You may not use any internet devices. You will be graded on functionality – but good style saves time and helps graders understand what you were attempting. You have 110 minutes. We hope this exam is an exciting journey.

**Note:** DO NOT WRITE in this booklet. Only work in the answer booklet will be graded.
Note: you may need to scroll to fully view blocks of code.

Answer the following short answer questions below.
A) If we interpret the hexadecimal 0xA2 as a signed 32-bit value, what is its value in decimal?
B) What is the decimal number -99 in 8-bit signed (two's complement)?
C) We have talked about the stack and the heap as two ways of storing data for our programs. Explain the difference between memory on the stack and the heap, and give a specific example of when you might choose one allocation technique over the other (you may provide code in your example, but are not required to).
D) What is the difference between a memory leak and a memory error? Give a specific example for each of when they might occur.
Scientists in a DNA lab have run into a problem; their recorded DNA strands somehow got corrupted and split into fragments! Luckily, some of the fragments overlap, so if they find out how the different strands overlap, they can reassemble them. That’s where you come in!

The scientists need your help implementing a function `repair_dna` that takes in two DNA fragments as strings, and returns a reassembled DNA strand, as a string. We reassemble the strands by looking at how much the start of the second DNA strand overlaps with the end of the first. If they overlap in this manner, we combine them by overlapping the second with the first as much as possible, and return a single string that represents this combination. Here are some examples of reassembling, with the overlapping region underlined in each example:

- `repair_dna("ACT", "TCG")` returns "ACTCG"
- `repair_dna("AAAC", "ACTT")` returns "AAACTT"
- `repair_dna("ACT", "GCC")` returns "ACTGCC"
- `repair_dna("ACT", "ACT")` returns "ACT"
- `repair_dna("ATAGA", "AGAC")` returns "ATAGAC"

The returned string should be heap-allocated (it will be the caller’s responsibility to free it). Notice how if the fragments are the same, the returned string is the same DNA sequence as either strand, and if there is no overlap, the returned string is the concatenation of the two strands.

The scientists have started implementing this function, but need your help to finish it. Their approach is to check overlap in decreasing amounts, with the idea that the first overlap they find will thus be the largest amount. Help them out by filling in the missing code portions below. The resulting implementation should compile cleanly and not have any memory errors or leaks. You may not need every blank. **You should have no loops in the code that you write.**
char *repair_dna(char *dna1, char *dna2) {
    int dna1_length = strlen(dna1);
    int dna2_length = strlen(dna2);
    int overlap;

    /* Find the largest amount that start of dna2 overlaps
    * with end of dna1
    */
    for (overlap = dna1_length; overlap > 0; overlap--) {
        /* If start of dna2 has an overlap of `overlap`
        * with end of dna1, stop.
        */
        if (________EXPRESSION 1 HERE______________) {
            break;
        }
    }

    // create and return the new heap-allocated string

    __________________0 OR MORE LINES HERE____________________
}
The `filter` operation is a common programming pattern in many languages. It is a function that takes in an array and a `select` function and returns a new array with only the elements for which the `select` function returns `true`.

A generic `filter` function in C would therefore have the following signature:

```c
void *filter(void *base, int nelems, int elem_size_bytes,
            int *new_nelems, bool (*select_fn)(void *));
```

It takes in the following parameters:

- `base`: a pointer to the first element in the array to filter
- `nelems`: the number of elements in the provided array
- `elem_size_bytes`: the size of a single provided array element, in bytes
- `new_nelems`: a pointer to an integer that should be updated to store the number of elements in the returned array
- `select_fn`: a function pointer that accepts a single parameter, a pointer to an element in the array, and returns `true` if that element should be included in the new array, or `false` otherwise.

`filter` should return a pointer to the first element of a new heap-allocated array containing just the elements for which `select_fn` returned `true`. It is the caller's responsibility to free the array when no longer needed.

Since `filter` does not know initially how big the resulting array will be, it should use a resize-as-you-go approach. The array should start out with space for `DEFAULT_CAPACITY` elements (a provided constant), and when the array fills up, it should double its current size.
A) Implement the generic C filter function. Your implementation should compile cleanly and not have any memory errors or leaks.
B) Implement the `select_large_ints` function that could be used as a select function for `filter` with an array of `ints` to filter out any numbers less than 5. Your implementation should compile cleanly and have no memory errors or leaks.
C) Implement the `select_valid_dna_strands` function that could be used as a select function for `filter` with an array of DNA strand strings to filter out any invalid strands. A DNA strand is considered invalid if any of the following are true:

- Its length is less than 3
- It contains characters other than (uppercase) A, C, T or G

You should not use any loops in your implementation of this `select_valid_dna_strands` function.

Then, using this function along with the `filter` function, finish the implementation of the `main` function below to select only valid DNA strands from the hardcoded array of strings, and print out these strands. Your implementation should compile cleanly and have no memory errors or leaks.

```c
int main(int argc, char *argv[]) {
    char *dna_strands[] = {
        "ACTG",
        "AA",
        "ACTGWW",
        "aCTG",
        "ACCCC",
        ""
    };
    int nelems = sizeof(dna_strands) / sizeof(dna_strands[0]);

    /* use filter along with select_valid_dna_strands to make
       * an array of dna_strands' values which are valid,
       * and print each of its elements. Specifically,
       * it should print all but the "AA", "ACTGWW", "aCTG" and "".
       */

    ___________________0 OR MORE LINES HERE___________________

        return 0;
} 
```
You and your friends are working on scheduling software for Stanford students to find times when they are free. You decide to efficiently represent a schedule using a 32-bit unsigned int, where each of the 32 bits represents a consecutive 45-minute block in your schedule. The bit is 1 if the person has a commitment at that time, or 0 if they do not have a commitment at that time. In other words, the most significant bit would represent the slot 12midnight - 12:45AM, the second most-significant-bit would represent the slot 12:45AM - 1:30AM, etc., and the least significant bit would represent 11:15PM - 12midnight (the 45 minute chunks make a 24 hour day easily divisible into 32 bits). Here's an example of a schedule encoded in this fashion:

```c
/* This schedule only has commitments from:
 * 12midnight - 12:45AM
 * 12:45AM - 1:30AM
 * 1:30AM - 2:15AM
 * 9:45PM - 10:30PM
 * 10:30PM - 11:15PM
 * 11:15PM - 12midnight
 */
unsigned int schedule = 0b11100000000000000000000000000111;
```

Now that you have chosen this as your representation, you need to implement various pieces of logic to compare and examine users' schedules. Implement the specified functions below using bit operations, without using any loops.
A) Implement the function `both_free` that returns a 32-bit unsigned int with a bit being 1 if both of the provided schedules have that slot free (i.e. does not have any commitment), or 0 otherwise. **You should not use any loops.**
B) Implement the function `at_least_one_free` that returns a 32-bit unsigned int with a bit being 1 if at least one of the provided schedules has that slot free (i.e. does not have any commitment), or 0 otherwise. You should not use any loops.
C) Implement the function `free_for_duration` that returns `true` if the provided schedule has no commitments between the specified start and end slots, or `false` otherwise. The start and end slots are specified as **bit indices** between 0 and 31 inclusive, where index 0 refers to the most significant bit, and index 31 refers to the least significant bit. The start and end slots are inclusive; meaning that if start = 0 and end = 3, the function should return `true` if and only if the provided schedule has no commitments for slots 0, 1, 2 or 3. **You should not use any loops.**

As an example, here are some sample calls to this function, and what they should return:

```c
/* This schedule only has commitments from:
   * 12midnight - 12:45AM
   * 12:45AM - 1:30AM
   * 1:30AM - 2:15AM
   * 9:45PM - 10:30PM
   * 10:30PM - 11:15PM
   * 11:15PM - 12midnight
   */
unsigned int schedule = 0b11100000000000000000000000000111;
free_for_duration(schedule, 0, 2);  // should return false
free_for_duration(schedule, 3, 5);  // should return true
free_for_duration(schedule, 2, 5);  // should return false
free_for_duration(schedule, 6, 6);  // should return true
free_for_duration(schedule, 3, 28); // should return true
```