CPSC 221 HW2

d6a2b, z0n1b

TOTAL POINTS

60.5 / 65

QUESTION 1

DeadHeads 25 pts

1.1 1a) constructor 2 / 2

✓ **- 0 pts Correct**

- 2 pts Missing line: head = new Node()

- 1 pts Missing constructor declaration: List::List()

- 0.5 pts Unnecessary code

- 0.5 pts Minor syntax error

- 2 pts Blank/Incorrect

1.2 1b) addAt no DH 2 / 2

✓ **- 0 pts Correct**

- 1 pts Missing or incorrect case: pos ==0

- 1 pts Missing or incorrect case: pos > 0

- 1 pts Incorrect Logic (either in walk() or assignment of head)

- 2 pts Blank/Incorrect

1.3 1c) addAt DH 2 / 2

✓ **+ 1 pts Correct walk(head, pos)**

✓ **+ 1 pts Assignment of next to new Node(elt, t->next)**

- 2 pts Blank/Incorrect

1.4 1d) cc and relation 2 / 2

✓ **+ 1 pts Dead Head**

✓ **+ 1 pts less than**

+ 0 pts Blank/Incorrect

1.5 1e) delAll 4 / 4

+ 2 pts Node* a = head;

✓ **+ 2 pts Node* t = a->next;**

✓ **+ 2 pts else a = a->next**

+ 0 pts Blank/Incorrect

- 1 pts Too many items chosen

1.6 1f) remove #1 3 / 3

✓ **- 0 pts Correct**

- 1 pts Fails to declare or advance a temporary node pointer to a useful place

- 1 pts Incorrect pointer reassignment

- 1 pts Fails to free memory associated with p

- 0.5 pts Missing or incorrect function declaration: void List::delNode(Node* p)

- 3 pts Blank/Incorrect

1.7 1g) 1f running time. 1 / 1

✓ **+ 1 pts Θ(n)**

+ 0 pts Blank/Incorrect

1.8 1h) remove #2 4 / 4

✓ **- 0 pts Correct**

- 1 pts Fails to declare a temporary node pointer to a useful place

- 1 pts Fails to preserve the data contained in p

- 1 pts Incorrect pointer reassignment

- 1 pts Fails to free memory associated with p

- 0.5 pts Missing or incorrect function declaration: void List::delNode(Node* p)

- 4 pts Blank/Incorrect

1.9 1i) analyze flaw 1 / 2

✓ **+ 1 pts Tail node**

+ 1 pts Valid description of design changes to fix algorithm (e.g. circular linked list, dummy tail, ...)

+ 0 pts Blank/Incorrect

We would want to change the list

1.10 1j) hack 2.5 / 3

- 0 pts Correct

- 1 pts Missing bullet 1: If p->data is a large object...

- **1 pts** Missing bullet 3: In order to work, the assignment...
- **1 pts** Missing bullet 7: An iterator over the list may be...

✓ **- 0.5 pts** **4 bullet points selected**

- **1 pts** 5 bullet points selectd
- **3 pts** Blank/Incorrect
- **1.5 pts** More than 5 bullet points selected

QUESTION 2

Structured Sorting 22 pts

2.1 2a) assure validity 2 / 2

✓ **+ 1 pts** **For all prefixes of the sequence, the number of \I's is at least the number of \O's.**

✓ **+ 1 pts** **The number of \I's in the whole sequence equals the number of \O's.**

+ **0 pts** Blank/Incorrect

2.2 2b) impossible sequence 2 / 2

✓ **- 0 pts** **2, 3, 1**

- **2 pts** Blank/Incorrect

2.3 2c) proof of impossible 8 / 8

✓ **- 0 pts** **Valid proof**

- **2 pts** Minor leap without justification
- **4 pts** Major leap without justification
- **8 pts** Blank/Incorrect

2.4 2d) input sequence 2 / 2

✓ **- 0 pts** **Correct: 3, 2, 4, 1**

- **2 pts** Blank/Incorrect

2.5 2e) prove whether or not SQ is enough 4 / 4

✓ **- 0 pts** **Correct: no + a valid permutation that cannot be output (e.g. 4, 5, 2, 3, 1)**

- **4 pts** Blank/Incorrect

2.6 2f) how about now? 4 / 4

✓ **- 0 pts** **Answers yes with adequate explanation**

- **1 pts** Minior Error in Explaination
- **2 pts** Inadequate explanation
- **4 pts** Blank/Incorrect

QUESTION 3

Loop Invariants 18 pts

3.1 findMaxHelper correct 2 / 5

- **0 pts** Correct proof

✓ **- 1 pts** **Missing/Incorrect explanation of base case: right-left=1**

✓ **- 1 pts** **Missing/Incorrect inductive hypothesis: true for (right-left) < k**

- **3 pts** Missing/Incorrect inductive step explanation: prove for k

✓ **- 1 pts** **Minor error in reasoning**

3.2 findMaxHelper recurrence 5 / 5

✓ **+ 1 pts** **Correct base case: T(1) <= b or T(1) = 1 or ...**

✓ **+ 2 pts** **Correct recurrence relation: T(n) <= 2T(n/2) + c for n > 1**

✓ **+ 2 pts** **Correct closed form: bn + cn - c (for some constants b and c, e.g. 2n-1)**

+ **0 pts** Blank/Incorrect
- **0.5 pts** minor error

3.3 LIS correctness 8 / 8

✓ **- 0 pts** **Correct**

- **1 pts** Missing/Incorrect base case
- **2 pts** Missing/Incorrect inductive hypothesis
- **5 pts** Missing/Incorrect inductive step
- **1 pts** Minor error/missing case
- **8 pts** Blank/Incorrect

1. **Dead Heads (25 points).**

   In some cases, it's easier to handle the special case of deleting the head of a linked-list or inserting a new node in front of the head of a linked-list by using a *dead head* (sometimes called a *dummy head*, or *sentinel*). The dead head is always the first node in the linked-list but it contains no data. (You can think of it as an empty node!) Its `next` pointer points to the "real" head of the linked-list, that is the first node in the linked-list that contains valid data. So an empty linked-list is represented by a dead head whose `next` pointer is `NULL`. The user specifies the linked-list by using a pointer to the list's dead head. For this problem, you will fix some implementations that use dead-headed linked-lists as a private member representation of a `List` class.

   Here is the `List` class definition that we will examine and expand:

```
class List{
  public:
    void addAt(int pos, int elt);
    void delAll(int x);
  private:
    struct Node{
      int data;
      Node * next;
      Node(int elt = 0,Node * p = NULL):data(elt),next(p){}
    };
    Node * head;
    Node * walk(Node * curr, int k);
    void delNode(Node * p);
};

List::Node * List::walk(List::Node * curr, int k){
  // walks curr forward k steps
  if (k<=0 || curr == NULL)
    return curr;
  else return walk(curr->next, k-1);
}
```

   (a) (2 points) Write the no-argument constructor for the above `List` class if the class is designed so that the empty list is represented by a single dead head node.

```
List::List(){
  head = new Node();
}
```

   (b) (2 points) Suppose that you have designed class `List` to contain a linked list implemented *without* a dead head. (We will refer to this implementation as the "Live Head" implementation, which is not an official term!) Write member function `addAt(int pos, int e)` which creates a new node at position `pos` whose data is `e`. Note that in this case we consider the first data element of the list to be position 0, and that in an empty list, `head == NULL`. Please use the given `walk` function to advance a pointer along the list.

1.1 1a) constructor 2 / 2

✓ **- 0 pts** **Correct**

- 2 pts Missing line: head = new Node()

- 1 pts Missing constructor declaration: List::List()

- 0.5 pts Unnecessary code

- 0.5 pts Minor syntax error

- 2 pts Blank/Incorrect

```
void List::addAt(int pos, int elt){
  //we may assume that pos is valid, only range is [0,size]
  //where pos=0 is add to beginning, before everything
  //and pos=size is add to end, after everything
  //piazza 693
  Node *offset = new Node;
  offset->next = head;
  Node *toAdd = walk(offset, pos);
  Node *newnode = new Node;
  newnode->data = elt;
  newnode->next = toAdd->next;
  toAdd->next = newnode;
  if (pos == 0) {
    head = newnode;
  }
  delete offset;
}
```

(c) (2 points) Rewrite the `addAt` member function for a list designed to use a dead head at the front of the list. Note that in this case we consider the first data element of the list to be position 0. You may assume that member variable `head` is not `NULL`, even if the list is empty.

```
void List::addAt(int pos, int elt){
  //we may assume that pos is valid, only range is [0,size]
  //where 0 is add before everything but after deadhead
  //and size is add after everything
  //piazza 693
  Node *toAdd = walk(head, pos);
  Node *newnode = new Node;
  newnode->data = elt;
  newnode->next = toAdd->next;
  toAdd->next = newnode;
}
```

(d) (2 points) Research the meaning of the term *cyclomatic complexity*, and use your understanding to answer the following questions.

Which implementation of `addAt` has lower cyclomatic complexity?
✓ Dead Head ○ Live Head

Given their relative cyclomatic complexities, we expect the number of test cases for the dead head implementation to be ○ greater than ✓ less than ○ equal to the number we would need for the live head implementation of `addAt`.

(e) (4 points) The following `List` class member function is intended to delete all nodes with data equal to x from a dead-headed linked-list whose last node has a `NULL` next pointer. It doesn't always work. Two lines of code need to be corrected. Write the correct lines

1.2 1b) addAt no DH **2 / 2**

✓ **- 0 pts Correct**

- 1 pts Missing or incorrect case: pos ==0

- 1 pts Missing or incorrect case: pos > 0

- 1 pts Incorrect Logic (either in walk() or assignment of head)

- 2 pts Blank/Incorrect

```
void List::addAt(int pos, int elt){
  //we may assume that pos is valid, only range is [0,size]
  //where pos=0 is add to beginning, before everything
  //and pos=size is add to end, after everything
  //piazza 693
  Node *offset = new Node;
  offset->next = head;
  Node *toAdd = walk(offset, pos);
  Node *newnode = new Node;
  newnode->data = elt;
  newnode->next = toAdd->next;
  toAdd->next = newnode;
  if (pos == 0) {
    head = newnode;
  }
  delete offset;
}
```

(c) (2 points) Rewrite the `addAt` member function for a list designed to use a dead head at the front of the list. Note that in this case we consider the first data element of the list to be position 0. You may assume that member variable `head` is not `NULL`, even if the list is empty.

```
void List::addAt(int pos, int elt){
  //we may assume that pos is valid, only range is [0,size]
  //where 0 is add before everything but after deadhead
  //and size is add after everything
  //piazza 693
  Node *toAdd = walk(head, pos);
  Node *newnode = new Node;
  newnode->data = elt;
  newnode->next = toAdd->next;
  toAdd->next = newnode;
}
```

(d) (2 points) Research the meaning of the term *cyclomatic complexity*, and use your understanding to answer the following questions.

Which implementation of `addAt` has lower cyclomatic complexity?
✓ Dead Head ○ Live Head

Given their relative cyclomatic complexities, we expect the number of test cases for the dead head implementation to be ○ greater than ✓ less than ○ equal to the number we would need for the live head implementation of `addAt`.

(e) (4 points) The following `List` class member function is intended to delete all nodes with data equal to x from a dead-headed linked-list whose last node has a `NULL` next pointer. It doesn't always work. Two lines of code need to be corrected. Write the correct lines

1.3 1c) addAt DH 2 / 2

✓ + 1 pts **Correct walk(head, pos)**

✓ + 1 pts **Assignment of next to new Node(elt, t->next)**

- 2 pts Blank/Incorrect

```
void List::addAt(int pos, int elt){
  //we may assume that pos is valid, only range is [0,size]
  //where pos=0 is add to beginning, before everything
  //and pos=size is add to end, after everything
  //piazza 693
  Node *offset = new Node;
  offset->next = head;
  Node *toAdd = walk(offset, pos);
  Node *newnode = new Node;
  newnode->data = elt;
  newnode->next = toAdd->next;
  toAdd->next = newnode;
  if (pos == 0) {
    head = newnode;
  }
  delete offset;
}
```

(c) (2 points) Rewrite the `addAt` member function for a list designed to use a dead head at the front of the list. Note that in this case we consider the first data element of the list to be position 0. You may assume that member variable `head` is not `NULL`, even if the list is empty.

```
void List::addAt(int pos, int elt){
  //we may assume that pos is valid, only range is [0,size]
  //where 0 is add before everything but after deadhead
  //and size is add after everything
  //piazza 693
  Node *toAdd = walk(head, pos);
  Node *newnode = new Node;
  newnode->data = elt;
  newnode->next = toAdd->next;
  toAdd->next = newnode;
}
```

(d) (2 points) Research the meaning of the term *cyclomatic complexity*, and use your understanding to answer the following questions.

Which implementation of `addAt` has lower cyclomatic complexity?
✓ Dead Head ○ Live Head

Given their relative cyclomatic complexities, we expect the number of test cases for the dead head implementation to be ○ greater than ✓ less than ○ equal to the number we would need for the live head implementation of `addAt`.

(e) (4 points) The following `List` class member function is intended to delete all nodes with data equal to x from a dead-headed linked-list whose last node has a `NULL` next pointer. It doesn't always work. Two lines of code need to be corrected. Write the correct lines

1.4 1d) cc and relation **2 / 2**

✓ **+ 1 pts** **Dead Head**

✓ **+ 1 pts** **less than**

+ 0 pts Blank/Incorrect

in the boxes next to the two lines that are incorrect.

`void List::delAll(int x){`	
`  a = head;`	
`  while ( a->next != NULL ) {`	
`  if ( a->next->data == x ) {`	
`  Node *t=a;`	$Node * t = a- > next;$
`  a->next = a->next->next;`	
`  delete t;`	
`  }`	
`  a = a->next;`	$else\{a = a- > next;\}$
`  }`	
`}`	

In this part of the problem, we will explore an algorithm for removing a node from a singly linked list containing n nodes *given a pointer to the node we wish to remove.*

(f) (3 points) This removal task would be simple if we had a pointer to the node *before* the one we wish to remove. We can be sure there *is* a node before the one we wish to remove if we implement the list with a dead head. Complete the code below to implement the `List` class member function that removes node `p` from the list whose `head` is a dead head, using the comments to guide your algorithm. You may assume that `p` is a valid data element from the linked list.

```
void List::delNode(Node *p){
  Node * t = head;

  /* advance t until it's in a useful place */
  //no need for t->next == NULL guard if we know p exist in list
  while (t->next != p) {
    t = t->next;
  }

  /* adjust pointers to remove appropriate node */
  t->next = t->next->next;

  /* free memory associated with p */
  delete p;
  p = NULL;
}
```

(g) (1 point) The worst case running time of `delNode` is ○ $\Theta(1)$ ○ $\Theta(\log n)$ ✓ $\Theta(n)$

(h) (4 points) If we are clever, we can solve the problem of removing a node, given a pointer to it, *without* iterating over the list. To solve this problem, consider the pointer you

1.5 1e) delAll **4 / 4**

+ **2 pts** Node* a = head;

✓ **+ 2 pts** **Node* t = a->next;**

✓ **+ 2 pts** **else a = a->next**

+ **0 pts** Blank/Incorrect

- **1 pts** Too many items chosen

in the boxes next to the two lines that are incorrect.

`void List::delAll(int x){`	
`  a = head;`	
`  while ( a->next != NULL ) {`	
`  if ( a->next->data == x ) {`	
`  Node *t=a;`	$Node * t = a- > next;$
`  a->next = a->next->next;`	
`  delete t;`	
`  }`	
`  a = a->next;`	$else\{a = a- > next;\}$
`  }`	
`}`	

In this part of the problem, we will explore an algorithm for removing a node from a singly linked list containing n nodes *given a pointer to the node we wish to remove.*

(f) (3 points) This removal task would be simple if we had a pointer to the node *before* the one we wish to remove. We can be sure there *is* a node before the one we wish to remove if we implement the list with a dead head. Complete the code below to implement the `List` class member function that removes node `p` from the list whose `head` is a dead head, using the comments to guide your algorithm. You may assume that `p` is a valid data element from the linked list.

```
void List::delNode(Node *p){
  Node * t = head;

  /* advance t until it's in a useful place */
  //no need for t->next == NULL guard if we know p exist in list
  while (t->next != p) {
    t = t->next;
  }

  /* adjust pointers to remove appropriate node */
  t->next = t->next->next;

  /* free memory associated with p */
  delete p;
  p = NULL;
}
```

(g) (1 point) The worst case running time of `delNode` is ○ $\Theta(1)$ ○ $\Theta(\log n)$ ✓ $\Theta(n)$

(h) (4 points) If we are clever, we can solve the problem of removing a node, given a pointer to it, *without* iterating over the list. To solve this problem, consider the pointer you

1.6 1f) remove #1 3 / 3

✓ **- 0 pts Correct**

- 1 pts Fails to declare or advance a temporary node pointer to a useful place

- 1 pts Incorrect pointer reassignment

- 1 pts Fails to free memory associated with p

- 0.5 pts Missing or incorrect function declaration: void List::delNode(Node* p)

- 3 pts Blank/Incorrect

in the boxes next to the two lines that are incorrect.

`void List::delAll(int x){`	
`  a = head;`	
`  while ( a->next != NULL ) {`	
`  if ( a->next->data == x ) {`	
`  Node *t=a;`	$Node * t = a- > next;$
`  a->next = a->next->next;`	
`  delete t;`	
`  }`	
`  a = a->next;`	$else\{a = a- > next;\}$
`  }`	
`}`	

In this part of the problem, we will explore an algorithm for removing a node from a singly linked list containing n nodes *given a pointer to the node we wish to remove.*

(f) (3 points) This removal task would be simple if we had a pointer to the node *before* the one we wish to remove. We can be sure there *is* a node before the one we wish to remove if we implement the list with a dead head. Complete the code below to implement the `List` class member function that removes node `p` from the list whose `head` is a dead head, using the comments to guide your algorithm. You may assume that `p` is a valid data element from the linked list.

```
void List::delNode(Node *p){
  Node * t = head;

  /* advance t until it's in a useful place */
  //no need for t->next == NULL guard if we know p exist in list
  while (t->next != p) {
    t = t->next;
  }

  /* adjust pointers to remove appropriate node */
  t->next = t->next->next;

  /* free memory associated with p */
  delete p;
  p = NULL;
}
```

(g) (1 point) The worst case running time of `delNode` is ○ $\Theta(1)$ ○ $\Theta(\log n)$ ✓ $\Theta(n)$

(h) (4 points) If we are clever, we can solve the problem of removing a node, given a pointer to it, *without* iterating over the list. To solve this problem, consider the pointer you

1.7 1g) 1f running time. **1 / 1**

✓ **+ 1 pts** **Θ(n)**

+ 0 pts Blank/Incorrect

are given, and ask yourself which node *would* be easy to remove. Is there a way to remove that node, while at the same time preserving its data? The following function is intended to effectively delete the node pointed to by `p` from a linked-list with a dead head. Complete the code below so that it takes constant time to do this operation. You may assume that `p` points to a valid data node (not the dead head).

```
void List::delNode(Node *p){
  //"delete the node pointed to by p" = delete p->next
  /* set up a pointer to a node that would be easy to remove */
  Node * t = p->next;

  /* preserve the data contained in that node */
  p->data = p->next->data;

  /* fix pointers */
  p->next = p->next->next;

  /* free appropriate memory */
  delete t;
}
```

(i) (2 points) You probably noticed that this algorithm fails if the input parameter `p` points to a particular node.

i. The code fails if `p` is a pointer to what node?

```
The last node, because p->next = NULL,
and p->next->data is undefined behaviour.
```

ii. Describe how you would change the design of the linked list so that the algorithm above would work for any valid `p` (where "valid" means that it points to a node with data).

```
Since there's only one edge case, adding a guard saying
if(p->next == NULL) {
  //walk the list to get node before p
  //set that node's next to NULL, and free p
} else {
  //do the same thing
}
is enough.
```

(j) (3 points) This mechanism for removing a node from a list is commonly referred to as a constant time "hack" because it exhibits some important disadvantages. Choose the applicable disadvantages from the list below.

1.8 1h) remove #2 4 / 4

✓ **- 0 pts Correct**

- 1 pts Fails to declare a temporary node pointer to a useful place

- 1 pts Fails to preserve the data contained in p

- 1 pts Incorrect pointer reassignment

- 1 pts Fails to free memory associated with p

- 0.5 pts Missing or incorrect function declaration: void List::delNode(Node* p)

- 4 pts Blank/Incorrect

are given, and ask yourself which node *would* be easy to remove. Is there a way to remove that node, while at the same time preserving its data? The following function is intended to effectively delete the node pointed to by `p` from a linked-list with a dead head. Complete the code below so that it takes constant time to do this operation. You may assume that `p` points to a valid data node (not the dead head).

```
void List::delNode(Node *p){
  //"delete the node pointed to by p" = delete p->next
  /* set up a pointer to a node that would be easy to remove */
  Node * t = p->next;

  /* preserve the data contained in that node */
  p->data = p->next->data;

  /* fix pointers */
  p->next = p->next->next;

  /* free appropriate memory */
  delete t;
}
```

(i) (2 points) You probably noticed that this algorithm fails if the input parameter `p` points to a particular node.

i. The code fails if `p` is a pointer to what node?

```
The last node, because p->next = NULL,
and p->next->data is undefined behaviour.
```

ii. Describe how you would change the design of the linked list so that the algorithm above would work for any valid `p` (where "valid" means that it points to a node with data).

```
Since there's only one edge case, adding a guard saying
if(p->next == NULL) {
  //walk the list to get node before p
  //set that node's next to NULL, and free p
} else {
  //do the same thing
}
is enough.
```

(j) (3 points) This mechanism for removing a node from a list is commonly referred to as a constant time "hack" because it exhibits some important disadvantages. Choose the applicable disadvantages from the list below.

1.9 1i) analyze flaw **1 / 2**

✓ **+ 1 pts** **Tail node**

+ 1 pts Valid description of design changes to fix algorithm (e.g. circular linked list, dummy tail, ...)

+ 0 pts Blank/Incorrect

We would want to change the list

- ✓ If `p->data` is a large object, the execution of the function may be very slow.
- ○ The compiler cannot optimize the amount of memory needed by the `Node`.
- ✓ In order to work, the assignment operator must be implemented for data's type.
- ○ There could be be a memory leak.
- ○ The implementation requires a dead head.
- ○ The constant time code will only work on lists with more than one data element.
- ✓ An iterator over the list may be invalidated.
- ✓ This code may result in a segmentation fault due to dereferencing a `NULL` pointer.

1.10 1j) hack **2.5 / 3**

- **0 pts** Correct
- **1 pts** Missing bullet 1: If p->data is a large object...
- **1 pts** Missing bullet 3: In order to work, the assignment...
- **1 pts** Missing bullet 7: An iterator over the list may be...

✓ **- 0.5 pts 4 bullet points selected**

- **1 pts** 5 bullet points selectd
- **3 pts** Blank/Incorrect
- **1.5 pts** More than 5 bullet points selected

2. **Structured Sorting (22 points).**

We want to sort a sequence of numbers using a stack. We imagine the numbers arrive at the stack in their input order. For example, if the input is 5,1,4,2,3 then 5 arrives first and 3 arrives last. When a number arrives, we must push it onto the stack and then we can perform any number of pop operations (including none). Each pop operation not only pops the top number off of the stack it also outputs the number. Pops on an empty stack are invalid. We want the sequence of numbers that are output to be the original input sequence in non-decreasing order.

For example, the following operation sequence sorts 5,1,4,2,3: $IIOIIOIOOO$ where I means "push In" and O means "pop Out".

(a) (2 points) Some operation sequences attempt to pop from an empty stack or end before the stack is empty. These are called *invalid* sequences. Describe two simple conditions that together ensure that an operation sequence is valid.

At any point of the sequence you cannot have more Os before Is

The sequence must have the same number of Is and Os if you want to end

(b) (2 points) What input sequence using the numbers 1,2,3 cannot be sorted in this way; meaning no sequence of operations will sort the given input.

2,3,1

(c) (8 points) Prove that it is impossible to sort an input sequence in which the number b is before the number c which is before the number a and $a < b < c$.

```
Input: b,c,a
Output: a,b,c

Steps:
1. With output of a,b,c, a has to be pop first. To
pop a, we have to push a. With input of b,c,a, we
have to push b,c first.
2. We can't pop b,c while pushing because a is
supposed to come first in output ordering, so the
order on the stack when the stack is full is
b,c,a.
3. Once we popped a, the top of stack is c, since
b was pushed first, so c must be the next to pop.

Since c comes after b, our output is not sorted.
If we decide to pop b before c, we'll contradict
step 2's logic and a is not in order. Therefore,
this sequence can't be sorted with just a stack.
```

2.1 2a) assure validity 2 / 2

✓ **+ 1 pts** **For all prefixes of the sequence, the number of \I's is at least the number of \O's.**

✓ **+ 1 pts** **The number of \I's in the whole sequence equals the number of \O's.**

+ 0 pts Blank/Incorrect

2. **Structured Sorting (22 points).**

We want to sort a sequence of numbers using a stack. We imagine the numbers arrive at the stack in their input order. For example, if the input is 5,1,4,2,3 then 5 arrives first and 3 arrives last. When a number arrives, we must push it onto the stack and then we can perform any number of pop operations (including none). Each pop operation not only pops the top number off of the stack it also outputs the number. Pops on an empty stack are invalid. We want the sequence of numbers that are output to be the original input sequence in non-decreasing order.

For example, the following operation sequence sorts 5,1,4,2,3: $IIOIIOIOOO$ where I means "push In" and O means "pop Out".

(a) (2 points) Some operation sequences attempt to pop from an empty stack or end before the stack is empty. These are called *invalid* sequences. Describe two simple conditions that together ensure that an operation sequence is valid.

At any point of the sequence you cannot have more Os before Is

The sequence must have the same number of Is and Os if you want to end

(b) (2 points) What input sequence using the numbers 1,2,3 cannot be sorted in this way; meaning no sequence of operations will sort the given input.

2,3,1

(c) (8 points) Prove that it is impossible to sort an input sequence in which the number b is before the number c which is before the number a and $a < b < c$.

```
Input: b,c,a
Output: a,b,c

Steps:
1. With output of a,b,c, a has to be pop first. To
pop a, we have to push a. With input of b,c,a, we
have to push b,c first.
2. We can't pop b,c while pushing because a is
supposed to come first in output ordering, so the
order on the stack when the stack is full is
b,c,a.
3. Once we popped a, the top of stack is c, since
b was pushed first, so c must be the next to pop.

Since c comes after b, our output is not sorted.
If we decide to pop b before c, we'll contradict
step 2's logic and a is not in order. Therefore,
this sequence can't be sorted with just a stack.
```

2.2 2b) impossible sequence **2 / 2**

✓ **- 0 pts** **2, 3, 1**

- 2 pts Blank/Incorrect

2. **Structured Sorting (22 points).**

We want to sort a sequence of numbers using a stack. We imagine the numbers arrive at the stack in their input order. For example, if the input is 5,1,4,2,3 then 5 arrives first and 3 arrives last. When a number arrives, we must push it onto the stack and then we can perform any number of pop operations (including none). Each pop operation not only pops the top number off of the stack it also outputs the number. Pops on an empty stack are invalid. We want the sequence of numbers that are output to be the original input sequence in non-decreasing order.

For example, the following operation sequence sorts 5,1,4,2,3: $IIOIIOIOOO$ where I means "push In" and O means "pop Out".

(a) (2 points) Some operation sequences attempt to pop from an empty stack or end before the stack is empty. These are called *invalid* sequences. Describe two simple conditions that together ensure that an operation sequence is valid.

At any point of the sequence you cannot have more Os before Is

The sequence must have the same number of Is and Os if you want to end

(b) (2 points) What input sequence using the numbers 1,2,3 cannot be sorted in this way; meaning no sequence of operations will sort the given input.

2,3,1

(c) (8 points) Prove that it is impossible to sort an input sequence in which the number b is before the number c which is before the number a and $a < b < c$.

```
Input: b,c,a
Output: a,b,c

Steps:
1. With output of a,b,c, a has to be pop first. To
pop a, we have to push a. With input of b,c,a, we
have to push b,c first.
2. We can't pop b,c while pushing because a is
supposed to come first in output ordering, so the
order on the stack when the stack is full is
b,c,a.
3. Once we popped a, the top of stack is c, since
b was pushed first, so c must be the next to pop.

Since c comes after b, our output is not sorted.
If we decide to pop b before c, we'll contradict
step 2's logic and a is not in order. Therefore,
this sequence can't be sorted with just a stack.
```

2.3 2c) proof of impossible 8 / 8

✓ **- 0 pts** **Valid proof**

- 2 pts Minor leap without justification

- 4 pts Major leap without justification

- 8 pts Blank/Incorrect

Suppose we add the choice of a queue to our sorting algorithm. Now we can use the operations I, O, E, and D, where I and O push and pop the stack, while E and D enqueue and dequeue the queue (dequeue also outputs the number that is dequeued). For example, we can sort 5,1,4,2,3 using: $IEIEEDDDOO$.

(d) (2 points) What input sequence using the numbers 1,2,3,4 is sorted by the sequence of operations $EIEIOODD$?

3,2,4,1

(e) (4 points) Is it possible to sort all input sequences using such a stack and queue? If yes then explain how. If no, give a permutation that cannot be output.

```
A sample sequence:
4,2,5,1,3
```

(f) (4 points) Suppose we add two new operations.
S : pop a number from the stack (without output) and enqueue it into the queue; and
Q : dequeue a number from the queue (without output) and push it onto the stack.
Is it possible to sort all input sequences using the operations I, O, E, D, S, and Q? If yes then explain how. If no, give a permutation that cannot be output.

```
Definition:
X: input array, with integers from [m,n] unsorted
Y: output array, with integers from [m,n] sorted
k: arbitrary integer from X
j: the largest number in X that is still smaller than k
i: current position while iterating through X, with max i being X.size-1
f: smallest previously sorted integer, ie top of stack (this definition used in
inductive step)
g: largest previously sorted integer, ir back of queue (this definition used in
inductive step)

We have an array input X, and we want output array Y. We know from part e that if
we have a sorted queue from k (front) to n (back), and a sorted stack of m (top)
to j (bottom), we can pop all integers from m to j, and dequeue all integers from
k to n, to have sorted output array. Therefore, this is an inductive proof of how
we can use operations to keep queue and stack sorted when receiving new integer
from input array.

Base case: Since we referenced 4 variables in our introduction, we can say that
the base case input has 4 integers. Suppose they are a,b,c,d for simplicity, where
a<b<c<d. There are 24 possible input arrangements, and therefore 24 possible ways
to create sorted queue {c,d} and stack {a,b}, written here using the previously
defined shorthands.
```

2.4 2d) input sequence 2 / 2

✓ **- 0 pts** **Correct: 3, 2, 4, 1**

- 2 pts Blank/Incorrect

Suppose we add the choice of a queue to our sorting algorithm. Now we can use the operations I, O, E, and D, where I and O push and pop the stack, while E and D enqueue and dequeue the queue (dequeue also outputs the number that is dequeued). For example, we can sort 5,1,4,2,3 using: $IEIEEDDDOO$.

(d) (2 points) What input sequence using the numbers 1,2,3,4 is sorted by the sequence of operations $EIEIOODD$?

3,2,4,1

(e) (4 points) Is it possible to sort all input sequences using such a stack and queue? If yes then explain how. If no, give a permutation that cannot be output.

```
A sample sequence:
4,2,5,1,3
```

(f) (4 points) Suppose we add two new operations.
S : pop a number from the stack (without output) and enqueue it into the queue; and
Q : dequeue a number from the queue (without output) and push it onto the stack.
Is it possible to sort all input sequences using the operations I, O, E, D, S, and Q? If yes then explain how. If no, give a permutation that cannot be output.

```
Definition:
X: input array, with integers from [m,n] unsorted
Y: output array, with integers from [m,n] sorted
k: arbitrary integer from X
j: the largest number in X that is still smaller than k
i: current position while iterating through X, with max i being X.size-1
f: smallest previously sorted integer, ie top of stack (this definition used in
inductive step)
g: largest previously sorted integer, ir back of queue (this definition used in
inductive step)

We have an array input X, and we want output array Y. We know from part e that if
we have a sorted queue from k (front) to n (back), and a sorted stack of m (top)
to j (bottom), we can pop all integers from m to j, and dequeue all integers from
k to n, to have sorted output array. Therefore, this is an inductive proof of how
we can use operations to keep queue and stack sorted when receiving new integer
from input array.

Base case: Since we referenced 4 variables in our introduction, we can say that
the base case input has 4 integers. Suppose they are a,b,c,d for simplicity, where
a<b<c<d. There are 24 possible input arrangements, and therefore 24 possible ways
to create sorted queue {c,d} and stack {a,b}, written here using the previously
defined shorthands.
```

2.5 2e) prove whether or not SQ is enough **4 / 4**

✓ **- 0 pts** **Correct: no + a valid permutation that cannot be output (e.g. 4, 5, 2, 3, 1)**

- 4 pts Blank/Incorrect

2.6 2f) how about now? 4 / 4

✓ **- 0 pts** **Answers yes with adequate explanation**

- 1 pts Minior Error in Explaination

- 2 pts Inadequate explanation

- 4 pts Blank/Incorrect

Suppose we add the choice of a queue to our sorting algorithm. Now we can use the operations I, O, E, and D, where I and O push and pop the stack, while E and D enqueue and dequeue the queue (dequeue also outputs the number that is dequeued). For example, we can sort 5,1,4,2,3 using: $IEIEEDDDOO$.

(d) (2 points) What input sequence using the numbers 1,2,3,4 is sorted by the sequence of operations $EIEIOODD$?

3,2,4,1

(e) (4 points) Is it possible to sort all input sequences using such a stack and queue? If yes then explain how. If no, give a permutation that cannot be output.

```
A sample sequence:
4,2,5,1,3
```

(f) (4 points) Suppose we add two new operations.
S : pop a number from the stack (without output) and enqueue it into the queue; and
Q : dequeue a number from the queue (without output) and push it onto the stack.
Is it possible to sort all input sequences using the operations I, O, E, D, S, and Q? If yes then explain how. If no, give a permutation that cannot be output.

```
Definition:
X: input array, with integers from [m,n] unsorted
Y: output array, with integers from [m,n] sorted
k: arbitrary integer from X
j: the largest number in X that is still smaller than k
i: current position while iterating through X, with max i being X.size-1
f: smallest previously sorted integer, ie top of stack (this definition used in
inductive step)
g: largest previously sorted integer, ir back of queue (this definition used in
inductive step)

We have an array input X, and we want output array Y. We know from part e that if
we have a sorted queue from k (front) to n (back), and a sorted stack of m (top)
to j (bottom), we can pop all integers from m to j, and dequeue all integers from
k to n, to have sorted output array. Therefore, this is an inductive proof of how
we can use operations to keep queue and stack sorted when receiving new integer
from input array.

Base case: Since we referenced 4 variables in our introduction, we can say that
the base case input has 4 integers. Suppose they are a,b,c,d for simplicity, where
a<b<c<d. There are 24 possible input arrangements, and therefore 24 possible ways
to create sorted queue {c,d} and stack {a,b}, written here using the previously
defined shorthands.
```

{a,b,c,d}: EIEEQ
{a,b,d,c}: EIIESQ
{a,c,b,d}: EEIQE
{a,c,d,b}: EEEIQ
{a,d,b,c}: EEIEQQS
{a,d,c,b}: EEEIQQS
{b,a,c,d}: IIEE
{b,a,d,c}: IIEEQS
{b,c,a,d}: IEIE
{b,c,d,a}: IEEI
{b,d,a,c}: IEIEQS
{b,d,c,a}: IEEIQS
{c,a,b,d}: EEIEQSQQS
{c,a,d,b}: EEEIQSQQS
{c,b,a,d}: EIIE
{c,b,d,a}: EIEI
{c,d,a,b}: EEEIQSQSQ
{c,d,b,a}: EEII
{d,a,b,c}: EEEEQSQSQQSQSQ
{d,a,c,b}: EEEIQSQ
{d,b,a,c}: EIIEQS
{d,b,c,a}: EIEIQS
{d,c,a,b}: EEEIQSQSQQS
{d,c,b,a}: EEIIQS

As a sidenote, there also are base cases where X's size is 0,1,2,3 which are not relevant to the inductive step, but still key to the proof as a whole. Since it is a small amount of cases, we can exhaustively prove that we can obtain Y. As previously define, a<b<c. For step, we can't divide X into a neat queue and stack pair, but we can just show directly that we can achieve Y.
{}: empty list is sorted by definition
{a}: list with one member is sorted definition, so just IO
{a,b}: ED
{b,a}: IO
{a,b,c}: EEEDDD
{a,c,b}: EEIDOD
{b,a,c}: IIEOOD
{b,c,a}: EEIODD
{c,a,b}: IEIDOO
{c,b,a}: IIIOOO

Inductive hypothesis: For all past numbers in input array from X[0] to X[i], we assume that they are sorted such that all integers are separated into two groups using an arbitrary k selected from X[0] to X[i], with integers from m to j is sorted in stack, and integers from k to n is sorted in queue.

Inductive step: For X[i+1], it has to either be (1)[m,f], (2) (f,j] , (3) [k,g), (4) [g,n]. We can break inductive step into four cases.

Case 1)
Since it's the smallest yet, it has to go on top of stack. Simply do I operation on X[i+1]. After this, our stack should look like {X[i+1],f...j}, and our queue is not modified. Inductive hypothesis is restored.

Case 2)
Since it is somewhere in the stack, we can do S operation on stack elements until X[i+1] is more the previously popped element (we will call this w) and less than or equal to the current top of stack (we will call this v). We can do I operation on X[i+1]. At this point, the queue has elements such that {k...n,f...w}. Then we can do QS operations (which simply puts front of queue to back of queue) on queue elements until w is in front of queue, then we do Q operation on w. After doing Q operation on w, the queue should look like {k...n,f...z}, where z is the next smallest integer from w. We can iterate QS and Q operations until all of f to z is moved back to stack. After this, our stack should look like {f...z,w,X[i+1],v...j}, and our queue should look like {k...n}. Inductive hypothesis is restored.

Case 3)
Since it is somewhere in the queue, we can do QS operation on queue elements until X[i+1] is more or equal to the previously operated queue element (we will call this w) and less than the current front of queue (we will call this v). We can do E operation on X[i+1]. At this point, the queue has elements such that {v...g,k...w,X[i+1]}. Then, we can keep doing QS operation on queue elements up until g. After this, queue should look like {k...w,X[i+1],v...g}, and stack is not modified. Inductive hypothesis is restored.

Case 4)
Since it's the largest yet, it has to go to back of queue. Simply do E operation on X[i+1]. After this, our queue should look like {k...g,X[i+1]}, and our stack is not modified. Inductive hypothesis is restored.

Conclusion: We can restore the inductive hypothesis for all 4 cases of X[i+1]. Therefore, we can say that the inductive hypothesis is proved and we can conclude that is a way to keep queue and stack sorted when receiving new integer from input array, which means that it is always possible to give output Y by popping all integers from m to j from stack, and dequeuing all integers from k to n from queue.

3.1 findMaxHelper correct 2 / 5

- **0 pts** Correct proof

✓ **- 1 pts** **Missing/Incorrect explanation of base case: right-left=1**

✓ **- 1 pts** **Missing/Incorrect inductive hypothesis: true for (right-left) < k**

- **3 pts** Missing/Incorrect inductive step explanation: prove for k

✓ **- 1 pts** **Minor error in reasoning**

Claim: `findMaxHelper(A, left, right)` returns the maximum element in `A[left..right-1]`.

Proof: Base case: Choose n = 1, for n=1 max1 and max2 findMaxHelper(A, 0, 1) will just return the only element in the array A, and since it's the only element, it is also the largest.

Inductive Hypothesis: Assume array with length n $\leq$ k where k is an arbitrary integer, findMaxHelper(A, 0, k) will return the maximum element in that array.

Inductive Step: IH must also be true for k+1 to successfully prove. In the recursive function findMaxHelper we can see that max1 will recursive be:
findMaxHelper(A, 0, k+1)
findMaxHelper(A, 0, (k+1)/2)
...
findMaxHelper(A, 0, 2)
findMaxHelper(A, 0, 1)
and since max2 is calculated after max1 it will be:
findMaxHelper(A, 1, 2);
...
findMaxHelper(A, (k+1)/2, k+1);

By our inductive hypothesis, since we know findMaxHelper will return the largest value in the array within the given length if length $\leq$ k. we know that
int max1 = findMaxHelper(A, 0, (k+1)/2) and
int max2 = findMaxHelper(A, (k+1)/2, k+1)
will return the max element in the first half (max1) and second half(max2) of the original array of length k+1. (Since both of their length are shorter than k) Afterwards, the subsequent inequality statement will compare max1 and max2 to return the maximum element of the array from 0 to k+1, which proves our inductive hypothesis to be correct.

□

3.2 findMaxHelper recurrence **5 / 5**

✓ **+ 1 pts** **Correct base case: T(1) <= b or T(1) = 1 or ...**

✓ **+ 2 pts** **Correct recurrence relation: T(n) <= 2T(n/2) + c for n > 1**

✓ **+ 2 pts** **Correct closed form: bn + cn - c (for some constants b and c, e.g. 2n-1)**

+ 0 pts Blank/Incorrect

- 0.5 pts minor error

ii. (5 points) Let $T(n)$ be the run time of this algorithm. Express $T(n)$ as a recurrence relation. Use repeated substitution (ignore floors/ceilings when calculating $n/2$) to determine a closed-form solution for $T(n)$, i.e., express $T(n)$ as a non-recursive function of n with no summations.

Recurrence:
T(1) = 1 since it's constant time when n = 1
T(n) = 2T(n/2) + C if n > 1

T(n) = 2T(n/2) + C

= 2(2T(n/2) + C) + C
= 4T(n/4) + 3c
= 4(2T(n/8) + C) + 2C
= 8T(n/8) + 7C
= $2^kT(n/2^k) + (2^k - 1)C$
= $nT(1) + (n - 1)c$

T(n) = n + nc - c

(a) (8 points) The following code finds the length of the longest subsequence of strictly increasing numbers in a given array:

```
// Returns the length of the longest subsequence of strictly increasing
// elements in the array A.  A subsequence of array A is not necessarily
// contiguous.
int LISlength(int *A, int n) {
  int *LISendAt = new int[n];
  int best = 0;
  for( int i=0; i<n; i++ ) {
    int r = 1;
    for( int j=0; j<i; j++ ) {
      if( A[j] < A[i] && r < LISendAt[j] + 1 )
        r = LISendAt[j] + 1;
```

3.3 LIS correctness **8 / 8**

✓ **- 0 pts Correct**

- 1 pts Missing/Incorrect base case

- 2 pts Missing/Incorrect inductive hypothesis

- 5 pts Missing/Incorrect inductive step

- 1 pts Minor error/missing case

- 8 pts Blank/Incorrect