CPSC 3620 Course Project: Closest-Pair Problem Utilizing Brute Force, and Pre-Sorting Divide and Conquer

CPSC 3620: Data Structures and Algorithms

Professor. John Zhang

Hasan Raza, Connor Pittman, Chamod Jayathilake

December 6, 2024

Table of Contents

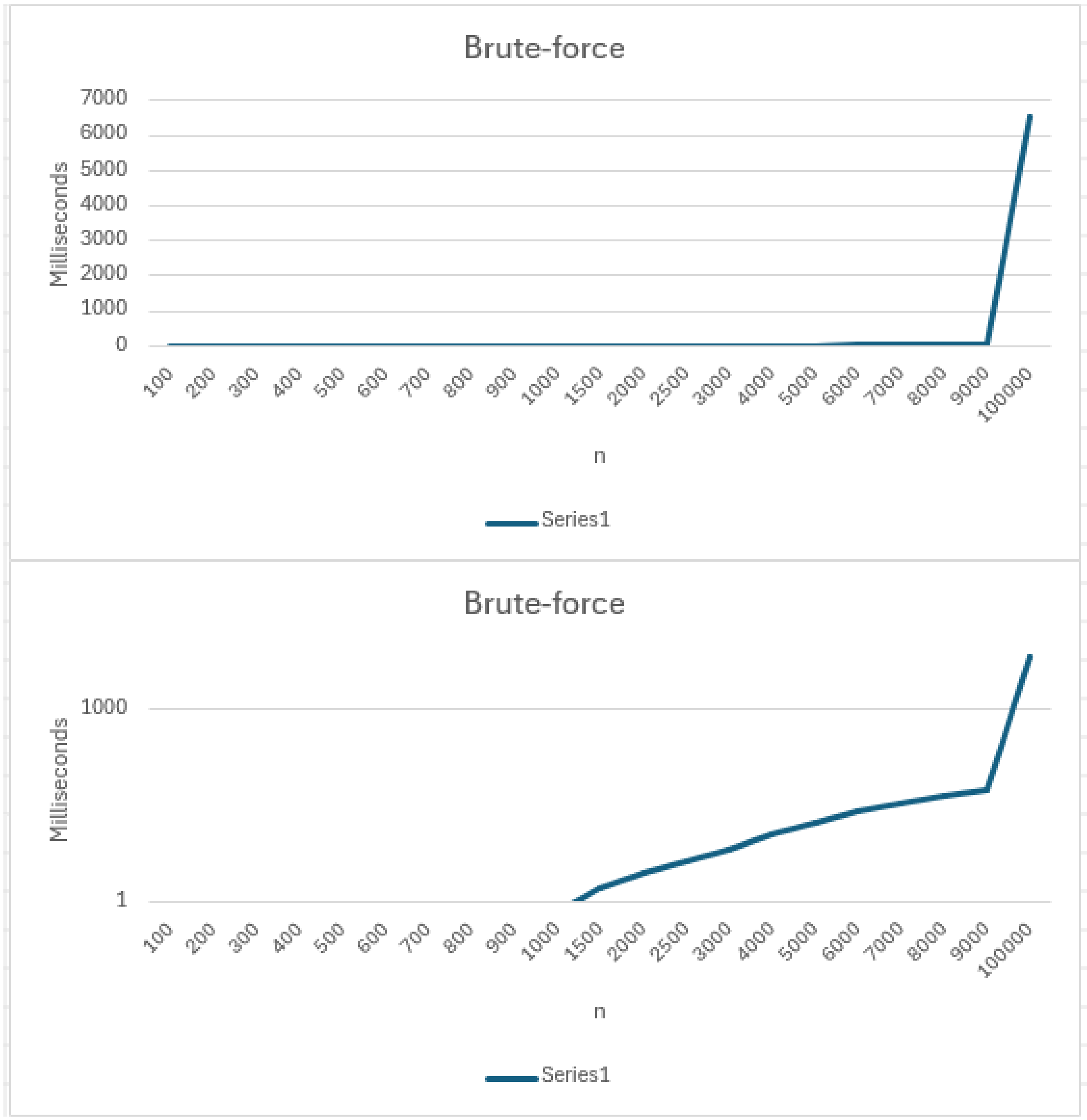
1.1 Introduction:

This project utlizes a program used to take N points and compute the closest pair of points. It does this in two ways, either using a brute force algorithm or by using a divide and conquer algorithm with pre-sorting. We outline the experiments we conducted and our findings. We also outline how the program works, the important variables, the data structures, and the algorithms that are used within it in detail. Additionally, we cover what we learned, the struggles we faced, and how this project will guide our learning in the future.

1.2 Experiments:

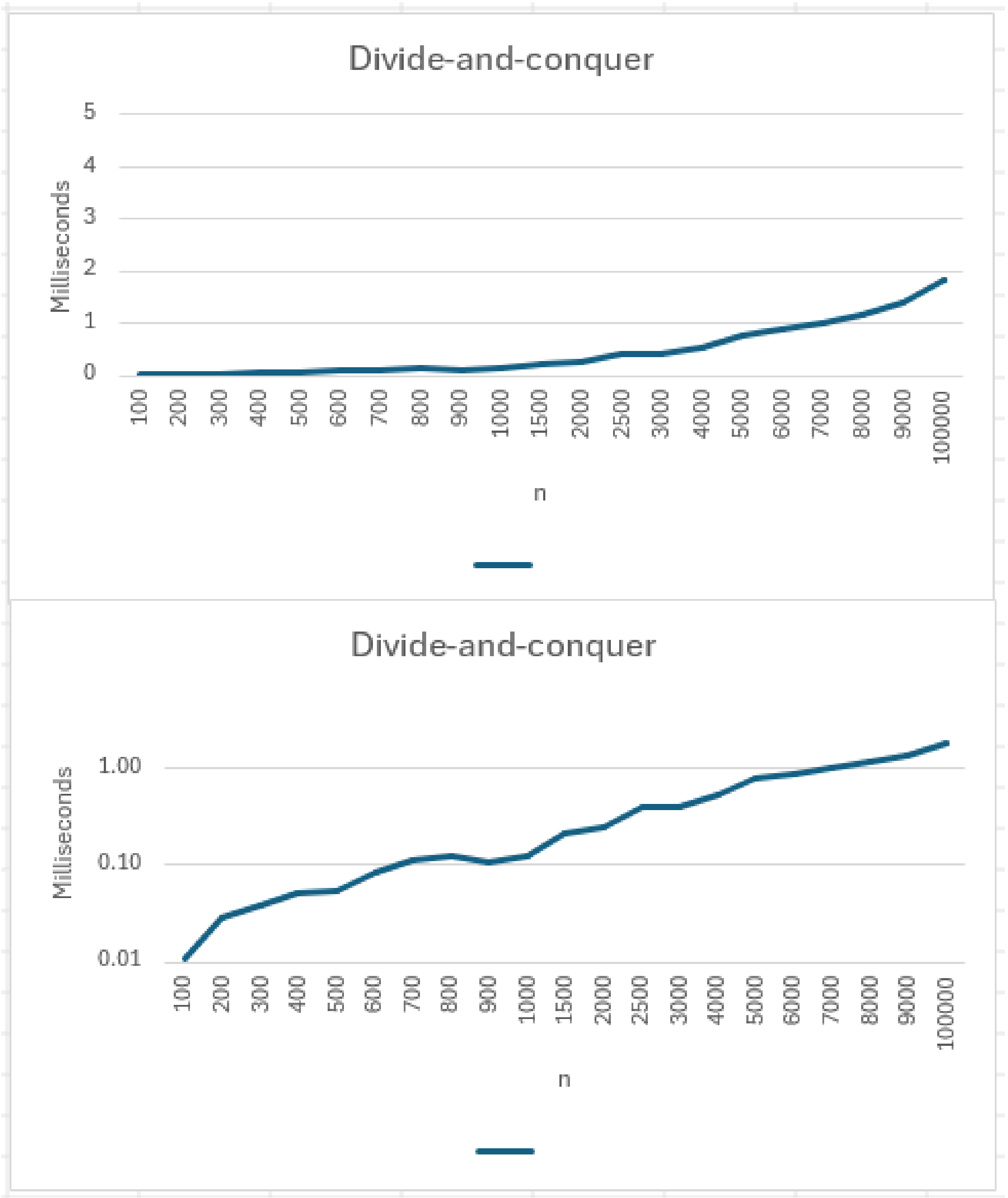
n	Brute-force run time (milliseconds)	Divide-and-conquer run time (milliseconds)
100	0.008017	0.01092
200	0.029016	0.02825
300	0.06369	0.039288
400	0.112102	0.051568
500	0.174978	0.053791
600	0.265981	0.085098
700	0.341517	0.115415
800	0.44571	0.123785
900	0.564104	0.110021
1000	0.697734	0.123394
1500	1.5679	0.215474
2000	2.7893	0.249303
2500	4.36254	0.401594
3000	6.294	0.40978
4000	11.2081	0.544979
5000	17.473	0.789339
6000	25.3483	0.882523
7000	34.0314	1.01474
8000	44.0561	1.017053
9000	57.5007	1.39438
100000	6522.76	1.84614

1.3 Brute-force charts:



Run Time in milliseconds over n (original scale, logarithmic scale)

1.4 Divide-and-conquer charts:



Run Time in milliseconds over n (original scale, logarithmic scale)

1.5 Time Complexity Analysis of Both Algorithms:

1.5.1 Brute Force:

The theoretical time complexity of a brute-force algorithm for the closest-pair problem is $O(N) + O(N^2) = O(N^2)$. Firstly, storing the set of coordinates into a vector prior to the brute force algorithm has a time complexity of O(N). This is because we are reading N points from outputN.txt and each coordinate takes O(1) to complete, for a total time complexity of O(N). Next, the brute-force algorithm compares each point against every other. Resulting in N comparisons for each point. This results in a total time complexity of $O(N) + O(N^2) = O(N^2)$ because O(N) is insignificant compared to $O(N^2)$.

The logarithmic scale provides a clearer representation of the brute-force algorithm's time complexity. The steady upward slope indicates a polynomial growth rate, which is consistent with a theoretical time complexity of O(N2). The nearly straight line on the logarithmic scale suggests that the implementation matches the expected behavior, as the slope aligns with the quadratic complexity. This confirms that the time required increases significantly as n grows, emphasizing the efficiency of the brute-force approach for large inputs. The visual jump in the time taken by the algorithm when N = 100,000 is due to how we were increasing N by a factor of 0.1 - 2 between tests, however, the jump from N = 9000 and N = 100,000 increases N by a factor of 11.1, leading to a visual jump in the chart.

1.5.1 Divide and conquer using Pre-Sorting:

The theoretical time complexity of a pre-sorting divide-and-conquer algorithm for the closest-pair problem is O(N) + O(N log N) + O(N log N) = O (N log N). Firstly, storing the set of coordinates into a vector prior to the pre-sorting divide and conquer algorithm has a time complexity of O(N). This is because we are reading N points from outputN.txt and each coordinate takes O(1) to complete, for a total time complexity of O(N). This algorithm makes use of pre-sorting using quick sort to sort the points by their X coordinates while ensuring the integrity of the coordinates is maintained. Quick sorting has a time complexity of O(N log N) utilizing pivots, partitioning, and swaps to sort the stored coordinates. This sorted vector of points is input into the divide and conquer algorithm which has a time complexity of

O (N log N). This is true because we recursively divide the points by their X value while ensuring that any points within the current minimum distance of the dividing line are compared with each other. This ensures that we do not miss any pair of points on opposite sides of the dividing line.

The logarithmic scale represents the time complexity of the divide-and-conquer algorithm combined with the quicksort algorithm utilized for pre-sorting, which reveals a more efficient growth rate compared to the brute-force approach. The gradual upward slope suggests a sub-quadratic complexity, likely O(N log N), which aligns with theoretical expectations for many divide-and-conquer algorithms. The nearly straight line on the logarithmic scale with a slope less steep than quadratic indicates that the implementation adheres to the theoretical time complexity. This demonstrates the divide-and-conquer approach’s improved scalability, making it suitable for larger input sizes compared to the brute-force method.

2.1 Main variables and data structures:

std::vector<int> xpoints: std::vector<int> ypoints;	This vector is used by both the brute force and the divide and conquer algorithms to store the coordinates of the points. Before the start of either algorithm, these vectors are populated from the txt file. They are then used as fundamental components of the Comparisions done within each algorithm.
const std::vector<int> acceptedNs{};	This vector is used to store the acceptable values of N. N is compared against this vector and if N is not present in the vector, the input is denied.

Double minXfirstpoint;
Double minYfirstpoint;
Double minXsecondpoint;
Double minYsecondpoint;
double minDist;
	These variables are used in both algorithms to store the closest discovered pair of points and the distance between them. The distance between two points is calculated using the pythagorean theorem and is compared against the current minDist. If the distance between the currently selected points is smaller than minDist, all five of these variables are overridden by the current selection and their distance.

Int N;	This variable is fundamental to the program. It is used to take user input and is used at every stage of the program. It is used to store the number of points that the user has selected to be used in their algorithm. It gets fed into the functions which then output the respective results.

struct Result // Structure to store the closest pair AND the distance between them.
{
double minDist; // Minimum distance
 	
double x1; // X value of first point in closest pair
double y1; // Y value of first point in closest pair
 	
double x2; // X value of second point in closest pair
double y2; // Y value of second point in closest pair
};
	This structure is used by the Result closestPairDivAndConquer function to return more than just a single value. It returns the minDist that it was able to find alongside the two closest points.

std::ofstream outFile();
std::ifstream inFile();
	The program uses input and output file handling as part of its data structures to write and read text to and from the outputN.txt files.

2.2 Algorithms, Files, and Functions,

and explaining how the Program Works

Important file or Function:	How each file/function works:
Src folder {	Contains all the .cc or .cpp files within the program.

Controller.cpp:	This file is simple and is the first one the user encounters during the compilation. It asks the user whether they would like to use a brute force approach or a divide and conquer approach to solve the closest point problem. After the user has made their decision and the input is deemed valid, the controller calls either divideConquerValidInputCheck() or bruteForceValidInputCheck().
Divide and Conquer Explanation	
Divideconquer.cc:	Divideconquer.cc holds the code for using presorting with divide and conquer to solve the closest pair problem. The divide and conquer algorithm used in this program utilizes presorting to ensure that the divide and conquer algorithm operates smoothly without issue. The specific details about how this algorithm works are detailed below within the individual functions making up this file. The distance, closest points, and the time taken to run the algorithm are returned at the end of the compilation.
void divideConquerValidInputCheck();	Firstly, the the user is prompted to input N. Their input is checked by divideConquerValidInputCheck() to make sure it's valid and an acceptable inpit. It is

	compared against std::vector<int> acceptedNs, which holds all the accepted values. 0 is an accepted value but this performs divide and conquer on all acceptable values of N. The function makes use of a while loop to ask the user what their input is. If the input is unaccepted 3 times in a row, the program is stopped. If the input is accepted, the function calls sortAndDivide(N)
void sortAndDivide(int N)	This function takes the user input N obtained in the previous function, reads the contents of outputN.txt and saves it to the vectors xpoints and ypoints. It then calls the quickSort function on the two points vectors to sort them by their X coordinate while maintaining the integrity of each point. After they are sorted, the divide and conquer algorithm is called on the presorted vector. This ensures that the divide and conquer
void quickSort(std::vector<int>& sortedVector, std::vector<int>& unsortedVector,int low, int high); int partition (std::vector<int>& sortedVector, std::vector<int>& unsortedVector,int low, int high);	quickSort() sorts the coordinates by their X coordinate while maintaining coordinate integrity. It does this by mirroring swaps to the xpoints vector within the partition function on the ypoints vector. The quickSort function operates exactly how a normal quicks sort algorithm would, utilizing pivots, swaps, and low/high values to sort the vector.

void closestPairDivAndConquer();	Void closestPairDivAndConquer() performs the divide and conquer algorithm on the sorted vectors. It does this by recursively dividing the graph using the sorted vector to find the closest pair. However, what if the closest pair exists along the line where the algorithm divides? As it recursively divides, it checks all points along the dividing line if they are within minDist of it. This ensures that it is only checking points near that dividing line that have a chance at being the new closest point. This algorithm takes a total of O(NlogN) time to solve, not including the sorting. Once the closest pair has been found, the distance, the two points, and the time taken to complete the algorithm is returned.
Brute Force Algorithm Explanation:	
Bruteforce.cc:	Bruteforce.cc holds the code for using the brute approach to solve the closest pair problem. It uses two for loops to compare each point with every other to discover the closest pair. The two points, the distance, and the time taken to run the brute force algorithm are returned.

void bruteForceValidInputCheck();	Valid input check for brute force does the exact same thing as the input check for divide and conquer. It ensures that the user input is acceptable. 0 is an accepted value but this performs brute force on all accepted values of N. The function makes use of a while loop to ask the user what their input is. If the input is unaccepted 3 times in a row, the program is stopped. If the input is accepted, the function calls closestPairBrute(N);
void closestPairBrute(int N);	This function is very simple. It uses two for loops to compare each point within the two coordinate vectors xpoints and ypoints. it returns the minDist, the closest points, and the time it took to run the algorithm. This function has a complexity of O(N^2).
Create.cc generateNPoints();	Create.cc is responsible for generating fresh outputN.txt files. If the user desire sto generate new ones. Using ‘make clean’ will call this file and command it to generate new output files. GenerateNpoints is the function that handles this. Once it has established the file path, a for-loop set to run N number of times for all values within acceptedNs (it runs 100 times for N = 100, 200 times

	for N = 200, etc), begins and the contents of each output file are created. This step has a time complexity of O(N) because each coordinate takes O(1) time to generate using rand().
Shared.cc: std::string selectTextFile(int N)	This file holds code that is used by the bruteforce.cc and the divideconquer.cc files. It holds the acceptedNs vector, which is used for both generating the output files and for ensuring that the input from the user is acceptable. It also has the selectTextFile function which is called by sortAndDivide and closestPairBrute to select the correct textfile for the user's input.
Textfile folder { }	Holds all text files.
Include folder { }	Holds all header files.
Readme.txt	Contains instructions about running the program.
Makefile	Makefile for compiling the project.

2.3 Learning & Improvements:

During the development of this program, the largest takeaway was how much a sophisticated divide and conquer algorithm is compared to a brute force approach. The divide and conquer algorithm consistently and reliably outperformed the brute force method. While we expected this, it was shocking how much faster it was. In our tests, we saw that when given an N value of

100,000, the divide and conquer approach had taken “Need to redo tests” milliseconds to compile. Even though the algorithm made use of quick sort AND a divide and conquer algorithm, both of which took O(NlogN) time to complete. The brute force algorithm was still completely outclassed. Additionally, when we began this project, we were unaware that we would need to include a sorting algorithm to ensure that the divide and conquer algorithm would operate effectively. Having to later include presorting to the divide and conquer algorithm, and seeing that it was still able to completely outclass the brute force algorithm with minimal changes to the total time cost reinforced our own beliefs about the inherent weaknesses of brute force algorithms and how they can be effortlessly surpassed by even basic divide and conquer algorithms. Additionally, it was eye-opening to see how divide and conquer algorithms increasingly outperform brute force algorithms for larger data sets. During this course, there was a consensus that there was no opportunity to see how the intelligent algorithms we have been learning about were superior to brute force algorithms. We knew that they were superior only on paper, as we did not have the hands-on experience comparing the two. Being able to directly compare pre-sorting alongside divide and conquer against a brute force approach was very illuminating.

2.4 Specific Issues:

During the development of this program, we were faced with several issues. Firstly, how do we design a divide and conquer algorithm for an array of unsorted points. We struggled with this question before we realized that we could use pre-sorting as a method to make it easier for the divide and conquer algorithm to discover the closest pair of points. However, we were hesitant to add a sorting algorithm with a time complexity of O(NlogN) to divide and conquer algorithm. However, once we realized that attempting to use divide and conquer to discover the closest pair within an unsorted array of points, was just a more complicated method of a brute force algorithm, we quickly implemented quickSort within the code. Another issue unrelated to the algorithms within the code was the file paths. We were unaware how the program would operate on a computer when we did not have control over the directories. Especially since we had added the ability for the outputN.txt files to be deleted and regenerated by the commands ‘make clean’ and ‘make’. We solved this issue by implementing the following code.

```
std::filesystem::path base_path = std::filesystem::current_path() / "textfiles";
```

This code implements a dynamic path for our text files, allowing the program to select the current path used by the file system and use that to write and read the text files that are generated. This approach works even when the project folder is placed in unexpected places within the computer’s storage. If the project folder's contents are not altered, the text files should be generated and correctly read every time.

2.5 Conclusion:

This project implements a brute force solution and a divide and conquer solution utilizing pre-sorting to the closest points problem. We compare the experiments and time complexity, outline the development process, evaluate the functions, variables, and data structures used within the code. We also covered the issues we faced and how this project helped us get hands on experience comparing two vastly different algorithms on large data sets of increasing size. This project was a pleasure to do and an amazing opportunity to experience how algorithms operate.