MPCS 55001 Algorithms Autumn 2023
Homework 1
Due 11:59 pm, Monday October 2 2023

Zhiwei Cao

Sep 27

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.** **Collaboration policy:** You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**. **Internet sources:** You must include the url of any internet source in your homework submission.

1 Square Roots (8 points)

(a) (4 points) Given a positive integer n, write **pseudocode** for a **recursive divide-and-conquer** algorithm to find the floor of the square root of n, $\lfloor\sqrt{n}\rfloor$.

Algorithm 1 Find Floor Square Root

```
1: function findFloorSquareRoot(n, left, right)
2: Input: n: An integer n.
3: Input: left: An integer 1, the left boundary of the checking list.
4: Input: right: An integer right, the right boundary of the checking list, which is the value of n.
5: Output: The floor of the square root of n
6: mid ← ⌊(left+right)/2⌋
7: if mid * mid = n then
8:     return mid
9: end if
10: if mid * mid < n then
11:     return findFloorSquareRoot(n,mid+1,right)
12:     // update the boundary
13: else
14:     return findFloorSquareRoot(n,left, mid-1)
15:     // update the boundary
16: end if
```

(b) (3 points) **Prove** the correctness of your algorithm using induction. (Recall that correctness proofs for recursive algorithms typically rely on induction, whereas correctness proofs for iterative algorithms typically rely on a loop invariant).
Base Case: When n = 0 or n = 1, the floor of the square root of n is themselves. Proved
Inductive Step: Assume the algorithm is correct for all integers k, for $k >= 2$
It is required to show the algorithm is true under the n = K+1 condition
In the while loop, there are three conditions.
If mid * mid = n, find the square root of the n
If mid * mid < n, the left value update to mid + 1 which shrink the interval. However, the inductive hypothesis still ensures that the floor root value of n insides the interval [mid+1,..., right]. At this moment, the mid value is fine to store as the results as the floor square root value, since the mid value is the minimum value of the interval.
If mid * mid > n, the interval will shrink to [left,...,mid-1]. Since, we already proved that the algorithm is

correct under the range from 0 to k, and now mid-1 is less than k. Under this condition, the statement is proved to be true.
In Sum, the inductive step is true, and the Base case is proved.

(c) (1 point) Write and solve a running time recurrence for your algorithm.
This algorithm always half shrinks the boundary of the solution, instead of splitting the question into several sub-problems. The algorithm is an increasing function that satisfies the recurrence relation. After each iteration, the search space is halved from n to n/2. As a result, the running time of the algorithm is $O(logn)$

2 Closest Pair (12 points)

In this problem, we will develop a divide-and-conquer algorithm for the following geometric task. CLOSEST PAIR *Input*: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$. *Output*: The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between p_i and p_j, that is,

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

is minimized. For simplicity, assume that n is a power of two, and that all the x-coordinates x_i are distinct, as are the y-coordinates. Here is a high-level overview of the algorithm:

- Find a value x for which exactly half the points have $x_i < x$, and half have $x_i > x$. On this basis, split the points into two groups, L and R.
- Recursively find the closest pair in L and in R. Say these pairs are p_L, $q_L \in L$ and $p_R, q_R \in R$, with distances d_L and d_R, respectively. Let d be the smaller of these two distances.
- It remains to be seen whether there is a point in L and a point in R that are less than distance d apart from each other. To this end, discard all points with $x_i < x - d$ or $x_i > x + d$ and sort the remaining points by y-coordinate.
- Now go through this sorted list, and for each point, compute its distance to the seven subsequent points in the list.[1] Let p_M, q_M be the closest pair found in this way.
- The answer is one of the three pairs $\{p_L, q_L\}$, $\{p_R, q_R\}$, $\{p_M, q_M\}$, whichever is closest.

(a) (2 points) In order to prove the correctness of this algorithm, start by proving the following property: any square of size $d \times d$ in the plane contains at most four points of L.

Consider step four, there is a strip whose width is 2d. Generate a square of size $d * d$ in the L side strip. At this moment, there are exactly 2 points that can exist in this square, in two corners. If there is another point exists, the value of d will be changed, which means the closest pair in the L changed and the side of the strip changed.
Now, put the square randomly in the L. It can only contain at most four points, which are in four corners. Since it already limited the d value ($min(dL, dR)$). If there is a fifth point in the square, then the dL is changed to a small value, and the d will be changed which contradicts the algorithm.
In Sum, with a given d value, there are only can exist at most 4 points in a square of size $d * d$ in the L.

(b) (4 points) Now prove that the algorithm is correct. The case which needs careful consideration is when the closest pair is split between L and R.

Since the algorithm is doing the split based on the central point all the time, and finds the d to limit the space for the next split. The point is that it only splits the space into L and R and finds the dL and dR separately in L and R. This process may lose the closest pair when these two points are located in L and R (different sides). As a result, the case that the closest pair is split between L and R is required to be carefully considered.

(c) (3 points) Write **pseudocode** for the algorithm.

[1] Try to figure out why 7 points are checked. It may be possible that this constant can be lowered given the specific restrictions for this problem, namely that the points are distinct.

Algorithm 2 Sort Closest Pair

function `sortAndFindMidPoint`($pointList$)
Input: $pointList$: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$.
Output: The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between p_i and p_j, that is, $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, is minimized.
$pointListx \leftarrow pointList$ sorted by the x coordinate
$pointListy \leftarrow pointList$ sorted by the y coordinate

Algorithm 3 Distance

function `distance`($p1, p2$)
Input: $p1, p2$: two points' coordinate
Output: Distance between p1 and p2
return $\sqrt{(p1.x - p2.x)^2 + (p1.y - p2.y)^2}$

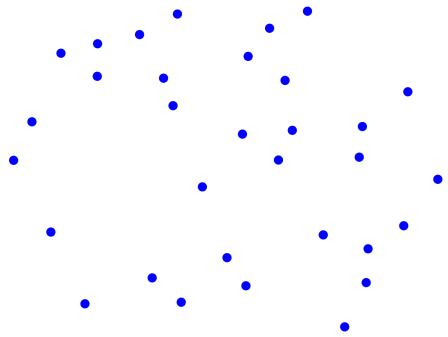
Algorithm 4 Claculate Closest Pair

function `claculateClosestPair`($poinntList$)
Input: $pointList$: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$.
Output: The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between p_i and p_j, that is, $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, is minimized.
$minDis \leftarrow \infty$
$minPair \leftarrow (poinntList[0], poinntList[1])$
for i in $1, \ldots, pointList - 1.length$ **do**
    for j in $i + 1, \ldots, pointList.length$ **do**
        if $distance(pointList[i], pointList[j]) < minDis$ **then**
            $minDis \leftarrow distance(pointList[i], pointList[j])$
            $minPair \leftarrow (stripList[i], stripList[j])$
        end if
    end for
end for
return $minPair$

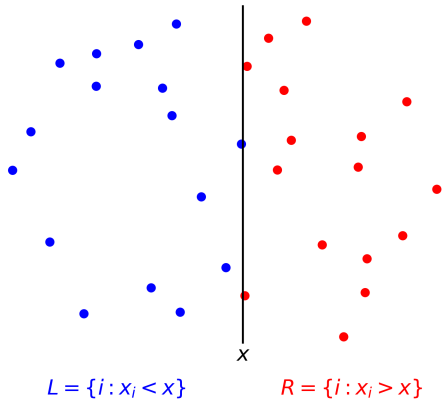
Algorithm 5 Find Closest Pair

return colsestPairRecurssive($pointListx, pointListy$)
function `colsestPairRecurssive`($pointListx, pointListy$)
Input: $pointListx$: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$ which sorted by the x coordinate
Input: $pointListy$: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$ which sorted by the y coordinate
Output: The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between p_i and p_j, that is, $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, is minimized.

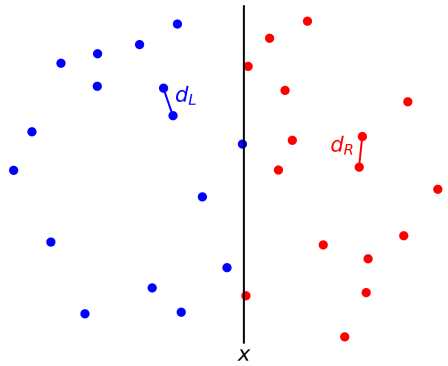
if $pointListx.length <= 3$ **then**
    return claculateClosestPair($pointListx$) // after recursive, it will provide sub-questions
end if
mid $\leftarrow \left\lfloor \frac{\text{pointListx.length}}{2} \right\rfloor$
$midPoint \leftarrow pointListx[mid]$
$leftByX \leftarrow pointListx[1, \ldots, mid-1] = \{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_{mid-1} = (x_{mid-1}, y_{mid-1})\}$
$rightByX \leftarrow pointListx[mdi, \ldots, n] = \{p_{mid} = (x_{mid}, y_{mid}), p_{mid+1} = (x_{mid+1}, y_{mid+1}), \ldots, p_n = (x_n, y_n)\}$
//https://hideoushumpbackfreak.com/algorithms/algorithms-closest-pair.html
$leftByY \leftarrow \emptyset$
$rightByY \leftarrow \emptyset$
for point in pointListy **do**
    if $point.x \leq midPoint.x$ **then**
        $leftByY \leftarrow leftByY \cup point$
    else
        $rightByY \leftarrow rightByY \cup point$
    end if
end for
$(xL, yL) \leftarrow colsestPairRecurssive(leftByX, leftByY)$
$(xR, yR) \leftarrow colsestPairRecurssive(rightByX, rightByY)$
//calculate the distances
$dL \leftarrow distance(xL, yL)$
$dR \leftarrow distance(xR, yR)$
$d \leftarrow \min\{dL, dR\}$
$stripList \leftarrow \emptyset$
for point in pointListy.length **do**
    if $|point.x - midPoint.x| < d$ **then**
        $stripList \leftarrow stripList \cup point$
    end if
end for
$stripList \leftarrow stripList$ sorted by the y coordinate
$minPair \leftarrow (stripList[0], stripList[1])$
for i in $0, \ldots, stripList.length$ **do**
    for j in $i+1, \ldots, \min\{i+7, stripList.length\}$ **do**
        // https://hideoushumpbackfreak.com/algorithms/algorithms-closest-pair.html
        if $distance(stripList[i], stripList[j]) < d$ **then**
            $d \leftarrow distance(stripList[i], stripList[j])$
            $minPair \leftarrow (stripList[i], stripList[j])$
        end if
    end for
end for
$(pM, qM) \leftarrow minPair$
if $dL \leq DR$ & $DL \leq distance(pM, qM)$ **then**
    return (pL, qL)
else if $dR \leq DL$ & $dR \leq distance(pM, qM)$ **then**
    return (pR, qR)
else
    return (pM, qM)
end if



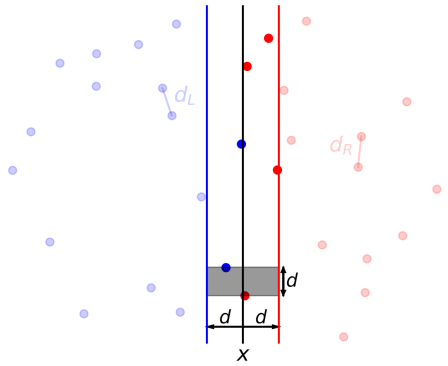
(i) Task: find the closest pair among all points



(ii) Split the points into two groups, L and R



(iii) Recursively find the closest pair in L and in R



(iv) Discard all points with $x_i < x - d$ or $x_i > x + d$

(d) (2 points) Argue that its running time is given by the recurrence:

$$T(n) = 2T(n/2) + O(n \log n).$$

You should refer to your pseudocode.
Splitting the list into two parts takes O(1) which is constant.
The algorithm always half the point list and does the recursive step, the running time is 2T(n/2)
In the sorting step which to sort the point based on the x-coordinate. This step takes O(n logn) time based on the worest case of sorting O(n) + O(n logn)
In sum, T(n) = 2T(n/2) + O(n) + O(n logn) = 2T(n/2) + O(n logn)

(e) (1 point) Show that the solution to this recurrence is $O(n \log^2 n)$.
Base on the recursive step. Step one only takes O(n logn), and step two has two recursive calls of (n/2) which provide 2 * ((n/2)*(log(n/2))) = O(n logn). Based on this pattern, step three has 4 * ((n/4)*(log(n/4))) = O(n logn), step four has 8 * ((n/8)*(log(n/8))) = O(n logn). Since the algorithm always halved, it runs this kind of step for logn time. It takes O(n logn) every time, so the solution of the recurrence is O(n log2n)

(f) (1 point) Can you improve the running time to $O(n \log n)$? Explain your proposal.
We could, when the algorithm reaches step 4 and is ready sort the points by y-coordinate. This sorting based on y-coordinate takes extra time for every recursive call. It is functional if the algorithm can maintain the sorted point list, which doesn't require sorting the list again. This could improve the time to $O(n \log n)$

(*Credit*: Dasgupta et al. 2008.)

3 Tracking Drones (12 points)

With drones becoming more prominent, it is useful to be able to quickly find their position. However your drone is no longer transmitting GPS data, but as it moves it connects to the cellular towers. You cannot ask the network where your drone is, but the cellular towers will answer you if your drone is still connected to them or tell you which tower they handed it off to. The cellular towers are arranged in an N-by-N grid of cells $(x, y) \in [N] \times [N]$. The drone's initial position is in the southwest corner $(x, y) = (1, 1)$ of the grid. The drone will stay in the grid, always moving from a square to an adjacent square in one of the four cardinal directions. It never leaves the grid, and never visits the same cell twice. You are given the N-by-N matrix A, whose entries are

- $A[i, j] =$ $*$ if the drone never visits cell (i, j).
- $A[i, j] = \uparrow$ if the drone visited cell (i, j) and departed upward.
- $A[i, j] = \leftarrow$ if the drone visited cell (i, j) and departed in the leftward direction.
- $A[i, j] = \downarrow$ if the drone visited cell (i, j) and departed downward.
- $A[i, j] = \rightarrow$ if the drone visited cell (i, j) and departed in the rightward direction.
- $A[i, j] =$ `DRONE` if the drone is connected to the tower in cell (i, j).

For example, $A[1, 1]$ will contain an arrow indicating which cellular tower the drone was handed over to.

(a) (6 points) Write **pseudocode** for an algorithm that makes only $O(N)$ queries to the matrix A and locates the drone. In other words, your algorithm should only read $O(N)$ entries of the matrix. (A maximum of 2 points will be given for correct $O(N \log N)$ algorithms.)
Consider the worst case first: it starts from the [1][1] and moves to right all the way, then moves up all the way, moves left all the way, then moves down until (1,3), which there is an unvisited cell between the last cell and the first cell

(b) (5 points) Prove the correctness of your algorithm. State your base case and state the invariant that your algorithm maintains.

Initialization: Start the For loop iteration. It is based on the direction provided in the Matrix[1][1]. Pass the if statement to determine what is the next cell it is going to. As a result, based on the single element M[1][1] which is the initial element of the iteration can figure out the next step.
Maintenance: Informally, the body of the for loop works by moving the (x,y) coordinates to different directions like (x+2,y) (x-2,y), (x+1,y+1), and so on until it finds the location of the drone. The sub-problem of each cell's direction could shrink the space required to check, which speeds up the algorithm. However, based on the problem description, the drone only stays in the N * N matrix, so the direction will not be over the boundary of the matrix, which maintains the correctness of the algorithm during the for loop iteration.
Termination: Finally, when we reach the end of the loop. The condition causing the for loop to terminate is that $index > N + 1$. Because each loop iteration increases $index$ by 1, we must have $index = N + 2$ at that time. At that mention, the algorithm reaches the worst case, which iterates all the cell of the matrix but finds the drone at the end.

(c) (1 point) Argue that your algorithm makes $O(N)$ queries.
For the worst case situation of the algorithm: it starts from the [1][1] and moves to right all the way, then moves up all the way, moves left all the way, then moves down until (1,3), which there is an unvisited cell between the last cell and the first cell. In this situation, it can only just a cell each time, at the end it requires an N+1 step to reach the drone. The time complexity at this moment is O(n+1). However, at this time complexity of O(n+1) simplifies to O(n) for the worst case. For other optimal cases, the time complexity is still O(n).

(*Credit*: This problem is due to Professor Andrew Drucker who was inspired by The Family Circus.)

4 Programming: Getting Drinks (20 points)

Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; do not include it here.

Algorithm 6 Find Drone

```
1: function findDrone(N, M)
2: Input: N: Given value, the width of the matrix
3: Input: M: Given N-by-N matrix
4: Output: (x, y): The location of the drone
5: for index = 1toN + 1 do
6:     // return condition
7:     if M[x, y] = DRONE then
8:         return (x,y)
9:     else if M[x, y] =↑ then
10:        if M[x, y + 1] = DRONE then
11:            return (x,y+1)
12:        else if M[x, y + 1]! = DRONE then
13:            if M[x + 1, y] = ∗ then
14:                if M[x, y + 1] =↑ then
15:                    y ← y + 2
16:                else if M[x, y + 1] =← then
17:                    x ← x − 1
18:                    y ← y + 1
19:                else if M[x, y + 1] =→ then
20:                    x ← x + 1
21:                    y ← y + 1
22:                end if
23:            else if M[x + 1, y]! = ∗ then
24:                if M[x + 1, y] =↑ then
25:                    x ← x + 1
26:                    y ← y + 1
27:                else if M[x, y + 1] =→ then
28:                    x ← x + 2
29:                else if M[x, y + 1] =↓ then
30:                    x ← x + 1
31:                    y ← y − 1
32:                end if
33:            end if
34:        end if
35:    else if M[x, y] =← then
36:        if M[x − 1, y] = DRONE then
37:            return (x-1,y)
38:        else if M[x − 1, y]! = DRONE then
39:            if M[x, y + 1] = ∗ then
40:                if M[x − 1, y] =↑ then
41:                    x ← x − 1
42:                    y ← y + 1
43:                else if M[x − 1, y] =← then
44:                    x ← x − 2
45:                else if M[x − 1, y] =↓ then
46:                    x ← x − 1
47:                    y ← y − 1
48:                end if
49:            else if M[x, y + 1]! = ∗ then
50:                if M[x, y + 1] =↑ then
51:                    y ← y + 2
52:                else if M[x, y + 1] =→ then
53:                    x ← x + 1
54:                    y ← y + 1
55:                else if M[x, y + 1] =← then
56:                    x ← x − 1
57:                    y ← y + 1
58:                end if
59:            end if
60:        end if
```

Algorithm 7 Find Drone

```
1: function findDrone (continue) (N, M)
2: Input: N: Given value, the width of the matrix
3: Input: M: Given N-by-N matrix
4: Output: (x, y): The location of the drone
5: if M[x, y] =↓ then
6:     if M[x, y − 1] = DRONE then
7:         return (x,y-1)
8:     else if M[x, y − 1]! = DRONE then
9:         if M[x + 1, y] = ∗ then
10:            if M[x + 1, y] =↓ then
11:                y ← y − 2
12:            else if M[x + 1, y] =← then
13:                x ← x − 1
14:                y ← y − 1
15:            else if M[x + 1, y] =→ then
16:                x ← x + 1
17:                y ← y − 1
18:            end if
19:        else if M[x + 1, y]! = ∗ then
20:            if M[x + 1, y] =↓ then
21:                x ← x + 1
22:                y ← y − 1
23:            else if M[x + 1, y] =↑ then
24:                x ← x + 1
25:                y ← y + 1
26:            else if M[x + 1, y] =→ then
27:                x ← x + 2
28:            end if
29:        end if
30:    end if
31: else if M[x, y] =→ then
32:    if M[x + 1, y] = DRONE then
33:        return (x+1,y)
34:    else if M[x + 1, y]! = DRONE then
35:        if M[x + 1, y] = ∗ then
36:            if M[x, y + 1] =→ then
37:                x ← x + 2
38:            else if M[x, y + 1] =↑ then
39:                x ← x + 1
40:                y ← y + 1
41:            else if M[x, y + 1] =↓ then
42:                x ← x + 1
43:                y ← y − 1
44:            end if
45:        else if M[x, y + 1]! = ∗ then
46:            if M[x, y + 1] =↑ then
47:                y ← y + 2
48:            else if M[x, y + 1] =← then
49:                x ← x − 1
50:                y ← y + 1
51:            else if M[x, y + 1] =→ then
52:                x ← x + 1
53:                y ← y + 1
54:            end if
55:        end if
56:    end if
57: end if
```