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

Problem	Score	Max Score
1	23	23
2	11	11
3	11	11
4	11	11
5	11	11
6	11	11
7	11	11
8	11	11
Total	100	100

Name:

Contact (e-mail address or phone number):

Question 1: Imagine evaluating the following expressions in order in the Python interpeter. For each expression 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. Each answer is worth 1 point.

```
>>> x = 5
>>> y = 7
>>> z = 2
>>> y / x - z
    -1
>>> (y / x) - z
    -1
>>> y / (x - z)
    2
>>> y % (x - z)
    1
>>> y % x
    2
>>> [[[]]][0]
    [[]]
>>> len([[1,[2,[3]]],[4],[5]])
    3
>>> len([[1,[2,[3]]],[4],[5]][0])
    2
>>> L = [[1,[2,[3]]],[4],[5]]
>>> L + L[0]
```

```
    [[1, [2, [3]]], [4], [5], 1, [2, [3]]]
>>> L + L[0][0]
```

Error (cannot use `+` with `list` and `int`).

```
>>> 'Ethiopia'[3]
    'i'
>>> 'Ethiopia'[3][0][0][:]
    'i'
>>> 'abcd'[:2]
    'ab'
>>> 'abcd'[2:]
    'cd'
>>> 'abcd'[1:2]
    'b'
>>> int('2')
    2
>>> w = n + 1
>>> w
```

Error (`w` is undefined).

```
>>> y = []
>>> w = 10
>>> while w > 0:
>>>     y += [[w]]
>>>     w /= 2
>>> y
```

```
     [[10], [5], [2], [1]]
>>> def fibonacci(n):
>>>    if n < 2:  return 1
>>>    return fibonacci(n-1) + n - 2
>>> fibonacci(-2)
     1
>>> fibonacci(4)
     4
>>> fibonacci(5)
     7
>>> [] != [[]]
     True
>>> x = 7
>>> x %= 2
>>> x
     1
```

Question 2: Write a function `countZeroes`(n) which takes as input a positive `int` n and outputs the number of zeroes in n. For example, `countZeroes`(10) is 1, `countZeroes`(50803) is 2, and `countZeroes`(547) is 0. What is the running time of your algorithm in terms of the number of digits D in n?

Example solution 1:

```
def countZeroes(n):
    if n == 0:
        return 0
    elif n%10 == 0:
        return 1 + countZeroes(n / 10)
    return countZeroes(n / 10)
```

Example solution 2:

```
def countZeroes(n):
    ans = 0
    while n > 0:
        if n%10 == 0:
            ans += 1
        n /= 10
    return ans
```

Example solution 3:

```
def countZeroes(n):
    s = str(n)
    ans = 0
    for i in xrange(len(s)):
        if s[i] == '0':
            ans += 1
    return ans
```

All three above solutions have running time $\Theta(D)$ (which is $\Theta(\log_2 n)$).

Question 3: You are given a `list` of pairs `L = [[`x_0`,`y_0`], [`x_1`,`y_1`], ..., [`x_{n-1}`,`y_{n-1}`]]` such that $y_i = x_{i+1}$ for all $0 \le i \le n-2$. When you combine adjacent pairs $[x_i, y_i]$ and $[x_{i+1}, y_{i+1}]$ (recall $y_i = x_{i+1}$), the new pair $[x_i, y_{i+1}]$ takes their place in the `list`. Combining this pair has cost $x_i \times y_i \times y_{i+1}$. You need to keep combining adjacent pairs until you're finally left with the single pair $[x_0, y_{n-1}]$, but you can choose the order in which you combine adjacent pairs. Write a function `bestCost(L)` which calculates the minimum cost of how to do this. What is your running time? (Justify your answer.)

For example, consider the input `L = [[1,5],[5,3],[3,7]]`. If you first combine `[1,5]` and `[5,3]`, the cost is $1 \times 5 \times 3 = 15$. Then you are left with the `list [[1,3],[3,7]]`, and combining these two has cost $1 \times 3 \times 7 = 21$. Thus the total cost is $15 + 21 = 36$. The other option is to first combine `[5,3]` and `[3,7]`, for a cost of $5 \times 3 \times 7 = 105$, producing the new pair `[5,7]`. The list is then `[[1,5],[5,7]]`, and combining these has cost $1 \times 5 \times 7 = 35$, for a total cost of $105 + 35 = 140$. Thus, the first option was better, and `bestCost([[1,5],[5,3],[3,7]])` should return 36.

Example solution: Let's rephrase the problem. We have a list of elements, which are pairs. Let's call these elements $x_0, \ldots, x_{n-1}$. We have an operation $\oplus$ that combines pairs (if you do `[a,b]` $\oplus$ `[b,c]`, you get `[a,c]`), and it "costs" you $a \times b \times c$. Now, we would like to evaluate the $\oplus$ operations in $x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$ in an order that minimizes the sum of costs. This is very similar to the "parenthesizing expressions" problem from Lecture 15. We would like to pick the order to do the operations, i.e. parenthesize the expression, so as to minimize the total cost.

The idea is then to use recursion with memoization. Let $f(i,j)$ be the cheapest way of combining the elements in `L[i:j]`. Then,

$$f(i,j) = \begin{cases} 0, & \text{if } i = j \\ \min_{i \le k < j} \{\texttt{L[i][0]} \cdot \texttt{L[k][1]} \cdot \texttt{L[j][1]} \\ \qquad + f(i,k) + f(k+1,j)\}, & \text{otherwise} \end{cases}.$$

The reason is that in any sequence of combining adjacent elements, there's always *some* pair of elements which is the last pair to be combined. That pair is the result of merging everything between i and k, and everything between $k+1$ and j, for some k. So, we try all possible k and take the best choice. Below is an implementation.

```
def f(L, i, j, mem):
    if i == j:
        return 0
    elif mem[i][j] != -1:
        return mem[i][j]
    mem[i][j] = float('infinity')
    for i in xrange(i, j):
        val = f(L, i, k, mem) + f(L, k + 1, j, mem)
        mem[i][j] = min(mem[i][j], L[i][0]*L[k][1]*L[j][1] + val)
    return mem[i][j]

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

The running time is $\Theta(n^3)$.

Question 4: You are given a chessboard which has n rows and m columns. The bottom-left square is $(0, 0)$, and the top-right is $(n-1, m-1)$. You want to get your piece from the bottom-left to the top-right. If your piece is at (x, y), the next step it can either move to $(x+1, y)$, $(x+1, y+1)$, or $(x, y+1)$, as long as the piece stays on the board. There is one catch though. Squares on your chessboard are either green or red, and on odd moves (your first, third, fifth, etc. moves) you can only move to red squares, and on even moves you can only move to green squares. If ever you can't make a move because all squares next to you are the wrong color, then your piece dies.

Write a function `isPossible(n,m,colors)` which returns `True` if it is possible to get from the bottom-left to the top-right corner without dying and `False` otherwise. Also, before writing the code, describe in words how your solution works: your description should not take more than a couple sentences. `colors` is a `list` of n strings each of length m. `colors[i][j]` is `'R'` if square (i, j) is red, and otherwise is `'G'`. For example, `isPossible(3,3,['GGG','RRG','GGG'])` gives `True`. The board is

G G G
R R G
G G G

Example solution: One way to solve this problem is to create a graph: each vertex represents a square in the chessboard together with whether your next move should be to a 'G' or 'R'. An edge goes from one vertex to another if we can go from that square to another and the next moves of both vertices should be different colors. Then we can do BFS or DFS to see whether either the vertex $(n-1, m-1,$ `'G'`$)$ or $(n-1, m-1,$ `'R'`$)$ is reachable from $(0, 0,$ `'R'`$)$. Here's an implementation using recursive DFS.

```
def makeArray(L, val):
    if len(L) == 1:
        return [val]*L[0]
    ans = []
    for i in xrange(L[0]):
        ans += [makeArray(L[1:], val)]
    return ans
```

```
# we're at vertex (x,y,c) where c is 0 if the next move should be red,
# and c is 1 otherwise
def dfs(x, y, c, n, m, colors, seen):
    if x==n-1 and y==m-1:
        return True
    dx = [1,1,0]
    dy = [1,0,1]
    for i in xrange(3):
        nx = x + dx[i]
        ny = y + dy[i]
        if nx>=0 and nx<n and ny>=0 and ny<m:
            cellColor = 0
            if colors[nx][ny] == 'G':
                cellColor = 1
            if c == cellColor and not seen[nx][ny][1-c]:
                # the cell (nx, ny) has the right color
                seen[nx][ny][1-c] = True
                if dfs(nx, ny, 1-c, n, m, colors, seen):
                    return True
    return False

def isPossible(n, m, colors):
    seen = makeArray([n, m, 2], False)
    return dfs(0, 0, 0, n, m, colors, seen)
```

Question 5: Describe an algorithm that does the following. You are given an integer $n \geq 2$ and must return the index i of the first Fibonacci number which is larger than n. Recall the Fibonacci numbers are $F_0, F_1, F_2, F_3, \ldots = 1, 1, 2, 3, \ldots$ (each number is the sum of the previous two). So, if $n = 7$, then the answer should be 5: the first Fibonacci number larger than 7 is $F_5 = 8$. If $n = 2$, then the answer should be 3, since the third Fibonacci number $F_3 = 3$ is the first Fibonacci number to be larger than 2.

You don't need to implement your solution, but you should describe how you would do it and also explain the running time assuming that all arithmetic operations take $O(1)$ time.

Example solution: Note the Fibonacci numbers are increasing (that is, $F_0 \leq F_1 < F_2 < F_3 < \ldots$) and also that $F_n \geq n$ for all $n \geq 1$. Thus, we can do a binary search. We know the answer must be between 0 and n, so we have $\log_2 n$ levels of binary search, and at each level we need to compute `fib(mid)` for some value `mid` between 0 and n. We can write a function `fib(x)` to compute the xth Fibonacci number using matrices and fast powering, as in Lecture 13, so that we can compute `fib(mid)` in $O(\log_2 n)$ arithmetic operations. Thus, this solution takes $\Theta((\log_2 n)^2)$ arithmetic operations.

It is possible to do better by, for example, first proving that for all $i \geq 4$, $F_i \geq 1.5^i$. So, for $n < F_4$, we can just compute the answer with `if` statements. For $n \geq 4$, we can binary search for the answer between 0 and $\lceil \log_{1.5} n \rceil$. Thus, the number of iterations of binary search is $\Theta(\log_2 \log_{1.5} n) = \Theta(\log_2 \log_2 n)$, and in each iteration we have to compute `fib(mid)` for `mid` $= O(\log_2 n)$ thus taking $O(\log_2 \log_2 n)$ arithmetic operations. Thus, overall, this solution takes $\Theta((\log_2 \log_2 n)^2)$ arithmetic operations. The previous paragraph was enough to get 9/11 points for this problem. In fact, a simple solution with a `while` loop also only requires $\Theta(\log_2 n)$ arithmetic operations.

```
def whichFib(n):
    at = 1
    a = 1
    b = 1
    while b <= n:
        c = a + b
        a,b = b,c
        at += 1
        if b > n:
            return at
```

The reason this solution only requires $\Theta(\log_2 n)$ arithmetic operations is that the `while` loop only goes until $F_{\text{at}} > n$, but since $F_{\text{at}} \geq 1.5^{\text{at}}$ for at ≥ 4, the loop can only go $\Theta(\log_2 n)$ steps.

Question 6: You are given a `list` `L` of `int`s that are all bigger than 1. Write a function `findPair(L)` that returns a list `[a,b]` such that $a^2 = b$ and `a`, `b` are both in `L`. If no such pair exists, return `[]`. For example, `findPair([9,5,2,7,3])` should return `[3,9]`. `findPair([5,2,25,4])` can either return `[2,4]` or `[5,25]`. `findPair([9,12,14])` should return `[]`.

Example solution: First we will sort `L`, then for each element a in `L` we will binary search for a^2.

```
# binary search for z in L[x:y]
# returns False if z is not in L[x:y], and True otherwise
def binarySearch(x, y, z, L):
    if x == y:
        return L[x] == z
    mid = (x + y)/2
    if L[mid] == z:
        return True
    elif L[mid] < z:
        return binarySearch(mid + 1, y, z, L)
    else:
        return binarySearch(x, mid - 1, z, L)

def findPair(L):
    L.sort()
    for a in L:
        if binarySearch(0, len(L) - 1, a*a, L):
            return [a, a*a]
    return []
```

Suppose `L` has n elements. Sorting in the beginning takes $\Theta(n \log_2 n)$ time. Then, there is a `for` loop taking n steps, and each time through the `for` loop we spend $\Theta(\log_2 n)$ time to binary search. Thus, the overall running time is $\Theta(n \log_2 n)$. It is possible to solve this problem in $\Theta(n)$ time using a technique known as *hashing*, but we haven't covered that in this course, so this $\Theta(n \log_2 n)$ solution is enough to get full credit.

In fact, it is possible to get $\Theta(n \log_2 n)$ time without binary searching:

```
def findPair(L):
    L.sort()
    at = 0
    for a in L:
        while at < len(L) and L[at] < a*a:
            at += 1
        if at == len(L):
            break
        elif a*a == L[at]:
            return [a, L[at]]
    return []
```

The solution above takes $\Theta(n \log_2 n)$ time to sort, then afterward only spends $\Theta(n)$ time in the `for` and `while` loops. The point is that we try the `a` from smallest to largest in the `for` loop since we've sorted. So, if for some `a` we try `L[i]`,`L[i+1]`,...,`L[j]` in the `while` loop and they were all smaller than a^2 except for `L[j]`, then when we try the next `b` in `L` we don't need to search from the beginning of `L` again: we can just keep searching from `L[j]` (if the previous elements in `L` were smaller than a^2, they'll also be smaller than b^2, so we don't need to look at them again.)

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). Furthermore, on odd moves (the first, third, fifth, etc. moves), you must take the edge out of your current location which is the longest (assume that no two edges have the same length). 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.

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

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. If t is odd though, we should make sure (z,w) is the longest edge leaving z to ensure we arrive at w in the next step.

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 $\Theta((n+m)k)$. Before doing the recursion we should also do a `for` loop over each vertex vertex then over each edge to figure out, for each vertex z, what the longest edge leaving it is.

Question 8: 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? Furthermore, we want to count the number of different ways of making change when we're only allowed to use an *even number of each coin type*. 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 2 ways: (1) give twelve 1-cent pieces, and (2) give two 5-cent pieces and two 1-cent pieces. The other ways would involve giving an odd number of some coin, so we can't do it.

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 2. What is the running time of your solution?

Example solution: This problem is reduces to the `change` problem from the practice exam. Having to use each type of coin an even number of times is the same as saying we can use each coin any number of times, but our coin values are actually $2 \cdot$`L[0]`,$2 \cdot$`L[1]`, etc. So, one way to solve this problem is to just first double each element of `L` then call the `change` function from the practice exam solutions. Or, you could just write it from scratch 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 2*L[x] <= n:
        mem[x][n] += recurse(L, x, n - 2*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.