San José State University
College of Science / Department of Computer Science
CS 146 Data Structures and Algorithms
Section 4/7, Spring 2015

Instructor: Dr. Angus Yeung

Assignment 1 Solutions

Due Date: Monday, February 9, 11:59 pm

PART A (40 Points)

1.1 (5 Points) Prove the following formula by induction: $\sum_{i=1}^{N} i^3 = (\sum_{i=1}^{N} i)^2$. You must show base case, inductive hypothesis and proof in your solution.

Solution (total 5 points):

A) Base Case: Must show the case when N = 1 (1 pt)
B) Inductive Hypothesis: Must mention either "Inductive Hypothesis:" or "…assume…" (1 pt)
C) Proof: Must show the derivation for N+1 case (3 pts)

$$\sum_{i=1}^{N+1} i^3 = (N+1)^3 + \sum_{i=1}^{N} i^3$$

$$= (N+1)^3 + \frac{N^2(N+1)^2}{4}$$

$$= (N+1)^2\left[\frac{N^2}{4} + (N+1)\right]$$

$$= (N+1)^2\left[\frac{N^2+4N+4}{4}\right]$$

$$= \frac{(N+1)^2(N+2)^2}{2^2}$$

$$= \left[\frac{(N+1)(N+2)}{2}\right]^2$$

$$= \left[\sum_{i=1}^{N+1} i\right]^2$$

1.2 (5 Points) Write a recursive method that returns the number of 1's in the binary representation of N. Use the fact that this is equal to the number of 1's in the representation of N/2, plus 1, if N is odd.

Solution (total 5 points):

A) The function must call itself within the function implementation – recursive funtion. (1 pt)
B) There must be an exit condition, e.g., if (n<2) (1 pt)
C) The function must be converging (or making progress toward the exit condition), e.g., n/2 (1 pt)
D) The code fragment must be implemented correctly (2 pts)

```
public static int ones( int n )
{
   if( n < 2 )
      return n;
   return n % 2 + ones( n / 2 );
}
```

1.3 (5 Points) Prove that $\sum_{i=1}^{N} i \times i! = (N+1)! - 1$ by induction. You must show base case, inductive hypothesis and proof in your solution.

Solution (total 5 points):

A) Base Case: Must verify the base case when N = 1 (1 pt)
B) Inductive Hypothesis: Must mention either "Inductive Hypothesis:" or "…assume…" (1 pt)
C) Proof: Must show the derivation for N+1 case (3 pts)

$$\sum_{i=1}^{N+1} i \times i! = (N+1) \times (N+1)! + \sum_{i=1}^{N} i \times i!$$

$$\sum_{i=1}^{N+1} i \times i! = (N+1) \times (N+1)! + (N+1)! - 1$$

$$= (N+1)! \times \big((N+1)+1\big) - 1$$

$$= (N+2)! - 1$$

1.4 (15 Points) List the functions below from the lowest to the highest order. If any two or more are of the same order, indicate which.

N^2	N	$\sqrt{N}$	$N \log N$	48
N^3	$N^2 \log N$	$N \log N$	$N^{1.5}$	$N \log \log N$
$N (\log N)^2$	$N \log(N^2)$	$1/N$	$2^{N/2}$	2^N

Solution (total 15 points):

$1/N$ (1 pt)
48 (1 pt)
$\sqrt{N}$ (1 pt)
N (1 pt)
$N \log \log N$ (1 pt)
$N \log N = N \log N = N \log(N^2)$ - they grow at the same rate. (3 pts) Note: $N \log N$ is repeated.
$N (\log N)^2$ (1 pt)
$N^{1.5}$ (1 pt)
N^2 (1 pt)
$N^2 \log N$ (1 pt)
N^3 (1 pt)
$2^{N/2}$ (1 pt)
2^N (1 pt)

1.5 (10 Points) For each of the following pairs of functions $f(N)$ and $g(N)$, determine whether $f(N) = O\big(g(N)\big)$, $g(N) = O\big(f(N)\big)$, or both. You must provide both **answers** and **explanations** for this question.

$f(N) = (N^2 - N + 3)/3, g(N) = 6N$
$f(N) = N + 2\sqrt{N}, g(N) = N^2$
$f(N) = N \log N, g(N) = N\sqrt{N}/2$
$f(N) = 2(\log N)^2, g(N) = \log N + 1$
$f(N) = 4N \log N + N, g(N) = (N^2 - N)/2$

Solution (total 10 points)
A. $f(N) = (N^2 - N + 3)/3, g(N) = 6N$
Answer: $g(N) = O\big(f(N)\big)$ (1 pt)

Explanation: Constant factors can be ignored; we pay attention to the largest (higher order) terms. N^2 outgrows N as N becomes very large. Therefore $f(N)$ outgrows $g(N)$. (1 pt)

B. $f(N) = N + 2\sqrt{N}, g(N) = N^2$
Answer: $f(N) = O\big(g(N)\big)$ (1 pt)
Explanation: N^2 outgrows N as N becomes very large. We ignore $2\sqrt{N}$ since N outgrows $\sqrt{N}$. (1 pt)

C. $f(N) = N \log N \,, g(N) = N\sqrt{N}/2$
Answer: $f(N) = O\big(g(N)\big)$ (1 pt)
Explanation: Both sides have a factor of N, so we ignore it. $\sqrt{N}$ outgrows $\log N$, so $g(N)$ grows bigger. (1 pt)

D. $f(N) = 2(\log N)^2, g(N) = \log N + 1$
Answer: $g(N) = O\big(f(N)\big)$ (1 pt)
Explanation: $(\log N)^2$ outgrows $\log N$. (1 pt)

E.$f(N) = 4N \log N + N, g(N) = (N^2 - N)/2$
Answer: $f(N) = O\big(g(N)\big)$ (1 pt)
Explanation: N^2 outgrows $N \log N$. (1 pt)

PART B (60 Points)

1.6 (15 Points) Define a class that provides `getLength` and `getWidth` methods. Using the `findMax` routines in Figure 1.18 (listed below), write a `main` that creates an array of `Rectangle` and finds the largest `Rectangle` first on the basis of area, and then on the basis of perimeter. (15 Points)

```
    // Generic findMax, with a function object.
    // Precondition: a.size( ) > 0.
    public static <AnyType>
    AnyType findMax( AnyType [ ] arr, Comparator<? super AnyType> cmp )
    {
        int maxIndex = 0;

        for( int i = 1; i < arr.size( ); i++ )
            if( cmp.compare( arr[ i ], arr[ maxIndex ] ) > 0 )
                maxIndex = i;

        return arr[ maxIndex ];
    }

class CaseInsensitiveCompare implements Comparator<String>
{
    public int compare( String lhs, String rhs )
      { return lhs.compareToIgnoreCase( rhs ); }
}

class TestProgram
{
    public static void main( String [ ] args )
    {
        String [ ] arr = { "ZEBRA", "alligator", "crocodile" };
        System.out.println( findMax( arr, new CaseInsensitiveCompare( ) ) )
    }
}
```

Figure 1.18 Using a function object as a second parameter to `findMax`; output is ZEBRA

Additional requirements for submission:

(1) Create the folder "`findRectangle`" that contains all required `.java` file(s).
(2) The folder should be part of the `hw1.zip` file that you upload to Canvas.
(3) Do not include any other file types inside the `findRectangle` folder except `.java` file(s).
(4) Do not declare and use any "`package`" in your `.java` file(s).
(5) Use "`findRectangle`" as the class name – so you can run as `%java findRectangle`.

There is one point penalty for each requirement that a student fails to follow (total penalty: 5 points).

Solution (total 15 points):

A) Additional Requirements for submission (5 pts, see the assignment question)
B) Declare a new class (1 pt)
C) The class must provide `getLength` and `getWidth` methods (1 pt)
D) The new class must call the `findMax` routines shown in Figure 1.18 (1 pt)
E) Must implement a `main` method (1 pt)
F) The `main` method must create an array of `Rectangle` (1 pt)
G) The array of `Rectangle` must be initiated with some entries (1 pt)

H) Program must find the largest `Rectangle` on the basis of area (1 pt)
I) Program must fine the largest `Rectangle` on the basis of perimeter (1 pt)
J) Program must be able to run as: `%java findRectangle` (1 pt)
K) Program must run without any errors or crashes (1 pt)

1.7 (30 Points) For each of the following six program fragments:

a. Give an analysis of the running time (Big-Oh will do).
b. Implement the code in Java, and give the running time for several values of N.
c. Compare your analysis with the actual running times.

```
1. sum = 0;
   for( i = 0; i < n; i++ )
       sum++;

2. sum = 0;
   for( i = 0; i < n; i++ )
       for( j = 0; j < n; j++ )
           sum++;

3. sum = 0;
   for( i = 0; i < n; i++ )
       for( j = 0; j < n * n; j++ )
           sum++;

4. sum = 0;
   for( i = 0; i < n; i++ )
       for( j = 0; j < i; j++ )
           sum++;

5. sum=0;
   for( i = 0; i < n; i++ )
       for( j = 0; j < i * i; j++ )
           for( k = 0; k < j; k++ )
               sum++;

6. sum=0;
   for( i = 1; i < n; i++ )
       for( j = 1; j < i * i; j++ )
           if( j % i == 0 )
               for( k = 0; k < j; k++ )
                   sum++;
```

The Big-Oh estimation and the analysis for actual running time must be submitted with other written questions in the same `hw1.pdf` file. Below is the additional requirements for Java program submission:

(1) Create the folder "`fragments`" that contains all required `.java` file(s).
(2) The folder should be part of the `hw1.zip` file that you upload to Canvas.
(3) Do not include any other file types inside the `fragments` folder except `.java` file(s).
(4) Do not declare and use any "`package`" in your `.java` file(s).
(5) Use "`fragment1`" as the class name – so you can run as `%java fragment1` for a working program that contains the first code fragment.
(6) Repeat Step (5) for other code fragment, e.g., `fragment2, fragment3,`…

A student will not receive full credits on this problem if the student fails to follow all steps listed in above.

Solution (30 points):
Big-Oh analysis: (8 points)
1. The running time is $O(N)$. Correct Answer (1 pt)
2. The running time is $O(N^2)$. Correct Answer (1 pt)
3. The running time is $O(N^3)$. Correct Answer (1 pt)
4. The running time is $O(N^2)$. Correct Answer (1 pt)
5. The running time is $O(N^5)$. Correct Answer (1 pts), Explanation (1 pt)
6. The running time is $O(N^4)$. Correct Answer (1 pts), Explanation (1 pt)

Code Implementation: (13 points)
1. Must implement working Java programs (six programs in total) that contains all the six code fragments. For example, program for code fragment 1 can run as `%java fragment1`. (6 x 1 pt)
2. Java program must run without major bugs or crash. (2 pts)
3. All java programs must follow the convention: `fragment1.java`, `fragment2.java`, etc. (2 pts)
4. Must provide the running time for at least two values of N. (6 x 0.5 pt)

Analysis with the actual running times: (9 points)
1. Must implement timer and show running time for each program. (6 x 0.5 pt)
2. Must analyze the running time and compare each value with Big-Oh numbers. (6 x 1 pt)

1.8 (15 Points) Suppose you need to generate a random permutation of the first N integers. For example, `{4, 3, 1, 5, 2}` and `{3, 1, 4, 2, 5}` are legal permutations, but `{5, 4, 1, 2, 1}` is not, because one number (1) is duplicated and another (3) is missing. This routine is often used in simulation of algorithms. We assume the existence of a random number generator, `r`, with method `randInt(i, j)`, that generates integers between i and j with equal probability. Here are three algorithms:

1. Fill the array `a` from `a[0]` to `a[n-1]` as follows: To fill a[i], generate random numbers until you get one that is not already in `a[0], a[1], . . . , a[i-1]`.
2. Same as algorithm (1), but keep an extra array called the `used` array. When a random number, `ran`, is first put in the array `a`, set `used[ran] = true`. This means that when filling `a[i]` with a random number, you can test in one step to see whether the random number has been used, instead of the (possibly) i steps in the first algorithm.
3. Fill the array such that `a[i] = i + 1`. Then

```
for( i = 1; i < n; i++ )
    swapReferences( a[ i ], a[ randInt( 0, i ) ] );
```

a. Prove that all three algorithms generate only legal permutations.
b. Give as accurate (Big-Oh) an analysis as you can of the expected running time of each algorithm.
c. Write (separate) programs to execute each algorithm 10 times, to get a good average. Run program (1) for N = 250, 500, 1,000, 2,000; program (2) for N = 25,000, 50,000, 100,000, 200,000, 400,000, 800,000; and program (3) for N = 100,000, 200,000, 400,000, 800,000, 1,600,000, 3,200,000, 6,400,000.
d. Compare your analysis with the actual running times.

All analysis of algorithms including Big-Oh, comparison analysis, and worst-case analysis list in above must be submitted with other written questions in the same `hw1.pdf` file. Below is the additional requirements for

Java program submission:

(1) Create the folder "`permutation`" that contains all required `.java` file(s).
(2) The folder should be part of the `hw1.zip` file that you upload to Canvas.
(3) Do not include any other file types inside the `fragments` folder except `.java` file(s).
(4) Do not declare and use any "`package`" in your `.java` file(s).
(5) Use "`permutation1`" as the class name – so you can run as `%java permutation1` for a working program that contains the first algorithm.
(6) Repeat Step (5) for other two algorithms, e.g., `permutation2` and `permutation3`.

A student will not receive full credits on this problem if the student fails to follow all steps listed in above.

Solution (15 Points):

a. All three algorithms generate only legal permutations because: (1.5 points = 3 x 0.5 points)
Algorithm 1: Has tests to guarantee no duplicates;
Algorithm 2: Has tests to guarantee no duplicates;
Algorithm 3: the third algorithm works by shuffling an array that initially has no duplicates.

b. Big-Oh analysis: (4.5 points)
Answer for Algorithm 1: $O(N^2 \log N)$; (1 pts)
Answer for Algorithm 2: $O(N \log N)$; (1 pts)
Answer for Algorithm 3: $O(N)$. (1 pts)

Attempt to explain how to reach each algorithm (3 x 0.5 pt)

Explanation for Algorithm 1: For the first algorithm, the time to decide if a random number to be placed in $a[i]$ has not been used earlier is $O(i)$. The expected number of random numbers that need to be tried is $N/(N-i)$. This is obtained as follows: i of the N numbers would be duplicates. Thus the probability of success is $(N-i)/N$. Thus the expected number of independent trials is $N/(N-i)$. The time bound is thus

$$\sum_{i=0}^{N-1} \frac{Ni}{N-i} < \sum_{i=0}^{N-1} \frac{N^2}{N-i} < N^2 \sum_{i=0}^{N-1} \frac{1}{N-i} < N^2 \sum_{j=1}^{N} \frac{1}{j} = O(N^2 \log N)$$

Explanation for Algorithm 2: The second algorithm saves a factor of i for each random number, and thus reduces the time bound to $O(N \log N)$ on average.
Explanation for Algorithm 3: Obviously it is linear.

c. Write (separate) programs to execute each algorithm 10 times, to get a good average.
Run program (1) for N = 250, 500, 1,000, 2,000;
program (2) for N = 25,000, 50,000, 100,000, 200,000, 400,000, 800,000; and
program (3) for N = 100,000, 200,000, 400,000, 800,000, 1,600,000, 3,200,000, 6,400,000.
Total Points for running programs: 6 points
   - Must deliver working Java programs for the three algorithms (3 x 1 points)
   - Must show results (written) for running each algorithm 10 times (3 x 0.5 points)
   - Must show results (written) for different N values shown in above (3 x 0.5 points)

d. Compare the analysis with the actual running times: 3 points (3 x 1 points)