Homework 8

Solution Key

This homework must be typed in LaTeX and handed in via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

Problem 1

1. Rob is building a fence for his farm to protect against the invading squirrels. To ensure he can always go to and from any place of his farm in the minimum distance possible, he constructed his fence in the shape of a convex polygon. Just as he finished constructing his fence he realizes he missed a post. Assume Rob's farm is on a grid and his current fence posts' positions are given by $Q = [q_0, q_1, \cdots, q_{n-1}]$ sorted in counterclockwise order with respect to q_0.

   Describe a linear time algorithm that Rob can use to fix his fence to ensure the missed pole is included in his fence (i.e. is either used as part of the fence or contained entirely within it). Show that your proposed algorithm is correct and analyze its running time.

2. To expand his farm Rob blindfolds himself, walks around his farm and the surrounding area randomly and plants a post into the ground. If required, he then rebuilds part of his fence to include the new post. He repeats this process as many times as he chooses.

   (a) Propose an algorithm that uses the Graham scan so that after the addition of n new posts the overall running time is $O(n^2 \log n)$. Prove the correctness and analyze the running time of the proposed algorithm.

   (b) Show how to improve this slightly, by showing that Rob can construct his fence in $O(n^2)$. Prove the correctness and analyze the running time of the proposed algorithm.

1. **Algorithm**

Algorithm 1 Rob's Farm Algorithm

Input: A convex hull Q given as a list of vertices, the first vertex being an anchor and a point p to be added/contained in the hull
Output: A new convex hull Q' that is composed of/contains all points in $Q \cup \{p\}$

```
procedure AddPointToConvexHull(Q, p)
    if q_0.y ≥ p.y then
        q_t ← q ∈ Q such that q.y is maximal
        Q ← [q_t, ..., q_{n-1}, q_0, ..., q_{t-1}]
    for i from 1 to n − 1 do                                  ▷ Insert p into Q in CCW order
        if Orientation(Q[0], Q[i], p) is CW then
            Q.insert(p, i)
            break
    return GrahamScan(Q)
```

Correctness: The only precondition necessary for valid inputs for Graham Scan is that the points be sorted around an anchor point. We maintain this precondition in all scenarios before running the Graham Scan.

- When the new point p doesn't replace the old anchor q_0, inserting p into Q would maintain the ordering.
- When the new point p replaces the old anchor q_0, we transition to using the top-most point q_t as the anchor and shift Q around q_t. This preserves the ordering between elements. Inserting p into the rearranged version of Q will still maintain the ordering.

Hence, the new convex hull is correct.

Running Time: We'll analyze each step of the algorithm.

1. Checking whether the new point displaces the old anchor is done in constant time.
2. If the new point displaces the old anchor, finding the top-most point and shifting Q both takes linear time with respect to the size of Q.
3. Inserting the new point into either Q or the rearranged version of Q takes worst-case linear time with respect to the size of Q.
4. Graham Scan runs in linear time with respect to the size of the input, which is Q.

Therefore, the algorithm overall runs in linear time with respect to the size of Q.

2. (a) This is a simple and naive algorithm that sorts and runs the Graham Scan on all points once a new point is received. This will run in $O(n \log n)$ since sorting takes $O(n \log n)$ in the worst case and the Graham Scan is linear in the size of the input. Because we will receive n points one-by-one, we need to run this procedure n times, making the overall running time $O(n^2 \log n)$.

   (b) We reuse our algorithm from part 1 on each new point we receive. Since we can preform each addition in $O(n)$ time and wish to add an additional n points one-by-one to our original convex hull, this algorithm will take $O(n^2)$ time to complete.

Problem 2

Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. Describe the overall algorithm and how you would approach computing the hashcodes used in Rabin-Karp. Your algorithm should not use more than $O(m)$ additional memory and should require at most $O(n^2 m)$ time. Prove the correctness of your solution, analyze its running time and memory space utilization.

Intuitively, we slide our pattern matrix P across our character matrix C to search for matches. We consider each row of our pattern matrix as a Rabin-Karp pattern and therefore store its hash in an array. We then also store the hash of the current subset of C that is currently being processed. If the two hashes match then we can employ a brute force check of the subset, otherwise we simply update our hash and check the next possible substring. We continue this until we have checked an entire $n \times m$ subset of C. If a match hasn't yet been found then we "slide" our pattern P down a row and repeat.

Let Σ represent our character set and $h : \Sigma \to \mathbb{Z}$ be a hash function. For example if $\Sigma =$ the English alphabet then we can choose h to be the hash function that maps a letter to its corresponding ASCII value. We can update our hashes for consecutive substrings using the following simple algorithm:

```
procedure updateHash(oldHash, oldChar, newChar, m)
    return (oldHash - h(oldChar)|Σ|^(m−1))|Σ| + h(newChar)
```

Algorithm 2 Multidimensional Rabin-Karp

```
Input: An n × n character matrix C and a m × m pattern matrix P
Output: A boolean representing if there exists a match of P in C
procedure mRabinKarp(C, P)
    patternHashes ← a list of size m where each element is a hash of
    the corresponding row in the pattern P. Horner's method is able
    to do this efficiently.
    for r ∈ [0, n − m] do
        currentHashes ← a list of size m where the ith index is given
        by the hash of C[r + i][0:m]
        for c ∈ [0, n − m] do
            if patternHashes = currentHashes and P = C[r:r + m][0:m] then
                return true
            if c ≠ n − m then
                for (i, oldHash) ∈ enumerate(currentHashes) do
                    oldChar ← C[r + i][c]
                    newChar ← C[r + i][c + m]
                    currentHashes[i] = updateHash(oldHash, oldChar, newChar, m)
    return false
```

Note: For a matrix M we use the operation $[*{:}*]$ to represent a slice of the matrix on a particular axis. This operation is linear in the size of the axis.

Correctness: Proving correctness is equivalent to proving that the algorithm returns `true` if

and only if the pattern exists in C. We prove this both ways.

1. Pattern exists $\implies$ returned `true`. By the method shown in lecture, the Rabin-Karp updating correctly produces hashes for patterns. If two 1D patterns consist of the same characters, they must hash to the same thing, since hashing is deterministic. Inductively, we extend this to say that if the $m \times m$ pattern is found, all m rows will hash to the corresponding target value (stored in `patternHashes`), and therefore equality will be checked and correctly confirmed. Since we loop over all possible positions of the pattern, we will eventually find it. Therefore, if the pattern exists, we eventually hash it, and return `true` once we confirm the equality.
2. Pattern does not exist $\implies$ returned `false`. In order to return `true`, the algorithm manually confirms that equal hashes are not just collisions, but rather that the two strings do indeed coincide. Because of this manual check, if the pattern does not exist, `true` can't be returned.

Runtime: Row-by-row analysis: Each row begins with m patterns of length m being hashed from the character array, which takes time $O(m^2)$. As we slide the window to the right, updating each pattern-row takes constant time, which means that updating all m of the rows takes $O(m)$ time. We do this for $O(n)$ shifts until we get to the end of the character array. This shifting process takes in total $O(nm)$ time. Since $n > m \implies nm > m^2$, each row takes time $O(nm + m^2) = O(nm)$. Because there are $O(n)$ rows, the overall time is $O(n^2m)$. Collisions may occur, but assuming that our hash function maps uniformly to a large interval, they are pretty unlikely. Also, a collision in this problem requires all m of the different pattern's hashes to be equal, which is significantly more unlikely than a collision between just two unique patterns.

Space: The extra space we store consist only of arrays of length m, so the space complexity is $O(m)$.

Problem 3

Anna is mixing paint for her house. Since her favorite color is green she's mixing some blue paint and yellow paint together with some paint thinner. She's managed to create n different shades of green with all the blue and yellow paint that she has.

For example, suppose she made three different shades of green.

		Samples		
% Compound		S_1	S_2	S_3
	Yellow	0.7	0.3	0.1
	Blue	0.2	0.1	0.7
	Paint Thinner	0.1	0.6	0.2

Then it is possible to produce a shade of green that is 35% yellow and 27.5% blue by mixing the shades she currently has in a $1:2:1$ ratio (25% S_1, 50% S_2, 25% S_3). However, it is **impossible** to create the shade of green which is 20% yellow and 10% blue.

Design an $O(n \log n)$ algorithm that checks whether it's possible to create a liquid with the specified percentage of yellow and blue. Argue the correctness of your algorithm.

Example Input: [(0.7, 0.2), (0.3, 0.1), (0.1, 0.7)], (0.35, 0.275)
Output: True

Algorithm

```
procedure PointInclusion(P, q)
    xMax ← p.x ∈ P such that p.x is maximal
    horizonalSegment ← (q, (xMax, q.y))
    E ← {}
    for i in range(0, S.size()) do
        j ← (i + 1) mod S.size()
        E.append((S[i], S[j]))
    intersectingEdges ← E.filter(e => Intersect(horizontalSegment, e))
    return intersectingEdges.size().isOdd()
procedure Mixable(S, l)
    S.sort(OrientationComparator)
    GrahamScan(S)
    return PointInclusion(S, l)
```

Correctness: The algorithm checks to see if a certain shade of paint is mixable by checking to see if it is contained in the convex hull formed by the original shades of paint. Let $C = \{v_1, \cdots, v_m\}$ denote the convex hull of the original shades of paint. Notice that a shade is only mixable if and only if it can be represented as a linear combination $a_1 v_1 + \cdots + a_m v_m$ where $a_1 + \cdots + a_m = 1$, i.e. a convex combination of $v_1, \cdots, v_m$. One can think of each a_i as the percentage of the shade v_i used to create the mixed shade. By the hint, the convex hull is the set of all convex combinations of the original n shades and thus every mixable paint is an element of the convex hull and every point in the convex hull represents a mixable paint.

Runtime: We begin by preforming an $O(n \log n)$ sort on the original set of paints. We then preform the Graham scan on this set of paints to form the convex hull, which is done in $O(n)$ time. Finally, we check to see if the input shade is contained within the convex hull using point inclusion. Determining the maximum x coordinate in our convex hull requires a linear scan of the hull which in the worst case contains $O(n)$ points. We can create the horizontal segment for the point we are checking in constant time and extract all the edges from our convex hull in linear time. Finally, filtering the set of edges to determine the number of intersections with our horizontal line segment also requires $O(n)$ time. In total checking if a point is included in the convex hull requires at worst $O(n)$ operations. Thus, the overall algorithm requires $O(n \log n)$ time where we are limited by time required to preform our original sort.