Suggested Solutions

Note: For selected questions only. The solutions provided here might not be the only solutions but would give you an idea of how they look like. For more information consult your instructor or TAs.

Q1. What will be the output of this code? Provide a brief explanation.

```
int main() {
  char ca = 'a';
  char cb = 'b';

  char* const ptr1_char = &ca;
  char* ptr2_char = &cb;
  ptr2_char = ptr1_char;

  printf("%c", *ptr1_char);
  printf("%c", *ptr2_char);
  return 0;
}
```

Solution: Run the code yourself and see the output. Before the printf functions, both ptr1_char and ptr2_char point to the same address (where ca is located in the memory), so when printing the content pointed to by these 2 pointers, the value stored by ca is printed. Notice that printf does not add a newline character.

Q2: Explain what's wrong with this code. Compile it in CSIL to see if it compiles and what it outputs. Compile it in replit to see if it compiles and what it outputs.

```
#include <stdio.h>

int main() {
  const char c = 'a';

  char* ptr_char = &c;
  *ptr_char = 'B';

  printf("%c", c);
  return 0;
}
```

Solution: The variable c is declared as a const char type but the pointer variable ptr_char is declared as a pointer that points to a regular char type. There is a type mismatch. Some compilers will still let it pass and make some adjustments automatically, but this is a bad practice as the output shows that the variable c is now storing a different value, which is not expected as it was declared with const.

Q3: Explain what's wrong with this code. Compile it in CSIL to see if it compiles and what it outputs. Compile it in replit to see if it compiles and what it outputs.

```
#include <stdio.h>

int main() {
  char ca = 'a';
  char cb = 'b';

  char* const ptr1_char = &ca;
  char* ptr2_char = &cb;
  ptr2_char = ptr1_char;
  ptr1_char = ptr2_char;

  printf("%c", *ptr1_char);
  printf("%c", *ptr2_char);
  return 0;
}
```

Hint: Think about the type for ptr1_char and what that means.

Q4: Write a function that gets an array of ints of length n, and returns the maximum value.

```
int max(const int array[], int n)
```

Solution:

```
int max(const int array[], int n) {
  int largest = array[0]; //assume first one is the largest
  for (int i=1; i<n; i++) { //start from 2nd and find the largest
    if (array[i] > largest) {
      largest = array[i];
    }
  }
  return largest;
}
```

Q5: Write a function that gets an array of ints of length n, and returns the sum of the two largest values.

```
int sum_max2(const int array[], int n)
```

Hint: Use a similar concept from Q4 to find the largest value, at the same time remember its index. Then repeat the process while skipping the index so the second largest value can be found. Finally return the sum of these two values.

Q6: Write a function that gets a 2d array of ints, and checks if it contains two equal rows.

```
bool contains_equal_rows(int height, int width,
                         const int array[height][width])
```

Solution: The answer to the question can be simplified by first creating a helper function to compare just 2 rows:

```
bool are_equal(int width,
          const int array1[width], const int array2[width]) {
  for (int i=0; i<width; i++) {
    if (array1[i] != array2[i]) {
        return false;
    }
  }
  //if reaches this point, all items are the same
  return true;
}
```

Then use a for-loop to compare the rows. Note how the iterators are set so every two rows are only compared once.

```
bool contains_equal_rows(int height, int width,
                     const int array[height][width]) {
  for (int i=0; i<height-1; i++) {
    for (int j=i+1; j<height; j++) {
      if (are_equal(width, array[i], array[j])) {
        return true;
      }
    }
  }
  //if reaches this point, no equal rows are found
  return false;
}
```

Q7: Write a function that gets an array of digits (given as ints) of length n, and prints the largest number possible using these digits.

```
void print_max_number(const int digits[], int n)
```

For example: on input {7, 5, 3, 8, 3, 0} the function should print 875330.

*Note: you are not required to return the number because it may be too large.

Hint: This question is tricky because you can't sort the digit array and at the time of the release of this problem set you are not supposed to use dynamic arrays. So, the solution to this question is to create a counting array (with a fixed size of 10). Then, for each digit in the array increment the corresponding item (e.g., if you see an 8 increase array[8] by 1). Finally, from 9 print the corresponding # of times.

Q8: Write a function that gets two arrays of ints a and b, and checks a[i] < b[i] for all i.

Decide on the correct signature for the function. Implement it.

Hint: The answer to this question is to use a for-loop to compare items from a and b one by one till one of the arrays are out of items. Then you have to decide if for unequal array lengths do you consider only up to the short length or they are not equal if the lengths are not equal.

Q9: Write a function that gets a string (array of chars ending with '\0') and reverses it in place.

```
void reverse(char str[])
```

For example, on input "hello", str should become "olleh".

Solution: The answer to this question is a series of swapping from the outside to the middle.

```
void reverse(char str[]) {
  int length = strlen(str); //or just count # items till \0
  for (int i=0; i<length/2; i++) {
    char temp = str[i];
    str[i]= str[length-i-1];
    str[length-i-1] = temp;
  }
}
```

Q10: Write a function that gets a string str and a char c, and returns the index of the first appearance of c in str. If c is not in str, the function returns -1.

```
int str_find(const char* str, char c)
```

For example, on input str="ABCa!i!aaD" and c='a' the function returns 3. on input str="ADaCFaDDa" and c='b' the function returns -1.

Hint: The answer to this question is basically a linear search, except the length of the string needs to be determined first to set up the boundary of the for-loop properly.

Q11: Write a function that gets a string (array of chars ending with '\0') and checks if it is a palindrome.

```
bool is_palindrome(const char* str)
```

For example, it outputs true if the input is "racecar", but returns false if the input is "engage".

Hint: The answer to this question is very similar to the answer to the reverse function in Q9, except instead of swapping the items you compare the items. It at any point the two items are not the same, return false, otherwise when the for-loop ends return true, as it means no items are mismatched.

Q12: Write a function that gets a string (array of chars ending with '\0') and returns the longest substring that is palindrome. The returned string should be allocated on the heap.

```
char* longest_substring_palindrome(const char* str)
```

For example, on the input is "ABCBDEFFFEDS", it needs to return "DEFFFED". Solve it in O(n^2) time.

Solution: There are a few ways to solve this problem. One simple way is to check every possible substring and keep track of the longest palindrome: use a nested-for-loop to determine the start and end of all the substrings and check for palindrome. But this is an O(n^3) algorithm. A better way is to treat each item the middle of a potential palindrome and expand to the left and right while keeping track of the longest one. This removes some of the duplicated checks where a substring is inside another substring (e.g., "AB" is only checked once instead once in "AB", "ABC", "ABCB", "ABCBD", …).

Using this approach, for each item visited two cases are considered: the item i is the middle of an odd palindrome, hence expand the search to i-1 and i+1, then i-2 and i+2, …etc.; and the item i is the right-middle of an even palindrome, hence expand the search to i-1 and i, then i-2 and i+1.

The O(n^2) time complexity can be roughly understood as have a for-loop that visits n-1 items, and for each item at most n items of the entire array is visited twice (odd then even palindrome).

```
char* longest_substring_palindrome(const char* str) {
  int length = strlen(str);
  int longestLength = 1;
  int longestStart = 0; //the index of the longest palindrome
  for (int i=1; i<length; i++) { //visit 1 item at a time
    //the odd palindrome case
    int left = i - 1;
    int right = i + 1;
    while (left >= 0 && right < length && str[left] == str[right]) {
      //does not go beyond boundary and the two ends equal
      if ((right - left + 1) >longestLength) {
        longestLength = right - left + 1; //update the length
        longestStart = left; //update where it starts
      }
      left--;
      right++;
    }
    //the even palindrome case
    left = i -1;
    right = i;
    while (left >= 0 && right < length && str[left] == str[right]) {
      //does not go beyond boundary and the two ends equal
      if ((right - left + 1) >longestLength) {
        longestLength = right - left + 1; //update the length
        longestStart = left; //update where it starts
      }
      left--;
      right++;
    }
```

```
  }
  //create the returned string on the heap
  char* result = (char*)malloc(sizeof(char) * (longestLength + 1));
  strncpy(result, &str[longestStart], longestLength);
  return result;
}
```

Q13: Write a function that gets an array of ints of length n, and outputs the length of the longest contiguous increasing subsequence.

```
int longest_incr_subsequence(const int arr[], int n)
```

For example, on input {1, 2, 6, 4, 3, 8, 9, 10, 2, 4} the output should be 4 because {3, 8, 9, 10} is the longest subsequence.

Solution: The idea of this answer is to keep track of the increasing sequence until the next item drops.

```
int longest_incr_subsequence(const int arr[], int n) {
  int largestLength = 1; //the shortest possible subsequence
  int currentLength = 1;
  int index = 0;
  while (index < n-1) {
    if (arr[index] < arr[index+1]) {
      currentLength++;
    }
    else {
      if (currentLength > largestLength) { //update if needed
        largestLength = currentLength;
      }
      currentLength = 1; //reset and count again
    }
    index++;
  }
  //in case the whole array is increasing
  if (currentLength > largestLength) {
        largestLength = currentLength;
  }
  return largestLength;
}
```

Q14: Write a function that gets a string that contains a number in binary, and returns the number as int. Assume that the number can fit into an int.

```
int binary_to_int(const char* binary)
```

For example, on input "11011" the return is 27.

Solution:

```
//assume there is at least one digit in the array
int binary_to_int(const char* binary) {
  int index = strlen(binary) - 1; //used to access the array
  int powerOfTwo = 1;
  int result = 0;
  while (index >= 0) {
    //use ASCII subtraction to get the actual value 0 or 1
    result += (binary[index] - '0') * powerOfTwo;
    powerOfTwo *= 2;
    index--;
  }
  return result;
}
```

Q15: Write a function that gets a string that contains a number in binary, and returns a string containing the number in decimal. You should not assume that the number can fit into an int or long. You should handle inputs up to length 1,000,000.

```
char* str_binary_to_dec(const char* binary)
```

For example, on input "11011" the return is "27".

Hint: Refer to our Self-Test Week 3. There is a similar question converting binary numbers into hexadecimal numbers.

Q16: We said that int take 4 bytes in the memory. Write a program that checks the order in which the number is stored in the memory. Does it start from most significant bits or from least significant bit? (most significant bit refers to the bit that stores the largest value. E.g., if “10” means 2, it’s the left bit.)

Hint: Use the right bit shift to remove the right most bit of the bit pattern and see the result. For example, if the value is 2,

2 starting from MSB: 00000000000000000000000000000010 >> 1 becomes 00000000000000000000000000000001 (=1)

5 starting from LSB: 01000000000000000000000000000000 >> 1 becomes 00100000000000000000000000000000 (=4)