2D Lists

Code Tracing

```
import  copy
def f(a):
        return  10*a[0][0]+a[1][0]

def ct2(a):
        b   =   copy.copy(a)
        c   =   copy.deepcopy(a)
        d   =   a
        e   =   a[0:len(a)]
        c[0][0] =   1
        d[0]    =   [2]
        e[1]    =   [3]
        b[0][0] =   4
        print(f(b), f(c),   f(d),   f(e))   # f is  defined above

a   =   [[5],   [6]]
ct2(a)
print(f(a)) #   don't   miss    this
```

Reasoning Over Code

```
def rc1(n):
        assert(isinstance(n, int) and
                (n >= 0) and (n < 1234))
        x = y = 0
        for i in range(2, 1234):
                if ((i % (i//2) > 0) and
                   ((i//10) == (i%10))):
                        (x, y) = (y, i)
        return  (x == n)
```

```
def rc2(M):
        assert(isinstance(M, list))
        (i, n) = (3, 1)
        for val in M:
                if (int(str(i)*n) != val): return False
                i -= 1
                if (i == 0): (i, n) = (3, n+1)
        return (len(M) == 7)
```

Big-O Conceptual Questions

Answer the following questions without look at any notes or the course website!!

State the Big-O of each of the following functions:

Function	**Big-O**
L.count(val)	O(N)
len(L)	O(1)
L.append(item)	O(1)
L.insert(0, item)	O(N)
max(L)	O(N)
min(L)	O(N)
sum(L)	O(N)
val in L	O(N)

List the ***worst-case scenario*** big-Os of selectionSort, bubbleSort, and mergeSort. Which is the fastest? Why?

List the ***best-case scenario*** big-Os of selectionSort, bubbleSort, and mergeSort. Which is the fastest? Why? Is this the same answer as the questions above?

Big-O Practice Questions

<table>
<tr><th>Function</th><th>Big-O</th></tr>
<tr><td><pre>
def bigOh1(L):
# assume L is a 1d list
N = len(L)
    for val in copy.copy(L):
        L += [val**2]
        i = N
        while (i > 0):
            L[i] += i
            i //= 4
return (sum(L) / len(L))
</pre></td><td></td></tr>
<tr><td><pre>
def bigOh2(L):
        # assume L is a pre-sorted 1d list
        # (don’t count the cost of sorting L in
        # your answer) assume binarySearch
        # is written as usual
        def f(L): # NlogN
            N = len(L)
            M = [ ]
            for val in L: # N
                M.append(binarySearch(L, val)) #log
            return M
        return f(f(f(L))) # note the nested calls
</pre></td><td></td></tr>
<tr><td><pre>
def bigOh3(x):
        N = math.log(x, 2)
        c = 1
        while (x > 0): (x, c) = (x//42, c+1)   #log x = N
        x = 1
        while (x**2 < c): x += 1   #sqrt(log x)
        return x
</pre></td><td></td></tr>
</table>

wordSearchWithPortals
You may assume that wordSearch and wordSearchFromCell are already written for you.

Given a rectangular board, return true if the word can be formed and false if the word cannot be formed. This will work like word search but with an addition. Instead of a board of all letters there can be tuples containing positions. The moment you see a tuple you should check that position of the board to see if the next letter matches and continue searching for the rest of the word from there. For example

```
board = [ [ 'd', ‘k’, 'g' ],
          [ (0,2), 'a', 'c' ],
          [ 'o', 'a', 't' ],
          [ 'u', 'r', 'k' ],
        ]
```