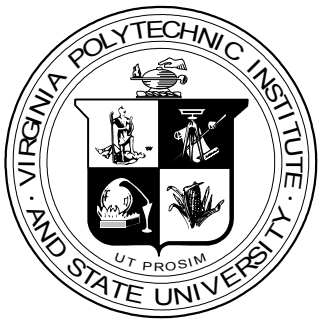

Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 6 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- **Note that either failing to return this test or discussing its content with a student who has not taken it is a violation of the Honor Code.**

Do not start the test until instructed to do so!

Name Solution
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed

1. [20 points] Circle TRUE or FALSE according to whether the statement is true or false:

a) $1000 - 7n + 2n^2$ is $O(n^3)$ (TRUE) FALSE

Directly from a theorem in the notes, 1000 – 7n + 2n^2 is O(n^2) since that's the dominant term. But big-O is just an upper bound, and clearly n^2 is O(n^3).

Or, using limits we can show it this way:

$$\lim_{n\to\infty}\frac{1000+7n+2n^2}{n^3}=\lim_{n\to\infty}\left(\frac{1000}{n^3}+\frac{7}{n^2}+\frac{2}{n}\right)=0$$

b) $5n^2 + 3n + 2$ is $\Omega((\log n)^2)$ (TRUE) FALSE

From the same theorem, 5n^2 + 3n + 2 is O(n^2) and we should suspect it's Theta(n^2). That's easily verified by taking the limit of (5n^2 + 3n + 2)/(n^2) as n goes to infinity and seeing it's 5. But how does that help, since this question is about Omega?

Well, remembering that Omega is about lower bounds, and remembering the hierarchy of complexity classes from the notes, n is Omega(log n) and so n^2 is Omega((log n)^2).

Alternatively, use the limit theorem and l'Hopital's Rule several times:

$$\lim_{n\to\infty}\frac{5n^2+3n+2}{(\log n)^2}=\lim_{n\to\infty}\frac{10n+3}{2(\log n)/(n\ln 10)}=\lim_{n\to\infty}\frac{(10n^2+3n)\ln 10}{2\log n}=\lim_{n\to\infty}\frac{(20n+3)\ln 10}{2/(n\ln 10)}=$$
$$\lim_{n\to\infty}\frac{(20n^2+3n)(\ln 10)^2}{2}=\infty$$

c) $3n^2 + 100(n \log n)^3$ is $\Theta(n^2)$ TRUE (FALSE)

First you need to expand this as 3n^2 + 100 n^3 (log n)^3. Clearly the second term is dominating the first since n^3 dominates n^2. So our theorem would say this is O(n^3 (log n)^3), which is Omega(n^2), and so this cannot be Theta(n^2).

Or, you can use limits again:

$$\lim_{n\to\infty}\frac{3n^2+100n^3(\log n)^3}{n^2}=\lim_{n\to\infty}\left(3+100n(\log n)^3\right)=\infty$$

d) $17n(2^n + n^2)$ is $\Theta(2^n)$ TRUE (FALSE)

Multiplying it out we have 17n 2^n + 17n^3. Now from our list of complexity classes, 2^n will dominate any power of n and so this is O(n 2^n) and that is clearly Omega(2^n).

Or, once again, using limits:

$$\lim_{n\to\infty}\frac{17n^3+17n2^n}{2^n}=\lim_{n\to\infty}\left(\frac{17n^3}{2^n}+17n\right)$$

Now the first term goes to zero (using l'Hopital's Rule) and the second goes to infinity, so this limit is infinite.

e) $\sum_{i=1}^{n^2} i \log n$ is $\Omega(n^3)$ TRUE FALSE

You really need to work out the summation to be sure of this one. As far as the summation itself goes, the n is a constant so you can write it as (log n) * Sum(i) and then apply the summation formula:

$$\log n\times\sum_{i=1}^{n^2} i=\log n\times\frac{n^2\left(n^2+1\right)}{2}$$

Now this is going to have the dominant term n^4 (log n) which is clearly Omega(n^3).

This can also be shown, tediously, using limits.

2. [20 points] The following algorithm prints an integer as a function of an input integer n, assumed to be positive.

```
read n;                     // 1 operation
x = 7 * n;                  // 2 operations
y = x + 5 * n;              // 3 operations
while (x + y > 0) {         // 2 operations for loop test
    x = x - 6;              // 2 operations
    y = y + 3;              // 2 operations
}
print x * y;                // 2 operations
```

We charge for operations at a rate of one arbitrary time unit for each assignment, arithmetic operation, comparison, integer input, and integer output.
Assume that n is a positive integer. Compute the time complexity function T(n) for the above algorithm. Also, determine the asymptotic (big-Θ) of T(n) as a simple function of n.

The only tricky thing here is the number of times the loop body is executed. Given the second and third statements, initially x + y equals 19n. The cumulative effect of the two statements inside the loop body is to decrease x + y by 3 on each pass. So, x + y will have the values 19n, 19n – 3, 19n – 6, etc. Roughly speaking, the loop body will execute 19n/3 times. If 19n is a multiple of 3, that will be exact. Otherwise, the correct value would be 19n/3 + 1 (assuming an integer divide is used). Fortunately there's an easy way to express this precisely – the ceiling function.

If you just add up the costs indicated above you get:

`T(n) = 1 + 2 + 3 + ceiling(19n/3)*6 + 2 = 8 + 6*ceiling(19n/3)`

If you assume 19n/3 is an integer then you have `T(n) = 8 + 38n`.

The analysis above ignores the "extra" execution of the loop test that causes the loop to terminate. Counting that as well just adds 2 operations.

Obviously, either way you analyze it, this is Theta(n).

3. [15 points] Consider the LinkedList and NodeT template interfaces given below:

```
template <class Data> class LinkedList {
private:
   NodeT<Data>* Head;
   NodeT<Data>* Tail;
public:
   bool  Insert(const Data&);
   bool  Append(const Data&);
   bool  Delete(const Data&);
   LinkedList SubList(Data&);
   Data* Find(const Data&);
   ~LinkedList();
};
```

```
template <class Data> class NodeT {
public:
   Data         Element;
   NodeT<Data>* Next;

   NodeT(Data Elem = Data(),
         NodeT<Data>* N = NULL);
};
```

Write the body of the public function `Find()`, which returns a pointer to the first data element in the list that matching its parameter or NULL if there is no such node. Your implementation must conform to the interface given below. You may assume that the implementation of `Data` provides overloads for any relational operators you need.

```
template <class Data>
Data* LinkedList<Data>::Find(const Data& Target) {

   NodeT<Data> *Curr = Head;           // slap a temp pointer on first node

   while ( Curr != NULL ) {            // quit when walk off end of list

      if ( Target == Curr->Element )   // check for match
         return ( &(Curr->Element) );  //    return address of data element

      Curr = Curr->Next;               // step to next node, if any
   }

   return NULL;                        // no match, so return NULL as spec'd
}
```

Notes:

- **Nothing above says that the list is sorted.**
- **Nothing implies that the head and/or tail nodes are dummy nodes (i.e., not storing data).**
- **You MUST handle the case where the list is empty, in which case Head would be NULL.**
- **You MUST NOT violate the encapsulation of the list by returning a pointer to a NODE rather than a pointer to a data element. Of course, the specified return type for Find() should have made that moot; unfortunately it did not.**

4. [15 points] Consider the LinkedList and NodeT template interfaces given below (same as problem 3):

```
template <class Data> class LinkedList {
private:
   NodeT<Data>* Head;
   NodeT<Data>* Tail;
public:
   bool  Insert(const Data&);
   bool  Append(const Data&);
   bool  Delete(const Data&);
   LinkedList SubList(Data&);
   Data* Find(const Data&);
   ~LinkedList();
};
```

```
template <class Data> class NodeT {
public:
   Data          Element;
   NodeT<Data>* Next;

   NodeT(Data Elem = Data(),
         NodeT<Data>* N = NULL);
};
```

Write the body of the public function `SubList()`, which returns a new list that is a **copy** of the sublist of the original list starting at the first node holding data matching the given parameter and going to the end of the original list. The original list is not modified.

```
template <class Data>
LinkedList LinkedList<Data>::SubList(const Data& Target) {

   NodeT<Data>    *Curr = Head;          // slap a temp pointer on head node
   LinkList<Data> sList;                 // create empty list

   while ( (Curr != NULL ) &&            // stop if walk off end of list
           (Curr->Element != Target) ) { //      OR if match is found

      Curr = Curr->Next;                 // step to next node, if any
   }

   while ( Curr != NULL ) {              // stop if walk off end of list

      sList.Append(Curr->Element);       // append node holding current data
                                         //   element to the sublist

      Curr = Curr->Next;                 // step to next node, if any
   }
   return sList;                         // return the sublist you built
}
```

Notes:

- **Again, you must deal with the possibility the list is empty.**
- **You must also deal with the possibility that the Target is not found, in which case you return an EMPTY LIST, not NULL.**
- **You cannot call Find() to do the search and set Curr because Find() returns a pointer to the DATA element, NOT to the NODE. You could call Find() to determine if Target is present in the list, and abort if it's not; however, that is inefficient since you'd just have to redo the search if Target was present.**
- **We didn't exactly say what the difference was between Insert() and Append(), but you should have chosen Append() to add elements to the sublist simply on the basis of its name.**
- **You should NOT be adding nodes to the sublist manually; i.e., NOT using the LinkedList interface to do it.**

5. Assume that an integer requires 4 bytes and a pointer requires 4 bytes. Assume that Data (e.g., problem 3) requires D bytes. We wish to implement a list of N records of Data.

   a. [5 points] How much space would the linked list of problem 3 require?

   Each NodeT<Data> object would require 4 bytes for a pointer and D bytes for the data element, and there would be N such node objects. AND, the linked list itself stores two more pointers so the total space cost would be:

   S(N) = N(D + 4) + 8.

   b. [5 points] How much space would a one-dimensional array implementation require?

   Each array cell would require D bytes, and there would be N cells. ALSO, the array name is really a pointer to the data (even if it's declared statically), and so that would require an additional 4 bytes:

   S(N) = ND + 4.

   c. [10 points] Assume N = 63. Estimate how much space a (randomized) skip list would require.

   For simplicity I'm assuming that there is no dummy tail node. Allowing for one would only change the results slightly.

   Each skip list node, including the head node, would have space for one data element (D bytes), an int to hold the node level number (4 bytes) and a pointer for the Forward array (4 bytes). So that gives us an estimate of:

   S(N) = 64*(D + 8) + space for Forward arrays

   How much space will the Forward arrays require? Recall that a skip list would expect to have 1/2 of its nodes in level 0, 1/4 in level 1, 1/8 in level 2, and so forth. Applying that here, with 63 data nodes, we'd expect to have the following data nodes:

Level	0	1	2	3	4	5
# Nodes	32	16	8	4	2	1
size of Forward[]	4	8	12	16	20	24

   The head node would also have to be in level 5. If we add all this up, we get an estimate of:

   S(N) = 64*(D + 8) + 32*4 + 16*8 + 8*12 + 4*16 + 2*20 + 2*24 = 64D + 1016

6. [10 points] Suppose we know that R is a transitive relation and that the following ordered pairs are in R:

(C,A), (D,C), (A,C), (E,A), (E,D)

Which of the following ordered pairs can we prove are **also** in R? (Circle all that apply.)

a. (A,A) **(A, C) and (C, A), so transitivity implies (A, A)**

b. (A,E)

c. (D,E)

d. (C,C) **(C, A) and (A, C), so transitivity implies (C, C)**

e. (E,C) **(E, A) and (A, C), so transitivity implies (E, C)**

f. (C,E)

g. (D,D)

h. (E,E)

i. (A,D)

j. (E,B)

None of the others can be established using what is given. If you had symmetry you could show some of the others; but you don't.