may not continue with "www."
- If the string does not contain "http://" the string may or may not start with "www."
- The web address must contain a domain name, after "http://" and "www." if they exist, which in this case only consists of one or more letters and numbers
- The web address must end with either ".com", ".org", or ".net"

So for example this regular expression should match:

```
http://www.facebook.com
http://www.example1.net
http://slashdot.org
www.google.com
google.com
www.npr.org
```

We have `http://`, or not:	**`(http://)?`**
We have `www.`, or not:	**`(www.)?`**
We have a nonempty, alphanumeric domain name:	**`[a-zA-z0-9]+`**
We have a domain category:	**`(.com|.org|.net)`**

So, this will do it:

```
(http://)?(www.)?[a-zA-z0-9]+(.com|.org|.net)
```

---

5. [12 points] Formulate a `grep` regular expression that would match all, and only, three-digit integers between 200 and 345. So, for example, it would match 243 and 322 but not 364.

`2[0-9]{2}`	**matches 200 - 199**
`3[0-3][0-9]`	**matches 300 - 339**
`34[0-5]`	**matches 340-345**

So, this will do it:

```
2[0-9]{2}|3[0-3][0-9]|34[0-5]
```

6. [16 points] Write an implementation of the C function that is specified below. Pay attention to the pre- and post-conditions, and the specified return value.

```
// Pre:
//         A[] contains AUsage values
// Post:
//         Barring an error, each element of A[] has replaced its
//         immediate successor, IF it divided its successor
//         evenly.
// Returns:
//         The number of elements that were replaced.
//
int Q6(int A[], int AUsage);
```

For example, given the arrays below, the function should respond as indicated:

```
A = {3, 9, 9, 2, 6, 12} → A = {3, 3, 9, 2, 2, 6} and returns 4
```

Here's one solution:

```
int Q6(int A[], int AUsage) {

   int numReplaced = 0;  // number of replacements that have been made

   // Trivial array traversal, EXCEPT that you must avoid looking for
   // a successor to the last element in the array, and you don't want
   // "propagate" changes upward:
   //
   for (int idx = AUsage - 1; idx > 0; idx--) {

      // Subtlety:  you have to avoid dividing by zero:
      if ( A[idx - 1] != 0 && A[idx] % A[idx - 1] == 0 ) {

         A[idx] = A[idx - 1];

         numReplaced++;
      }
   }

   return numReplaced;
}
```

You can also do an "upward" traversal of the array, starting at index 0, but you have to use a temporary variable to hold the "old" value of the successor so you use that when you take your next step.

7. Consider the following short C function. The given code, with appropriate include directives, compiles with no errors or warnings.

```
// Copies the contents of the array A[] into the arrays B[] and C[],
// alternating the elements.
//
// Example:
//                                                B[]:  1  4  6  8  9 12
//    A[]: 1  3  4  5  6  7  8  9  9 11 12 -->
//                                                C[]:  3  5  7  9 11
//
// Pre:      A[] contains ASZ elements in non-descending order.
//           B[] is large enough to hold ASZ/2 + 1 elements.
//           C[] is large enough to hold ASZ/2 + 1 elements.
// Post:
//           A[] is unchanged.
//           B[] and C[] contain alternating elements of A[],
//           as shown above.
//
void Q7(int32_t A[], uint32_t ASZ, int32_t B[], int32_t C[]) {

   uint32_t Apos = 0, Bpos = 0, Cpos = 0;

   while ( Apos < ASZ ) {         // walk A[] and copy...

      B[Bpos] = A[Apos];
      C[Cpos] = A[Apos + 1];

      Apos = Apos + 2;
      Bpos++;
      Cpos++;
   }
}
```

a) [6 points] The implementer of the C function shown above has assumed something that is not guaranteed by the preconditions. What is that?

The implementation copies pairs of elements from `A[]` into `B[]` and `C[]`.

That only makes sense if the number of elements in `A[]`, `ASZ`, is even.

There were lots of incorrect answers here, including (but not limited to):

- A[] might be too large for its size to be specified by a uint32_t value; if so, you've got bigger problems, since an array that large will require 16GB of memory.
- One of the parameters passed to the function might be of the wrong type; e.g., the actual parameter corresponding to ASZ might be a signed int; if so, you've got the "caller is an idiot" error, but that will not automatically cause an error, since the caller's value will simply be mapped into the range of a uint32_t when it's passed.
- B[] or C[] might not be empty; nothing in the function specification says that any previous contents are supposed to be preserved, in fact the second post-condition and the example should make it clear that any previous contents will be over-written.
- The implementation assumes the values in A[] are in ascending order, not non-descending order; that's simply false, although the example is in ascending order.

b) [8 points] Assuming all of the given preconditions are satisfied, how might this assumption affect the execution of the given function?

The preconditions say nothing about whether `ASZ` is even or odd; if `ASZ` is odd, then on the last pass through the loop, the code will attempt to access a memory location just past the end of the array.

The actual effect at runtime depends on ownership of that memory location; but, if there's not a runtime error, then the code will copy an unknown value into `C[]`.

The common issue here was that three things need to be said:

- an array-out-of-bounds error will occur if ASZ is odd
- that could lead to a segmentation fault at runtime
- if not, it will lead to the copying of a value into C[] that should not go there

And remember, there is no such thing as automatic bounds checking for arrays in C, nor is there any such thing as an "array index out of bounds" exception (or any other exceptions) in C.