Closest Pair

1 Problem

Consider a problem in computational geometry: Given a set of n 2D points, the goal is to find the closest pair of points in the set. Coordinates are given like so:

$$P_i = (x_i, y_i), \qquad i = 0, \ldots, n-1$$

a) Design a brute force algorithm $ClosestPair(n, X, Y)$ to solve the problem, where n is the number of points and X and Y are arrays of size n storing x and y coordiantes, respectively, of the n points. Analyze the complexity of your algorithm.

b) Design a divide and conquer algorithm $ClosestPair(n, X, Y)$ to solve the problem, where n is the number of points and X and Y are arrays of size n storing x and y coordiantes, respectively, of the n points. Analyze the complexity of your algorithm.

2 Algorithms

2.1 Brute Force

The brute force solution can be described as so: Compute the distances between all possible pairs of points and pick the pair with the smallest distance.

```
ClosestPair( n, X, Y )
  minDistance = infinity
  for i = 0 to n
      for j = i + 1 to n
          if ( sqrt( ( X[i] - X[j] )^2 + ( Y[i] -Y[j] )^2 ) < minDistance )
              minDistance = sqrt( ( X[i] - X[j] )^2 + ( Y[i] -Y[j] )^2 )
              closestPair   = ( ( X[i], Y[i] ), ( x[j], y[j] ) )
  return closestPair
```

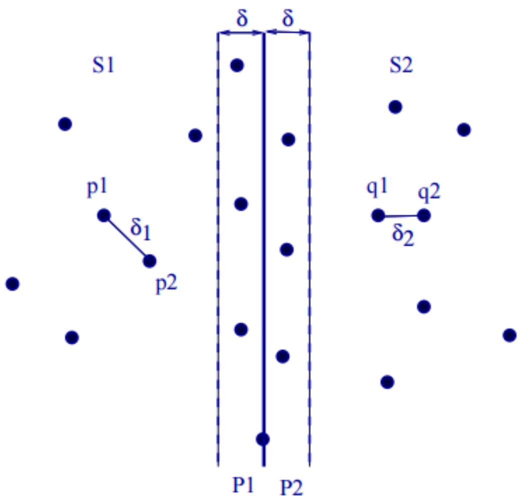
In the actual implementation, I create a Point object with x coordinates and y coordinates to make things easier.

The comlexity of this algorithm is $\mathcal{O}(n^2)$ since you are checking all pairs. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ paris for which you do a finite set of operations. This results in $\frac{n^2-n}{2}$ finite operations which is $\mathcal{O}(n^2)$.

2.2 Divide and Conquer

The problem can also be solved in $\mathcal{O}(n \log n)$ time using a recursive divide and conquer approach like so:

1. Sort the points according to their x-coordinates.
2. If n, number of points, is less than or equal to 3, use brute force algorithm.
3. Split the set of points P into two equal subsets by a vertical line $x = P[mid].x$
4. Solve the problem recursively in the left and right subsets. This gives the left-side and right-side closest pairs.
5. Find which pair of points is closer, the left closest or right closest and call this pair of points d.
6. From the above 4 steps we now have an upper bound of the minimum distance between the closest points. Now we need to consider pairs that contain points on either side of our dividing line $x = P[mid].x$. We can create a region in which points that are closer than d must lie. The points can be no farther from $x = P[mid].x$ than the distance between the points in d. Create an array $strip[]$ of all such points. To make this clearer here is an illustration:



δ= distance between pair d, $P1 = P[0] -> P[mid]$, $P2 = P[mid+1]P[n-1]$

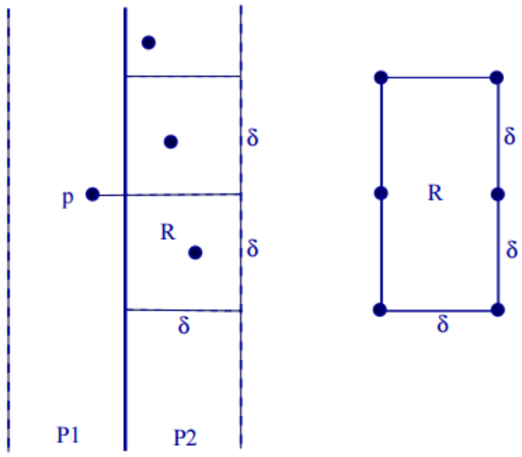
Credits for this image go to Professor Suri here at UCSB

7. Sort $strip[]$ according to y coordinates.
8. Look for a pair in $strip[]$ thats closer than the pair d. If found put pair in s

9. If s is not empty, return s, else return d.

The time analysis of this algorithm is rather complicated. We will analyse the time of each step.

1. $\mathcal{O}(n \log n)$ using merge sort.
2. This is $\mathcal{O}(1)$ since the $\mathcal{O}(n^2)$ algorithm is only used for $n \leq 3$.
3. This is $\mathcal{O}(n)$ since you are just splitting an array by either passing new bounds to the recursive call later or creating two new arrays copying at most n items from P
4. T(n/2) + T(n/2)
5. $\mathcal{O}(1)$, simply compare two distances.
6. $\mathcal{O}(n)$ since you will be adding at most n points, all of them, to $strip[]$.
7. $\mathcal{O}(n \log n)$ using merge sort on at most n items.
8. This is the key to the run time. This appears to be $\mathcal{O}(n^2)$, but it is not, because for each point in $strpi[]$, you only have to calculate the distance between it at most and at most 6 other points. Consider a point $p \in S_1$ where S_1 is the set of points that lie to the left of $x = P[mid].x$ and are in $strip[]$. All points within S_2, the points to the right of S_1 in $strip[]$, that are within distance δ of p must lie in a $\delta \times 2\delta$ rectangle R:



δ= distance between pair d, $P1 = P[0]-> P[mid]$, $P2 = P[mid+1]P[n-1]$

Credits for this image go to Professor Suri here at UCSB

There can be at most 6 points within R since all points in S_1 and S_2 are at a minimum δ apart from all other points in S_1 and S_2 respectively. Because we have already sorted *strip*[] according to their y coordinates, we can linearly go through the every point in *strip*[] comparing the next 6 points to it to check for a distance less than δ. This results in $6n$ sets of finite operations which is $\mathcal{O}(n)$. This onlly works when *strip*[] is presorted according to y coordinates because you can move down the points sequentially.

9. this is $\mathcal{O}(1)$

This overall cost is an addition of all the steps:

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

$$T(n) = 2T(n/2) + 2\mathcal{O}(n\log n) + 3\mathcal{O}(n) + 3\mathcal{O}(1)$$

The $\mathcal{O}(n)$ and $\mathcal{O}(1)$ parts of the equation become dominated by $\mathcal{O}(n\log n)$. We are left with:

$$T(n) = 2T(n/2) + \mathcal{O}(n\log n) \tag{1}$$

To solve this let us take $n = 2^m$.

$$T(2^m) = 2T(2^{m-1}) + 2^m \log(2^m) = 2T(2^{m-1}) + m2^m$$

Calling $T(2^m)$ as $f(m)$, we get

$$\begin{aligned} f(m) &= 2f(m-1) + m2^m \\ &= 2(2f(m-2) + (m-1)2^{m-1}) + m2^m \\ &= 4f(m-2) + (m-1)2^m + m2^m \\ &= 4(2f(m-3) + (m-2)2^{m-2}) + (m-1)2^m + m2^m \\ &= 8f(m-3) + (m-2)2^m + (m-1)2^m + m2^m \end{aligned}$$

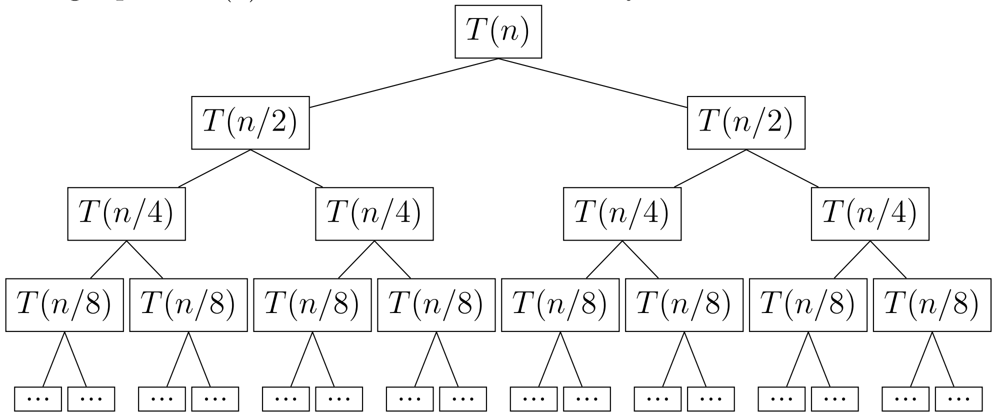
Following this pattern:

$$\begin{aligned} f(m) &= 2^m f(0) + 2^m(1+2+3+\cdots+m) = 2^m f(0) + \frac{m(m+1)}{2}2^m \\ &= 2^m f(0) + m(m+1)2^{m-1} \end{aligned}$$

So

$$T(n) = nT(1) + n\left(\frac{\log(n)(1+\log(n))}{2}\right) = \Theta(n\log^2 n) \tag{2}$$

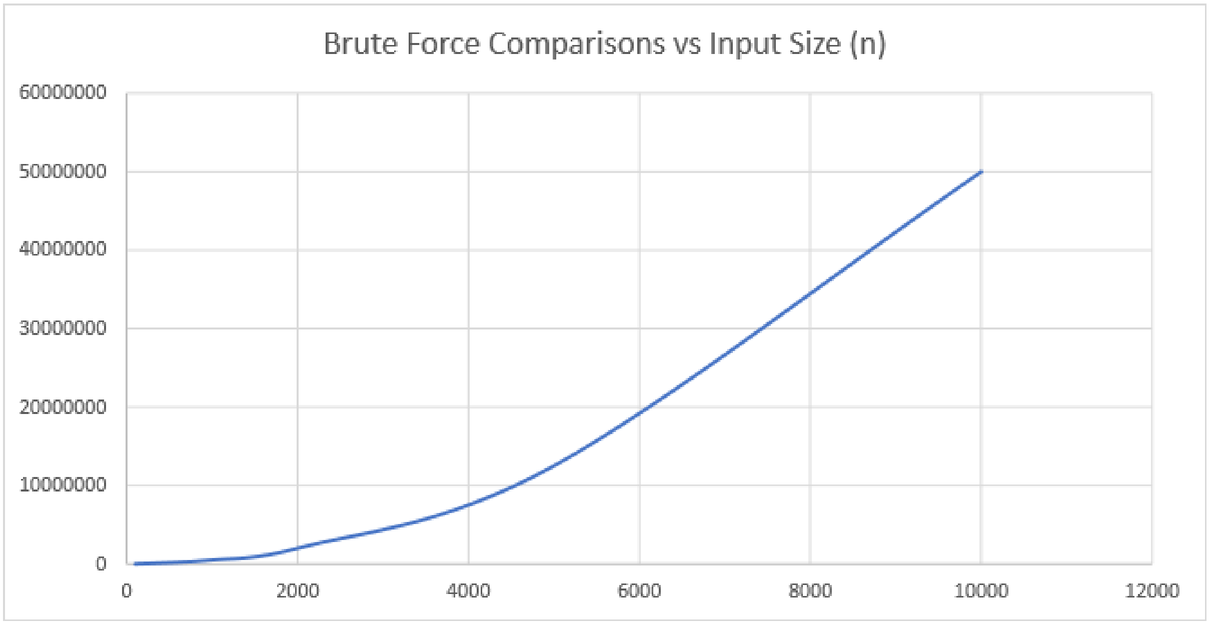
Using equation (1) we can also do a tree analysis:

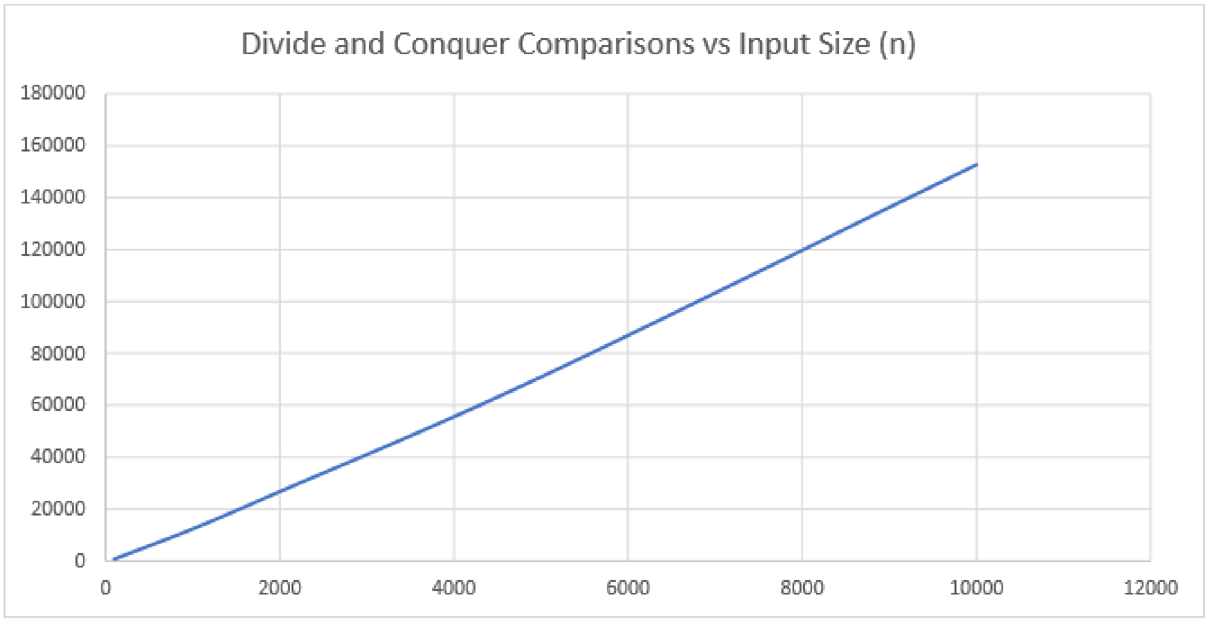

$T(n)$
$T(n/2)$ $T(n/2)$
$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$
$T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$
$\cdots$

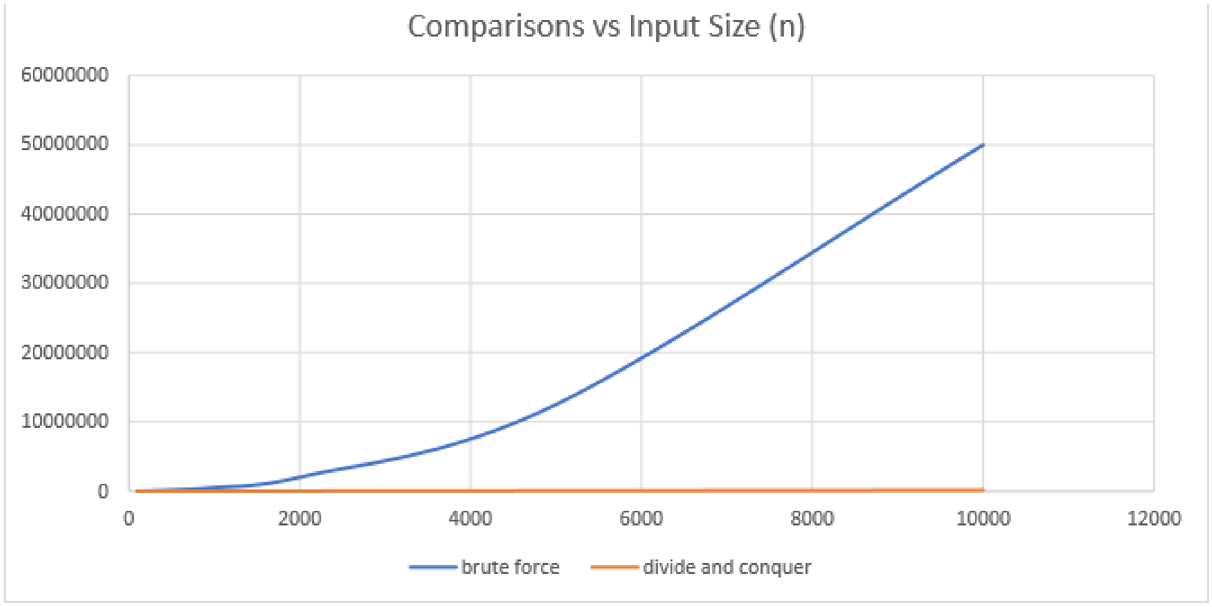
Where at each stage we do $n \log(n)$ operations. So we do $n \log n$ operations $\log n$ times. This results in $n \log n \cdot \log n$ operations which is

$$\Theta(n \log^2 n)$$

3 Results







It is easy to observe that the divide and conquer algorithm performs many less comparisons than the brute force algorithm espicially as the input size becomes larger and larger.

A Implementation in C++

The `C++` implementation with comments: To compile, type 'make all'. The executable file is named 'prog1'

`main.cpp`

```
// Alexander Bauer
// CS 130B Programming Assignment 1
// Last Edited: 4/24/2017

#include <string>   /* std::string */
#include <cmath>  /* std::abs */
#include <limits> /* std::numeric_limits */
#include <math.h> /* sqrt */
#include <vector> /* std::vector */
#include <cstdlib> /* qsort */
#include <iostream> /* std::cout, std::cin */

int numComparisons = 0; //Global variable for counting comparisons

// Point struct to make things easier
struct Point{

    double x;
    double y;

    //CONSTRUCTOR
    Point( double xVal = 0, double yVal = 0 )
    : x{ xVal }, y{ yVal } { }

};

//COMPARISON FUNCTIONS
//These two functions are used to sort points. Used in conjunction
//with std::qsort
int compareX( const void* a, const void* b ){
    Point *p1 = (Point *)a;
    Point *p2 = (Point *)b;
    double val = p1->x - p2->x;
        if ( val < 0 )
            return -1;
        if ( val > 0 )
            return 1;
        return 0;
}

int compareY( const void* a, const void* b ){
    Point *p1 = (Point *)a;
    Point *p2 = (Point *)b;
    double val = p1->y - p2->y;
    if ( val < 0 )
        return -1;
    if ( val > 0 )
        return 1;
    return 0;
}


//DISTANCE FUNCTION
double distanceBetween( Point p1, Point p2 ){
    numComparisons++; // Update number of comparisons
    return sqrt( ( p1.x - p2.x ) * ( p1.x - p2.x )
               +( p1.y - p2.y ) * ( p1.y - p2.y ) );
}

Point* bruteForce( Point P[], int numPoints ){
    double minDistance = std::numeric_limits<double>::max();
    Point* closestPair = new Point[2];
    for( int i = 0; i < numPoints; i++ ){
        for ( int j = i+1; j < numPoints; j++ ){
            double currentDistance = distanceBetween( P[i], P[j] );
            if ( currentDistance < minDistance){
                minDistance = currentDistance;
                closestPair[0].x = P[i].x;
                closestPair[0].y = P[i].y;
                closestPair[1].x = P[j].x;
                closestPair[1].y = P[j].y;
            }
        }
    }
    return closestPair;
}
```

```
// Helper function that finds the closest pair in the strip of given
// size. Points are sorted according to there y coordinates
Point* stripClosest( Point strip[], int numPoints, double d){
    double minDistance = d;
    Point* closestPair = new Point[2];
    // sort by y coordinate
    qsort( strip, numPoints, sizeof(Point), compareY );

    // Search for pair closer than distance d
    for( int i = 0; i < numPoints; i++ ){
        for ( int j = i + 1; j < numPoints && ( strip[j].y - strip[i].y ) < d
            double currentDistance = distanceBetween( strip[i], strip[j] );
            if ( currentDistance < minDistance){
                minDistance = currentDistance;
                closestPair[0].x = strip[i].x;
                closestPair[0].y = strip[i].y;
                closestPair[1].x = strip[j].x;
                closestPair[1].y = strip[j].y;
            }
        }
    }
    return closestPair;
}

Point* divideConquer( Point* P, int numPoints ){
    // P has already been sorted according to x values
    // If there are 2 or 3 points, use the bruteForce algorithm since
    // it has become trivial with this many points;
    if ( numPoints <= 3 )
        return bruteForce( P, numPoints );

    // Decide on a middle point
    int mid = numPoints / 2;

    // Calculate clsoest pair left of midPoint.x dl
    Point* dl = divideConquer( P, mid );

    // Calculate closest pair right of midPoint.x dl
    Point* dr = divideConquer( P + mid, numPoints - mid );

    // Find smaller between dl & dr, d
    Point* d = new Point[2];
    if( distanceBetween( dl[0], dl[1]) < distanceBetween( dr[0], dr[1] )){
        d[0].x = dl[0].x;
        d[0].y = dl[0].y;
        d[1].x = dl[1].x;
        d[1].y = dl[1].y;
    }
    else{
        d[0].x = dr[0].x;
        d[0].y = dr[0].y;
        d[1].x = dr[1].x;
        d[1].y = dr[1].y;
    }
    // Build strip[] that contains points closer than
    // distanceBetween(d[0], d[1]) to the vertical line x=P[mid].x
    std::vector<Point> strip;
    double smallestDistance = distanceBetween( d[0], d[1] );
    for( int i = 0; i < numPoints; i++){
        numComparisons++; //I added this update because I think you are
                          //technically comparing everytime, but I was maki
                          //it easier to code
        if ( std::abs( P[i].x - P[mid].x ) < smallestDistance ){
            strip.push_back( P[i] );
        }
    }

    // Find the closest points in the strip
    Point* s = stripClosest( &strip[0], strip.size(), smallestDistance );
    if( distanceBetween( s[0], s[1] ) < .000001 ){
        return d;

    }
    else{
        return s;

    }

}

int main() {
    int numPoints = 0;
    double x;
    double y;
    std::cin >> numPoints;
    std::vector<Point> P;
    for( int i = 0; i < numPoints; i++){
        std::cin >> x;
        std::cin >> y;
        P.push_back( Point( x, y ) );
    }
    // Sort the array according to x coordinates
    std::qsort( &P[0], numPoints, sizeof(Point), compareX );

    Point* closestPointsDC = divideConquer(&P[0], numPoints);
    int divideConquerComparisons = numComparisons; //Record total Divide conq
    numComparisons = 0; // Reset for brute force

    Point* closestPointsBF = bruteForce(&P[0], numPoints);
    int bruteForceComparisons = numComparisons; //Record total brute force

    //Output Results
    std::cout << closestPointsBF[0].x << "␣" << closestPointsBF[0].y
              << "␣" << closestPointsBF[1].x << "␣" << closestPointsBF[1].y
              << std::endl << bruteForceComparisons << std::endl;

    std::cout << closestPointsDC[0].x << "␣" << closestPointsDC[0].y
              << "␣" << closestPointsDC[1].x << "␣" << closestPointsDC[1].y
              << std::endl << divideConquerComparisons << std::endl;

    return 0;
}
```

References

[1] Professor Suri *https://www.cs.ucsb.edu/ suri/cs235/ClosestPair.pdf.*