Exercise 3.3. Let P be a set of n points in $\mathbb{R}^2$. Design and analyze an algorithm that computes a list of all pairs of points with distance within a factor of 2 of the minimum distance between all pairs of points.

`find-pairs`(P): /* *Use the divide-and-conquer algorithm from lecture to find minimum distance in P.* */.

1. Let $\mathcal{M}$ be an empty list
2. Let $\delta =$ `min-distance`(P) // $\mathcal{O}(n \log n)$
3. For $p \in P$: // $\mathcal{O}(n)$
   A. Let $Q = \{(x, y) \in P \ : \ |p.x - x| < 2\delta \text{ and } |p.y - y| < 2\delta\}$
   B. For $q \in Q$: // $\mathcal{O}(1)$
      1. If $(q, p) \notin \mathcal{M}$ AND $\sqrt{(p.x - q.x)^2 + (p.y - q.y)^2} < 2\delta$:
         a. Add (p, q) to $\mathcal{M}$
4. Return $\mathcal{M}$

Proof of Correctness. We know that the subroutine `min-distance`(P) will return the minimum distance between any two points in P in $\mathcal{O}(n \log n)$ time. Then as we iterate through each point in P (taking $\mathcal{O}(n)$ time), we create a square Q with side length 4δ centered at p.

Claim 1. *Q contains at most 31 points.*

Proof of Claim 1. We know that the minimum distance between any two points in P is δ. For each $q \in Q \cap P$, draw a ball of radius $\delta/2$ centered at q. Then each ball B_p has area $A_{B_p} = \pi\delta^2/4$ and lies in the "padded" square $\hat{Q}$ with side length 5δ and area $A_{\hat{Q}} = 25\delta^2$. Obviously every B_p must be disjoint, so the maximum number of B_p in $\hat{Q}$ must be

$$\left\lfloor \frac{25\delta^2}{\pi\delta^2/4} \right\rfloor = \left\lfloor \frac{100}{\pi} \right\rfloor = \lfloor 31.8 \rfloor = 31.$$

■

Thus there exists at most a constant number of points in Q and we can iterate through them in $\mathcal{O}(1)$ time. As we iterate through each $q \in Q$, we confirm that the pair (q, p) is not already in our list of pairs $\mathcal{M}$ and whether the distance between p and q is less than 2δ. If both conditions are true, we add (p, q) to $\mathcal{M}$. Thus `find-pairs` will comprehensively find all valid pairs in $\mathcal{O}(n \log n) + \mathcal{O}(n)\mathcal{O}(1) = \mathcal{O}(n \log n)$ time. □

Exercise 3.6. [The example] parallel algorithm is very fast, but has a $\mathcal{O}(\log n)$-factor more work than the obvious sequential algorithm. The goal of this exercise is to develop a (recursive) parallel divide-and-conquer algorithm that is just as fast — $\mathcal{O}(\log n)$ parallel time — but has $\mathcal{O}(n)$ total work. In addition to designing the algorithm, you should analyze the running time and the work similar to how we did [for the example].

`parallel-sums`($A[1..n]$):

/ Given an input array of numbers $A[1..n]$, returns an array of the prefix sums of A. */*

1. If $n = 1$, return A
2. Let B be an empty array of size $n/2$
3. In parallel for integers $i \in [0, \frac{n}{2} - 1]$:
   A. $B[i] = A[2i] + A[2i + 1]$
4. $B =$ `prefixSum`(B)
5. In parallel for integers $i \in [0, \frac{n}{2} - 1]$:
   A. $A[2i] = B[i]$
   B. $A[2i + 1] = B[i] + A[2i + 1]$
6. return A

Proof of correctness. WLOG, we may assume n is even. In the case $n_{\neq 1}$ is odd, we simply pad A with a zero.

The base case is trivial, if $n = 1$ then the array is already its own prefix sum.

We begin by summing pairs into B and then recursing on B, which gives us the partial sums of pairs up to each index i. We then distribute these sums back into A, adjusting each pair by the total prefix sum of all previous pairs. Thus, the element $A[2i]$ becomes the sum up to its own index, and $A[2i+1]$ becomes the sum up to index $2i + 1$. □

Proof of complexity. Let $W(n)$ be the function representing the work complexity. Our algorithm is represented by the recurrence

$$\begin{aligned} W(n) &= W(n/2) + \mathcal{O}(n) \\ &= W(n/4) + \mathcal{O}(n/2) + \mathcal{O}(n) \\ &= W(n/2^k) + \sum_{j=0}^{k-1} \mathcal{O}(n/2^j). \end{aligned}$$

It is a basic fact that a decreasing geometric series is convergent, whence our recurrence solves to a final work complexity of $W(n) = \mathcal{O}(n)$.

Let $T(n)$ be the function representing the time complexity. Then by running them in parallel, we can compute $\mathcal{O}(n)$ sums in $O(1)$ time. Thus our algorithm is represented by the recurrence

$$\begin{aligned} T(n) &= T(n/2) + \mathcal{O}(1) \\ &= T(n/4) + \mathcal{O}(1) + \mathcal{O}(1) \\ &= T(n/2^k) + \mathcal{O}(1). \end{aligned}$$

Thus our recurrence gives us a final time complexity of $T(n) = \mathcal{O}(\log n)$. □

Let X and Y be two sets of integers. We define the *unique sums* of X and Y as the set of integers of the form

$$z = x + y$$

where $x \in X,\ y \in Y$, and the choice of $(x, y) \in X \times Y$ is unique. You may assume that all the integers in X are distinct (amongst themselves) and that all the integers in Y are distinct (amongst themselves).

Exercise 4.2.1. Suppose X and Y each have n integers. Design and analyze a $\mathcal{O}(n^2 \log n)$ time algorithm to compute the unique sums of X and Y.

`unique-sums`(X, Y):

/ Given two sets of integers X and Y, returns the set of unique sums of X and Y. Preprocessing: sort X and Y. */*

1. Let $\mathcal{S}$ and $\mathcal{U}$ be empty lists
2. For $x \in X$: // $\mathcal{O}(n)$
   A. For $y \in Y$: // $\mathcal{O}(n)$
      1. Let $s = x + y$
      2. Add $(s, (x, y))$ to S
3. Sort $\mathcal{S}$ by value of s // $\mathcal{O}(n^2 \log n)$
4. For integers $i \in [1, n^2]$:
   A. If $\left[\exists \mathcal{S}[i+1] \text{ AND } \mathcal{S}[i+1].s = \mathcal{S}[i].s\right]$ AND $\left[\ \exists \mathcal{S}[i-1] \text{ AND } \mathcal{S}[i-1].s = \mathcal{S}[i].s\right]$:
      1. Skip this iteration
   B. Add $\mathcal{S}[i].s$ to $\mathcal{U}$
5. Return $\mathcal{U}$

Proof of correctness. We first add all possible combinations of $x \in X$ and $y \in Y$ and add them to the list $\mathcal{S}$. We then sort elements of $\mathcal{S}$ by their sum attributes. Next, we iterate through $\mathcal{S}$ and check if any adjacent elements share the same sum. If they do, we skip that iteration. Otherwise, we add the sum to the list $\mathcal{U}$. This guarantees that only unique sums are added to $\mathcal{U}$. □

Proof of complexity. We first iterate through all possible combinations of $x \in X$ and $y \in Y$, which takes $\mathcal{O}(n^2)$ time. We then sort $\mathcal{S}$ by sum, which takes $\mathcal{O}(n^2 \log n)$ time. Finally, we iterate through $\mathcal{S}$ to find unique sums, which takes $\mathcal{O}(n^2)$ time. Thus, the final time complexity for `unique-sums` is

$$\mathcal{O}(n^2) + \mathcal{O}(n^2 \log n) + \mathcal{O}(n^2) = \mathcal{O}(n^2 \log n).$$

□

Exercise 4.2.2. Suppose all the integers in X and Y are between 0 and M for some $M \in \mathbb{N}$. Design and analyze a $\mathcal{O}(M \log M)$ time algorithm to compute the unique sums of X and Y.

unique-sums-M(X, Y, M):

/ Given two sets of integers X and Y and an integer M, returns the set of unique sums of X and Y. Assumes use of subroutines F_n^* and F_n from lecture. Preprocessing: sort X and Y. */*

1. Let A and B be arrays of length $M+1$, initialized to 0
2. For $x \in X$: // $\mathcal{O}(n)$
   A. $A[x] = 1$
3. For $y \in Y$: // $\mathcal{O}(n)$
   A. $B[y] = 1$
4. Let $\mathcal{N}$ be the smallest power of 2 greater than $2(M+1)$
5. Let $\tilde{A}$ and $\tilde{B}$ be arrays of length $\mathcal{N}$, initialized to 0
6. For $i \in [0, M]$: // $\mathcal{O}(M)$
   A. $\tilde{A}[i] = A[i]$
   B. $\tilde{B}[i] = B[i]$
7. For $i \in [M+1, \mathcal{N}-1]$: // $\mathcal{O}(M)$
   A. $\tilde{A}[i] = 0$
   B. $\tilde{B}[i] = 0$
8. Let $\hat{A} = F_N(\tilde{A})$ and let $\hat{B} = F_N(\tilde{B})$ // $\mathcal{O}(M \log M)$
9. Let $\hat{C}$ be an array of length $\mathcal{N}$, initialized to 0
10. For $k \in [0, \mathcal{N}-1]$: // $\mathcal{O}(M)$
    A. $\hat{C}[k] = \hat{A}[k] \cdot \hat{B}[k]$
11. Let $C_{aux} = F_N^*(\hat{C})$
12. For $k \in [0, \mathcal{N}-1]$: // $\mathcal{O}(M)$
    A. $C_{aux}[k] = C_{aux}[k]/\mathcal{N}$
13. Let $\mathcal{U}$ be an empty list
14. For $z \in [0, 2M]$:
    A. If $round(C_{aux}[z]) = 1$:
       1. Append z to $\mathcal{U}$
15. Return $\mathcal{U}$

Proof of correctness. We form "indicator" arrays A and B of length $M+1$, then create extended copies $\tilde{A}$ and $\tilde{B}$ of length $N \geq 2(M+1)$. Applying F_N to each padded array, we multiply their transforms pointwise and take the inverse transform F_N^*. We then divide by N, because $F_N^*(F_N(\cdot)) = N \cdot (\cdot)$. A well known DFT property implies that this returns the (linear) convolution of A and B. Hence each index z of the output equals $\sum_{x+y=z} A[x]\, B[y]$, i.e. the number of pairs (x, y) summing to z. Therefore, z is a unique sum exactly if this convolution value is 1. □

Exercise 4.2.3. Suppose now that we have k sets $X_1, \ldots, X_k$, each consisting of integers between 1 and M. (You may again assume no duplicates within each set.) A unique sum of $X_1, \ldots, X_k$ is defined as an integer of of the form

$$z = x_1 + \cdots + x_k$$

where $x_i \in X_i$ for all $i \in [k]$, and the choice of $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ is unique. Design and analyze an algorithm to compute the unique sums of $X_1, \ldots, X_k$ in $\mathcal{O}(kM \log(Mk) \log(k))$ time. You may assume for simplicity that k is a power of 2.