# 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.

   Insertion sort. Insertion sort is faster for a list that is nearly sorted. The problem goes from $O(n^2)$ to around $O(n)$ given this condition.

2. In the worst-case, what is the time complexity of quicksort? Intuitively, can you explain how this happens?

   Quicksort is $O(n^2)$. Suppose that we have a list that is nearly sorted. If our pivot chosen is either the first or last element of the list, we'd have $n$ groups that each need to be iterated through $n$ times.

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?

   You need an auxiliary array for the merge step. Merge sort will always be $O(nlogn)$

4. What is the primary assumption a developer has to make before using the binary search algorithm?

   Data has to be sorted.

5. Given the following piece of code, determine the output:

```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;
}
```

BASE BASE BASE BASE DERIVED BASE

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`.

A pure virtual function makes a class abstract. You cannot create an instance of an abstract class.

```cpp
class Person {
public:
    std::string get_name() = 0;
};
```

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

   Encapsulation is the idea of hiding data and functions within a single unit. In C++, we can do this via classes.

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;
}
```

OUTPUT:

Constructor with two values

Copy Constructor

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.

```C++
#include <iostream>
#include <vector>
#include <string>

class Sorting {
public:
    virtual std::string worst_case_runtime() = 0;
    virtual void sort(std::vector<int>& vec) = 0;
};

class InsertionSort: public Sorting {
public:
    std::string worst_case_runtime() override {
        return "O(n^2)";
    }

    void sort(std::vector<int>& vec) override {
        // implement this however you want
        int key, j;
        int n = vec.size();

        for (int i = 1; i < n; i++) {
            key = vec[i];
            j = i - 1;

            while (j >= 0 && vec[j] > key) {
                vec[j + 1] = vec[j];
                j--;
            }

            vec[j + 1] = key;
        }
```

```
    }
};

class QuickSort: public Sorting {
public:
    std::string worst_case_runtime() override {
        return "O(n^2)";
    }

    void sort(std::vector<int>& vec) override {
        quickSort(vec, 0, vec.size()-1);
    }

    void quickSort(std::vector<int>& vec, int left, int right) {
        if (left < right) {
            int pivot = partition(vec, left, right);
            quickSort(vec, left, pivot - 1);
            quickSort(vec, pivot + 1, right);
        }
    }

    int partition(std::vector<int>& vec, int left, int right) {
        int pivot = vec[left];
        int from_left = left + 1;
        int from_right = right - 1;
        int tmp;

        while (from_left != from_right) {
            if (vec[from_left] <= pivot) {
                from_left++;
            } else {
                while ((from_left != from_right) && (pivot <= vec[from_right])) {
                    from_right--;
                }
                std::swap(vec[from_right], vec[from_left]);
            }
        }

        if (vec[from_left] > pivot) {
            from_left--;
        }
        vec[left] = vec[from_left];
        vec[from_left] = pivot;

        return from_left;
```

```
    }
};

class MergeSort: public Sorting {
public:
    std::string worst_case_runtime() override {
        return "O(n log n)";
    }

    void sort(std::vector<int>& vec) override {
        mergeSortHelper(vec, 0, vec.size()-1);
    }

    void mergeSortHelper(std::vector<int>& vec, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            mergeSortHelper(vec, left, mid);
            mergeSortHelper(vec, mid + 1, right);

            merge(vec, left, mid, right);
        }
    }

    void merge(std::vector<int>& vec, int left, int mid, int right) {
        int i, j, k;
        std::vector<int> L(mid - left + 1, 0);
        std::vector<int> R(right - mid, 0);

        for (i = 0; i < mid - left + 1; i++) {
            L[i] = vec[left + i];
        }

        for (j = 0; j < right - mid; j++) {
            R[j] = vec[mid + left + j];
        }

        i = 0;
        j = 0;
        k = left;

        while (i < mid - left + 1 && j < right - mid) {
            if (L[i] <= R[j]) {
                vec[k] = L[i];
                i++;
```

```
            } else {
                vec[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < mid - left + 1) {
            vec[k] = L[i];
            i++;
            k++;
        }

        while (j < right - mid) {
            vec[k] = R[j];
            j++;
            k++;
        }
    }
}
```

10. Given the struct, implement the following functions.

C++

```
#include <algorithm>

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
  ListNode* prev = nullptr;
  ListNode* cur = head;

  while (cur->next != nullptr) {
    ListNode* next = cur->next;
    cur->next = prev;
    prev = cur;
    cur = next;
  }

  return prev;
```

```
}

bool is_target_in_list(ListNode* head, int target){
  // check if the linked list contains the target
  while (head != nullptr) {
    if (head->val == target) {
      return true;
    }
  }

  return false;
}

int find_max(ListNode* head){
  // find the largest value in the node
  int max = head->val;

  while (head != nullptr) {
    max = std::max(max, head->val);
    head = head->next;
  }

  return max;
}
```

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 3
    12838374920495869584747 4
    19238473749395849584847 3
    19238374839584954023421 0
    09384738263473920398429 8
    ```

    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.)

    C++

    ```
    #include <iostream>
    #include <fstream>
    ```

```cpp
#include <list>

std::list<int> get_integer(const std::string& filename) {
    std::list<int> big_int;
    std::ifstream input(filename);
    std::string line;

    if (input.is_open()) {
        while (getline(input, line)) {
            for (auto num: line) {
                int n = num - '0';
                big_int.push_front(n);
            }
        }
    } else {
        std::cout << "Could not open file." << std::endl;
    }

    return big_int;
}

std::list<int> add_integers(const std::list<int>& l1, const std::list<int>& l2) {
    std::list<int> result;

    // our assumption is that both lists are of the same size
    int carry = 0;
    for (auto itr1 = l1.begin(), itr2 = l2.begin();
        itr1 != l1.end() && itr2 != l2.end(); itr1++, itr2++) {
        int total_val = *itr1 + *itr2 + carry;
        int insert_val = total_val % 10;
        carry = total_val / 10;
        result.push_back(insert_val);
    }

    if (carry > 0) {
        result.push_back(carry);
    }

    return result;
}

int main() {
    auto big_num1 = get_integer("/filepath/to/num1.txt");
    auto big_num2 = get_integer("/filepath/to/num2.txt");
```

```
    auto result = add_integers(big_num1, big_num2);

    for (auto n : result) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

12. Implement a `Stack` class using queues.

C++

```
#include <queue>
#include <iostream>

using namespace std;

class Stack
{
    queue<int> q1, q2;
    int curr_size;

public:
    Stack()
    {
        curr_size = 0;
    }

    void pop()
    {
        if (q1.empty())
            return;

        // Leave one element in q1 and
        // push others in q2.
        while (q1.size() != 1)
        {
            q2.push(q1.front());
            q1.pop();
        }

        // Pop the only left element
        // from q1
        q1.pop();
```

```
        curr_size--;

        // swap the names of two queues
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }

    void push(int x)
    {
        q1.push(x);
        curr_size++;
    }

    int top()
    {
        if (q1.empty())
            return -1;

        while( q1.size() != 1 )
        {
            q2.push(q1.front());
            q1.pop();
        }

        // last pushed element
        int temp = q1.front();

        // to empty the auxiliary queue after
        // last operation
        q1.pop();

        // push last element to q2
        q2.push(temp);

        // swap the two queues names
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
        return temp;
    }

    int size()
    {
        return curr_size;
```

```
    }
};

// Driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    cout << "current size: " << s.size()
         << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    cout << "current size: " << s.size()
         << endl;
    return 0;
}
```

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++

```
#include <stack>

using namespace std;

class MinStack {
private:
    stack<int> s;
    stack<int> min;
public:
    /** initialize your data structure here. */
    MinStack() {
        return;
    }

    void push(int x) {
        if(s.empty() || x <= min.top()){
            min.push(x);
        }
        s.push(x);
    }

    void pop() {
        if(s.empty()){return;}
        if(s.top() == min.top()){
            min.pop();
        }
        s.pop();
    }

    int top() {
        return s.top();
    }

    int getMin() {
        if(s.empty()){return -1;}
        return min.top();
    }
};
```

14. Given the struct, implement the following functions:

C++

```
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
    if (p == nullptr || q == nullptr) return (p == q);
    return (p->val == q->val && isSameTree(p->left, q->left) &&
        isSameTree(p->right, q->right));
}

// Given a binary tree, check whether it is a mirror of itself
// (ie, symmetric around its center).
bool isSymmetricHelp(TreeNode* left, TreeNode* right){
    if (left == nullptr || right == nullptr)
        return left == right;
    if (left->val != right->val)
        return false;
    return isSymmetricHelp(left->left, right->right)
        && isSymmetricHelp(left->right, right->left);
}
bool isSymmetric(TreeNode* root) {
    // TODO
    return root == nullptr || isSymmetricHelp(root->left, root->right);
}

// Given a binary tree, implement level order traversal
// you can also change this to void by just logging out the numbers as you see them
vector<vector<int>> levelOrder(TreeNode* root) {
// TODO
    vector<vector<int>> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
    q.push(nullptr);
    vector<int> cur_vec;
    while(!q.empty()) {
        TreeNode* t = q.front();
        q.pop();
        if (t== nullptr) {
            result.push_back(cur_vec);
            cur_vec.resize(0);
```

```
            if (q.size() > 0) {
                q.push(nullptr);
            }
        } else {
            cur_vec.push_back(t->val);
            if (t->left) q.push(t->left);
            if (t->right) q.push(t->right);
        }
    }
    return result;
}
```