Thomas Jefferson Invitational Open in Informatics 2012

Exam in Java

Contest Part I (Theoretical—Short Answer)

Explanation Document

Do not open until told to do so.

Rules and Instructions:

1. You will have 60 minutes to complete this section. This may not be enough time to complete the entire contest. Do not be alarmed if you do not finish every problem.
2. For each problem, write your answer clearly, neatly, and legibly in the space provided in the answer document.
3. For code analysis problems, assume that all necessary headers and libraries have been included, and that there are no compile or run-time errors.
4. Each problem will be weighted differently, for a total of 50 points. Partial credit may be awarded for partially-correct answers.
5. Work on these problems within your team.
6. Please direct all non-content-related questions to the proctors.
7. Good luck, and have fun!

Short Answer Problem 0

(2 points)

It is helpful to re-write the triangle in a more standard row-column form than the original triangle. If the original triangle looked like this:

Pascal's Triangle

row 0:						1					
row 1:					1		1				
row 2:				1		2		1			
row 3:			1		3		3		1		
row 4:		1		4		6		4		1	
row 5:	1		5		10		10		5		1

Than the final triangle should look like this:

Realligned Pascal's Triangle

row 0:	1					
row 1:	1	1				
row 2:	1	2	1			
row 3:	1	3	3	1		
row 4:	1	4	6	4	1	
row 5:	1	5	10	10	5	1

As can be confirmed by examination, each element (except for the first one) is the sum of the element directly above it and the one above it and to the left in the tree (that is, its row minus one and its column minus one). Of the five options presented, only C, `return pascal(row - 1, col - 1) * pascal(row - 1, col)`, adds those two values.

The correct code, then, should read:

```
public static int pascal(int row, int col){
    if(col == 0 || col == row)
        return 1;
    else
        return pascal(row - 1, col - 1) + pascal(row - 1, col); // *
}
public static void main(String[] args){
    System.out.println(pascal(9, 5));
}
```

Short Answer Problem 1

(3 points)

Oh, silly Albert. His code is almost right. However, he makes two main mistakes:

1. He looks for the maximum element at each pass. There are valid selection sorts that use this technique, but then he would have to iterate backwards through the array. Instead, he should look for the minimum.

2. He sets the last unsorted value to the maximum value found in his pass of the array but does not set the spot that held the previous maximum to the value at the previously unsorted index (phew!). Rather than swapping elements, he simply copies over elements from one part of the array to another part of the array.

More correct code should look like this:

```
static int[] numbers = {7, 3, 5, 2, 4};

static void selection_sort(int len) {
    for(int i = 0; i < len; i++) {
        int max_index = i;
        for(int j = i; j < len; j++)
            if(numbers[j] < numbers[max_index])
                max_index = j;
        int temp = numbers[i];
        numbers[i] = numbers[max_index];
        numbers[max_index] = numbers[i];
    }
}

public static void main(String[] args) {
    selection_sort(5);
    System.out.println(numbers[0] + " " + numbers[1] + " " + numbers[2]
        + " " + numbers[3] + " " + numbers[4]);
}
```

As it was, as his code when through the array, it first copied the `7` to the first index of the array (where it already was), then the `5` to the second index, then the `5` to the third index, then the `4` to the fourth index, and finally the `4` to the last index. In other words, it constructed a ”postfix-max” array where each element was the maxiumum of that element and every element to the right. The final array, which was the information requested, was, `[7, 5, 5, 4, 4]`.

Short Answer Problem 2

(3 points)

This was a fairly straightforward bit-string flicking problem. One important thing to remember for this problem and bit-string flicking problems in general is the order of operations. In Boolean algebra, AND has precedence over OR (as AND is analagous to multiplication and OR is analagous to addition). Also, the unary NOT has precedence over the binary operations.

In this problem, most of the operations were parenthesized. However, that unary NOT may have caused some confusion. The correct series of bit-string flicking operations should be:

$$(\mathrm{NOT}(01111)\,\mathrm{OR}(\mathrm{NOT}(01001)\,\mathrm{AND}\,01011)) = AB010\,\mathrm{AND}\,C10D1$$

$$(10000\,\mathrm{OR}(10110\,\mathrm{AND}\,01011)) = AB010\,\mathrm{AND}\,C10D1$$

$$(10000\,\mathrm{OR}\,00010) = AB010\,\mathrm{AND}\,C10D1$$

$$10010 = AB010\,\mathrm{AND}\,C10D1$$

$$10010 = (A\,\mathrm{AND}\,C)B0D0$$

If we compare the right-hand side of the equation to the left-hand side of the equation bit-by-bit, we find that both A and C (that is, Alex and Chloe) should go to the party (together, as it would seem) and D (Doug) is also going as a third-wheel. B (Brian), whose bit matches up with a zero, does not go the party.

Short Answer Problem 3

(4 points)

a) For reference, here is the code in question.

```
public static int function(int n){
    int a = 1,i = 1;
    for(i = 1;i <= n + 1;i++){
        int b = 0,j = 1;
        for(j = 1;j <= i + 1;j++)
            b = b + a;
        a = b;
    }
    return a;
}
public static void main(String[] args){
    System.out.println(function(3));
}
```

The input was small enough that we could simply step through the different statements of the function. As it turns out, there is also an explicit formula for `function(n)`: $(n+2)!$. In the inner loop, `a` is added to `b` `i + 1` times. In other words, we set `b` to `(i + 1)*a`. Then we set `a` to `b`, so the net effect of the loop is `a = (i + 1)*a`. As we loop `n + 1` times and each time, multiply `a` by `i + 1`, we get the factorial of `n + 2`. If this does not make sense, we suggest copying the program over and running it on your own machine for different values of `n`.

b) Again, here is the code for reference.

```
public static int function(int a, int b){
    if(a == 0)
        return 1;
    if(b == 0)
        return 2;
    if(a > 5)
        return -1;
    return function(a+b, Math.abs(a-b)) + function(a+b, Math.abs(a-b)-1);
}
public static void main(String[] args){
    System.out.println(function(2, 1));
}
```

There may be an explicit solution to this, but it is easier here to just run through the recursion. Using $f(a, b)$ as shorthand for `function(a, b)`, the recursion reads:

$$f(2,1) = f(3,1) + f(3,0) = f(4,2) + f(4,1) + 2 = f(6,2) + f(6,1) + f(5,3) + f(5,2) + 2$$

$$= -1 - 1 + f(8,2) + f(8,1) + f(7,3) + f(7,2) + 2 = -1 - 1 - 1 - 1 - 1 - 1 + 2 = -4$$

So, `function(2, 1)` evaluates to -4.

Short Answer Problem 4

(3 points)

The value returned in this function is x, so the variable we change in our loop should be x. We need both a and b to remain constant during the operation because we use them to increment x and c. Therefore, any option that sets a can ruled out.

The variable c is a multiple of b (as it is calculated by adding b multiple times). It is incremented by b repeatedly, so its value goes from b to 2*b to 3*b and so on. It is easy to see, then that a - c is the next value that x needs to multiplied by in order to determine the bth factorial. Answer choice C is the only option where this is done.

The correct code should read:

```
public static int buggy(int a, int b){
    int c = b;
    int x = a;
    while(a - c > 0)
    {
        x = x * (a-c); // *
        c = c + b;
    }
    return x;
}
```

Short Answer Problem 5

(4 points)

a) The expression $A + B \bullet C$ is TRUE under two conditions: A is TRUE or $B \bullet C$ is TRUE. The set of triples where A is TRUE is $(1, *, *)$ and $B \bullet C$ is TRUE for the triplets $(*, 1, 1)$. Explicitly, these triples are: $(1, 0, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, $(1, 1, 1$, and $(0, 1, 1)$.

b) This expression is similar to the first part, except that we want $B \bullet C$ to be FALSE so that $\overline{B \bullet C}$ can be TRUE. The triplets that make $B \bullet C$ FALSE are $(*, 0, 0)$, $(*, 0, 1)$, and $(*, 1, 0)$. Explicitly, the triplets are: $(1, 0, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, $(1, 1, 1)$, $(0, 0, 0)$, $(0, 0, 1)$, and $(0, 1, 0)$.

c) This is a complicated Boolean algebra expression. While the answer could be computed with a truth table, it is much simpler to simplify the Boolean algebra expression and then determine when that equivalent and simpler string is TRUE. The steps to simplify the expression, in order, should be:

$$\overline{\overline{(A + B) \bullet C} \bullet ((A \bullet C) + C) \bullet (B \bullet C + A) + \overline{(A \bullet C) + (A \bullet B)}}$$

$$\overline{\overline{(A + B) \bullet C} \bullet ((A \bullet C) + C) \bullet (B \bullet C + A)}((A \bullet C) + (A \bullet B))$$

$$(((A + B) \bullet C) + \overline{((A \bullet C) + C)} + \overline{(B \bullet C + A)})((A \bullet C) + (A \bullet B))$$

$$(A \bullet C + B \bullet C + \overline{C \bullet (A + 1)} + \overline{(B \bullet C + A)}) \bullet A \bullet (C + B)$$

$$(A \bullet C + B \bullet C + \overline{C} + \overline{(B \bullet C + A)}) \bullet A \bullet (C + B)$$

$$(A \bullet C + B \bullet C + \overline{C} + (\overline{B} + \overline{C}) \bullet \overline{A}) \bullet A \bullet (C + B)$$

$$(A \bullet A \bullet C + A \bullet B \bullet C + A \bullet \overline{C} + A \bullet (\overline{B} + \overline{C}) \bullet \overline{A}) \bullet (C + B)$$

$$(A \bullet C + A \bullet B \bullet C + A \bullet \overline{C}) \bullet (C + B)$$

$$A \bullet (C + B \bullet C + \overline{C}) \bullet (C + B)$$

$$A \bullet (C + B)$$

This is (almost) the simplest the equation gets (but we do not gain anything by distributing A, so it is left where it is). The expression is TRUE only when A is TRUE (the tuples $(1, *, *)$) and $B + C$ is TRUE (the tuples $(*, 1, 1)$, $(*, 1, 0)$, and $(*, 0, 1)$). The intersection of the sets, then, is explicitly $(1, 1, 1)$, $(1, 1, 0)$, $(1, 0, 1)$.

Short Answer Problem 6

(4 points)

This requiresa fairly straight-forward implementation of the standard algorism with numbers in other bases. The work comes out to:

a) $1001_2 + 110111_2 = 1000000_2$

```
  11111
    1001
+ 110111
--------
 1000000
```

b) $10010100111001_2 + 1010100111_2 = 10011111100000_2$

```
        111111
 10010100111001
+    1010100111
---------------
 10011111100000
```

c) $A3C01FF_{16} + 29DD92C1B_{16} = 28152E1A_{16}$

```
   111  11
    A3C01FF
+ 29DD92C1B
-----------
  2A8152E1A
```

d) $10010101_2 - 1100111_2 = 101110_2$

```
             <-- It’s hard to write here, but you need to carry ones
  10010101       to make 10’s just like 2nd grade but in binary.
-  1100111
----------
    101110
```

e) $427_8 \cdot 31_8 = 15477_8$

```
  12
  427
*  31
```

```
   -----
    427
+ 1505
-------
  15477
```

f) $\frac{10101000_2}{100_2} = 101010_2$

```
       101010
    ---------
100| 10101000
    -100
    ----
       101
      -100
      ----
         100
        -100
        ----
           00
```

g) $FFFFF_{16}^3 + 3 \cdot FFFFF_{16}^2 + 3 \cdot FFFFF_{16} + 1 = 1000000000000000_{16}$
This problem is somewhat unique. Rather than computing the value of the expression directly as was done with the other prolbems, it is easier here to write the expression as

$$(100000_{16} - 1)^3 + 3 \cdot (10000_{16})^2 + 3 \cdot (100000_{16} - 1) + 1$$

and evaluate that. This is an interesting problem, and hopefully, with the above examples as models, you the reader can do this on your own. Any complaints should be directed to Saketh Are.

Short Answer Problem 7

(4 points)

The original problem explained how the values were calculated for the first game. If we apply a similar process to the other games:

2^{nd} game:
Pickle: 161_{10}
Rhonun: 221_{10}

As he rolled a 161_{10}, Pickle must convert to base $(1+6+1) = 8$. In base-8, $161_{10} = 241_8$. His score, then, is $(2+4+1) = 7$. Rhonun has had a little more luck, as $(2+2+1) = 5$, $221_{10} = 1341_5$, and $(1+3+4+1) = 9$. So Rhonun beats Pickle in this round.

3^{rd} game:
Pickle: 734_{10}
Rhonun: 803_{10}

Pickle: $(7+3+4) = 14$, $734_{10} = 3A6_{14}$, and $(3+10+6) = 19$. Rhonun: $(8+0+3) = 11$, $803_{10} = 670_{11}$, and $(6+7+0) = 13$. This round goes to Pickle.

As you can see, Rhonun won the first two rounds, and Pickle won the third. Therefore, Rhonun was the overall winner as he won the most rounds. There was some confusion as to what “total score” meant. If it means “sum of all the individual scores,” then Pickle had a total score of $(2+7+19) = 28$ and Rhonun had a score of $(3+9+11) = 23$. If “highest total score” was interpreted as “of all the scores, which one was the highest,” then Pickle’s score of 19 was the best. Pickle wins either way, and both 28 and 19 were accepted as answers.

Short Answer Problem 8

(4 points)

This is an interesting problem. Though it is certainly possible to just walk through the program for the smaller test cases, one does simply walk through a loop of size $100,000,000$ or even 99. Here is the code for reference:

```
public static int function(int n){
    int[] mystery = new int[n + 1];
    for(int i = 0; i <= n; i++)
        mystery[i] = i;
    for(int i = 2; i <= n; i++)
        if(mystery[i] == i)
            for(int j = 2 * i; j <= n; j += i)
                mystery[j] += i;
    return mystery[n];
}
public static void main(String[] args){
    System.out.println(function(12));
    System.out.println(function(16));
    System.out.println(function(39));
    System.out.println(function(99));
    System.out.println(function(100000000));
}
```

We walk through the case where `n` is 12 and then generalize our result to find the other problems. The first thing the loop does is initialize an array of size 13 with elements set from 0 to 12. That is:

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

Then, starting with 2, the code runs from logic. If the value is still set to its index, then it adds itself to values that are at indices that are multiples of the starting index. If we run that, the array goes through the following states:

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

$$[0, 1, 2, 3, 6, 5, 8, 7, 10, 9, 12, 11, 14]$$

$$[0, 1, 2, 3, 6, 5, 11, 7, 10, 12, 12, 11, 17]$$

$$[0, 1, 2, 3, 6, 5, 11, 7, 15, 12, 12, 11, 17]$$

Note that we only have to loop through the array adding a number if it has not already been marked by another index, that is, it is not a multiple of any number before it other than 0 or 1. This is equivalent to saying that it is prime. So, what the program really does is add all the prime numbers to its multiples. We want to find the value of the last number in the array (our argument), so rather than walking through the program, we can find the prime factorization of each number and use that to calculate the output.

For $n = 12$, 12 can be factored into $4 * 3$, which can further be factored into $2 * 2 * 3$. The program only adds each prime factor once, so the output of `function(12)` should be $(12 + 2 + 3) = 17$ which is the answer we determined by walking through the code. For $n = 16$, we can notice that $16 = 2^4$, so it only has one unique prime factor: 2. Given this, `function(16)` evaluates to $(16 + 2) = 8$. Thirty-nine has a prime factorization of $3 * 13$, so `function(39)` is $(39 + 3 + 13) = 55$, and 99 has a prime factorization of $3^2 * 11$, so `function(99)` is $(99 + 11 + 2) = 113$. For the monster case of $n = 100,000,000$, note that $100,000,000 = 10,000^2 = 100^4 = 10^8 = (2 * 5)^8 = 2^8 * 5^8$. The function at $100,000,000$, then, is $100,000,000 + 2 + 4 = 100,000,007$. Interestingly, this test case probably cannot be run on a standard home computer because the numbers are so large. A more efficient alogirithm is needed.

Short Answer Problem 9

(4 points)

Unlike the last problem, the simplest (and perhaps, dare I say, only) way to solve this problem is to simply walk through each line of input. That's what we are going to do here. Using a theoretical data structure that can switch between being a stack and queue. An asterisk will indicate what side of the structure we will be popping from and will always push, by convention, to the right side of the structure. Initially, the structure is set up as a queue:

1 A, 1 B, 1 C, 1 D: $*$[A, B, C, D]

Then the direction gets reversed:

3: [A, B, C, D]$*$

We add some more:

1 E, 1 F, 1 G, 1 H: [A, B, C, D, E, F, G, H]$*$

The direction is reversed again:

3: $*$[A, B, C, D, E, F, G, H]

There are some pops:

2, 2: $*$[C, D, E, F, G, H], Popped: A, B

It reverses:

3: [C, D, E, F, G, H]$*$

More pops:

2, 2: [C, D, E, F]$*$, Popped: H, G

And the rest of the lines:

1 I: [C, D, E, F, I]$*$

2: [C, D, E, F]$*$, Popped: I

3: $*$[C, D, E, F]

1 J, 1 K: $*$[C, D, E, F, J, K]

From this point, it alternatively pops from the front and the end. The last couple pops were: C, K, D, and J. The list of values popped in order, then, are A, B, H, G, I, C, K, D, and J.

Short Answer Problem 10

(5 points)

There are 12 statements in this section that need to be converted into two other notations each for a total of 24 different conversions. It would be tedious and there would be little learning if all 24 conversions were worked out here. We have selected a few statements that we found particularly enlightening or illustrate certain principles that can be used to then do the rest of the conversions.

1. `(1 + 3)/(2 + 8)`: The first thing to notice is that there are naturally two sub-expressions of the bigger expression: `(1 + 3)` and `(2 + 8)`. First, let's convert these two sub-expressions into prefix and postfix: `+ 1 3` and `+ 2 8` and `1 3 +` and `2 8 +`. Then, we express the division of the two sub-expressions in the correct notation. For prefix, the expression becomes `/ + 1 3 + 2 8`, and for postfix, the expression becomes `1 3 + 2 8 + /`.

2. `1 + 3 / 2 + 8`: Notice that even though this is the same problem as the previous one without any parentheses, the grouping is completely different. By the order of operations, we know that we have to evaluate `3 / 2` before we do any adding. It makes sense, then, to convert that expression into postfix and prefix notation before the rest of it. In these forms, it looks like `3 2 /` and `/ 3 2` respectively. Now we can convert the addition operations, and the phrase becomes `1 3 2 / + 8 +` and `+ + 1 / 3 2 8` in postfix and prefix.

3. `1 - 2 + 3 * 4 ↑ 5`: Just as before, we should convert from one notation to the others in the order of the order of opertions. The operation with the highest precedence in the order of operations is the exponentiation `4 ↑ 5`, which becomes `↑ 4 5` in prefix and `4 5 ↑` in postfix. Then, we must multiply the expression by `3`, so we now have `* 3 ↑ 4 5` and `3 4 5 ↑ *`. Then, we subtract `2` from `1` and then add it to what we have before. The final phrases, then, are `+ - 1 2 * 3 ↑ 4 5` and `1 2 - 3 4 5↑ * +`.

For the last part, there are a few tricks that can be utilized to decide whether an expression is in prefix notation, infix notation, or postfix notation. The simplest is to note that the *only* notation where an operater is allowed to be (and always is) the first item is prefix. When the last item is an operator, the notation is postfix. Else, it is written in infix notation.

Short Answer Problem 11

(5 points)

The first three parts of this problem can be solved by drawing the binary tree associated, because the number of rows is small.

```
                    (0,0)
       (1, 0)                    (1, 1)
(2, 0)         (2, 1)   (2, 2)          (2, 3)
```

Note that in a tree, there is at most one path between any two nodes. Let us define the "least common ancestor" of a pair of nodes to be the deepest (lowest) node that is an ancestor of both nodes. An ancestor of a node is a node that is on the path from the node to the root (it's one of the parents of a parents of this node). The least common ancestor of two nodes can be one of the nodes themselves, if one node happens to be an ancestor of the other. By observing patterns, it is possible to see that the shortest path between two nodes in a binary tree is equal to the sum of the path lengths of the first node and the second node to their least common ancestor, because the path consists of moving upward from one node until the least common ancestor is reached, and then moving downward to the other node.

The fourth part of this problem is slightly tricky, because the row numbers are high and it might be annoying to draw the diagram. However, the problem is greatly simplifyed by seeing that the two nodes are actually next to each other and share the same parent. That gives a shortest path length of 2.

The final part is the trickest. This is where looking at the binary representations of the column numbers comes in handy. Consider a node in column 3. The binary representation of 2 is 11_2. The parent of node 2 has a binary representation of 1_2. A node in column 2, or 10_2 has the same parent. By looking at this sort of pattern, we can see that the parent of a node in column x is in column $x/2$, where / represents integer division. Dividing by 2 using integer division is equivalent to removing the last digit of the binary representation of the number.

The binary representation of 3 is 11_2. The binary representation of 102 is 1100110_2. However, the node in column 102 is two rows below the node in column 3, so we can shift the node up two rows safely (the least common ancestor is no deeper than the most shallow node in the pair). Dividing by two twice (or removing the last two digits) of $102 = 1100110_2$ gives 11001_2. Now that the two nodes are in the same row, we can move them up rows together until they are in the same column, effectively finding their least common ancestor. It takes 5 more moves for both nodes until both are in column 0 and row 95, which is their least common ancestor. The total number of moves, which is equal to the number of steps in the shortest path, was $5 + 5 + 2 = 12$.

Short Answer Problem 12

(5 points)

The error in the method comes in integer division. If we are trying to calculate 2^8, the function correctly decides that $2^8 = (2^4)^2$. However, when the exponent is odd instead of even, then a value is lost when the exponent is divided by 2. $2^9 \neq (2^4)^2$, for instance. To fix this, we must reconsider what should happen if the exponent is odd (because the program is does the correct breakdown for even exponents, although the answer is not always correct for even exponents). The correct way to figure out 2^9 is to break it down into $(2^4)^2 \cdot 2$, multiply by the value of a another time to account for the extra power. See the correct code below for details.

However, it is incorrect to say that the method is correct whenever b (the exponent) is even. Even if the method correctly decides that $a^b = \left(a^{\frac{b}{2}}\right)^2$, when we recur on $a^{\frac{b}{2}}$, we that part might be calculated incorrectly if $\frac{b}{2}$ is odd. Essentially, the exponent is repeatedly divided by 2, and if the result is ever odd (and greater than 1), then the power method will make a mistake. The only numbers that can be repeatedly divided by 2 without ever reaching an odd number (until getting to 1) are powers of 2. Thus, the power method shown is only correct for $b = 2^k$, where k is any non-negative integer. The power method is also correct when $b = 0$ because of the way the base case is coded.

The runtime complexity of "power of 2" exponentiation is $O(\log b)$, or any variation of that. This can be deduced based on the fact that b, the exponent, is repeatedly divided by 2. The recursive method is called once for each time the exponent is divided by 2. The number of times an integer N can be divided by 2 is proportional to $O(\log N)$, so we have that as the runtime complexity of this algorithm.

```
public static int pow(int a, int b){
    if (b <= 0){
        return 1;
    }
    else if (b == 1){
        return a;
    }
    else{
        int tmp = pow(a, b / 2);
        if(tmp % 2 == 0)
            return tmp * tmp;
        else
            return tmp * tmp * a;
    }
}
```