HW 3

Ian O'Brien

02/20/19

1. Design a recursive algorithm for computing 2^n for any non-negative integer, n ,that is based on the formula $2^n = 2^{n-1} + 2^{n-1}2$:

   a)

   $n \leftarrow$ any non-negative integer...
   test(n):
   if $n == 0$:
   return 1;
   return test(n-1) + test(n-1);

   b)

   basic operaton for this algorithm is addition in the recursive call and we can represent the number of times it was done by a recurrence relation...

   T(0) = 0
   T(1) = 1
   T(2) = 3
   T(3) = 7
   T(4) = 15
   T(5) = 31
   soo....it appears that the recurrence relation is...

   T(n) = $2^n - 1$ for the number of additions in this algorithm

   c)

   I would say for this specific problem that this is not a best solution, we could find an iterative algorithm that performs its

basic operation, multiplaction, n-2 times with the following...

The following algorithm will do 2 comparisons + n-2 additions,
which is an improvement from the previous recursive algorithm.

This is linear complexity as oppposed to exponential, big improvement

```
def test(n):
        answer ← 2;
        if i == 0:
            return 1;
        if i == 1:
            return 2;
        else:
            for i ← 0...(n-1):
                answer *= 2;
        return answer;
```

2. Starting with a map represented by a rectangular grid with height h and width w. Cells are numbered (0, 0) in the top left to (h−1, w−1) in the bottom right. Each cell in the grid is either empty or contains an impassable object. All movements on the map are done as a single step to a cell that is adjacent horizontally or vertically. No diagonal movement is permitted.

   a)

   breadth first search algorithm to find shortest path in maze with blocks:
   **keeps count of shortest path, and route of shortest path
   in previous[ ] array**

```
        // visit each vertex that isnt a block and
        // mark as not visited
        for each vertex v:
            if block(v) == false:
                visited(v) = false;
                previous[v] = NULL;
```

```
// initalize empty queue and add
// starting vertex to it
Q = empty-queue
visited(v) = true;
count ← 1
end-of-maze ← false
enqueue(Q, v);
// while queue is not empty visit each vistable vertex
// adjacent to starting vertex
while(Q != empty):
        v = dequeue(Q)
        ++count
                for each vertex, a, adjacent to v:
                        if blocked(a) == false  a != NULL:
                                if visited(a) == false:
                                        visited(a) = true;
                                        previous[a] = [v].append(previous[v]);
                                        if a == END OF MAZE:
                                                end-of-maze ← true
                                                break;
                                        enqueue(Q, a);
        if end-of-maze == true
                return count;
        else
                return -1;
```

b)

The worst case time complexity for this algorithm is the case that every vertex is unblocked and has to be visited. The most "costly" operation for the breadth first search is probably visiting a node, in this case for our maze it is the case. (it would depend on the work you are doing at each visit). So our complexity can be calculated...

worst case: $A = \sum_{i=1}^{n} 1 = n$ $\Rightarrow$ O(n), linear complexity

it is worth noting, though, it is not the case for our maze that a dense graph with different rules on traverseal would have a different

complexity closer to O(vertexes + edges)

c)

The algorithm I implemented in part A has a built in array for all previous nodes visited at each node, each time you visit a node in the algo you add the current node to a list of the previous nodes visited to get to that node. Once you get to the end of the maze you can look at the final nodes data and see the path that was taken to get there by looking at that nodes previous [ ] array. This does not introduce any more time complexity since we are still doing a constant amount of work at each vertex, however; this does increase the amount of memory used significantly for this problem. Each node visited with have an array with the path of all the previous nodes visited to get there, this would be fairly close to $O(n^2)$ space complexity.