Practice Problems for Exam 3

We've compiled some exercises before the third exam.

Don't expect this to be reflective of the exam.

---

1. Given a nearly sorted vector of integers, would you prefer to use quicksort or insertion sort? Explain why.
2. In the worst-case, what is the time complexity of quicksort? Intuitively, can you explain how this happens?
3. What is a drawback, in terms of space efficiency, of merge sort? What advantage does merge sort have, in terms of time efficiency, over quicksort?
4. What is the primary assumption a developer has to make before using the binary search algorithm?
5. Given the following piece of code, determine the output:

C++

```cpp
class Base {
public:
    Base() {}

    virtual void f() {
        std::cout << "BASE ";
    }
};

class Derived: public Base {
public:
    Derived(): Base() {}

    virtual void f() {
        std::cout << "DERIVED ";
    }
};

int main() {
    std::vector<Base> base_objs;
    Base obj1 = Derived();
    Derived obj2 = Derived();
    Base obj3 = Base();

    base_objs.push_back(obj1);
    base_objs.push_back(obj2);
    base_objs.push_back(obj3);

    for (Base base_obj : base_objs) {
        base_obj.f();
    }

    obj1.f();
    obj2.f();
    obj3.f();

    return 0;
}
```

6. What makes a class abstract? Is it possible to create an instance of an abstract class? Implement an abstract class `Person` with a pure virtual function `move`.

7. What is the notion of encapsulation in terms of object-oriented programming?

8. Evaluate the following piece of code:

```cpp
class A {
private:
    int x;
    int y;
public:
    A(): x(0), y(0) {
        std::cout << "Default constructor" << std::endl;
    }

    A(int x, int y): x(x), y(y) {
        std::cout << "Constructor with two values" << std::endl;
    }

    A(const A& other) {
        std::cout << "Copy Constructor" << std::endl;
        x = other.x;
        y = other.y;
    }
};

int main() {
    A a(1, 2);
    A b = a;

    return 0;
}
```

9. For this question, we'll do some OOP and inheritance. You are tasked with writing an abstract class `Sorting` with the pure virtual functions `sort(std::vector<int>)` and `worst_case_runtime()`. This class is then subclassed into three classes which you are responsible for: `InsertionSort`, `QuickSort`, and `MergeSort`. Within these classes, you are to implement a function `sort` which sorts a vector of integers according to the type of sorting algorithm you have. (You are free to implement this however you wish.) `worst_case_runtime()` would then return a `std::string` containing the worst case runtime of the algorithm.

10. Given the struct, implement the following functions.

```cpp
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int val): val(val), next(nullptr) {}
};

ListNode* reverse_linked_list(ListNode* head) {
    // Reverse the linked list and return the new head after reversing
}

bool is_target_in_list(ListNode* head, int target) {
    // check if the linked list contains the target
}

int find_max(ListNode* head) {
    // find the largest value in the node
}
```

11. **(Hard)** Big Integer Addition: Suppose that we have two text files, `num1.txt` and `num2.txt`. For simplicity, suppose that each file contains `n` lines of integers of length `m`.

    For example, `num1.txt` could look like this

    ```
    12833648473834937493948
    12838374920495869584747
    19238473749395849584847
    19238374839584954023421
    09384738263473920398429
    ```

    This file would constitute one large number. The case will be similar for `num2.txt`.

    Your task is to read the numbers into an appropriate data structure and allow for the addition of these two big integers and output the result to a text file. (Do this problem if you have time.)

12. Implement a `Stack` class using queues.

13. Implement a `MinStack`. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

    - push(x) -- Push element x onto stack.
    - pop() -- Removes the element on top of the stack.
    - top() -- Get the top element.

- getMin() -- Retrieve the minimum element in the stack.

C++

```
class MinStack {
private:

public:
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        // Check if the stack is empty or not

    }

    void pop() {

    }

    int top() {

    }

    int getMin() {

    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

14. Given the struct, implement the following functions:

```cpp
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int val): val(val), left(nullptr), right(nullptr) {}
};

// Given two binary trees, write a function to check if they are the same or not.
// Two binary trees are considered the same if they are structurally identical
// and the nodes have the same value.
bool isSameTree(TreeNode* p, TreeNode* q) {
    // TODO
}

// Given a binary tree, check whether it is a mirror of itself
// (ie, symmetric around its center).
bool isSymmetric(TreeNode* root) {
    // TODO
}

// Given a binary tree, implement level order traversal
void levelOrder(TreeNode* root) {
    // TODO
}
```