Practice Exam Solutions
Algorithms and Programming for High Schoolers (AddisCoder)

Question 1: Imagine evaluating the following expressions in order in the Python interpeter. For each expression written in red, write down what the expression would evaluate to in the space below. If the expression would cause an error in Python, then write Error.

```
>>> x = 5

>>> y = 7

>>> z = 2

>>> x * y + z

    37

>>> (x * y) + z

    37

>>> x * (y + z)

    45

>>> x % y

    5

>>> y % x

    2

>>> [] * z
```

```
      []
>>> [] * ‘2’
```

Error. (can’t multiply a `list` by a `str`)

```
>>> [1] * z + x
```

Error. (can’t add the `list` `[1,1]` to the `int` 5)

```
>>> str(2)
      ‘2’
>>> [[]][0]
      []
>>> [[]][1]
```

Error. The `list` `[[]]` has no 1st element.

```
>>> len([[[]]])
      1
>>> [[]][]
```

Error. This is invalid Python syntax.

```
>>> x = [2, 3]
>>> y = []
>>> for i in xrange(x[0]**x[1]):
>>>    y += [i*2]
>>> y
      [0, 2, 4, 6, 8, 10, 12, 14]
>>> def isPrime(x):
>>>    if x < 2:  return False
>>>    for i in xrange(2, x):
```

```
>>>       if i*i >= x:  break
>>>       if x % i == 0:  return False
>>>    return True
>>> isPrime(1)
      False
>>> isPrime(2)
      True
>>> isPrime(9)
      True (to be correct the loop should break if i*i > x)
>>> isPrime(12)
      False
```

Question 2: Consider the `lists` `[]`, `[[]]`, `[[[]]]`, ... The *depth* of such a `list` is the number of nested layers of brackets. So, `depth([])` is 0, `depth([[]])` is 1, `depth([[[]]])` is 2, etc. Write a function `depth`(L) which takes such a `list` and computes its depth. What's the running time of your function in terms of n, the number of brackets in the list?

Example solution:

```
def depth(L):
    if L == []: return 0
    return 1 + depth(L[0])
```

The running time is $\Theta(n)$.

Question 3: Consider the following code for testing whether a number is prime or not.

```
def isPrime(n):
    if n < 2: return False
    for i in xrange(2, n):
        if n % i == 0: return False
        return True
```

What is the running time of this code as written? Is it correct? If it's not correct, suggest a minor change to the code which would make it correct.

Example solution: The running time of the code as written is $\Theta(1)$: the first time through the loop, the code returns. In fact, as written, the code is incorrect: it will just return `True` if either $n = 2$, or $n \geq 3$ and n is odd. A minor change to make the code correct would be to indent the `return True` differently to be directly under the `for` loop instead of being a part of the code block of the `for` loop.

Question 4: Write a function `cubeRoot`(n) which takes a positive integer n and outputs the largest integer x such that $x^3 \leq n$.

(a) Give a solution with running time $O(n)$.

(b) Give a solution with running time $O(\log_2 n)$.

The running times above are assuming you can do arithmetic on integers up to n in $O(1)$ time.

Example solution: Note that any function which is $O(\log_2 n)$ is also $O(n)$, so it suffices to just give a solution with running time $O(\log_2 n)$. Here is such a solution using binary search:

```
# do a binary search amongst integers in [x,y] for the answer
def binarySearch(x, y, n):
    if x == y:
        return x
    # let mid be ceil((x+y)/2) (which is why we put the + 1)
    # if you don't put the + 1, the code will run forever if y == x+1
    # and the elif occurs (because mid will just be x again).
    # alternatively, above you could have a separate check for y ==
    # x+1 and handle that case separately
    mid = (x + y + 1) / 2
    cube = mid**3
    if cube == n:
        return mid
    elif cube < n:
        return binarySearch(mid, y, n)
    else:
        return binarySearch(x, mid - 1, n)

def cubeRoot(n):
    return binarySearch(0, n, n)
```

It's also possible to solve just problem (a) more simply without binary search, just using a `while` loop. The following code takes $\Theta(n)$ time.

```
def cubeRoot(n):
    x = 0
    while x*x*x <= n:
        x += 1
    return x - 1
```

Question 5: Suppose we have a list of numbers L[0],L[1],...,L[n-1]. An *inversion* in the list is a pair i, j such that $i < j$ but L[i] $>$ L[j]. In other words, an inversion is a pair of indices where the larger number comes before the smaller number.

Describe an algorithm for counting the number of inversions in a list, then implement your algorithm in Python as `countInversions`(L). Faster running times get more points. Hint: The problem is solvable in $\Theta(n \log_2 n)$ time, though an easier solution takes $\Theta(n^2)$ time. You get more points for giving a slow, correct solution than a fast, incorrect solution.

Example solution: Here is a simple implementation taking $\Theta(n^2)$ time. We simply try all pairs and count the number of inversions.

```
def countInversions(L):
    ans = 0
    for i in xrange(0, len(L)):
        for j in xrange(i + 1, len(L)):
            if L[i] > L[j]:
                ans += 1
    return ans
```

A faster solution, taking only $\Theta(n \log_2 n)$ time, is to modify `mergeSort`. The basic idea is that during a `merge` operation, everytime the next smallest item comes from the right half instead of the left half, that means the element from the right half is inverted with respect to all the remaining items in the left half, so we can add that many inversions. The code follows.

```
# merge(A, B) returns a list with two elements.  The 0th element is
# the merged list of A,B, and the 1st element is the number of
# inversions between elements of A and elements of B (i.e., how many
# pairs (a,b) with a in A and b in B have a > b).
```

```
def merge(A, B):
    inversions = 0
    C = []
    atA = 0
    atB = 0
    for i in xrange(len(A) + len(B)):
        if atA == len(A):
            C += B[atB:]
            break
        elif atB == len(B):
            C += A[atA:]
            break
        elif A[atA] < B[atB]:
            C += [A[atA]]
            atA += 1
        else:
            C += [B[atB]]
            atB += 1
            inversions += len(A) - atA
    return [C, inversions]

# mergeSort(L) returns a list with two elements.  The 0th element is
L, sorted, and the 1st element is the number of inversions in L.
def mergeSort(L):
    A = L[:len(L)/2]
    B = L[len(L)/2:]
    C = mergeSort(A)
    D = mergeSort(B)
    E = merge(C[0], D[0])
    return [E[0], C[1] + D[1] + E[1]]

def countInversions(L):
    return mergeSort(L)[1]
```

Question 6: We've discussed making change using the least number of coins possible. What if we want to count how many different ways there are of making change? For example, if the coins we have available are [1,5,10,25] cents and we want to make change for 12 cents, there are 4 ways: (1) give all 1-cent pieces, (2) give a 10-cent piece and two 1-cent pieces, (3) give two

5-cent pieces and two 1-cent pieces, and (4) give one 5-cent piece and seven 1-cent pieces. So, the answer in this case is 4.

Write a function `change`(L, n) which outputs the number of ways to make change for n cents when the coin denominations available are those in L. For example, `change`([1,5,10,25], 12) should return 4. What is the running time of your solution? Faster running times get more points.

Example solution: The basic idea is to use recursion and memoization. Let $f(x, n)$ be the number of ways to make change for n cents using only the coins `L[x:]`. Then,

$$f(x,n) = \begin{cases} 1, & \text{if } n = 0 \\ 0, & \text{if } x = \text{len}(L) \text{ and } n > 0 \\ f(x+1, n), & \text{if } x < \text{len}(L) \text{ and } L[x] > n \\ f(x, n - L[x]) + f(x+1, n), & \text{otherwise} \end{cases}$$

The corresponding code is as follows.

```
def recurse(L, x, n, mem):
    if n == 0:
        return 1
    elif x == len(L):
        return 0
    elif mem[x][n] != -1:
        return mem[x][n]
    mem[x][n] = recurse(L, x+1, n, mem)
    if L[x] <= n:
        mem[x][n] += recurse(L, x, n - L[x], mem)
    return mem[x][n]

def change(L, n):
    mem = []
    for i in xrange(len(L)):
        mem += [[-1]*(n+1)]
    return recurse(L, 0, n, mem)
```

If the length of `L` is m, the running time is $\Theta(nm)$ in the worst case.

Question 7: Given a directed graph where each edge has a length, describe an algorithm that takes as input two vertices u, v and an integer $k \geq 0$ and outputs the length of the shortest path from u to v which takes *exactly* k steps. The path is allowed to visit vertices multiple times (for example, the path $1 \to 3 \to 2 \to 3 \to 7$ is a valid path from 1 to 7 of length 4, even though it visits vertex 3 twice). What is the running time of your algorithm? You do not have to write the code for it.

Example solution: For a vertex w and integer t, let $f(w,t)$ be the length of the shortest path from u to w taking exactly t steps. Then we have the following.

$$f(w,t) = \begin{cases} 0, & \text{if } w = u \text{ and } t = 0 \\ \infty, & \text{if } w \neq u \text{ and } t = 0 \\ \min_{z:(z,w) \text{ is an edge}} \ell(z,w) + f(z,t-1), & \text{otherwise} \end{cases},$$

where $\ell(z,w)$ is the length of the edge (z,w). In other words, the shortest way to get to w in t steps goes to some other vertex z in $t-1$ steps then takes the edge from z to w, so we try all possibilities for z. This can be implemented using recursion and memoization, and we will want to calculate $f(v,k)$. When calculating $f(w,t)$ for different values of w, t along the way, there are at most n values for w and $k+1$ vales for t. Also, if m is the number of edges in the graph, when you consider all possible vertices in the place of w, all loops combined loop over all edges once, for a total of m. So the runtime is $\Theta((n+m)k)$. A solution which said running time $O(n^2k)$ would get almost all the points (there are n possibilities for w and $k+1$ for t, and the loop that takes the min of all possibilities is at most n steps).

Question 8: In class I described Karatsuba's algorithm for multiplying two n-digit numbers, which recursively multiplied three pairs of $n/2$-digit numbers then combined the results in $O(n)$ time to get an overall running time of $\Theta(n^{\log_2 3})$.

Suppose that there existed an algorithm for multiplying two n-digit numbers which recursively multiplied *two* pairs of $n/2$-digit numbers then combined the results in $O(n)$ time. What would the running time then be?

Example solution: If $T(n)$ is the running time for multiplying two n-digit numbers, then we would have $T(n) = 2T(n/2) + O(n)$. This is exactly the recurrence for `mergeSort`, so overall we would have a running time of $O(n \log_2 n)$. One can also draw the recursion tree and see that there are $\log_2 n$ levels, each of which has total work n (level k will have 2^k nodes, each doing work $O(n/2^k)$).