CS 170 HW 5 (Optional)

Due **2021-10-04, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

You may submit your solutions if you wish them to be graded, but they will be worth no points

2 True and False Practice

For the following problems, justify your answer or provide a counterexample.

(a) True or False: Kruskal's works with negative edges.

Solution: True. The cut property holds even with negative edges. We can add a constant to each edge to make all positives and run Kruskal's if we want positive edges.

(b) We modify a graph G with negative edges by adding a large positive constant to each edge, making all edges positive. Let's call this modified graph G'.

True or False: If we run Dijsktra's on G', the resulting shortest paths on G' are also the shortest paths on G.

Solution: False. Longer paths are penalized in G'. This means that a shortest path with many edges in G is no longer the shortest path in G'.

(c) Let $G = (V, E)$ be a DAG with positive edge weights. We first run Dijkstras algorithm to compute the distance from the source s to every other vertex v. Afterwards, we store the vertices in increasing order of their distance from s.
True or False: this sequence of vertices be a valid topological sort of G.

Solution: False. Consider the graph below: a topological sort of the vertices would be A, B, C, but using the method in this question, we get A, C, B.

$$\begin{tikzcd}
A \arrow[r, "5"] \arrow[rr, bend right, "1"'] & B \arrow[r, "2"] & C
\end{tikzcd}$$

(d) Let $G = (V, E)$ be an undirected graph. Let $G' = (V', E')$ where $V' = V \bigcup \{u\}$ and $E' = E \bigcup E_u$, where E_u is some set of edges that include u.
True or False: any MST of G is the subset of some MST of G'.

Solution: False. Consider the case when the new vertex u in G' has edges to all other vertices in G'. If each of these edges from u are lighter than any edge of G, then the MST of G' is the set of all edges from u.

3 Bounding Sums

Let $f(\cdot)$ be a function. Consider the equality

$$\sum_{i=1}^{n} f(i) \;\in\; \Theta(f(n)),$$

Give a function f_1 such that the equality holds, and a function f_2 such that the equality does not hold.

Solution: There are many possible solutions.
$f_1(i) = 2^i$: $\sum_{i=1}^n 2^i = 2^{n+1} - 2 \in \Theta(2^n)$.
$f_2(i) = i$: $\sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2) \neq \Theta(n)$.

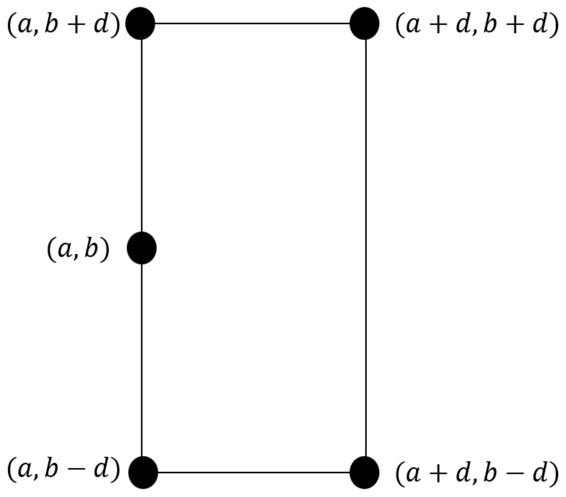
4 Agent Meetup

Manhattan has an "amazing" road system where streets form a checkerboard pattern, and all roads are either straight North-South or East-West. We simplify Manhattan's roadmap by assuming that each pair x, y, where x and y are integers, corresponds to an intersection. As a result, the distance between any two intersections can be measured as the *Manhattan* *distance* between them, i.e. $|x_i - x_j| + |y_i - y_j|$. You, working as a mission coordinator at the CS 170 Secret Service Agency, have to arrange a meeting between two of n secret agents located at intersections across Manhattan. Hence, your goal is to find the two agents that are the closest to each other, as measured by their Manhattan distance.

In this problem, you will devise an efficient algorithm for this special purpose. As mentioned before, you can assume that the coordinates of the agents are integer values, i.e. the ith agent is at location (x_i, y_i) where x_i, y_i are integers.

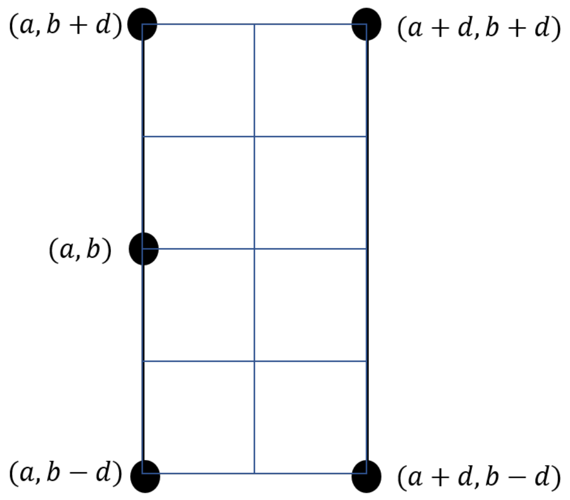
Note: This problem is very geometric, we suggest you draw examples when working on it!

(a) Let (a, b) be an arbitary intersection. Suppose all agents i for which $x_i > a$ are Manhattan distance strictly greater than d apart from each other, where $d > 0$. Give an upper bound on the number of agents i that satisfy $a \le x_i \le a + d$ and $b - d \le y_i \le b + d$. In other words, how many agents can fit in this rectangle (visualized below) without two of these agents being Manhattan distance d or less apart?



Briefly justify your answer. A reasonable bound suffices, you are not expected to provide the tightest possible bound.

Solution: 8 agents can fit in this rectangle (visualized below) without two of these agents being Manhattan distance d or less apart.



At most one agent can be in each square, since the distance between every pair of points within each square is at most d.

Since we assume all locations are integer, a better bound is possible but likely not as straightforward.

Since any constant will result in the same asymptotic result in the next part, any reasonable constant with a correct justification should get full credit for this problem.

(b) Design a divide and conquer algorithm to find the minimum Manhattan distance between any two agents. **Give a three-part solution.** A solution that has runtime within logarithmic factors of the optimal runtime will still get full credit.

Hint: Try sorting the list of agents by y-coordinate, and then sorting the list of agents by x coordinate. Note that since all distances are integer values, we only need to consider pairs that are $d-1$ values away from each other, where d is the smallest distance that we have computed so far

Solution: We give an algorithm whose runtime is $O(n \log^2 n)$. It is possible to improve it to $O(n \log n)$ by optimizing parts of the algorithm, to simplify the solution we won't bother to do so here.

Main idea.

The high level idea is similar to maximum subarray sum. Define L and R to be the left and right half of the agents when sorted by x-coordinate; why this is the right way to split the agents will become clear later. The closest pair of agents are either (1) both in L, (2) both in R, or (3) one is in L and one is in R. We can handle cases (1) and (2) recursively. It might seem like to handle case (3) we have to find the closest distance between an agent in L and an agent in R. The key idea is that letting the smallest distance we found in cases (1) and (2) be $d+1$, we only need to consider pairs of agents in case (3) that are distance at most d apart (recall all distances are integer). This idea will allow us ignore many pairs of agents in L and R.

If m_x is the median x-coordinate of all agents, note that any agent in R with $x_i > m_x + d$ can't be distance d or less from any agent in L. So let R_{close} be all agents in R for which

$x_i \leq m_x + d$; we only need to consider agents in R_{close} in case (3). Now consider any agent i in L at position (x_i, y_i). We can skip computing the distance between agent i and any agent in R_{close} whose y-coordinate does not lie in the range $[y_i - d, y_i + d]$. If we sort R_{close} by y-coordinate beforehand, we can quickly identify agents in R_{close} in this range by binary searching.

So our non-recursive work is: for each agent i in L, identify the agents in R_{close} in the y-coordinate range $[y_i - d, y_i + d]$ and compute the distance between i and these agents. We then output the smallest distance we found either between the recursive calls and this non-recursive step.

As a base case, if there are only two agents, we can just report the distance between them.

Correctness. If there are only two agents, our algorithm is of course correct. Otherwise, we proceed by induction.

If the closest pair of agents are distance $d + 1$ apart and both in L or both in R, one of the recursive calls returns the right answer by our inductive hypothesis. Otherwise, the closest pair of agents is distance at most d apart and split between L and R. By part (a) and the definition of R_{close}, every pair of agents in $L \times R$ at distance at most d has their distance computed by the algorithm, in which case the final output will be correct.

Runtime analysis

The non-recursive work we do is as follows.

- Sorting the list of all agents by x-coordinate, which takes $O(n \log n)$ time.
- Sorting R_{close} by y-coordinate, which takes $O(n \log n)$ time.
- Locating the agents in R_{close} to compare agents in L to. Since this takes $O(\log n)$ time per agent, this overall takes $O(n \log n)$ time.
- By part (a), for each agent $i \in L$, there are $O(1)$ agents j for which $m_x \leq x_j \leq m_x + d, y_i - d \leq y_j \leq y_i + d$. So we do $O(n)$ non-recursive distance computations in $O(n)$ time.

So the recurrence relation is $T(n) = 2T(n/2) + O(n \log n)$. We can't use the master theorem, but drawing the tree of subproblems, we can see the i th level of recursion has 2^i subproblems doing $O(n/2^i \log(n/2^i))$ work. So the work per level is $O(n \log n)$, i.e. the total work is $O(n \log^2 n)$. (We can also show a lower bound of $\Omega(n \log^2 n)$ since levels 0 to $\frac{1}{2} \log n$ do $\Omega(n \log n)$ work.)

5 Box Union

There are n boxes labeled $1, \ldots, n$, and initially they are each in their own stack. You want to support two operations:

- put(a, b): this puts the stack that a is in on top of the stack that b is in.
- under(a): this returns the number of boxes under a in its stack.

The amortized time per operation should be the same as the amortized time for find$(\cdot)$ and union$(\cdot, \cdot)$ operations in the union find data structure.
Hint: use "disjoint forest" and augment nodes to have an extra field z stored. Make sure this field is something easily updateable during "union by rank" and "path compression", yet useful enough to help you answer under$(\cdot)$ *queries quickly. It may be useful to note that your algorithm for answering under queries gets to see the z values of all nodes from the query node to its trees root if you do a find.*

Solution: At any given time, let $u(s)$ denote the number of boxes under box s. In the disjoint forest union, let $z(s)$ denote the augmented field stored at node s. If s is a root, we additionally store a parameter size(s), which represents the total number of boxes in a given stack. At any given time, we will maintain the invariant that when s is a root node, $z(s)$ is equal to $u(s)$, and otherwise $z(s)$ is equal to $u(s) - u(p(s))$ where $p(s)$ denotes the parent of s in the disjoint forest data structure. To obtain the value of $u(s)$, we perform a find(s) operation, and output the sum of $z(s')$ for s' in the path between s and the root r; this sum can be verified to equal $u(s)$. When the data structure is initialized, setting all $z(s)$ to 0, along with setting size(s) to 1 maintains the invariant. Whenever a union operation is performed, one root s is made a child of another root s' — in this case, we

(1) if s' is in the "upper" stack, update $z(s')$ to $z(s') + \text{size}(s)$ and update $z(s)$ to $z(s) - z(s')$; otherwise if s' is in the "lower" stack, keep $z(s')$ unchanged and update $z(s)$ to $z(s) + \text{size}(s') - z(s')$,

(2) replace size(s') with size(s) + size(s').

Before a path compression operation is performed, we can compute the value of $u(s)$ for all s whose parent is updated to root r and replace each $z(s)$ with $u(s) - z(r)$.