Part A [18 Points] There are 6 **multiple-choice** questions in this part.

A correct choice for one question will get you 3 points. To choose an answer, simply draw a circle around the bullet ( ). An incorrect answer, marking several answers or selecting nothing for a question will get you -1. If you believe that more than one answer is correct, select the best answer. The minimum total mark for this part is 0.

1. Here is an incorrect kind of pseudo code a student provided for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

```
declare a character stack
while ( more input is available)
{
   read a character
   if ( the character is a '(' )
      push it on the stack
   else if ( the character is a ')' and the stack is not empty )
      pop a character off the stack
   else
      print "unbalanced" and exit
}
print "balanced"
```

Which of these unbalanced sequences does the above code think is balanced?

a) ((())
b) ())(()
c) ((()))
d) (()))()

2. Suppose we have a circular array implementation of the queue class, with ten items in the queue stored at data [2] through data [11]. The current capacity is 42. Where does the insert method place the new entry in the array?

a) data[1]
b) data[2]
c) data[11]
d) data[12]

3. In the linked list implementation of the queue class, where does the insert method place the new entry on the linked list?

a) At the head
b) At the tail
c) After all other entries that are greater than the new entry.
d) After all other entries that are smaller than the new entry.

ID:

4. You have implemented the queue with a linked list, keeping track of a front node and a rear node with two reference variables. Which of these reference variables will change during an insertion into a NONEMPTY queue?

a) Neither changes
b) Only front changes.
c) Only rear changes.
d) Both change.

5. How many recursive calls will be made if the following method is called with 6?

```
void greeting(int n)
{
   if (n > 0)
   {
      System.out.println("Hello!");
      greeting(n+1);
   }
}
```

6

a) 5
b) 6
c) infinitely
d) 7

6. Consider the following List ADT operations(where the p_i's represent positions):
`p1=insertFirst(Ann), p2=insertAfter(p1,Alex), p3=insertBefore(p2,Lee), p4=insertFirst(Jim), remove(p4), swapElement(p1,p2), replaceElement(p3,Kim), p5=insertAfter(first(),Bob).`

Which of the following options describes the final list in a **correct** way?

a) (Ann,Alex,Lee, Jim)
b) (Alex,Kim,Lee,Bob)
c) (Alex,Bob,Kim, Ann)
d) (Lee,Kim, Alex,Jim)

[ann]
[ann, alex]
p1 p2
[ann, lee, alex]
p1 p3 p2
[jm, ann, lee, alex]
p4 p1 p3 p2
[ann, lee, alex]
p1 p3 p2
[alex, lee, ann]
p2 p3 p1
[alex, kim, ann]
p2 p3 p1
[alex, bob, kim, ann]

ID:

Part B [57 Points]

B.1 [33 Points] A palindrome is a string that reads the same forward and backward, capitalization and space are ignored. For example *deed, go dog, level*.... are palindromes.

a) **[7 Points]** Write an iterative algorithm in pseudo code that tests whether a string is a palindrome.

```
Algorithm palindreme(palindromeString)
    length ← lenght of palindromeString
    for i ← 0 to floor of(lenght/2) -1 do
        if charAt(i) ≠ charAt(lenght-1-i) of palindromeString the
            return false
    return true
    end of algorithm
```

b) **[10 Points]** Write a recursive algorithm in pseudocode that tests whether a string is a palindrome.

```
Algorithm palindrome(testString, index, lenght)
    if index > floor of (lenght/2)-1 then
        return true
    else if charAt(i) = charAt(lenght-1-i) in testString then
        palindrome(testString, index+1, lenght)
    else then
        return false
```

c) **[10 Points]** Describe how you could use a *Stack* or *Queue* to check whether a string is a palindrome, and write the pseudo code for that.

```
Algorithm palindrome(stringPal)
    for i ← 0 to floor of (lenght/2)-1 do
        push(charAt(i))
    for k ← ceiling of (lenght/2) to lenght-1 do
        if pop() ≠ charAt(k) then
            return false
    return true
    end
```

d) **[6 Points]** What are the worst-case runtimes of above algorithms (a), (b), (c) (use the big-Oh notation). Justify your answers.

algorithm a is O(n) since one for loop is used

algorithm b is O(n) because we do n/2 recursive calls

algorithm c is O(n) because we use 2 separate for loops

ID:

B.2 [24 Points] A group of students candidates went for a summer job interview with an IT company, here are some of the programming/algorithmic solutions they wrote. Describe the worst case running time of the following solutions in Big-Oh notation in terms of the variable *n*. Provide the answer in the box beside each algorithm. A correct choice for one question will get you 4 points.
Showing your work is not required (– don't spend a lot of time showing your work)

I.

```
void silly(int n, int x, int y) {
   if (x < y) {
      for (int i = 0; i < n; ++i)
         for (int j = 0; j < n * i; ++j)
            System.out.println("y = " + y);
   } else {
      System.out.println("x = " + x);
   }
}
```

$O(n^3)$

II.

```
void silly(int n) {
   for (int i = 0; i < n * n; ++i) {
      for (int j = 0; j < n; ++j) {
         for (int k = 0; k < i; ++k)
            System.out.println("k = " + k);
         for (int m = 0; m < 100; ++m)
            System.out.println("m = " + m);
      }
   }
}
```

$O(n^5)$

III.

```
void silly(int n) {
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j)
         System.out.println("j = " + j);
      for (int k = 0; k < i; ++k) {
         System.out.println("k = " + k);
         for (int m = 0; m < 100; ++m)
            System.out.println("m = " + m);
      }
   }
}
```

$O(n^2)$

ID:

IV.

```
void nero(int n) {
  for(int i=0; i < n; i++) {
    System.out.println("!");
  }
  for(int k=0; k < n*n*n; k++) {
    System.out.println("!");
  }
}
```

$O(n^3)$

V.

$200(m\ n)/3$

```
int vespasian(int n, int m) {
  for(int i=m; i > 0; i--) {
    for(int j=0; j < 200; j++) {
      for(int k=0; k < n; k += 3) {
        System.out.println("$");
  } } }
}
```

$O(mn)$

VI.

```
void claudius(int n) {
  for(int i=0; i < n; i++) {
    int j = 1;
    while(j < n) {
      System.out.println("j = " + j);
      j = j * 2;
  } }
}
```

$O(n \log n)$

ID: 40006118

Part C [25 Points]

For each of the **5** questions in this part, mark **T** if the given statement is **ALWAYS** true. Otherwise mark **F** and **justify** your answer. If you do not justify the FALSE case you will lose $^4/_5$ of the mark. There is **no penalty** for selecting a wrong answer. **Hint**: a correct counter example and/or correct specification will give you better marks. A correct answer will get you 5 points.

1. If $f(n) = 5n^2$ then $f(n) \in \Omega(2^n)$ ☐T ☒F

for $f(n)$ to be in $\Omega(g(n))$
$f(n)$ must be bigger of equal to $c\,g(n)$
in that case, $g(n)$ is 2^n ; therefore, there is no way $f(n) \in \Omega(2^n)$. It would work if it was big-O

2. The worst-case asymptotic running time for the best algorithm for finding something in a sorted array? is $O(n)$ ☐T ☒F

the best algorithm for finding something is the binary search. using this method, we halve the data to analyze every iteration. the worst case of asymptotic runtime is then $O(\log n)$ not $O(n)$

3. The worst-case asymptotic running time of finding and removing all values greater than 12 from a stack implemented with a linked-list (leaving the rest of the stack in its original order) is O(1) ☐T ☒F

this is false since we would have to pop every element. Remove all the values greater than 12 and push the remaining element in reverse order. this would have a complexity of at least $O(n)$.

4. Removing at the tail of a singly linked list is very efficient: performed in a constant time O(1). ☐T ☒F

Depending how one might implement a linked list, he might not include a pointer to the tail. To remove the tail, he would have to traverse the whole list of n elements and then remove the tail. this would be $O(n)$. Because of this exception, it is not always true, therefore I answered false

5. Having $f(n) \in O(g(n))$ *implies* $g(n) \in \Omega(f(n))$ ☒T ☐F

big-O implies that $f(n) \leq c\,g(n)$ for $n \geq n_0$

big-Omega implies that $g(n) \geq c\,f(n)$ for $n \geq n_0$

We can move the C from the big-Omega to the left handed side by using a division. $\frac{1}{C}$ is a constant so we can rewrite it to C_2

$f(n) \leq c\,g(n) \iff C_2\,g(n) \geq f(n)$ $C_1 = C_2$

Same equation

End of Exam!