Practice test for Midterm 1 – Solutions

September 25, 2019

1 C++ review

► Write a function which takes a vector of `int` and an `int` m and returns true if all the values in the vector are factors of m.

```
bool has_factors(vector<int> facs, int m) {
  for(int i = 0; i < facs.size(); ++i)
    if(m % facs[i] != 0)
      return false;

  return true;
}
```

► What does the following function do:

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

Checks to see if the vector is the same backwards and forwards. E.g.,

```
{ 1, 2, 3, 2, 1} // True
{ 1, 2, 3, 4, 5} // False
```

► Given the following ordered array class

```
class ordered_array {
  public:
    ...
    int count(int x);
  private:
    int* data;
    int current_size;
    int max_size;
};
```

Implement the `count` method, which should count how many times an element x occurs in the array.

```
int ordered_array::count(int x) {
  int c = 0;
  for(int i = 0; i < current_size; ++i)
    if(data[i] == c)
      ++c;

  return c;
}
```

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

► This function checks whether a vector a is a "subset" of another vector b: i.e., whether every element of a is also an element of b.

```
bool subset_of(const vector<int>& a, const vector<int>& b)
{
  for(int i = 0; i < a.size(); ++i) {
    bool found = false;

    for(int j = 0; j < b.size(); ++j)
      if(b[j] == a[i]) {
        found = true;
        break;
      }

    if(!found)
      return false;
  }
```

```
  return true;
}
```

Assume that a has size m and b has size n. What is the worst-case big-O complexity of this function? What is the best-case complexity?

Worst case is $O(mn)$. Best case is either $O(m+n)$ (i.e., the smaller of the two vectors).

► Here is a function that checks a vector to see if it is *bitonically sorted* (begins ascending, and then switches to descending at some point):

```
bool is_bitonic(vector<int> v)
{
  // Check ascending section
  int i;
  for(i = 0; i < v.size() - 1; ++i)
    if(v[i] > v[i+1])
      break;

  // Check descending section
  for(int j = i; j < v.size() - 1; ++j)
    if(v[j] < v[j + 1])
      return false;

  return true;
}
```

Analyze the cost of this function, in terms of the number of comparisons C *between vector elements* (i.e., do not count `i < v.size()`) the number of increments I, and the number of vector lookups L. What is the best case cost? What is the worst case cost? When (for what inputs) do the best/worst cases occur?

	Best	Worst
C	3	$n-1$
I	1	$n-1$
L	6	$2n-2$

The best case is when the input starts with a descending pair, followed by an ascending pair: e.g., `{2,1,2,...` The worst case is when the input actually is bitonically ordered.

► 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;
    data = new int[1 + 5 * capacity / 3];

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

    capacity = 1 + 5 * capacity / 3;
    delete[] old_data;
  }

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

PB	Size	Capacity	Cost
	0	0	
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	12	8
9	9	12	1
10	10	12	1

- Given the following `list` definition

```
class list {
  public:

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

    bool is_sorted();

  private:
    node* head = nullptr;
}
```

Implement the `is_sorted` method, which checks a list to see if it is sorted, and returns true if it is.

```
bool list::is_sorted()
{
  for(list::node* n = list.head;
      n != nullptr && n->next != nullptr;
      n = n->next)
  {
    if(n->value > n->next->value)
      return false;
  }

  return true;
}
```

► Suppose we have a singly-linked list with methods `head()` and `at()`, and the above `node` type. The following function constructs the reversed version of a list (a new list with the elements in reverse order)

```
list reverse(list l) {
  list out;
  for(int i = 0; i < l.size(); ++i)
    out.push_front(l.at(i));

  return out;
}
```

What is the time complexity of this function? If necessary, rewrite the function so that its time complexity is $O(n)$.

Time complexity is $O(n^2)$. The correct version is

```
list reverse(list l) {
  list out;
  for(list::node* n = l.head;
      n != nullptr;
      n = n->next)
    out.push_front(n->value);

  return out;
}
```

► Given the following `stack` type (array based), write the `push` and `pop` methods:

```
class stack {
  public:
```

```
    ...
    void push(int x);
    void pop();

  private:
    int* data;     // Array of stack elements
    int  size;     // Max size
    int  top = -1; // Index of top element
};


void stack::push(int x) {
  top++;
  data[top] = x;
  size++;
}

void stack::pop() {
  top--;
  size--;
}
```

► Given the following queue type, implement the queue operations enqueue and dequeue:

```
class queue {
  public:
    void enqueue(int x);
    void dequeue();

  private:
    struct elem {
      int   value;
      elem* next;
    };

  elem* back = nullptr;  // back of the queue
  elem* front = nullptr; // Front of the queue
};
```

(This is a list-based queue, but with the list built-in to the queue class itself.)

```
void queue::enqueue(int x) {
  back = back->next = new elem{x, nullptr};
}
```

```
void queue::dequeue() {
  elem* t = front;
  front = front->next;
  delete t;
}
```

► Suppose we have an array-based queue of size 8. Draw the state of the array after the following queue operations have been executed. For each dequeue() operation, write next to it the element that would be removed from the queue.

```
enqueue(7);
enqueue(5);
enqueue(3);
dequeue();
dequeue();
enqueue(2);
enqueue(1);
dequeue();
dequeue();
```

0	1	2	3	4	5	6	7
7							
7	5						
7	5	3					
	5	3					
		3					
		3	2				
		3	2	1			
			2	1			
				1			

► Given a stack-of-char class

```
class stack {
  public:
    void push(char c);
    char pop();
    char top();
    bool empty();
  ...
};
```

Use this class to write a function which checks a string containing [] and () parentheses to see if they are properly nested and matched.

```
bool proper_nesting(string s) {
  stack st;

  for(char c : s) {
    if(c == '[' || c == '(')
      st.push(c);
    else if(c == ']') {
      if(st.empty())
        return false; // Not enough [
      else if(st.top() != '[')
        return false; // Mismatched
      else
        st.pop();
    }
    else if(c == ')') {
      if(st.empty())
        return false; // Not enough [
      else if(st.top() != '(')
        return false; // Mismatched
      else
        st.pop();
    }
  }

  return st.empty();
}
```