Practice test for Midterm 1

March 16, 2026

1 C++ review

► Write a function which takes a vector of int and returns true if all the elements of the vector are positive (≥ 0) and false otherwise:

```
bool all_pos(vector<int> data)
{
```

Solution:

```
bool all_pos(vector<int> data)
{
    for(int i = 0; i < data.size(); ++i)
        if(data[i] < 0)
            return false;

    return true;
}
```

► From assignment 1, the ordered_array class, we want to add a function to insert multiple copies of a given value:

```
class ordered_array {
  public:
    ⋮
    void ordered_array::insert_multiple(int x, int n);
    ⋮
  private:
    int* data;
    int sz, cap;
};
```

which inserts n copies of x. Since the array is ordered, all copies will be inserted at the same position. Write the code for `insert_multiple`. You can use the existing `insert` function, if you want.

Solution:

```
void ordered_array::insert_multiple(int x, int n)
{
    // Figure out where to insert
    for(int i = 0; i < sz; ++i)
        if(data[i] > x) {
            // Insert before index i

            // Shift up by n elements
            sz += n;
            for(int j = sz-1; j > i + n; --j)
                data[j] = data[j-n];

            // Fill with n copies of x
            for(int j = i; j < i + n; ++j)
                data[j] = x;

            return;
        }
}
```

- What does the following function do:

```
bool f(vector<int> x)
{
  bool g = true;
  for(int i = 0; i < x.size() - 1; ++i)
    if(x[i] > x[i+1])
      g = false;

  return g;
}
```

Give examples of non-empty vectors, one which will cause the function to return true, and one which will cause it to return false.

Solution: It returns false if the elements of the vector are not in ascending order. To return true, give a vector sorted in ascending order, e.g., `1, 2, 3, ...`. To return false, give a vector where any pair of elements are `x[i] > x[i+1]`: `1, 2, 1, ...`.

- Given the following ordered array class

```
class ordered_array {
  public:
    ...
    void remove(int x); // Removes one copy
    void remove_multiple(int x);
  private:
    int* data;
    int sz;
    int cap;
};
```

Implement the `remove_multiple` method, which should remove all the copies of x in the array (the normal `remove` only removes one copy). You can use the existing `remove` function if you want.

```
void ordered_array::remove_multiple(int x)
{
```

Solution:

```
void ordered_array::remove_multiple(int x)
{
  // remove until the size stops changing
  int last_sz;
  do {
    last_sz = sz;
    remove(x);
  } while(last_sz ≠ sz);
}
```

2 Big-O analysis; Vectors, lists, stacks, and queues

- This function searches for all occurrences of a pattern p in a string s:

```
void find_all(string s, string p)
{
  for(int i = 0; i < s.length() - p.length(); ++i) {
    bool found = false;
    for(int j = 0; j < p.length(); ++j)
      if(s[i + j] ≠ p[j])
        found = false;

    if(found)
      cout << "Found at " << i << endl;
  }
}
```

It p has length m and s has length n, and we assume that n is much larger than m, what is the big-O complexity of this function? Does it have different best- and worst-case complexities?

Solution: best/worst case is $O(mn)$. It has the same best/worst case because we cannot exit either loop early. (We *could* exit the j-loop early, as soon as `s[i+j]` $\neq$ `p[j]`, which would change the best-case, but the code doesn't do this.)

- Here is a function that checks a vector to see if it is sorted:

```
bool is_sorted(vector<int>& v) {
  for(int i = 0; i < v.size() - 1; ++i)
    if(v[i] > v[i+1])
      return false;
  return true;
}
```

Analyze the cost of this function, in terms of the number of comparisons `v[i] > v[i+1]`, the number of increments `++i`, and the number of vector lookups `v[...]`. What is the best case cost? What is the worst case cost? When (for what inputs) do the best/worst cases occur?

Solution:

	Best	Worst
`v[i] > v[i+1]`	2	$n-1$
`++i`	0	$n-1$
`v[...]`	2	$2(n-1)$
	`v[0] > v[1]`	Sorted ascending

► Given the following implementation of `vector::push_back` trace through the cost of the first 10 pushbacks, if a "cheap" pushback (i.e., a single copy) has a cost of 1, and the initial size and capacity are 0.

```
void vector::push_back(int x)
{
  if(size == capacity) {
    // Full, reallocate to make room
    int* old_data = data;
    int new_cap = 1 + capacity + capacity / 2
    data = new int[new_cap];

    // Copy everything to the new array
    for(int i = 0; i < capacity; ++i)
      data[i] = old_data[i];

    capacity = new_cap;
    delete[] old_data;
  }

  // Add new element
  data[size++] = x;
}
```

PB #	Size	Capacity	Cost
	0	1	
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Solution:

PB #	Size	Capacity	Cost
	0	1	
1	1	1	1
2	2	2	2
3	3	4	3
4	4	4	1
5	5	7	5
6	6	7	1
7	7	7	1
8	8	11	8
9	9	11	1
10	10	11	1

- Given the following `list` class definition

```
class list {
  public:

    struct node {
      int value;
      node* next;
    };

    node* find(int x);

  private:
    node* head;
}
```

Implement the `find` method searches through the list for the node with value x, returning a pointer to it if it exists, or `nullptr` if it does not exist.

```
list::node* list::find(int x)
{
```

Solution

```
list::node* list::find(int x)
{
  node* n = head;
  while(n ≠ nullptr and n->value ≠ x)
    n = n->next;

  return n;
}
```

► While it's not possible *in general* to make the linked-list `.at(i)` operation run in better than $O(n)$ time, it is possible to make `.at(i)` run in $O(1)$ time when used in this specific way:

```
for(int i = 0; i < sz; ++i)
    ... l.at(i) ...
```

When `.at(`i_1`)` is called, it saves both the index and `node*` into class member variables; when `.at(`i_2`)` is called later, we check to see if $i_2 \geq i_1$; if so, we use the saved `node*` to start the search for i_2 at i_1

Using the following `list` class, make the necessary changes to `at` so that it runs in $O(1)$ time when used as above.

```
class list {
  public:
    ...
    node* at(int i);
    ...
  private:
    node* hd;
    int sz;

    // at() has not been called yet, so we don't have a saved node*
    node* last_at_ptr = nullptr;
    int   last_at_ind = -1;
};

list::node* list::at(int i)
{
```

Solution:

```
list::node* list::at(int i)
{
  node* n;
  if(last_at_index == -1) {
    // No previous .at(), start from scratch
    n = hd;
    last_at_ind = 0;
  }
  else if(i ≥ last_at_ind) {
    // Start search where the last one left off
    i -= last_at_ind;
    n = last_at_ptr;
  }

  while(n ≠ nullptr and i ≠ 0) {
    --i;
    ++last_at_ind;
    n = n->next;
  }

  if(n == nullptr)
    last_at_ind = -1; // Went off the end, no node to save

  last_at_ptr = n;

  return n;
}
```