

# CS 32 Solutions Week 6

**Concepts:** Recursion, Templates, STL

## Recursion Problems

1. (10 min) 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

Solution:

```
int getMax(int a[], int n) {  
    if (n == 1)  
        return a[0];           // base case: array is 1 element  
    int x = getMax(a, n-1);     // reduce: getMax of n-1 elements  
    if (x > a[n-1])             // compare max to last element  
        return x;  
    else  
        return a[n-1];  
}
```

2. (15 min) 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 elements of an array of nonnegative numbers 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;
}
```

Solution:

```
int sumOverThreshold2(int x[], int length, int threshold){
    return sumOverThreshold2Helper(x, length, threshold, 0);
}

// sum argument holds the running sum so far before looking at
// current element
int sumOverThreshold2Helper(int x[], int length, int threshold,
int sum)
{
    if (sum > threshold) {
        return sum;
    }
    if (length == 0) { // base case: end of array reached
        return -1;
    }
    return sumOverThreshold2Helper(x+1, length-1, threshold,
sum+x[0]);
} // reduce by adding first element of array to sum
```

```
/******
```

```
OR
```

```
*****//
```

```
int sumOverThreshold3(int x[], int length, int threshold) {  
    if(threshold < 0){  
        return 0;  
    } else if (length == 0) { // base case: end of array reached  
        return -1;  
    }  
  
    int returnOfRest = sumOverThreshold3(x+1, length-1, threshold-x[0]);  
    if (returnOfRest == -1){  
        return -1;  
    } else {  
        return x[0] + returnOfRest;  
    }  
}
```

```
// reduce: find sum over new threshold t - x[0] in x after x[0]
```

3. (15 min) Given a string *str*, recursively compute a new string such that all the 'x' chars have been moved to the end.

Hint: <https://www.cplusplus.com/reference/string/string/substr/>

```
string endX(string str);
```

Example:

```
endX("xrxe") → "rexx"
```

Solution:

```
string endX(string str) {  
    if (str.length() <= 1)    // base case: no x's to shift to end  
        return str;  
    if (str[0] == 'x')  
        return endX(str.substr(1)) + 'x'; // reduce: handle str[0]  
    // first  
    else  
        return str[0] + endX(str.substr(1));  
}
```

4. (15 min) 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.

Examples:

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

Solution:

```
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);
}

// reduce: slowly take out x to check (a-1)x + by = c - x
// or slowly take out y to check ax + (b-1)y = c - y
```

## Template/STL Problems

5. (5 min) The following code has 3 errors that cause either runtime or compile time errors. Find and fix all of the errors.

```
class Potato {
public:
    Potato(int in_size) : size(in_size) { };
    int getSize() const {
        return size;
    };
private:
    int size;
};

int main() {
    vector<Potato> potatoes;
    Potato p1(3);
    potatoes.push_back(p1);
    potatoes.push_back(Potato(4));
    potatoes.push_back(Potato(5));

    vector<int Potato>::iterator it = potatoes.begin();
    while (it != potatoes.end()) {
        potatoes.erase(it);
        it++;
    }

    for (it = potatoes.begin(); it != potatoes.end(); it++) {
        cout << it.getSize() << endl;
    }
}
```

Solution:

```
class Potato {
public:
    Potato(int in_size) : size(in_size) { };
    int getSize() const {
        return size;
    };
};
```

```

private:
    int size;
};

int main() {
    vector<Potato> potatoes;
    Potato p1(3);
    potatoes.push_back(p1);
    potatoes.push_back(Potato(4));
    potatoes.push_back(Potato(5));

    vector<int Potato>::iterator it = potatoes.begin(); // 1
    while (it != potatoes.end()) {
        it = potatoes.erase(it); // 2
    }

    for (it = potatoes.begin(); it != potatoes.end(); it++) {
        cout << it->getSize() << endl; // 3
    }
}

```

1: potatoes.begin() gives iterator of potatoes, which is a vector<Potato>, so the iterator given will be of the type vector<Potato>::iterator

2: After calling erase with the iterator it, it is invalidated. Instead of incrementing it, the return value of potatoes.erase(it) should be assigned to it. The erase method returns the iterator of the element that is after the erased element.

3: Iterators use pointer syntax, so the last for loop should use it->getSize() instead of it.getSize().

6. (15 min) Implement a stack class *Stack* that can be used with any data type using templates. Use a linked list (not an STL `list`) to store the stack and implement the functions *push()*, *pop()*, *top()*, *isEmpty()*, a default constructor, and a destructor that deletes the linked list nodes.

Solution:

```
template<typename Item>
class Stack {
public:
    Stack() : m_head(nullptr) {}

    bool isEmpty() const {
        return m_head == nullptr;
    }

    Item top() const {
        // We'll return a default-valued Item if the Stack is empty,
        // because you should always check if it's empty before
        // calling top().
        if (m_head != nullptr)
            return m_head->val;
        else
            return Item();
    }

    void push(Item item) {
        Node* new_node = new Node;
        new_node->val = item;
        new_node->next = m_head;
        m_head = new_node;
    }

    void pop() {
        // We'll simply do nothing if the Stack is already empty,
        // because you should always check if it's empty before
        // popping.
        if (m_head == nullptr) {
            return;
        }
    }
};
```



```
    Node* temp = m_head;
    m_head = m_head->next;
    delete temp;
}

~Stack() {
    while (m_head != nullptr) {
        Node* temp = m_head;
        m_head = m_head->next;
        delete temp;
    }
}

private:
    struct Node {
        Item val;
        Node* next;
    };
    Node* m_head;
};
```

7. (15 min) Implement a vector class `Vector` that can be used with any data type using templates. Use a dynamically allocated array to store the data. Implement only the `push_back()` function, default constructor, and destructor.

Solution:

```
template <typename T>
class Vector {
public:
    Vector();
    ~Vector();
    void push_back(const T& item);
private:
    // Total capacity of the vector -- doubles each time
    int m_capacity;
    // The number of elements in the array
    int m_size;
    // Underlying dynamic array
    T* m_buffer;
};

template <typename T>
Vector<T>::Vector()
: m_capacity(0), m_size(0), m_buffer(nullptr)
{}

template <typename T>
Vector<T>::~~Vector() {
    delete[] m_buffer;
}

template <typename T>
void Vector<T>::push_back(const T& item) {
    // if space is full, allocate more capacity
    if (m_size == m_capacity)
    {
        // double capacity(doesn't have to be doubled, but recommended);
        //special case for capacity 0
        if (m_capacity == 0)
            m_capacity = 1;
        else
```

```
    m_capacity *= 2;

    // allocate an array of the new capacity
    T* newBuffer = new T[m_capacity];

    // copy old items into new array
    for(int i = 0; i < m_size; i++)
        newBuffer[i] = m_buffer[i];

    // delete original array (harmless if m_buffer is null)
    delete [] m_buffer;

    // install new array
    m_buffer = newBuffer;
}

// add item to the array, update m_size
m_buffer[m_size] = item;}
m_size++;
}
```

## Extra Practice

### Recursion:

1. (10 min) Implement the function `sumOfDigits` recursively. The function returns the sum of all of the digits in the given *positive* integer `num`.

*Hint: Use integer division*

```
int sumOfDigits(int num);  
  
sumOfDigits(176); // return 14  
sumOfDigits(111111); // return 6
```

Solution:

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

2. (15 min) 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);

// deletes all nodes whose keys/data == value, returns list head
// use the delete keyword
Node* deleteAll(Node* head, int value);

// 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);
```

Solution:

```
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;
}

```

## Template/STL:

1. (5 min) Will this code compile? If so, what is the output? If not, what is preventing it from compiling?

*Note: We did not use namespace std because std has its own implementation of max and namespace std will thus confuse the compiler.*


```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

int main() {
    std::cout << max(3, 7) << std::endl;    // line 1
    std::cout << max(3.0, 7.0) << std::endl; // line 2
    std::cout << max(3, 7.0) << std::endl;  // line 3
}
```

Solution:

On Xcode, it gives the following error messages:

```
int main()
{
    std::cout << max(3, 7) << std::endl;
    std::cout << max(3.0, 7.0) << std::endl;
    std::cout << max(3, 7.0) << std::endl;
    return 0;
}
```

 No matching function for call to 'max'

For max, the compiler expects two arguments that are of the same type, as indicated in the template declaration T. In the third call, 3 is an integer and 7.0 is a double, so there is no matching function call for this instance.

If we were to remove line 3, lines 1 and 2 would both output 7.

2. (20 min) You are given an STL `set<list<int>*>`. In other words, you have a set of pointers, and each pointer points to a list of ints. Consider the sum of a list to be the result of adding up all elements in the list. If a list is empty, treat its sum as zero. Write a function that removes the lists with odd sums from the set. The lists with odd sums should be deleted from memory and their pointers should be removed from the set. This function should return the number of lists that are removed. You may assume that none of the pointers is null.

Solution:

```
int deleteOddSumLists(set<list<int>*>& s) {
    int numDeleted = 0;

    // iterate over the set
    set<list<int>*>::iterator set_it = s.begin();
    while (set_it != s.end())
    {
        // iterate over each list and get the sum
        int sum = 0;
        list<int>::iterator list_it = (*set_it)->begin();
        list<int>::iterator list_end = (*set_it)->end();
        while (list_it != list_end) {
            sum += *list_it;
            list_it++;
        }

        // delete list and remove from set if sum is odd
        // otherwise, proceed to check the next list
        if (sum % 2 == 1) {
            delete *set_it;
            set_it = s.erase(set_it);
            numDeleted++;
        }
        else set_it++;
    }
    return numDeleted;
}

// Sample driver code:
int main()
{
```



```
set<list<int>*> s;  
list<int>* l1 = new list<int>;  
l1->push_back(1);  
l1->push_back(2);  
list<int>* l2 = new list<int>;  
l2->push_back(1);  
l2->push_back(1);  
list<int>* l3 = new list<int>;  
l3->push_back(1);  
l3->push_back(0);  
s.insert(l1);  
s.insert(l2);  
s.insert(l3);  
cout << deleteOddSumLists(s) << endl;  
}
```