CPSC 221 2017W2: Midterm Exam 1 SOLUTION

January 29, 2018

1 Who gets the marks? [1 marks]

Please enter your 4 or 5 digit CSID in this box: Cinda

2 Choices, miscellany, and some originality [16 marks]

Unless otherwise specified, select the single best answer among the choices.

MISC1 [2 marks]

```
void addOne(int & n) {
   n = n + 1;
}
void addTwo(int * n) {
   *n = *n + 2;
}
int addFour(int n) {
   return n + 4;
}

int main() {
   int p = 0;

   p = addFour(p);
   addTwo(&p);
   addOne(p);

   cout << p << endl;

   return 0;
}
```

What is the result of executing these statements?

- ● 7 is sent to standard out. (Fill in the blank with the appropriate integer.)
- ○ This code exhibits unpredictable memory behavior.
- ○ This code does not compile because of a type mismatch.
- ○ This code has a memory leak.

MISC2 [2 marks]

Consider this simple example, and assume the standard `iostream` library has been included.

```
1        int ** p; // pointer to an integer pointer
2        int x = 8;
3        p = new int*;
4 // WHAT LINE GOES HERE?
5        **p = 12;
6        cout << x << endl;
```

Complete Line 4 above to associate variables `p` and `x` in such a way that the output of the code is 12:

*p = &x

MISC3 [3 marks]

Suppose we have defined a class `Dessert` with a default constructor, copy constructor, assignment operator, and destructor. For the code segment below, how many times is each function called by the time `main` returns? Write your answers in the spaces provided.

```
int main() {
  Dessert* pudding;
  Dessert cookie; // Default Constructor
  pudding = new Dessert(cookie); // Copy Constructor
  Dessert* custard = &cookie;
  *custard = *pudding; // assignment operator
  Dessert* cake = pudding;
  pudding = new Dessert(); // Default Constructor
  delete cake; // Destructor
  //Destructor as cookie goes out of scope.
}
```

2	Default constructor
1	Copy constructor
1	Assignment operator
2	Destructor

MISC4 [2 marks]

In the C++ function below, give the tightest asymptotic upper bound that you can determine for the function's **runtime**, in terms of the input parameter.

```
void mittens(int n) {
   for (int i = n*n*n; i > 1; i = i/2) {
        cout << "It's grey day number: " << i << endl;
   }
```

```
}
```

Running time for `mittens`:	$O(\log n)$

MISC5 [2 marks]

In the C++ function below, give the tightest asymptotic upper bound that you can determine for the function's **runtime**, in terms of the input parameter.

```
int touque(int n) {
   for (int i = 0; i < 2*n; i++){
      for (j = 0; j < i; j += 3) {
         cout << "The number of people who spell it toque: " << j << endl;
      }
      for (j = i; j < 2*n; j += 2) {
         cout << "The number of people who spell it tuque: " << j << endl;
      }
   }
}
```

Running time for `touque`:	$O(n^2)$

MISC6 [1 marks]

Suppose that you have an unordered, null-terminated singly-linked list containing n nodes with head and tail pointers. What is the running time of an efficient algorithm to remove all instances of a specified value?

- ○ $O(1)$
- ○ $O(\log n)$
- ● $O(n)$
- ○ $O(n^2)$
- ○ None of the options is correct.

MISC7 [2 marks]

Suppose that you have an unordered, circular doubly-linked list containing n nodes and only a head pointer. What is the running time of an efficient algorithm to remove the last element of the list? (In a circularly linked list, `head->prev` points to the last element in the list, and `head->prev->next == head`.)

- ● $O(1)$
- ○ $O(\log n)$
- ○ $O(n)$
- ○ $O(n^2)$
- ○ None of the options is correct.

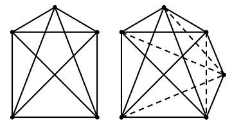
MISC8 [2 marks]

Suppose that you have a sorted array containing n values. What is the best-case running time for inserting an element in the exact middle of the array?

- ○ $O(1)$
- ○ $O(\log n)$
- ● $O(n)$
- ○ $O(n^2)$
- ○ None of the options is correct.

3 Many sides to the issue [10 marks]

A diagonal in a convex polygon is a line segment connecting two non-neighbouring vertices. Polygons for $n = 5$ and $n = 6$ with their diagonals are shown in the figure below:



Prove using induction, that the number of diagonals in a convex n-sided polygon ($n \geq 3$) is $\frac{n(n-3)}{2}$.

(a) **[3 marks]**

Base case: For $n = \boxed{3}$, the polygon is a triangle and it has no diagonals. The formula that counts diagonals gives $\frac{3(3-3)}{2} = 0$, which is what we want.

(b) **[3 marks]** Inductive hypothesis: For any $3 < j < n$, a polygon of j edges has $\frac{j(j-3)}{2}$ diagonals.

Note that there are other appropriate ways of stating this hypothesis. A strong statement isn't necessary here!

(c) **[4 marks]**

Complete your inductive step: For any $n > 3$, a polygon of size n, together with all of its diagonals, can be decomposed into a polygon with $n-1$ sides and its diagonals, and one additional vertex and the edges incident on that vertex. (Proof of this not required here, but suffice to say, if it weren't possible, then the polygon wouldn't be convex.) Counting diagonals, we have: $(n-1)((n-1)-3)/2$ from the polygon of size $n-1$, $n-3$ from the diagonals incident on the additional vertex, and one additional diagonal from the boundary of the $n-1$ sized polygon.

$$\frac{(n-1)(n-4)}{2} + (n-3) + 1 = \frac{(n-1)(n-4) + 2(n-2)}{2} = \frac{(n^2 - 3n)}{2} = \frac{n(n-3)}{2}$$

Note that there are several alternative correct proofs. For example, students may prove $P(k) \implies P(k+1)$, or they may take a different approach entirely, counting total edges and subtracting to get the result. All of this is fine. We DO want them to do an inductive proof, rather than a combinatorial proof, so correct combinatorial proofs only get partial credit.

4 Laundry is the correct thing to do [17 marks]

Geoff has just finished washing n smelly socks of many different colours and must organize them. Black socks must be paired together so that he is presentable on work days, but for other days he does not care if his socks do not match. The following algorithm below puts black socks next to one another in pairs. Geoff owns an even number of Black socks.

```
void MatchSocks(vector<int>& socks)
{
  bool odd = (socks[0].colour == BLACK); // indicates odd or even Black socks so far
  for (int i = 1; i < socks.size(); i++) {
    if (socks[i].colour == BLACK) odd = !odd; // flip between odd and even
    if (socks[i].colour != BLACK && socks[i-1].colour == BLACK && odd) {
      swap(socks[i-1], socks[i]); // swap vector elements
    }
  }
}
```

(a) **[2 marks]** Trace the algorithm on the small example below. Black socks are denoted by a B.

Index:	0	1	2	3	4	5	6	7
Original:	B	O	B	B	O	P	O	B
Result:	O	B	B	O	P	O	B	B

(b) **[2 marks]** Fill in the blanks for the following loop invariant:

Before iteration i of the loop:

- If socks[$\boxed{0}$] to socks[$\boxed{\text{i-1}}$] contains $2k+1$ black socks, for some k (an odd number), then the socks are arranged in k neighboring pairs, and socks[$\boxed{\text{i-1}}$] is black.
- If socks[$\boxed{0}$] to socks[$\boxed{\text{i-1}}$] contains $2k$ black socks, for some k (an even number), then the socks are arranged in k neighboring pairs.

(c) **[13 marks]** Perform a loop invariant analysis to prove that at termination, all the BLACK socks appear in pairs in the socks vector.

Initialization (base case):

- If `socks[0] == BLACK`: Then, vacuously, all but one of the black socks are paired, and the one black sock is in the most recently considered position. Odd should be set to true, since we have seen exactly one black sock.
- If `socks[0] != BLACK`: Then, vacuously, all but one of the black socks are paired. Odd should be set to false, since we've seen an even number of black socks (0).

Maintenance (inductive case):

- If `socks[i] == BLACK`:
  - case 1: there are an odd number of black socks in socks[0..i-1]. The loop invariant (above) says that socks[i-1] is black. Since we're observing another, we've found a pair! We simply set the variable odd to false and the loop is complete. The invariant holds at the end of the loop because by adding a pair to those already paired, we're maintaining the fact that all black socks are paired if we've seen an even number of them.
  - case 2: there are an even number of black socks in socks[0..i-1]. In this case, we just set the variable to odd and move onward. The loop invariant holds at the end of the iteration because odd is true, and the last observed sock is black.
- If `socks[i] != BLACK`: (Note that the value of variable odd doesn't change in this case since we're not seeing another black sock.)
  - case 1: there are an odd number of black socks in socks[0..i-1]. The loop invariant (above) says that socks[i-1] is black. the loop swaps the socks in locations i-1 and i, maintaining the fact that the most recently seen sock is black.
  - case 2: there are an even number of black socks in socks[0..i-1]. In this case, the loop simply increments and no state change is affected. Since there are an even number of socks, they're all still paired, and the

Termination (end case): The above holds for all i, and in this case the last value of i is socks.size()-1. At the end of that iteration, all black socks are paired!

5 Sort of neat... [12 marks]

In this problem you will show us that you understand the concept of *loop invariants* in sorting algorithms. Each of the lists of names below has been created by invoking a sorting algorithm, and stopping it after some number of iterations, k. For each list, and for each sorting algorithm, fill in the box with the *maximum* possible value of k. If no iterations could have occurred, then $k = 0$. In the table, "Sel" stands for selection sort, and "Ins" stands for insertion sort. In this problem we are considering an "iteration" to be one execution of the *outer* loop of the algorithm.

	ex 1	ex 2	ex 3
0	Asuna	Akame	Erza
1	Winry	Akeno	Jubia
2	Shiro	Asuna	Lucy
3	Yuri	Erza	Lucy
4	Lucy	Jubia	Maka
5	Shana	Lucy	Nami
6	Erza	Taiga	Riza
7	Saeko	Yuri	Shana
8	Taiga	Shana	Shiro
9	Jubia	Nami	Taiga
10	Lucy	Shiro	Winry
11	Maka	Saeko	Yuri
12	Nami	Maka	Asuna
13	Akame	Lucy	Akame
14	Akeno	Riza	Akeno
15	Riza	Winry	Saeko
Ins	2	8	12
Sel	0	6	0

GRADING NOTE: We should allow for the insertion sort line to be 1, 7, 11, since the outer loop may not iterate over a one element segment of the array.

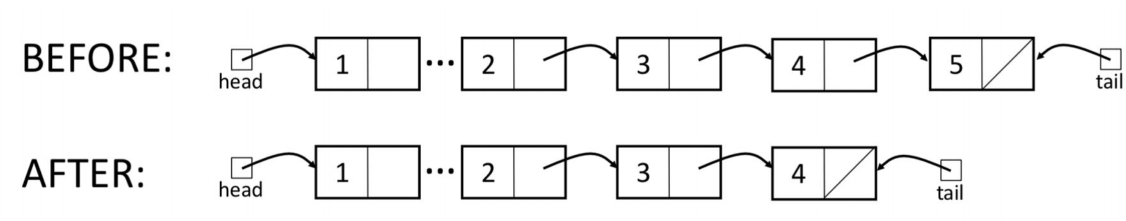
6 A twist in the list [9 marks]

In each of the problem segments below, we have given you "before and after" models of linked lists. Your task is to transform the "before" into the "after" using simple pointer manipulations on the list nodes. Refer to the elements of the list nodes using the `Node` class below. Your solutions should follow these guidelines:

- You may declare `Node` pointer variables to use in navigating the lists. When you are finished with them, set them to `NULL`.
- You must never refer to the `data` member of the `Node` class.
- You may write loops to simplify your solutions, but your answers don't need to be general... they just need to work on the given lists. (Don't worry about even/odd length, or empty lists, for example.)
- Any variables listed in the picture can be used in your solution. If they do not appear in the "after" diagram, they should be set to `NULL`.
- If a node is removed from a list, be sure to free its memory!

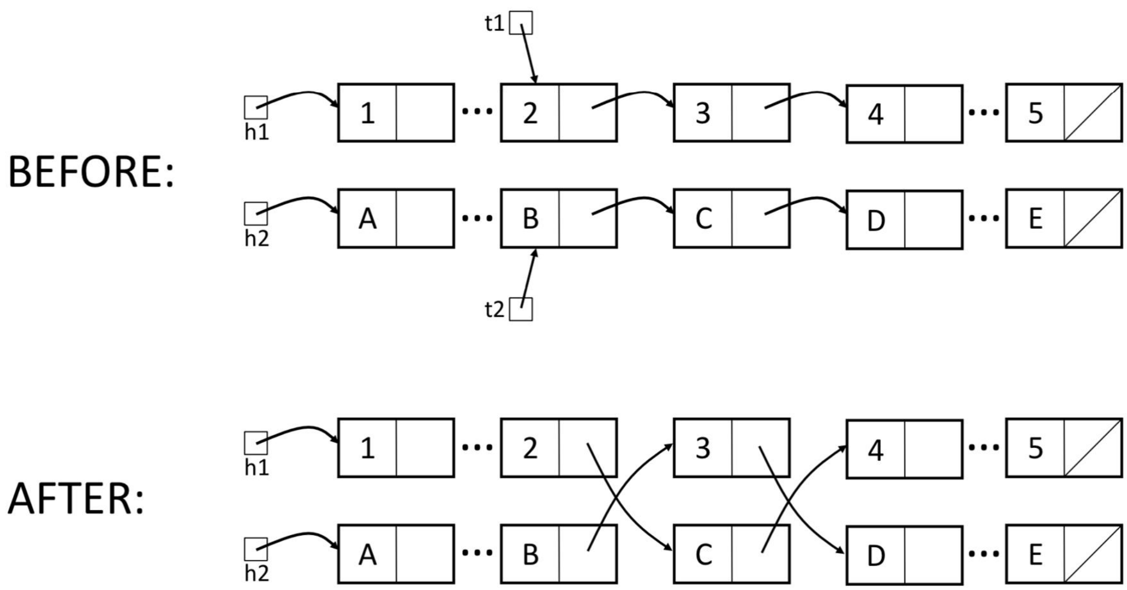
```
class Node {
   public:
      int data;
      Node * next;
      Node(int e): data(e), next(NULL) {} };
```

(a) **[4 marks]**



line #	
1	`Node * t = head;`
2	`While (t->next != tail)`
3	`    t = t->next;`
4	`delete tail;`
5	`t->next = null;`
6	`tail = t;`
7	`t = null;`
8	

(b) **[5 marks]**



line #	
1	`Node * p1 = t1->next;`
2	`Node * p2 = t2->next;`
3	`t1->next = p2;`
4	`t2->next = p1;`
5	`t1 = p1->next;`
6	`t2 = p2->next;`
7	`p1->next = t2;`
8	`p2->next = t1;`
9	`p1 = p2 = t1 = t2 = NULL;`
10	
11	
12	

This page intentionally left (almost) blank.

If you write answers here, you must CLEARLY indicate on this page what question they belong with AND on the problem's page that you have answers here.

This page intentionally left (almost) blank.

If you write answers here, you must CLEARLY indicate on this page what question they belong with AND on the problem's page that you have answers here.