CME213, Spring 2022
Parallel Computing
Julian Cooper

Homework 1
Due Friday, April 8th via GradeScope

Question 0: Our graph is undirected, but we could use it to build a *directed* graph or *digraph*. A digraph's edges are *ordered* pairs of nodes (n_i, n_j), rather than unordered pairs $\{n_i, n_j\}$; all operations look like their undirected counterparts and have similar complexity.

Here's one digraph representation using our graph class.

```
template <typename V>
class Digraph {
...
private:
Graph<V> g_;
};
```

Explain the representation by giving an abstraction function and representation invariant. Note that `Digraph` is not `Graph`'s friend and can only use `Graph`'s public interface.

Idea: Want to represent each node of the directed graph with two nodes of the undirected graph , i.e. an output (even idx) and input (odd idx).

$AF(\text{Digraph}) = (N,\, E)$ where

- $N = \{\{n_{2i}, n_{2i+1}\} \mid 0 <= i < \text{g\_.num\_nodes()}\}$
- $E = \{\{n_i, n_j\} \mid \text{j - i \% 2 == 1, i} \in \text{adj\_[j]}\}$

$RI(\text{Digraph})$:

- Size of nodes set N = 2 * g_.num_nodes()
- Size of edges set E = g_.num_edges()

Question 1: The `vector<T>::resize` operation extends a vector with *memory initialized* elements. (For example, after `vector<int> v; v.resize(20)`, `v` contains 20 zeroes in memory.) This is almost always what we want, but it can be useful to extend a vector with *uninitialized memory*. Here's an example: uninitialized memory helps build a *sparse vector*, in which elements are initialized only when first referenced. We assume a special `garbage_vector` type whose `resize` method does not initialize new memory. **See next page**
Note: This question will not completely make sense until you understand what the code is doing below!

```
template <typename T>
class sparse_vector {
garbage_vector<size_t> position_;
vector<size_t> check_;
vector<T> value_;
public:
// Construct a sparse vector with all elements equal to T().
sparse_vector() {}
// Return a reference to the element at position @a i.
T& operator[](size_t i) {
    // If out of bounds, we must re-size our array and add
    // (uninitialized) memory!
    if (i >= position_.size())
    position_.resize(i + 1);
    // If we haven't initialized the memory yet, go ahead and do so now.
    if (position_[i] >= check_.size() || check_[position_[i]] != i) {
    position_[i] = check_.size(); // First reference to the element ati.
    check_.push_back(i);          // We associate an index...
    value_.push_back(T());        // ...alongside a value.
    }
    return value_[position_[i]];
}
};
```

An abstract `sparse_vector` value is an **infinite** vector.

Write an abstraction function and representation invariant for `sparse_vector`.

Idea: Want to describe the functionality and representation invariants of our infinite vector. In particular, how does the implementation above guarantee operator [] works for all positive integer inputs i without initializing memory address of locations for each resize.

AF(sparse_vector) = an infinite vector V such that $V[i]$ returns an integer for all $i \in$ positive integers

RI(sparse_vector):

- check_.size() == value_.size()
- *max_element(check_.begin(), check_.end()) + 1 = position_.size()