CSC 225 --- FALL 2015
Algorithms and Data Structures I
Assignment 1

Total out of 60 (10 marks for each question)

----- Question 1 -----

The correct order of functions according to Big-Oh notation
from smallest to largest:

2^100
log(log n)
sqrt(log n)
(log n)^2
3n^(0.5)
6n*(log n)
4^(log n) = n^2
n^3
4^n
2^(2^n)

Justification can be done using (R1--R8) on slide 82 of
Lecture 3 and 4, as well as slide 84, or even slide 94 with
a combination of the rules on slide 82.

For example, as n tends to infinity for the following limit:

lim log(log n)/sqrt(log n)
= lim (2*sqrt(log n))/(ln(2)*log(n))
= lim 1/(ln(2)*sqrt(log n))
= 0

by L'Hospital's Rule applied twice, which shows the proper
ordering of the second and third functions in the list.

As a simpler example, we could reference slide 94 for the
functions 3n^(0.5) and 6n*(log n) with a further justification

by two applications of R1 on slide 82 using a=3, then a=6, to deal with constants and obtain a proper ordering.

Showing the reasoning for one pair of successive functions would be enough to get full marks here.

----- Question 2 -----

(a)

Let x = floor(log N) to simplify our writing.
Let T(N) be the number of steps in the code fragment.
The outer loop iterates (x + 1) times, because
the outer loop runs for values of k that satisfy:

```
n = N/(2^k) >= 1
         N >= 2^k
     log N >= k
```

which implies 0 <= k <= x.

The following sum of (x + 1) terms gives the total number of steps, where the ith term in the sum is the number of steps that the inner loop takes during the ith run of the outer loop:

```
T(N) = N + N/2 + N/4 + ... + N/(2^i) + ... + N/(2^(x-1)) + N/(2^x)
T(N) = N*( 1 + 1/2 + 1/4 + ... + 1/(2^i) + ... 1/(2^(x - 1)) + 1/(2^x) )
T(N) < N*2
```

for all N >= 1. Therefore, T(N) is O(N).

Note that the infinite geometric sum (1 + 1/2 + 1/4 + ...) = 2.

(b)

We continue to make use of x as defined in part (a).

Now let y = ceil(log N) to simplify our writing.
Let K(N) be the number of steps in the code fragment.
The outer loop iterates y times, because the outer
loop does *not* run for values of k that satisfy:

i = 2^k >= N
    k >= log N

which implies 0 <= k <= (y-1).

The following sum of y terms gives the total number of
steps, where the ith term in the sum is the number of steps
that the inner loop takes during the execution of the ith
iteration of the outer loop:

K(N) = 1 + 2 + ... + 2^(i) + ... + 2^(y - 1)

Although in reverse order, this sum is the same as in T(N),
except K(N) may be missing the largest term, depending on
whether we have x = y or x < y.

Therefore, K(N) <= T(N) for all N >= 1,
and so K(N) is also O(N).

(c)

We continue to make use of y as defined in part (b).
Let B(N) be the number of steps in the code fragment.
The outer loop runs y iterations by the same reasoning
as in part (b). However, now the inner loop obviously
runs N iterations, and our total number of steps
is simply:

B(N) = N*y <= N*(log N + 1) = N*(log N) + N
B(N) < 2N*(log N)

for all N >= 1. Therefore, B(N) is O(N*log(N)).

(When working with growth order, you can use inequalities
to avoid having to deal with nasty formulae or unwieldy
algebraic manipulations, but then you have to be careful
not to relax the expression so much that the bound is no
longer in the smallest growth order possible.)

----- Question 3 -----

You have your nicely written sum notation, but this file
format forces me to have to make the following compromise.
Let sum(a,b)[f(i)] denote the sum of terms given by a
function f(i) where a <= i <= b.

We want to prove sum(1,n)[2i-1] = n^2 for all n >= 1.

First, check the base case for n = 1: (or in other words, does
the statement we want to prove hold true for this value of n?)

(Notice the value we are checking is the first possible value
that n is allowed to take to satisfy the statement we want
to prove, i.e. n >= 1.)

We have sum(1,1)[2i-1] = 2(1) - 1 = 1
and we also have (1)^2 = 1

Since both expressions evaluate to the same value, the base
case holds true.

Induction Hypothesis:

** Suppose sum(1,k)[2i-1] = k^2 for some k >= 1.

(Notice that k is >= our base case value.)

(Now we have to show that the statement holds true for the
next possible value that n can take, i.e. n = k + 1.)

(Start with the left hand side of the equality and
manipulate algebraically to get the right hand side.)

(Somewhere along the manipulation, you have to apply **
but convince yourself that ** is ONLY true and applicable
for the specific value of k at this point.)

sum(1,k+1)[2i-1] = (2(k+1) - 1) + sum(1,k)[2i-1]
(this step is justified by rule of sums, simply
removing the largest term from the sum and
writing that term explicitly by itself.)

 = (2k + 2 - 1) + k^2
(simple algebra for the first terms, and applying **
to the term sum(1,k)[2i-1].)

 = k^2 + 2k + 1
(simple algebra)

 = (k+1)(k+1)
(by factoring)

 = (k+1)^2

And so the statement we are trying to prove holds true for
(k+1) WHEN it holds true for (k).

This is dragging the point on, and you don't have to be this
detailed about it, because this is to help convince everyone
how induction works. Since the statement holds true for the
base value n = 1, then it holds true for n = 2. We already
did the algebra to justify this, so just replace k with the
value 1 in our proof to convince yourself. Similarly,
because it holds true for n = 2, then it holds true for
n = 3, and so on, and so forth, as you can see, for any
value of n >= 1.

A small warning: an induction proof falls apart if there is no base value that holds true, so the danger is that it can be easy to delude yourself a statement is true by induction when in fact it may be false. So pay close attention to what looks to be like the trivial part of the argument.

----- Question 4 -----

Less explanation this time, and towards ideal brevity in justification of the proof's steps.

We want to prove sum(1,n)[1/(i(i+1))] = n/(n+1), for all n >= 1.

The statement is true for n = 1, since,
sum(1,1)[1/(i(i+1))] = 1/(1(1+1)) = 1/2,
and
1/(1+1) = 1/2.

Induction Hypothesis:

Suppose sum(1,k)[1/(i(i+1))] = k/(k+1), for some k >= 1.

Then

sum(1,k+1)[1/(i(i+1))] = 1/((k+1)(k+1+1)) + sum(1,k)[1/(i(i+1))]
= 1/((k+1)(k+2)) + k/(k+1)
(sum(1,k) replacement justified by Induction Hypothesis)
= (1 + k(k+2)) / ((k+1)(k+2))
= (k^2 + 2k + 1) / ((k+1)(k+2))
= (k+1)^2 / ((k+1)(k+2))
= (k+1)/(k+2)
= (k+1)/((k+1)+1)

Therefore, this shows the statement must hold true for all values of n >= 1.

(Notice that the last line of algebraic manipulation made the expression look *exactly* as if we had replaced n for k+1 in the right hand side of the statement equality, n/(n+1).)

(If you really wanted to, and some induction proof isn't working out well with your attempts, you could try the algebraic manipulations in the reverse order and the proof would still be correct. Sometimes the other direction is easier, but it depends on the manipulations involved. Typically, I try to pick algebraic manipulations where I know I only have to cancel factors or collect terms, instead of algebraic manipulatios that require splitting things up in an unforseeable or unintuitive way.)

----- Question 5 -----

Let A be an array containing n-1 unique integers in the range [0,n-1].

We want to describe an algorithm in pseudocode for finding the missing integer in the range specified in O(n) time, and without using more than O(log n) additional space outside the array A.

Since the sum of all integers from 0 to n-1 is
y = (1/2)*(n-1)*(n) < n^2, for all n > 0, to keep track
of a sum of integers in A we would need at most
log(n^2) = 2*log(n) bits, which is clearly O(log n) space.

Calculating y takes a constant number of steps using the formula we have given in terms of n.

Let S be the sum of all entries in the array A.
To sum all the integers in A takes a linear number of steps, one addition for each of the n entries.

The solution of the missing integer is then simply (y - S), a single subtraction.

Altogether, this shows our algorithm is O(n).

PSEUDOCODE

INPUT:
n >= 2
A an array of n-1 integers in the range [0,n-1]

OUTPUT:
The missing integer in A from range [0,n-1]

```
MissingInteger(n, A) {

  y <-- (1/2)*(n-1)*(n)

  // note that sum uses only 2*log(n) bits
  sum <-- 0
  for (i=0; i<(n-1); i++) {
    sum += A[i]
  }

  return (y - sum)

}
```

----- Question 6 -----

Let A be an input array with n >= 2 elements.

We want to describe an algorithm that determines the number of pairs of equal elements in A that runs in O(n*log n) time.

First, it is much easier to compare elements in the array once we sort them, and there is no issue if we use a sorting algorithm that runs in O(n*log n) time. So say, for example, we use Heapsort is listed on slide 96 of Lecture 3 and 4 as running in this time.

We have to be careful, because say A contains only the same value in every entry, and therefore has n choose 2 number of pairs, or (n_C_2) = (1/2)*n*(n-1), which is O(n^2), which means we cannot check each pair individually even if they are nicely sorted. So just calculate them instead with the formula we've used for choosing pairs here. All we need then are the number of equal elements, which we can get in linear time checking every element of the array once, in sorted order.

Thus, it actually takes longer to sort the array than it does to check for all the equal entries in A. Altogether, our algorithm runs in O(n*log n) time.

PSEUDOCODE

INPUT:
A an array of n values

OUTPUT:
The number of pairs of values in A that are equal.

```
NumberOfEqualPairs( A ) {

  // our call to sorting routine in time O(n*log n)
  Heapsort( A )

  // integers to keep track of number of *successive
  //   equal entries* NOT number of pairs
  value <-- A[0]
  count <-- 1

  // keep track of number of equal pairs
  n_pairs <-- 0

  for (i=1; i<n; i++) {
    if ( A[i] == value ) {
      // found a successive equal value
      count++;
    } else {
      // this subset of successive entries equal
      //   to "value" has ended
      n_pairs += (1/2)*(count)*(count-1)

      // now start checking for next subset
      value <-- A[i]
      count <-- 1
    }

  } // done checking array A for equal entries

  return n_pairs
}
```

(If you really want to, you can allow for n=1 and immediately return zero for that case.)