CSC 212 Practice Midterm Exam 1B
Problems marked with (*) are challenging and problems marked with (**) are hard

Your Name: ______________________________

1. (10 points) (*) Write a formula $T(n)$ that counts the number of multiplications performed by the following function `bar` on an input of size $n \geq 1$. You do not need to find a closed form for $T(n)$.

```
int foo(int n) {
    int result = 0;
    for (int i = 1; i <= n; i *= 2)
        result += i;
    return result;
}
```

Solution: $T(n) = \lfloor \log_2 n \rfloor + 1$

2. (10 points) Find a closed form for $\sum_{i=2}^{n} i$.

Solution: $\frac{n(n+1)}{2} - 1$

3. (10 points) Rate the growth rate of the following functions from greatest to least:

$$3n \lg n \qquad 2^{n-5} \qquad 3^{100} \qquad 8n^2 + 18n$$

Solution: $2^{n-5}, 8n^2 + 18n, 3n \lg n, 3^{100}$.

4. (10 points) Suppose `v` is a grow-by-one dynamic array with size 0 and capacity 1. Give a Θ-bound on the time complexity of calling `push_back` n times.

Solution: $\Theta(n^2)$. We must copy one element, then two, then three, and so on, up to n elements. Summing up, we find that the total number of copies is $\Theta(n^2)$.

5. (10 points) (**) Suppose `v` is a grow-by-factor dynamic array containing n elements. If `push_back` is called and there is no room for new elements, `v` will increase the capacity by 1%. When resizing, the capacity is always increased by at least one. Give a Θ-bound on the time complexity of calling `push_back`. Indicate if your bound is amortized.

Solution: $\Theta(1)$ amortized. The analysis is the same as grow-by-doubling. Any grow-by-factor dynamic array with a constant growth factor (not dependent on n) enables `push_back` in $\Theta(1)$ amortized time.

6. (10 points) What is the output of the following program?

```
stack<int> s;
s.push(0);
s.push(1);
s.push(2);
s.pop();
cout << s.top() << ' ';
s.pop();
cout << s.top();
```

Solution: 1 0

7. (10 points) (*) Give a Θ-bound on the time complexity of the following program. Justify your answer.

```
int qux(const vector<int>& v) {
    queue<int> q;
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        if (v[i] < 0)
            while (!q.empty())
                q.pop();
        q.push(i);
        result = max(result, q.size());
    }
    return result;
}
```

Solution: $\Theta(n)$. There are n possible values of `i` and each is pushed to the queue once, and so can be popped at most once. Hence, although the `while` loop is nested, it runs at most a total of n times.

8. (10 points) What is the output of the following program?

```
priority_queue<int> q; // max-priority queue
q.push(1);
q.push(2);
q.push(0);
q.pop();
cout << q.front() << ' ';
q.pop();
cout << q.front();
```

Solution: 1 0

9. (10 points) What are the contents of v after this program executes?

```
vector<int> v{3, 1, 2, 0, 4};
make_heap(v.begin(), v.end()); // max-heap
```

Solution: 4, 3, 1, 0, 2

10. (10 points) (*) You are an operating systems engineer designing a file management system. Users can navigate to directories by specifying paths, which are strings that describe the sequence of directories to reach a file or folder. Before accessing a path, the system must simplify it to its canonical form.

    In a path,

    - `/` separates directories.
    - `.` represents the current directory.
    - `..` represents moving up one directory.

    The **top-level directory** is `/`. Moving up does not change this directory.

    The **canonical form** of a path is an equivalent path without any `.` and `..` components. For example:

    - `/home/usr/../share/./bin` has canonical form `/home/share/bin`
    - `/home/share/../../usr` has canonical form `/home/`
    - `/../` has canonical form `/`

    What abstract data type is best for converting paths to canonical paths? Here, best means efficiently solves the problem. Justify your answer.

    Solution: Stack or deque. Consider parsing the path directory by directory. If we encounter `..`, we pop the stack if it is not empty. If we encounter `.`, we do nothing. If we encounter a directory, we push it to the stack. After the input is parsed, we reverse the directories on the stack, and then merge them into a single string. This takes $\Theta(n)$ time.