CS 32 Worksheet 5

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.

Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please go to any of the LA office hours.

Concepts

Recursion

Problems

1. What does the following code output and what does the function LA_power do?

```
#include <iostream>
using namespace std;

int LA_power(int a, int b)
{
   if (b == 0)
       return 0;
   if (b % 2 == 0)
       return LA_power(a+a, b/2);

   return LA_power(a+a, b/2) + a;
}

int main()
{
  cout << LA_power(3, 4) << endl;
}
```

It outputs 12. LA_power returns the result of multiplying its arguments.

2. Given a singly-linked list class LL with a member variable *head* that points to the first *Node* struct in the list, write a function to recursively delete the whole list. Assume each Node object has a next pointer.

```
void LL::deleteList()

void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
     return;

  deleteListHelper(head->next);
  delete head;
  head = nullptr;
}

void LL::deleteList() {
  deleteListHelper(m_head);
}
```

3. Implement the function `getMax` recursively. The function returns the maximum value in `a`, an integer array of size `n`. You may assume that `n` will be at least 1. **(Easy)**

```
int getMax(int a[], int n);

int getMax(int a[], int n) {
      if (n == 1)
            return a[0];
      int x = getMax(a, n-1);
      if (x > a[n-1])
            return x;
      else
            return a[n-1];
}
```

4. Given a string *str*, recursively compute a new string such that all the 'x' chars have been moved to the end.

```
string endX(string str);

Example:
endX("xrxe") → "rexx"
```

```
string endX(string str) {
  if (str.length() <= 1)
    return str;
  if (str[0] == 'x')
    return endX(str.substr(1)) + 'x';
  else
    return str[0] + endX(str.substr(1));
}
```

5. Implement the following function in a recursive fashion:

   ```
   bool isSolvable(int x, int y, int c);
   ```

   This function should return true if there exists nonnegative integers *a* and *b* such that the equation *ax + by = c* holds true. It should return false otherwise.

   Ex: `isSolvable(7, 5, 45) == true //a == 5 and b == 2`
   Ex: `isSolvable(1, 3, 40) == true //a == 40 and b == 0`
   Ex: `isSolvable(9, 23, 112) == false`

   ```
   bool isSolvable(int x, int y, int c) {
     if (c == 0)
       return true;
     if (c < 0)
       return false;

     return isSolvable(x, y, c - x) || isSolvable(x, y, c - y);
   }
   ```

6. A robot you have programmed is attempting to climb a flight of stairs, for which each step has an associated number. This number represents the size of a leap that the robot is allowed to take backwards or forwards from that step (the robot, due to your engineering prowess, has the capability of leaping arbitrarily far). The robot must leap this exact number of steps.

   Unfortunately, some of the steps are traps, and are associated with the number 0; if the robot lands on these steps, it can no longer progress. Instead of directly attempting to reach the end of the stairs, the robot has decided to first determine if the stairs are climbable. It wishes to achieve this with the following function:

   ```
   bool isClimbable(int stairs[], int length);
   ```

This function takes as input an array of int that represents the stairs (the robot starts at position 0), as well as the length of the array. It should return true if a path exists for the robot to reach the end of the stairs, and false otherwise. (Note : the robot doesn't have to only end up at the first position past the end of the array)

Ex: `isClimbable({2, 0, 3, 0, 0}, 5) == true`
`//stairs[0]->stairs[2]->End`
Ex: `isClimbable({1, 2, 4, 1, 0, 0}, 6) == true`
`//stairs[0]->stairs[1]->stairs[3]->stairs[2]->End`
Ex: `isClimbable({4, 0, 0, 1, 2, 1, 1, 0}, 8) == false`

```
bool isClimbableHelper(int stairs[], bool visited[], int
length, int pos) {
  if (pos < 0)
    return false;
  if (pos >= length)
    return true;

  if (stairs[pos] == 0 || visited[pos])
    return false;
  visited[pos] = true;

  return isClimbableHelper(stairs, visited, length, pos -
stairs[pos]) || isClimbableHelper(stairs, visited, length, pos
+ stairs[pos]);
}

bool isClimbable(int stairs[], int length) {
  if (length < 0)
    return false;

  bool* visited = new bool[length];
  for (int x = 0; x < length; x++)
    visited[x] = false;

  bool res = isClimbableHelper(stairs, visited, length, 0);
  delete[] visited;
  return res;
}
```

7. Implement the function `sumOfDigits` recursively. The function returns the sum of all of the digits in the given *positive* integer `num`.

```
int sumOfDigits(int num);

sumOfDigits(176); // return 14
sumOfDigits(111111); // return 6

int sumOfDigits(int num) {
      if (num < 10)
            return num;
      return num % 10 + sumOfDigits(num/10);
}
```

8. Implement the function `isPalindrome` recursively. The function should return whether the given string is a palindrome. A palindrome is described as a word, phrase or sequence of characters that reads the same forward and backwards.

```
bool isPalindrome(string foo);

isPalindrome("kayak"); // true
isPalindrome("stanley yelnats"); // true
isPalindrome("LAs rock"); // false (but the sentiment is true
:))

bool isPalindrome(string foo) {
      int len = foo.length();
      if (len <= 1)
            return true;
      if (foo[0] != foo[len-1])
            return false;
      return isPalindrome(foo.substr(1, len-2));
}
```

9. Write the following linked list functions recursively.

```
// Node definition for singly linked list
struct Node {
      int data;
      Node* next;
};

// inserts a value in a sorted linked list of integers
//         returns list head
```

```
// before: 1 → 3 → 5 → 7 → 15
// insertInOrder(head, 8);
// after: 1 → 3 → 5 → 7 → 8 → 15
Node* insertInOrder(Node* head, int value) {
	if (head == nullptr || value < head->data) {
		Node* p = new Node;
		p->data = value;
		p->next = head;
		head = p;
	} else
		head->next = insertInOrder(head->next, value);
	return head;
}
// deletes all nodes whose keys/data == value, returns list head
Node* deleteAll(Node* head, int value) {
	if (head == nullptr)
		return nullptr;
	else {
		if (head->data == value) {
			Node* temp = head->next;
			delete head;
			return deleteAll(temp, value);
		}
		else {
			head->next = deleteAll(head->next, value);
			return head;
		}
	}
}

// prints the values of a linked list backwards
// e.g. 0 → 2 → 1 → 4 → 1 → 7
// reversePrint(head) will output 714120
void reversePrint(Node* head) {
	if (head == nullptr)
		return;
	reversePrint(head->next);
	cout << head->data;
}
```

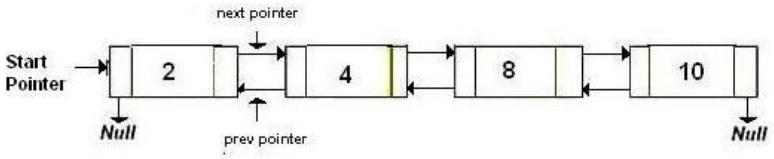
10. Write a recursive function `isPrime` to determine whether a given integer input is a prime number or not.

Example:
isPrime(11) → true
isPrime(4) → false

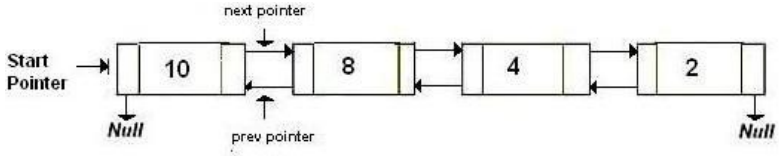
```
// We notice that without a secondary parameter to keep count
      of
// where we are in our check, this problem is impossible
// recursively. Thus the solution can be done with either a
// default parameter or with an auxiliary helper function.

bool isPrime(int num) {
      return isPrimeHelper(num, 2); // start with testing
                                    // divisibility by 2
}
bool isPrimeHelper(int num, int i) {
      if (num <= 2)
            return num == 2;
      if (num % i == 0)  // not prime if divisible by i
            return false;
      if (i*i > num)  // is prime if exhausted all divisors
            return true;
      return isPrimeHelper(num, i + 1); // increment i and see
      if it is divisible by it
}
```

11. Implement `reverse`, a recursive function to reverse a doubly linked list. It returns a pointer to the new head of the list. The integer value in each node must not be changed (but of course the pointers can be).

Example:
Original:



After:



```
// Node definition for doubly linked list
struct Node {
      int val;
      Node* next;
      Node* prev;
};

Node* reverse(Node* head) {
      if (head == nullptr)
            return head;
      // Swap next and prev
      Node* temp = head->next;
      head->next = head->prev;
      head->prev = temp;
      // If previous is null then we are done
      if (head->prev == nullptr)
            return head;
      return reverse(head->prev);
}
```

12. Implement the following recursive function:

```
string longestCommonSubsequence(string s1, string s2);
```

The function should return the longest common subsequence of characters between the two strings s1 and s2. Basically, it should return a maximum length string of characters that are common to both strings and are in the same order in both strings.

Example:

```
string res = longestCommonSubsequence("smallberg",
      "nachenberg");
//res should contain "aberg" as seen in the green chars
res = longestCommonSubsequence("los angeles", "computers");
//res should contain the string "oes"

string longestCommonSubsequence(string s1, string s2) {
    if (s1.empty()  ||  s2.empty()) // base case: either empty
        return "";
    // split the strings into head and tail for simplicity
    char s1_head = s1[0];
    string s1_tail = s1.substr(1);
    char s2_head = s2[0];
```

```
    string s2_tail = s2.substr(1);

    // if heads are equal, use the head and
    // recursively find rest of common subsequence
    if (s1_head == s2_head)
        return s1_head + longestCommonSubsequence(s1_tail,
     s2_tail);
    // heads different, so check for common subsequences not
    // including one of the heads
    string if_behead_s1 = longestCommonSubsequence(s1_tail,
     s2);
    string if_behead_s2 = longestCommonSubsequence(s1,
     s2_tail);

    // return the longer of the subsequences we found
    return if_behead_s1.length() >= if_behead_s2.length() ?
     if_behead_s1 : if_behead_s2;
}
```

13. Implement the recursive function `merge` that merges two sorted linked lists `l1` and `l2` into a single sorted linked list. The lists are singly linked; the last node in a list has a null next pointer. The function should return the head of the merged linked list. No new Nodes should be allocated while merging.

    Example:

    ```
    l1:   1 -> 4 -> 6 -> 8
    l2:   3 -> 9 -> 10
    ```

    After merge: `1 -> 3 -> 4 -> 6 -> 8 -> 9 -> 10`

```
// Node definition for singly linked list
struct Node {
    int val;
    Node* next;
};

Node* merge(Node* l1, Node* l2) {
    // base cases: if a list is empty, return the other list
    if (l1 == nullptr)
        return l2;
    if (l2 == nullptr)
        return l1;

// determine which head should be the head of the merged list
```

```
      // then set head->next to the head returned from recursive
calls
            Node* head;
            if (l1->val < l2->val) {
                  head = l1;
                  head->next = merge(l1->next, l2);
            }
            else {
                  head = l2;
                  head->next = merge(l1, l2->next);
            }

            // return the head of the merged list
            return head;
      }
```

14. Rewrite the following function recursively. You can add new parameters and completely change the function implementation, but you can't use loops.

    This function sums the numbers of an array from left to right until the sum exceeds some threshold. At that point, the function returns the running sum. Returns -1 if the threshold is not exceeded before the end of the array is reached.

```
int sumOverThreshold(int x[], int length, int threshold) {
  int sum = 0;
  for(int i = 0; i < length; i++) {
    sum += x[i];
    if (sum > threshold) {
      return sum;
    }
  }

  return -1;
}
int sumOverThreshold2(int x[], int length, int threshold){
      return sumOverThreshold2Helper(x, length, threshold, 0);
}
int sumOverThreshold2Helper(int x[], int length, int threshold,
                            int sum)
{
  if (sum > threshold) {
    return sum;
```

```
  }
  if (length == 0) {
    return -1;
  }
  return sumOverThreshold2Helper(x+1, length-1, threshold,
                                 sum+x[0]);
}
/******************
OR
******************/
int sumOverThreshold3(int x[], int length, int threshold) {
  if(threshold < 0){
    return 0;
  } else if (length == 0) {
    return -1;
  }

  int returnOfRest = sumOverThreshold3(x+1, length-1,
     threshold-x[0]);
  if (returnOfRest == -1){
    return -1;
  } else {
    return x[0] + returnOfRest;
  }
}
```

15. Given the following program, give the output of each function call for parts a, b, and, c.

```
void fizzbuzz(int x){
  if (x == 0) {
    cout << "fizzbuzz" << endl;
    return;
  }

  cout << "fizz" << endl;
  fizzbuzz(x-1);
  fizzbuzz(x-1);
}
```

   a. fizzbuzz(1);

fizz
fizzbuzz
fizzbuzz

b. fizzbuzz(2);
fizz
fizz
fizzbuzz
fizzbuzz
fizz
fizzbuzz
fizzbuzz

c. fizzbuzz(3);
fizz
fizz
fizz
fizzbuzz
fizzbuzz
fizz
fizzbuzz
fizzbuzz
fizz
fizz
fizzbuzz
fizzbuzz
fizz
fizzbuzz
fizzbuzz