**Divide And Conquer Based Approach:  
  
1) Divide:** Recursively break a given problem into sub problems of same type.

**2)Conquer:** Recursively solve the subproblems.

**3)Combine:** Appropriately combine the result of subproblems to get original answer.

**Divide And Conquer vs Dynamic Programming:**Difference is related to overlapping subproblems.

**Divide And Conquer Related Problems:  
  
1) Binary Search**

**2)Merge sort**

**3)Quick sort**

I know these things.

1. **Calculate pow(x,n):**

/\* Extended version of power function that can work

for float x and negative y\*/

#include<stdio.h>

float power(float x, int y)

{

float temp;

if( y == 0)

return 1;

temp = power(x, y/2);

if (y%2 == 0)

return temp\*temp;

else

{

if(y > 0)

return x\*temp\*temp;

else

return (temp\*temp)/x;

}

}

/\* Program to test function power \*/

int main()

{

float x = 2;

int y = -3;

printf("%f", power(x, y));

return 0;

}

**Now, for negative (temp\*temp)/x that is very important.**

**Because, we all know, in case of positive numbers, the result will be:**

If(y%2==0)

{

return temp\*temp;

}

else

{

return x\*temp\*temp;

}

**Count Of Inversions:**Now, count of inversion is a popular problem. I know at least three solutions of it. Now, first solution is using merge sort:

(using enhanced merge sort)

Suppose we know the number of inversions in the left half and right half of the array (let be inv1 and inv2), what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().

In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j]

I.e. suppose, first sorted partition contains arr[0…i…mid] elements and second sorted partition contains arr[mid+1..n-1]

Now, if arr[i]> arr[j[ that necessarily means arr[i+1..mid] is greater than arr[j] hence, 1+(mid-i-1)=(mid-i) number of inversions will be added.

**Closest Pair Of Points:**

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.



Following are the detailed steps of a O(n (Log2n)2) algorithm.

**Input:** An array of n points P[]

**Output:** The smallest distance between two points in the given array.

As a pre-processing step, input array is sorted according to x coordinates.

1) Find the middle point in the sorted array, we can take P[n/2] as middle point.

2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].

1. Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d. (This dl and dr are minimum distance between pairs where either both points are lying in the first subarray or both points are lying in the second subarray)   
     
   mindis

4)From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. **Consider the vertical line passing through passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.  
  
  
closepair**

5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.

6)Find the smallest distance in strip[]. This is tricky. From first look, it seems to be a O(n2) step, but it is actually O(n). It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See this for more analysis.

7) Finally return the minimum of d and distance calculated in above step (step 6)

**// A divide and conquer program in C/C++ to find the smallest distance from a**

**// given set of points.**

**#include <stdio.h>**

**#include <float.h>**

**#include <stdlib.h>**

**#include <math.h>**

**// A structure to represent a Point in 2D plane**

**struct Point**

**{**

**int x, y;**

**};**

**/\* Following two functions are needed for library function qsort().**

**Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/**

**// Needed to sort array of points according to X coordinate**

**int compareX(const void\* a, const void\* b)**

**{**

**Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;**

**return (p1->x - p2->x);**

**}**

**// Needed to sort array of points according to Y coordinate**

**int compareY(const void\* a, const void\* b)**

**{**

**Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;**

**return (p1->y - p2->y);**

**}**

**// A utility function to find the distance between two points**

**float dist(Point p1, Point p2)**

**{**

**return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +**

**(p1.y - p2.y)\*(p1.y - p2.y)**

**);**

**}**

**// A Brute Force method to return the smallest distance between two points**

**// in P[] of size n**

**float bruteForce(Point P[], int n)**

**{**

**float min = FLT\_MAX;**

**for (int i = 0; i < n; ++i)**

**for (int j = i+1; j < n; ++j)**

**if (dist(P[i], P[j]) < min)**

**min = dist(P[i], P[j]);**

**return min;**

**}**

**// A utility function to find minimum of two float values**

**float min(float x, float y)**

**{**

**return (x < y)? x : y;**

**}**

**// A utility function to find the distance between the closest points of**

**// strip of given size. All points in strip[] are sorted according to**

**// y coordinate. They all have an upper bound on minimum distance as d.**

**// Note that this method seems to be a O(n^2) method, but it's a O(n)**

**// method as the inner loop runs at most 6 times**

**float stripClosest(Point strip[], int size, float d)**

**{**

**float min = d; // Initialize the minimum distance as d**

**qsort(strip, size, sizeof(Point), compareY);**

**// Pick all points one by one and try the next points till the difference**

**// between y coordinates is smaller than d.**

**// This is a proven fact that this loop runs at most 6 times**

**for (int i = 0; i < size; ++i)**

**for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)**

**if (dist(strip[i],strip[j]) < min)**

**min = dist(strip[i], strip[j]);**

**return min;**

**}**

**// A recursive function to find the smallest distance. The array P contains**

**// all points sorted according to x coordinate**

**float closestUtil(Point P[], int n)**

**{**

**// If there are 2 or 3 points, then use brute force**

**if (n <= 3)**

**return bruteForce(P, n);**

**// Find the middle point**

**int mid = n/2;**

**Point midPoint = P[mid];**

**// Consider the vertical line passing through the middle point**

**// calculate the smallest distance dl on left of middle point and**

**// dr on right side**

**float dl = closestUtil(P, mid);**

**float dr = closestUtil(P + mid, n-mid);**

**// Find the smaller of two distances**

**float d = min(dl, dr);**

**// Build an array strip[] that contains points close (closer than d)**

**// to the line passing through the middle point**

**Point strip[n];**

**int j = 0;**

**for (int i = 0; i < n; i++)**

**if (abs(P[i].x - midPoint.x) < d)**

**strip[j] = P[i], j++;**

**// Find the closest points in strip. Return the minimum of d and closest**

**// distance is strip[]**

**return min(d, stripClosest(strip, j, d) );**

**}**

**// The main function that finds the smallest distance**

**// This method mainly uses closestUtil()**

**float closest(Point P[], int n)**

**{**

**qsort(P, n, sizeof(Point), compareX);**

**// Use recursive function closestUtil() to find the smallest distance**

**return closestUtil(P, n);**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};**

**int n = sizeof(P) / sizeof(P[0]);**

**printf("The smallest distance is %f ", closest(P, n));**

**return 0;**

**}**

Now, this function is required to sort points based on x coordinate. As, the sort function in algorithm header accepts an optional function pointer.

**int compareX(const void\* a, const void\* b)**

**{**

**Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;**

**return (p1->x - p2->x);**

**}**

Now, this function is required to sort points based on y coordinates.

**int compareY(const void\* a, const void\* b)**

**{**

**Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;**

**return (p1->y - p2->y);**

**}**

This actually calculates the distance between Point P1 and P2

**float dist(Point p1, Point p2)**

**{**

**return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +**

**(p1.y - p2.y)\*(p1.y - p2.y)**

**);**

**}**

This function will be called when there is atmost 3 points.

**float bruteForce(Point P[], int n)**

**{**

**float min = FLT\_MAX;**

**for (int i = 0; i < n; ++i)**

**for (int j = i+1; j < n; ++j)**

**if (dist(P[i], P[j]) < min)**

**min = dist(P[i], P[j]);**

**return min;**

**}**

Now, after sorting all the points based on x coordinate, we will need to recursively find dl and dr and find minimum of them d. After, that, this function will be called

**float stripClosest(Point strip[], int size, float d)**

**{**

**float min = d; // Initialize the minimum distance as d**

**qsort(strip, size, sizeof(Point), compareY);**

**//sort it based on y coordinate. Now, all points are strip is already sorted based on x coordinate**

**// Pick all points one by one and try the next points till the difference**

**// between y coordinates is smaller than d.**

**// This is a proven fact that this loop runs at most 6 times**

**for (int i = 0; i < size; ++i)**

**for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)**

**if (dist(strip[i],strip[j]) < min)**

**min = dist(strip[i], strip[j]);**

**return min;**

**}**

Note that, if for some j strip[j].y-strip[i].y>=min we will no longer check more values of j for current i.

**float closestUtil(Point P[], int n)**

**{**

**// If there are 2 or 3 points, then use brute force**

**if (n <= 3)**

**return bruteForce(P, n);**

**// Find the middle point**

**int mid = n/2;**

**Point midPoint = P[mid];**

**// Consider the vertical line passing through the middle point**

**// calculate the smallest distance dl on left of middle point and**

**// dr on right side**

**float dl = closestUtil(P, mid);**

**float dr = closestUtil(P + mid, n-mid);**

**//recursively calling closestUtil**

**// Find the smaller of two distances**

**float d = min(dl, dr);**

**// Build an array strip[] that contains points close (closer than d)**

**// to the line passing through the middle point**

**Point strip[n];**

**int j = 0;**

**for (int i = 0; i < n; i++)**

**if (abs(P[i].x - midPoint.x) < d)**

**strip[j] = P[i], j++;**

**// Find the closest points in strip. Return the minimum of d and closest**

**// distance is strip[]**

**return min(d, stripClosest(strip, j, d) );**

**}**