CS 325 Implementation Assignment 1: Report

Troy Diaz, Balakrishna Thirumavalavan, Andrew Brandenfels

January 25, 2025

1. Brute Force

Algorithm Design and Pseudo-code

```
1) Calculate distance between all pairs.
2) Store minimum distance and each pair that has that distance.

O(n^2) complexity

Pseudo code design:

1) Set min_distance to be inf.
2) Initialize an empty list for pairs we're interested in.
3) For each pair, compare distance. If distance is less than min_distance,
store distance and add that pair to the list.
4) If distance is the same for the next pair, add to the list.
5) Sort list of closest pairs.
6) Return the minimum distance and pair list.
```

Asymptotic Analysis of Runtime

The brute-force method for finding the minimum distance between all pairs of points in a list has a time complexity of $O(n^2)$.

In our approach, we proceed as follows:
1. Iterate through n points, $O(n)$:

```
for i in range(len(points)):
    ...
```

2. For each point, calculate the distance to all other points, $O(n-1)$:

```
for j in range(i+1, len(points)):
    ...
```

3. Calculate the distance in the inner loop, $O(1)$:

```
dist = distance(points[i], points[j])
    ...
```

4. Outside sorting, $O(nlogn)$:

```
closest_pairs = sort_pairs(closest_pairs)
    ...
```

Thus, the time complexity of this brute-force method can be expressed as:

$$T(n) = n \cdot (n-1) \cdot (nlogn) \cdot O(1) = O(n^2).$$

Empirical Runtime

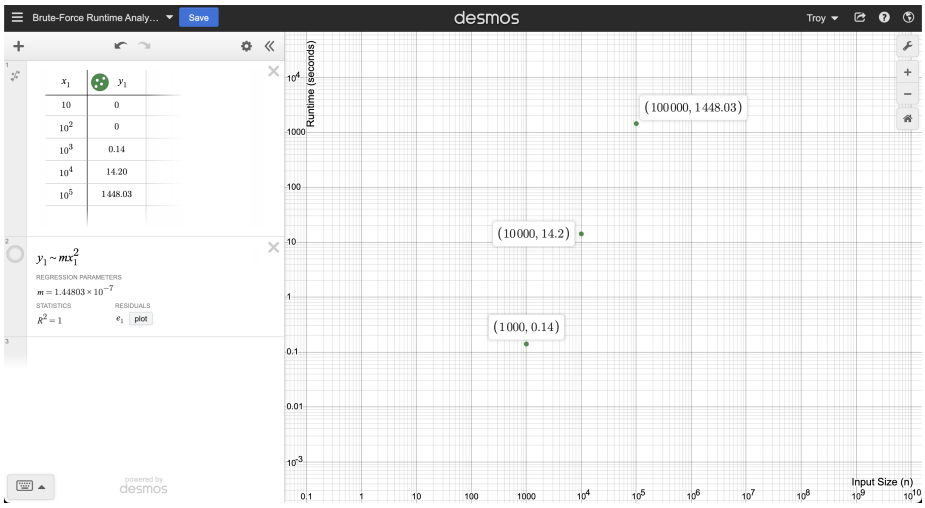

Fig 1.1 Empirical runtime Data for a Brute-Force Algorithm on a Log Scale. Scatter plot of runtime versus input size (n) for a brute-force algorithm.

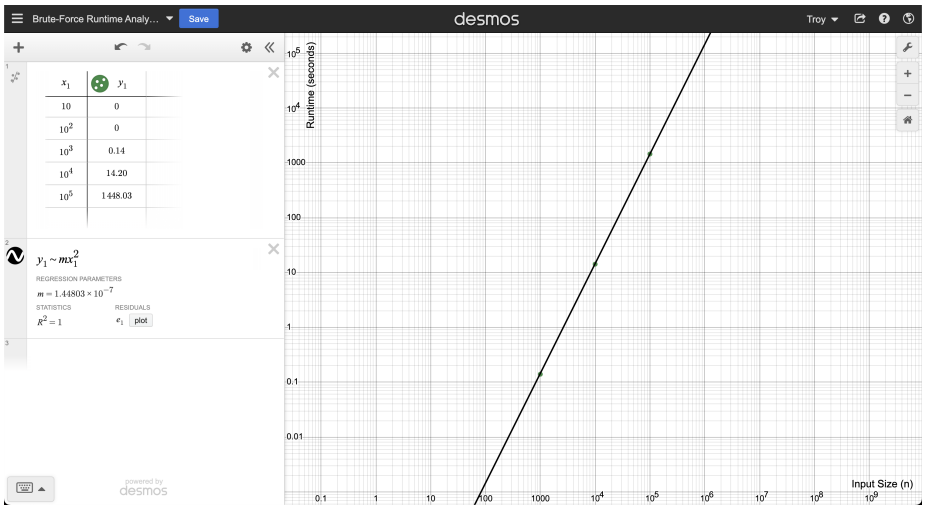

Fig 1.2 Quadratic Runtime Growth of a Brute-Force Algorithm on a Log Scale. The plot shows the relationship between runtime and input size (n), demonstrating a quadratic pattern with $R^2 = 1$.

Interpretation and Discussion

The goal of brute force is to iteratively loop through each pair and calculate the distance between other pairs. The closest pairs are returned, along with the minimum distance. Fig 1.1 illustrates the empirical runtime data for a brute-force algorithm. The runtime plot compares the runtime as the sample sizes grow, and, as shown by the asymptotic analysis of the runtime above, the growth is $O(n^2)$ as the sample size n approaches a large number.

The runtime plot shows the time in seconds as the sample sizes increase (10^2, 10^3, 10^4, 10^5). As shown in Fig 1.2, the points align with the x^2 line, indicating a clear quadratic growth.

The experimental data matches the theoretical $O(n^2)$ growth. While sorting has a complexity of $O(n \log n)$, it is negligible compared to the $O(n^2)$ growth, especially for larger datasets. Smaller data sets may see a difference; however, as n gets larger, the quadratic term dominates.

Hardware optimizations may speed up execution but do not significantly affect the overall runtime trend. In conclusion, the brute-force method exhibits

quadratic growth.

2. Divide and Conquer

Algorithm Design and Pseudo-code

The following steps detail the design of the naive divide and conquer algorithm:

1. Divide and Conquer Closest Pair
    1. Sort Points by x-coordinate
        (a) Begin by sorting the list of points based on their x-coordinates.
    2. Call Recursive Function
        (a) Use the sorted list of points as input to the `recursive_closest_pair()` function, and return its result.
2. Recursive Closest Pair
    1. Divide Points
        (a) Split the list of points into two halves: a left half and a right half.
    2. Find the Midpoint
        (a) Identify the x-coordinate of the midpoint, which will divide the points into two regions.
    3. Recursive Calls for Left and Right Halves
        (a) Call `recursive_closest_pair()` on the left half to find the minimum distance `(d1)` and the closest pairs in that region `(pairs1)`.
        (b) Similarly, call `recursive_closest_pair()` on the right half to get the minimum distance `(d2)` and closest pairs `(pairs2)`.
    4. Determine the Smaller Distance
        (a) Set `d` to the smaller of the two distances, `d1` and `d2`.
    5. Build the Strip of Points Near the Midpoint
        (a) Collect all points within a distance `d` from the midpoint into a new list called `strip`.

6. Merge Closest Pairs from the Strip

    (a) Use the `merge_strip` function to compute the minimum distance `(d_strip)` and closest pairs `(pairs_strip)` within the strip.

7. Compare and Return Results

    (a) If `d_strip` is smaller than `d`, return `d_strip` and `pairs_strip`.

    (b) If `d_strip` equals `d`, return `d` along with the combined pairs from `pairs1`, `pairs2`, and `pairs_strip`.

    (c) Otherwise, return `d` and the combined pairs from `pairs1` and `pairs2`.

3. Merge Strip

1. Sort Strip by y-coordinate

    (a) Arrange all points in the strip in ascending order of their y-coordinates.

2. Initialize Variables

    (a) Set `min_distance` to `d`.

    (b) Create an empty list `closest_pairs` to store the closest pairs of points found.

3. Find Closest Pairs in the Strip. For each point `p` in the strip,

    (a) Compare `p` with subsequent points `q` in the strip.

    (b) If the difference in their y-coordinates is greater than or equal to `min_distance`, stop checking further for this point.

    (c) Calculate the distance `(d_ij)` between `p` and `q`.

    (d) If `d_ij` is smaller than `min_distance`,

        i. Update `min_distance` to `d_ij`.

        ii. Replace `closest_pairs` with the pair `(p, q)`.

    (e) If `d_ij` equals `min_distance`,

        i. Add `(p, q)` to the `closest_pairs`.

4. Return Results

    (a) Return the `min_distance` and the list of `closest_pairs`.

```
Algorithm DivideAndConquerClosestPair(points)
    Sort points by x-coordinate
    Return RecursiveClosestPair(points)

Algorithm RecursiveClosestPair(points)
    if length(points) <= 1
        Return infinity, []

    mid <- floor(length(points) / 2)
    left <- points[0:mid]
    right <- points[mid:]
    midpoint_x <- points[mid].x

    (d1, pairs1) <- RecursiveClosestPair(left)
    (d2, pairs2) <- RecursiveClosestPair(right)
    d <- min(d1, d2)
    pairs <- pairs1 + pairs2

    strip <- []
    for each point in points
        if abs(point.x - midpoint_x) < d
            strip.append(point)

    (d_strip, pairs_strip) <- MergeStrip(strip, d)

    if d_strip < d
        Return d_strip, pairs_strip
    else if d_strip == d
        Return d, pairs + pairs_strip
    else
        Return d, pairs

Algorithm MergeStrip(strip, d)
    Sort strip by y-coordinate
```

```
min_distance <- d
closest_pairs <- []

for i <- 0 to length(strip) - 1
    for j <- i + 1 to length(strip) - 1
        if (strip[j].y - strip[i].y) >= min_distance
            break
        d_ij <- Distance(strip[i], strip[j])
        if d_ij < min_distance
            min_distance <- d_ij
            closest_pairs <- [(strip[i], strip[j])]
        else if d_ij == min_distance
            closest_pairs.append((strip[i], strip[j]))

Return min_distance, closest_pairs
```

Asymptotic Analysis of Runtime

This naive approach to divide and conquer can be broken into four main components:

1. The initial sorting of the points
2. The recursive division of points
3. The merge step, where the closest pairs across the dividing line are identified
4. Solving the recurrence

Initial Sorting of Points:
At the start of the algorithm, we sort the input points by their x-coordinates. This step uses a comparison-based sorting algorithm, which takes $O(n \log n)$. This sorting is efficient and is not repeated in subsequent recursive calls, so its contribution to the runtime is $O(n \log n)$.

Recursive Division of Points:
The divide and conquer algorithm splits the input into two halves recursively, solving the closest pair problem on the left and right halves. At each level of

recursion, the points are divided in half. This creates the recurrence relation for the runtime:

$$T(n) = 2T(\frac{n}{2}) + \text{cost of merge step}$$

At each level, the merge step combines the results.

Merge Step:
In the merge step, we identify the closest pair of points that span the dividing line. This involves:

- Collecting points in the strip (within distance d of the midpoint) in $O(n)$, since all n points are checked.
- The strip is resorted at each level of recursion, taking $O(n \log n)$ in the worst case.
- Each point in the strip is compared with at most 7 other points, resulting in $O(n)$ comparisons.

So, the total cost for the merge step is:

$$O(n \log n) + O(n) = O(n \log n)$$

Solving the Recurrence:
The recurrence relation for the naive algorithm is:

$$T(n) = 2T(\frac{n}{2}) + O(n \log n)$$

Using Master Theorem, we can analyze this such that:

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

Here,

- $a = 2$ (the number of sub-problems)
- $b = 2$ (the factor by which the problem size is divided)
- The non-recursive term is $O(n \log n)$, which is of the form $O(n^d \log^k n)$ with $d = 1$ and $k = 1$, since $n \log n$ is a product of n and $\log n$.

Recall that Case 2 dictates if $a = b^d$ ($d = \log_b a$), then $T(n) = O(n^d \log^{k+1} n)$

$$\log_b a = \log_2 2 = 1$$

After substitution, we get:

$$T(n) = O(n^1 \log^{1+1} n) = O(n \log^2 n)$$

Thus, the overall runtime is $O(n(\log n)^2)$.

Empirical Runtime

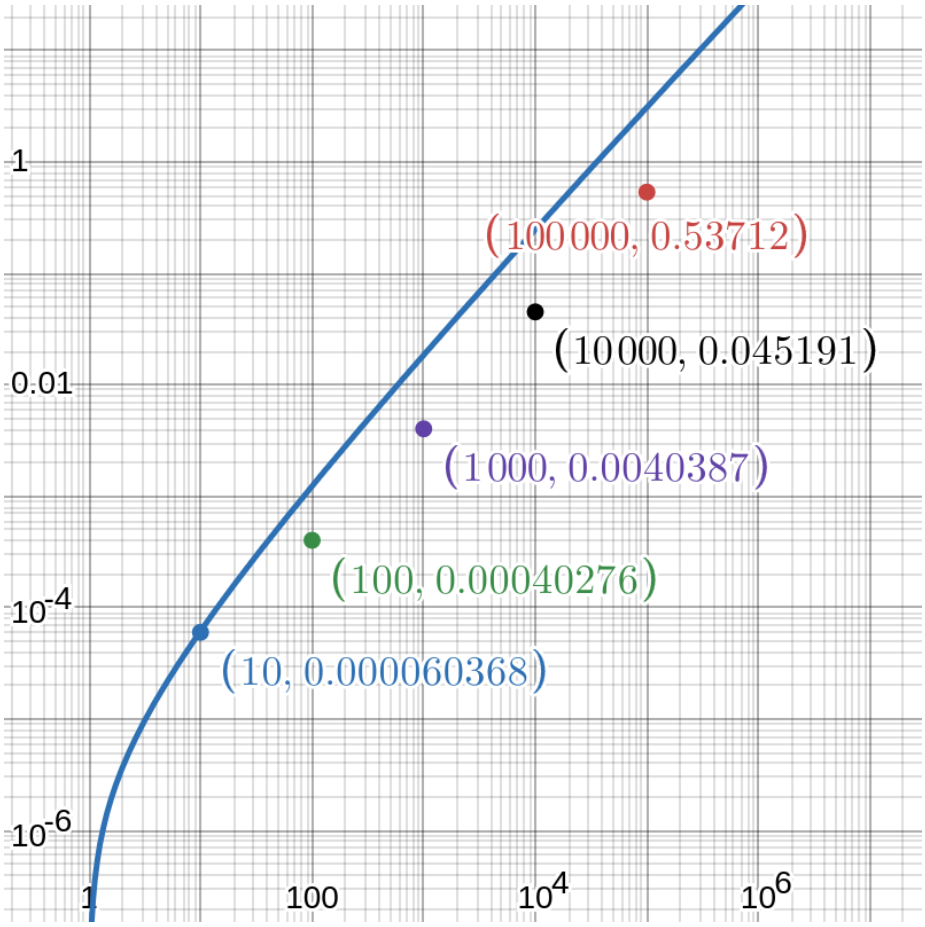

Figure 1: Naive DNC - Blue curve is $f(n) = cn \ln n$ for $c = 2.7 \times 10^{-6}$

Interpretation and Discussion

3. Enhanced Divide and Conquer

Algorithm Design and Pseudo-code

The enhanced divide and conquer algorithm completes the following steps:

1. Sort the points by ascending x and y coordinates in two lists.
2. Call `enhanced_dnc_recursive()` on the n points.
3. Return the minimum distance and sorted pairs.

The enhanced recursive function performs the following algorithm:

1. If $n = 2$, return the only pair.
2. If $n = 3$, compute each pairwise distance and return the smallest distance and associated points.
3. Otherwise, call `enhanced_dnc_recursive()` separately on the left and right regions separated by the median x value.
4. Compare and store the smallest distance δ of the left and right sides, and the associated point sets.
5. Remove stored point sets that both lie exactly on the median line, to avoid double-counting.
6. Store in M all points within δ x-distance of the median line from the y-sorted array, retaining the y-sort.
7. For each point A in M, compare its y-distance with the next point B in M. If the y-distance is greater than δ, end the iteration and continue checking the next point A. Otherwise, compute the distance from A to B. If this distance is less than δ, then set δ to be the new minimum value and store the associated points, overwriting the previously stored wider pairs of points. If the distance of AB is equal to delta, then simply append points [A,B] to the list of closest points.
8. Return the minimum distance δ and closest points.

```
enhanced_dnc(points)
    xSorted = points.xSort()
    ySorted = points.ySort()
    distance, pairs = enhanced_dnc_recursive(xSorted, ySorted, n)
    return distance, sort_pairs(pairs)

enhanced_dnc_recursive(n)
    if (n <= 3)
        return brute_force

    median = floor(n / 2)
    leftMinDistance, leftClosestPoints = enhanced_dnc_recursive(1, ..., median)
    rightMinDistance, rightClosestPoints = enhanced_dnc_recursive(median + 1, ..., n)

    if leftMinDistance < rightMinDistance
        d = leftMinDistance
        closestPoints = leftClosestPoints
    else if rightMinDistance < leftMinDistance
        d = rightMinDistance
        closestPoints = rightClosestPoints
    else
        d = leftMinDistance
        closestPoints = leftClosestPoints.append(rightClosestPoints)

    for pointSet in closestPoints:
        if pointSet.A.x == pointSet.B.x == median.x
            closestPoints.remove(pointSet)

    for point in ySorted
        if point.x < 0 or n < point.x
            break
        if abs(point.x - medianPoint.x) < d
            M.append(point)
    d_m = d
    for A in M
```

```
        for B in M[i + 1:]
            if abs(B.y - A.y) >= d
                continue
            else
                abDistance = computeDistance(A, B)
                if abDistance < d_m
                    closestPointsInM = [[A, B]]
                elif abDistance == d_m
                    closestPointsInM.append([A, B])
    if d_m < d
        closestPoints = closestPointsInM
        d = d_m
    elif d_m == d
        closestPoints.append(closestPointsInM)
    return d, closestPoints
```

Asymptotic Analysis of Runtime

We define $F(n)$ as the outer non-recursive function, and $T(n)$ as the recursive function. The standard Python library `sorted()` function sorts the points by x and y coordinates using a Timsort algorithm. This algorithm combines mergesort and insertion sort, and is known to be $O(n \log n)$. We note that the return statement will be a lower order term and therefore can be ignored in the asymptotic analysis. Therefore we obtain that

$$\begin{aligned} F(n) &= 2cn \log n + T(n) \\ &= O(n \log n) + T(n). \end{aligned}$$

In the recursive function, we note that steps 1 and 2 are the base case and are considered constant time. In step 3, there are two recursive calls, each with $\frac{n}{2}$ points, for a cost of $2T(\frac{n}{2})$. Steps 4 and 5 are also constant time. Step 6 is $O(n)$, because each point in the y-sorted array must be checked to determine if it lands in the M strip. In step 7, the outer for loop on A is $O(n)$, because the size of M is linearly proportional to the input n. The inner loop over B runs at most 7 times, due to the fact that δ is established as the minimum distance of the left and right sides separately, and because we end the inner loop when the A and B y-values differ by at least δ. Therefore the inner for loop of step

7 and its associated comparisons and storage is considered $O(1)$, and step 7 is $O(n)$ overall. In step 8, we simply perform another $O(1)$ constant operation. Summing these results yields

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

Let $a = 2$, $b = 2$, and $d = 1$. Then applying the Master Theorem, we note that $\frac{a}{b^d} = 1$, so

$$\begin{aligned} T(n) &= O(n^d \log n) \\ &= O(n \log n). \end{aligned}$$

Applying this result to the outer function $F(n)$ shows that the asymptotic runtime for the enhanced divide and conquer algorithm is

$$\begin{aligned} F(n) &= O(n \log n) + O(n \log n) \\ &= O(n \log n). \end{aligned}$$

Empirical Runtime

Interpretation and Discussion

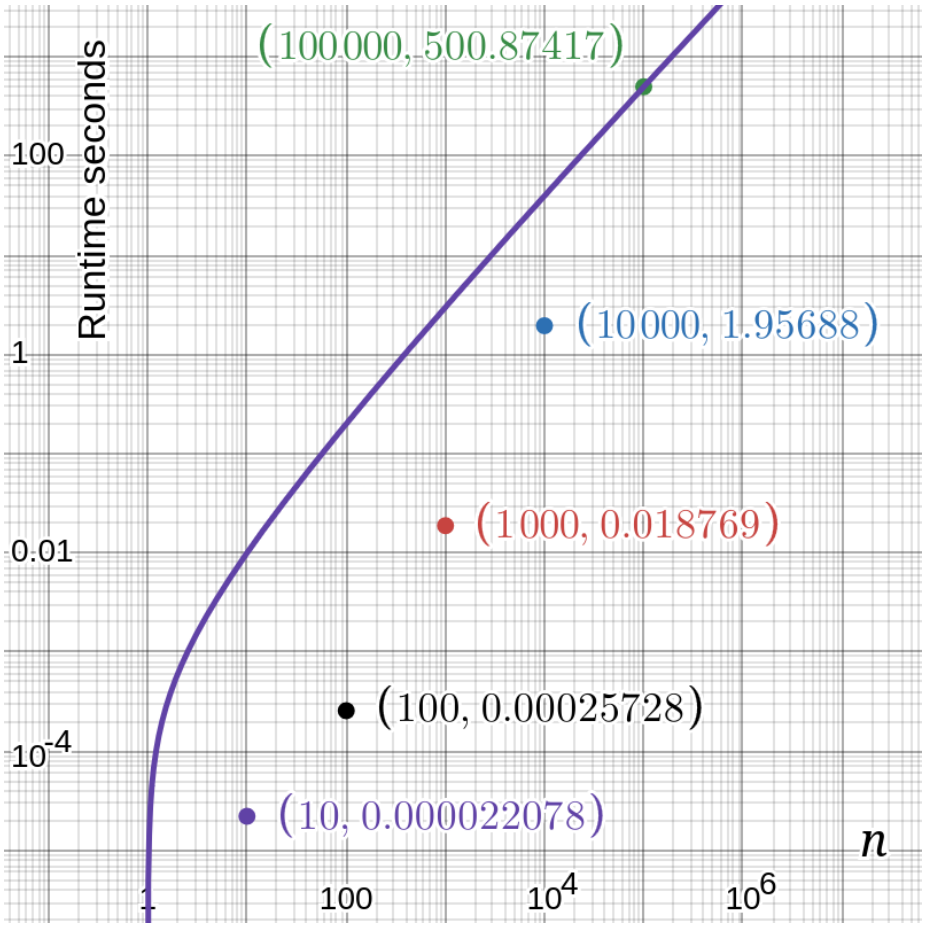

Figure 2: **Enhanced divide & conquer runtime test results**
Each point represents the average of 10 trials using random inputs of different seeds for each trial. Note that the axes are on a logarithmic scale. The purple curve is $f(n) = cn \log n$ for $c = 0.001$.