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

Your Name: _____

1. (10 points) Write a function $T(n)$ that counts the number of multiplications performed by the following function `foo` 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 = 1;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= i; ++j)
            result = result * i * j;
    return result;
}
```

Solution: $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 2$

2. (10 points) Which of the following asymptotic descriptions of $n^2 + 3n \lg n$ are correct?

- √ $\mathcal{O}(n^2)$
- ◯ $\mathcal{O}(n \lg n)$
- ◯ $\mathcal{O}(n)$
- √ $\Omega(n^2)$
- √ $\Omega(n \lg n)$
- √ $\Omega(n)$
- √ $\Theta(n^2)$
- ◯ $\Theta(n \lg n)$
- ◯ $\Theta(n)$

3. (10 points) What is the growth rate of the following function?

$$2n + 3 \lg n + 5$$

- ◯ Constant
- ◯ Logarithmic
- √ **Linear**
- ◯ Linearithmic
- ◯ Quadratic
- ◯ Cubic
- ◯ Exponential

4. (10 points) (*) Suppose algorithm A always runs in $\Theta(n)$ time and algorithm B always runs in $\Theta(\lg n)$ time. Which algorithm should you prefer for small values of n? Which algorithm should you prefer for large values of n? Justify your answer.

Solution: Asymptotic notation only applies for large values of n. For small values of n, there isn't enough information to determine which algorithm is faster. For large values n, prefer B since $\Theta(\lg n)$ grows more slowly than $\Theta(n)$.

5. (10 points) Suppose `v` is a grow-by-doubling dynamic array with size 0 and capacity 1. What is the capacity of v after 5 calls to `push_back`?

Solution: 8. The capacity is doubled to 2, then 4, then 8.

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

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

Solution: 1 2

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

```
int baz(const vector<int>& v) {
    stack<int> s;
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        while (!s.empty() && v[i] >= v[s.top()])
            s.pop();
        int x = i + 1;
        if (!s.empty())
            x = i - s.top();
        result = max(result, x);
        s.push(i);
    }
    return result;
}
```

Solution: $\Theta(n)$. Since stack operations take $\Theta(1)$ time, its clear (with the exception of the nested `while` loop) that the `for` loop takes $\Theta(n)$ time. In fact, the nested `while` loop runs at most n total times across all iterations of the `for` loop. There are n possible values of `i`, and each value of `i` is pushed to the stack exactly once, and so can be popped at most once. Therefore, the stack is popped at most n times, and so the `while` loop runs at most n times despite being nested.

8. (10 points) What are the contents of `v` after this program executes?

```
vector<int> v{0, 1, 4, 2, 3}; // min-heap
pop_heap(v.begin(), v.end());
v.pop_back();
v.push_back(1)
push_heap(v.begin(), v.end())
```

Solution: 1, 1, 4, 3, 2

9. (10 points) (**) Give an input of size n such that `heapsort` sorts it in linear time. Justify your answer.

Solution: There are many valid approaches. Consider an array of n zeroes as input. First, `heapsort` runs `make_heap` which always takes $\Theta(n)$ time. Then, `heapsort` calls `pop_heap` n times. For each call to `pop_heap`, only one swap is performed between the first element and the last. The new root, which is zero, is never swapped downwards since it is greater than or equal to its children (also zero). Thus, the calls to `pop_heap` take $\Theta(n)$ time. Hence, `heapsort` will sort the array in linear time.

10. (10 points) (*) You are a software engineer working on a cloud computing platform. Users submit computational jobs to the system, and the system executes them on a server. A **job** is a unit of work, like running a simulation or processing a dataset. Jobs arrive in the order that customers submit them. Some jobs are marked as **urgent**, and should be executed as soon as possible.

    For example, suppose jobs are represented by letters, and the system has two servers:

    1. Job A (not urgent) arrives, Job B (not urgent) and Job C (not urgent) arrive.
        - Server 1 executes A.
        - Server 2 executes B.
    2. Job D (urgent), Job E (not urgent) and Job F (urgent) arrive.
        - Server 1 executes D.
        - Server 2 executes F.
    3. No new jobs arrive.
        - Server 1 executes C.
        - Server 2 executes E.

    What abstract data type best models the system's waiting area for jobs before they are sent to a server? Here, best means efficiently solves the problem. Justify your answer.

    Solution: There are two valid approaches:

    1. Priority queue. Assign urgent jobs a higher priority than not urgent jobs. Always take the highest priority job as the next job. This takes $\mathcal{O}(n \lg n)$ time.
    2. Queue or deque. Maintain two queues, one of urgent jobs and one of not urgent jobs. If there is an urgent job available, always take it. Otherwise, take a not urgent job. This takes $\mathcal{O}(n)$ time.