Youssef Naguib <ymn4543@rit.edu>
CS261
Write-up for hw-3 questions 3-5

3.
My algorithm calculates the number of right triangles given n 2d points. At first I was not sure how to do this without iterating through every triplet possible and using trigonometry or linear algebra. After much thought I devised a much more efficient algorithm. The total right angles variable is set to 0 to begin with. The algorithm iterates through every point and calculates the slope to each other point, keeping track of undefined slopes. The slopes are stored in an array, and sorted using merge sort. After the array is sorted, the algorithm iterates through each slope and find the inverse slope by using the formula (-1/slope). If the slope is 0, the number of right angles possible at that angles is equal to the number of undefined slopes. Otherwise, Binary search is used on the slope list to find if the inverse slope is in the slope list. If it is, neighboring indices are also checked to see if there are duplicate inverse slopes. The right triangle variable is increased accordingly.
This algorithm is valid for the following observations:

- For every point, every slope to other point is calculated and stored in an array. Every right angle possible at that vertex can be found using these slopes, as any two inverse slopes that are both in the array are perpendicular.
- To account for floating point errors, I store slopes as mini arrays that represent fractions, where the numerator is the first element and the second is the denominator. This way the inverse of the inverse is never calculated and no floating point errors are possible. My modified merge sort stores the slopes by dividing the numerator and denominator, but the actual decimal slope is not used for comparison.
- For any slopes equal to 0, the number of right angles possible with that slope would be undefined slopes. This is a variable I keep track of. All possibilities are accounted for.

Running time analysis:

- Iterate through each point O(N)
- Iterate through all other points and find slope O(N)
- Use Merge sort on slopes O(NlogN)
- Binary search on N-1 slopes O(logN)
- Many O(1) operations in between
  Total complexity is O(N) * (O(N) + O(NlogN) + O(logN) + O(1)) = O(N) * O(NlogN)

= O(N^2logN)

4.

The solution to the candy problem is a greedy one. We are spending all the money we have, and the goal is to purchase as many types of candies possible. Each candy has an associated price, and we want to maximize the types of candy bought, not the number of candies bought. (This means we would not want to buy the same type twice). My approach to this problem was to buy the cheapest candy available, and repeat until no more candies could be bought. This is done using a greedy algorithm. First the candy prices are stored in an array. The select_rand() function is used to find the k'th smallest element in an array, so I set k = 0 to find the cheapest candy. I then subtracted that candy's price from the allowance and removed that candy price from the array of prices, so that it's never bought again. This process is repeated, and k never changes, until an candy price is returned that is too expensive to be bought with the remaining allowance. This approach is valid because it maximizes the number of candies bought by choosing the cheapest candy each time, minimizing cost.

Running time analysis:

- Select_rand() has an expected running time of O(N), and it is repeated a constant number of times.
- Total running time = C * O(N) = O(N)

5.

Heart of the Algorithm:

tile_list is the dynamic programming array

1- **Verbal description**: tile_list[k] = number of possible tiling's for a backsplash of dimensions 2*k.

2- **Solution:** Solution will be found at tile_list(n) where n is the width of the backsplash wall.

3- **Recurrence:** tile_list[k] = (tile_list[k-1]) + (4 * tile_list[k-2]) + (2 * tile_list[k-3])

Base cases:

K	tile_list[k]

0	1
1	1
2	5

Running time analysis:

For each subproblem in range of 3 to k, where tile_list[k] represents the solution, we do a constant amount of work .

= O(N) * O(1) = O(N)