Instructions: Opscan forms will be passed out in class and collected at class on Thursday, Nov 14. No late opscans will be accepted.

1. Which of the following statements could describe the base case of a recursive algorithm?

   1) The function returns its parameter value times the function performed on one-half its parameter value.
   2) If the parameter value is 100, the function returns zero.
   3) The function operates on floating point numbers.
   4) 1 and 2 only
   5) None of these

2. Of the choices given in question 1, which could describe the general (recursive) case of a recursive algorithm?

For questions 3 and 4, consider the following recursive function:

```
void PrintIt(int n) {

   if (n != 0) {                // Line 1
      cout << "Again" << endl; //      2
      PrintIt(n - 1);          //      3
   else {
      cout << n << endl;       //      4
      return;                  //      5
   }
}
```

3. Which line(s) relate to the base case?

   1) lines 2 and 3 2) lines 4 and 5 3) There is no base case.

4. Which line(s) relate to the general (recursive) case?

   1) lines 2 and 3 2) lines 4 and 5 3) There is no base case.

5. Are there any values for `x` that would cause an infinite recursion if the call `PrintIt(x)` were made?

   1) Yes, if `x > 0` 3) Yes, `x >= 0` 5) No
   2) Yes, if `x < 0` 4) Yes, `x <= 0`

---

6. Given the recursive function below, what is the value of `Func(2)`?

```
int Func( int n ) {
   if (n == 5)
      return 5;
   else
      return ( 2 * Func(n + 1) );
}
```

   1) 5 3) 40 5) None--the result is infinite recursion
   2) 20 4) 80 6) None of these

7. The following function sums the integers from `Low` through `Limit`, inclusive:

```
int Sum( int Low, int Limit ) {

    if ( Low > Limit ) return 0;
    if (_______________)
       return Limit;
    else
       return ( Low + Sum(Low + 1, Limit) };
}
```

What should the missing condition in the `if` statement be?

1) `Low == 1`
2) `Limit == 1`
3) `Low == Limit`
4) `Low > Limit`
5) `Low < Limit`
6) None of these

---

8. Given the recursive function:

```
void PrintArr(const int array[], int first, int last) {

   if (first > last)
      cout << "Done";
   else {
      PrintArr(array, first + 1, last);
      cout << array[first];
   }
}
```

which code segment below produces the same output as the function call: `PrintArr(arr, 0, 5)` ?

1)
```
for (int i = 0; i < 6; i++)
    cout << array[i];
cout << "Done";
```

2)
```
cout << "Done";
for (int i = 0; i < 6; i++)
    cout << array[i];
```

3)
```
for (int i = 0; i < 6; i++) {
    cout << array[i];
    cout << "Done";
}
```

4)
```
for (int i = 5; i >= 0; i--)
    cout << array[i];
cout << "Done";
```

5)
```
cout << "Done";
for (int i = 5; i >= 0; i--)
    cout << array[i];
```

6) None of these

---

9. Given the recursive function below, what is the value of the expression `Sum(5)` ?

```
int Sum( int n ) {
   if ( n < 8 )
      return ( n + Sum(n) );
   else
      return 2;
}
```

1) 5
2) 13
3) 20
4) 28
5) None--the result is infinite recursion
6) None of these

10. If the following function is called with a value of 2 for `n`, what is the resulting output?

```
void Quiz( int n ) {

   if (n > 0) {
      cout << 0;
      Quiz(n - 1);
      cout << 1;
      Quiz(n - 1);
   }
}
```

1) `00011011`
2) `11100100`
3) `10011100`
4) `01100011`
5) `001101`
6) None of these

For questions 11 through 15, consider the following function, `isThere()`, and the associated helper function `ValueInList()`, which are intended to indicate whether a specified `Value` occurs in a given array holding `Size` elements:

```
bool isThere(int Value, const int Array[], int Size) {

   return ValueInList(Value, Array, Size);
}

bool ValueInList(int Value, const int Array[], int Size) {

   if (Size <= ________)                              // Line 1
      return _________;                               // Line 2
   else if (Array[Size-1] == ___________)             // Line 3
      return _________;                               // Line 4
   else
      return ValueInList(Value, Array, ________);     // Line 5
}
```

11. How should the blank in Line 1 be filled?

1) `false`
2) `true`
3) `0`
4) `Value`
5) `-1`
6) None of these

12. How should the blank in Line 2 be filled?

1) `false`
2) `true`
3) `Size++`
4) `Size - 1`
5) `Value`
6) None of these

13. How should the blank in Line 3 be filled?

1) `Value`
2) `true`
3) `Array[0]`
4) `false`
5) `Array[Size]`
6) None of these

14. How should the blank in Line 4 be filled?

1) `false`
2) `true`
3) `Value`
4) `Size-1`
5) `Array[Size]`
6) None of these

15. How should the blank in Line 5 be filled?

1) `false`
2) `true`
3) `Size++`
4) `Size - 1`
5) `Value`
6) None of these

For questions 16 through 20, consider the following recursive function, which is intended to print the data elements from a `SList` object (with member functions as specified in the notes) in reverse order. Note: we assume that there is an `operator<<` for the type `Item`.

```
void RevPrint(SList& LL, ostream& Out) {

   if ( _________________ ) return;          // Line 1: terminate recursion

   LL.goToTail();                            // Line 2
   Item toPrint;                             // Line 3
   _____________________;                    // Line 4: obtain data to print
   Out << toPrint << endl;                   // Line 5

   _____________________;                    // Line 6: recursive case
}
```

16. How should the blank in Line 1 be filled?

1) `!LL.isEmpty()`
2) `LL.isEmpty()`
3) `LL.Head != NULL`
4) Nothing goes there
5) None of these

17. How should the blank in Line 4 be filled? (The comment may not tell the whole story.)

1) `toPrint = LL.Get()`
2) `LL.Advance()`
3) `LL.Delete(toPrint)`
4) Nothing goes there
5) None of these

18. How should the blank in Line 6 be filled?

1) `return`
2) `return RevPrint(LL, Out)`
3) `LL.RevPrint()`
4) `RevPrint(LL, Out)`
5) None of these

19. Does the function `RevPrint()` have any unfortunate side effect on the list `LL`?

1) Yes, it reverses the list.
2) Yes, it leaves the list empty.
3) Yes, it disconnects the list.
4) No.
5) None of these

20. What would happen if the parameter `LL` were passed by value instead of by reference?

1) The function would not work correctly, but there would not be a runtime error.
2) There would be a runtime error.
3) The function would work correctly, but it would make one copy of the list (in addition to the original).
4) The function would work correctly, but it would make at least M copies of the list (in addition to the original), where M is the number of elements in the list.
5) Nothing goes here

For questions 21 and 22, consider the following recursive function:

```
int findFactorPower(int D, int N) {

   if ( (D <= 0) || (N < 0) ) return 0;     // catch invalid parameters

   int Times;                               // # of times D goes into N

   if ( N % D == 0)                         // Does D go into N?
      Times = 1 + findFactorPower(D, N/D);  // Count it and see if it
                                            //  goes in again.
   else
      Times = 0;                            // Nope, don't check further.

   return Times;
}
```

21. How many recursive calls result from making the call: `findFactorPower(4, 48)` ?

1) `0`
2) `1`
3) `2`
4) `3`
5) `4`
6) None of these

22. What is returned from the call: `findFactorPower(4, 48)` ?

1) `0`
2) `2`
3) `3`
4) `4`
5) `12`
6) None of these

For questions 23 through 25, consider the Knapsack Problem stated in the course notes, and the recursive function given there for solving the problem. If there is a solution, it is a set of values; the function discovers the solution by performing a sequence of recursive calls until the goal sum is achieved, or is found to be impossible. Although the given function does not build a representation of the solution set, it could be easily modified to do so. In any case, the function does build the solution in a virtual sense, and each time the function is called it determines whether the set it is currently considering is a possible solution. Keeping that in mind… suppose that the function is called with an initial goal of 10, and the following array of "candidate" values:

```
int ray[5] = {3, 5, 7, 11, 13};
```

23. When the function is called initially (non-recursively), the solution set it is currently considering is:

1) `empty`
2) `{3}`
3) `{3, 5}`
4) `{3, 5, 7}`
5) `{3, 7}`
6) `{3, 5, 11}`
7) `{3, 5, 13}`
8) None of these

24. One of the recursive calls will be made when the current candidate solution set is `{3, 5, 13}`. What is the candidate solution set when the next recursive call is made?

1) `empty`
2) `{3}`
3) `{3, 5}`
4) `{3, 5, 7}`
5) `{3, 7}`
6) `{3, 5, 11}`
7) `{3, 5, 13}`
8) None of these

25. Suppose that the given function implementation is modified so that the recursive calls are managed as follows:

```
return ( Knapsack(ray, goal-ray[start], start+1, end) ||
         Knapsack(ray, goal, start+1, end) );
```

The effect of the change is that:

1) The function will no longer reach the correct conclusion in any case.
2) The function will reach the correct conclusion if there is a solution, but fail if there is not.
3) The function will reach the correct conclusion if there is no solution, but fail if there is one.
4) The function will still reach the correct conclusion in all cases.
5) The function will require more recursive calls.
6) The function will require fewer recursive calls.
7) 4 and 5 only
8) 4 and 6 only
9) None of these