CS375-A0 Assignment 2 (Fall 2013)
(Answer)

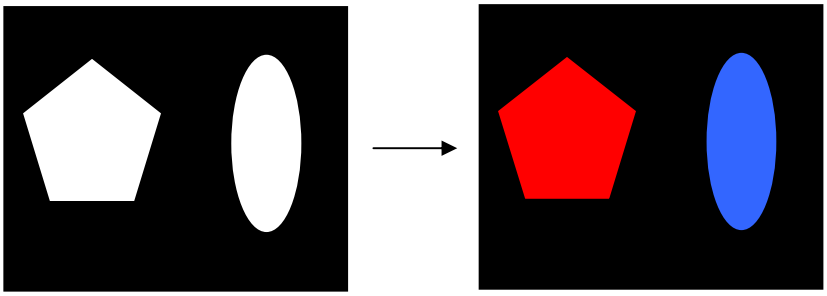
Objective:

(1) Design and analyze an algorithm using divide and conquer strategy
(2) Establish recurrence equation and solve it.
(3) Enhance the concept of proof of correctness for algorithms.

There are two parts in this assignment: (A) Theory part and (B) programming part

[Part A] Theory [77%]:

1. (9%) Given an image which shows two white regions, design an algorithm to fill the region 1 by the red color, and fill the region 2 by the blue color. Assume the image is represented by a Matrix with the size of N by N (e.g., color[x, y]), use the *recursive* algorithm to solve this problem. (Assume the value of color is either WHITE, or RED, or BLUE). White a pseudo-code.



Answer:

```
Void Start(Color[N, N])
{
For (x=0; x<N; x++)
For(y=0; y<N, y++)
{
 If Color[x, y]= = WHITE
          Fill(x, y, RED, WHITE);

  If Color[x, y] = = WHITE
          Fill(x, y, BLUE, WHITE);
}
}

Void Fill(x, y, FillColor, InteriorColor)
{
  If Color[x,y] = = InterriorColor;
   {
```

```
    Color[x, y] = FillColor;
    Fill(x+1, y, FillColor, InterriorColor);
    Fill(x-1, y, FillColor, InterriorColor);
    Fill(x, y+1, FillColor, InterriorColor);
    Fill(x, y-1, FillColor, InterriorColor);
  }
 }
```

2. (9%) Given a sorted array of distinct integers A[1, …, n], you want to find out whether there is an index i for which A[i]=i. Give a divide-and-conquer algorithm to solve this problem. Derive the time complexity. (Note: the running time much be less than O(n)).

Answer:

Same as the binary search algorithm, divide the array into two, and check whether the middle entry has A[m]=m;

If A[m] > m, then search the segment of A[1, …, m-1]
If A[m] < m, then search the segment of A[m+1, …, n]

Repeat the procedure for each segment.

e.g.,

```
FindM(L, R, A[])
{
  Middle point: m
 If (m= = A[m]) return m;

 If (A[m] > m)  {index=FindM(L, m-1, A[]); if (index != -1) return (index);}

 If(A[m] <m) { index =FindM(m+1, R, A[]); if (index != -1) return (index);}


 Return -1;
}
```

3. [10%] Write a piece of code to plot the following graph. Assuming that the plotting function has been provided as DrawSquare(x, y, r), which draw a square of size 2r with center (x, y).

(1) Pseudo-Code to plot following graph. (4%)

(2) Write the recurrence equation for your algorithm. (3%)

(3) Plot the recursion tree to derive the solution of T(r). (3%)

(Note: r is the input size, which is the power of 2. The time complexity for drawing one square is O(1)). (Hint: The input of function can be: int x, int y, int r)

Answer:

The following program output the result when the input size is r=4.

```
//  Assume r is power of 2
star(int x, int y, int r)
{
    if (r>0)
     {
          star(x-r, y+r, r/2);
          star(x+r, y+r, r/2);
          star(x-r, y-r, r/2);
          star(x+r, y-r, r/2);
      DrawSquare (x,y,r); //Draws a square of size 2r with center (x,y) takes time O(1)
        }
       else { return;}
    }
```

Or:

```
star(int x, int y, int r)
{
    if (r= =0) return;
     else
      DrawSquare (x,y,r); //Draws a square of size 2r with center (x,y) takes time O(1)
          star(x-r, y+r, r/2);
          star(x+r, y+r, r/2);
          star(x-r, y-r, r/2);
          star(x+r, y-r, r/2);

        }
}
```

Recurrence equation:

T(r) = 4T(r/2) + O(1)
T(1)=O(1);
T(0) = 0;

k=lg(r)

$$T(r) = O(1)\sum_{i=0}^{k} 4^i = O(1)\frac{4^{k+1}-1}{4-1} = O(1)(4 \cdot r^2 - 1)/3 = O(r^2)$$

Recursion tree:

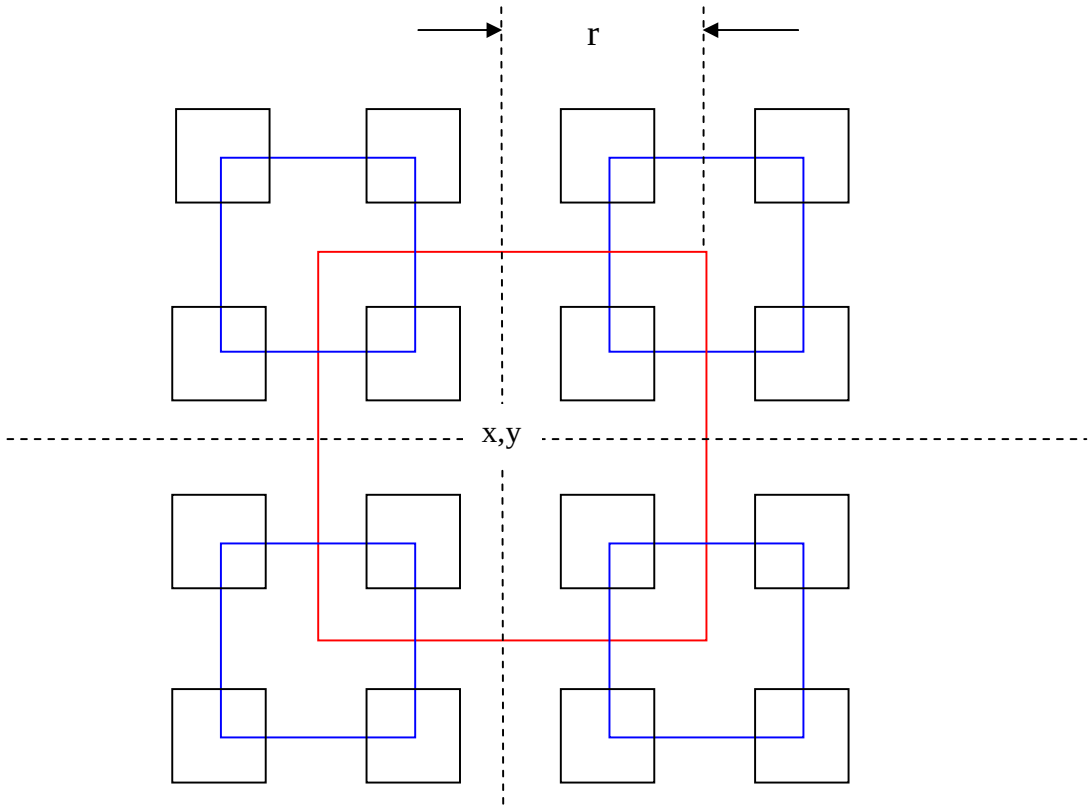
O(1) 4^0

O(1) O(1) O(1) O(1) 4^1

O(1)O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) 4^2

……..

4^k



4. (9%) An Array A[1,…,n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. Show how to solve this problem in O(nlgn) time.(Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A?) Note that it is required to use a divide-and-conquer approach with O(nlgn) time complexity.

Answer :

```
// suppose ME( ) returns the value of an array (N, M)
// N is the number of the majority (if no majority, N="-1")
// M is the majority element (if no majority, M = "NULL")

ME(A, n)
{
//base case
If(n= =1) return (1, A[1]);

A1[1, .., n/2]= A[1, …, n/2];
A2[1, …. n/2]= A[n/2+1, …, n];

(N1, m1) = ME(A1, n/2);   // m1 is the majority element; N1 is the number of m1
(N2, m2) = ME(A2, n/2);   // m2 is the majority element; N2 is the number of m2


//// no majority element found in A1 and A2
If (m1= = NULL and m2 = = NULL) return (N="-1"; M= NULL);

Else   ////two majority elements equal
    If (m1= = m2) then return (N = N1+N2; M = m1);

//Otherwise, if (m1 != m2) (including m1 = =NULL or  m2 = =NULL)
//then search the m1 in A2 or search the m2 in A1, get counting number C;
// Time complexity: O(n)

   Else
      if (m1 != NULL)
            C= search(m1, A[n/2+1, ….n]); //assume the "search" function is provided
            If ((N1 +C)> n/2) return (N = N1 +C, M = "m1");
       if (m2 != NULL)
            C= search(m2, A[1, ….n/2]); //assume the "search" function is provided
            If ((N2 +C )> n/2)  return (N = N2 +C,  M = "m2");

//// none of above cases
return (N=-1, M=NULL)

  }
```

$T(n) = 2T(n/2) + Q(n)$
$a = 2$, $b = 2$ so $n\log_b a = n\log_2 2$
Since $n\log_2 2 = n$, $Q(n)/n = Q(1) = Q(\lg^0 n)$,
we are dealing with Case 2.
$T(n) = Q(n\log_b a\ lg^{k+1} n) = Q(n\lg n)$

Note 1:

Some students know how to divide the array into half and recursively call the ME. However, the "search" of m1 or m2 is not conducting on half of the array. Instead, it searches the entire array (see sample below from a student). As a result, it increases the time complexity dramatically. For the base case, it searches ALL single element in the entire array, which almost equals to the brute-force approach. However, since it uses the divide-and-conquer approach, we can still give the full credit for such case.

```
Element getMajority(List[] l)
{
  if (list.length == 1) return l[0];
  maj1 = getMajority ( makeList ( l, 0, N / 2) );
  maj2 = getMajority ( makeList ( l, N / 2 +1, N) );
  count1 = countInList(l, maj1); // see how many times its repeated
  count2 = countInList(l, maj2); // see how many times its repeated
  if (maj1==null && maj1==null) return / *this may be   eliminated since there is the return
at the end if neither count1
nor count2 are above N/2*/


if (count1 > N / 2 ) return maj1;
  if (count2 > N / 2) return maj2;
  return
}
```

Note 2: Alternative Thoughts (This example is not required, just for reference)

Majority Element:

```
    sort array A
    key = A[n/2]
    indexL = searchL(0, (n/2)-1, key)
    indexR = searchR((n/2)+1, n, key)
    if(indexR - indexL > n/2)
            return true
    else
            return false

searchL(low, high, key)
    mid = (low + high)/2
    if(low = high)
            return mid
    if(A[mid] = key)
            if(A[mid-1] != key)
                    return mid-1
            else
                    return searchL(low, mid-1, key)
    if(A[mid] != key)
```

```
        if(A[mid+1] = key)
                return mid+1;
        else
                return searchL(mid+1, high, key)

searchR(low, high, key)
    mid = (low + high)/2
    if(low = high)
        return mid
    if(A[mid] = key)
        if(A[mid+1] != key)
            return mid+1
        else
            return searchR(mid+1, high, key)
    if(A[mid] != key)
        if(A[mid-1] = key)
            return mid-1
        else
            return searchR(low, mid-1, key)
```

5. (9%) Suppose you are choosing between the following three algorithms:
   - Algorithm A solves problems by dividing them into five sub-problems of half size, recursively solving each sub-problem, and then combining the solutions in linear time.
   - Algorithm B solves problems of size n by recursively solving two sub-problems of size n-1 and then combining the solutions in constant time.
   - Algorithm C solves problems of size n by dividing them into nine sub-problems of size n/3, recursively solving each sub-problem, and then combining the solutions in $O(n^2)$ time.

   What are the running times of each of these algorithms (in big-O notation), and which would you choose? (Note: the solving strategy of the problem and the sub-problems is same).

   Answer:

   i. $T(n) = 5T(n/2)+O(n)$ ➔ $T(n)=O(n^{\log_2 5})$ (Master method case 1)

   ii. $T(n) = 2T(n-1) + O(1)$ ➔ $T(n)=O(2^0+2^1+\ldots+2^{(n-1)})=O(2^n-1)=O(2^n)$

   iii. $T(n) = 9T(n/3) +O(n^2)$ ➔ $T(n)=O(n^2 \lg n)$ (Master method case 2)

   We will choose (C)

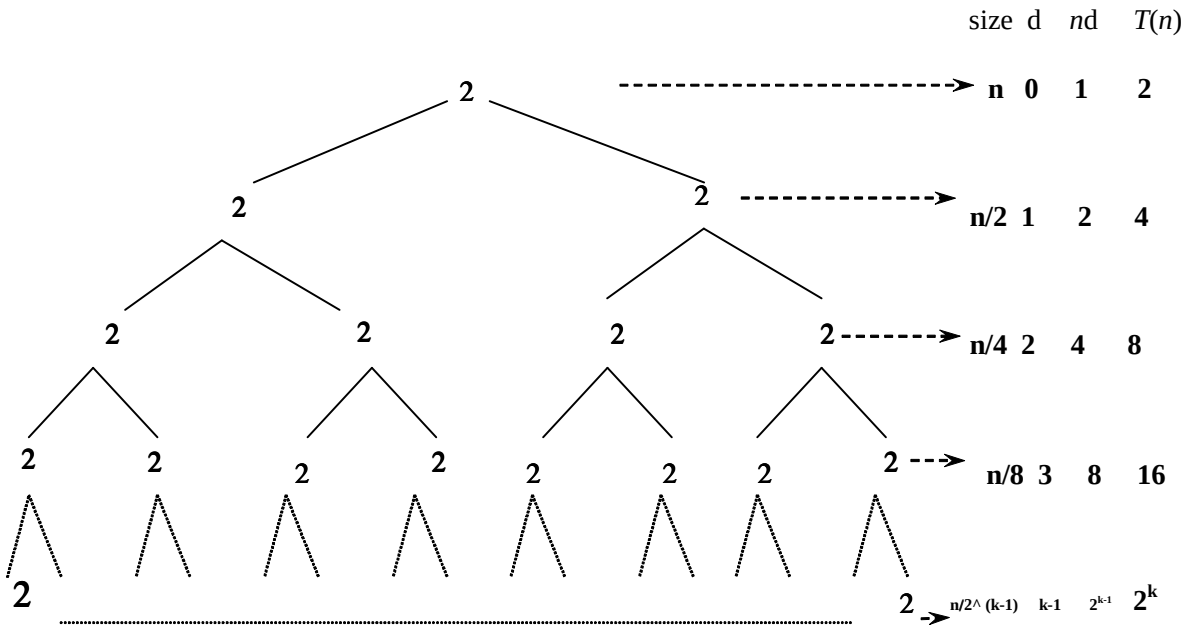
6. [10%]

a. [3%]Write the recurrence equation for the code below. Use the number of comparisons as your barometer operation (The *min* operation requires 1 comparison and the *max* operation requires 1 comparison*): (Note: you can count min and max as the comparison operation and skip the rt-lt <=1)*

```
MinMax(A,lt,rt)
// return a pair with the minimum and the maximum
   if (rt - lt ≤ 1)
       return (min(A[lt], A[rt]), max(A[lt],A[rt]));
   (min1, max1) = MinMax(A,lt, ⌊(lt+rt)/2⌋ );
   (min2, max2) = MinMax(A, ⌊ (lt+rt)/2 ⌋ +1,rt);
   return (min (min1, min2), max(max1, max2));
```

$$T(n) = \begin{cases} 2 \text{ if } n \le 2 \\ 2T(n/2) + 2 \text{ otherwise} \end{cases}$$

b. [4%] Show the recursion tree and solve the recurrence equation for this code. For simplicity assume n = 2^k. This algorithm requires a smaller number of comparisons than the one in question 4.

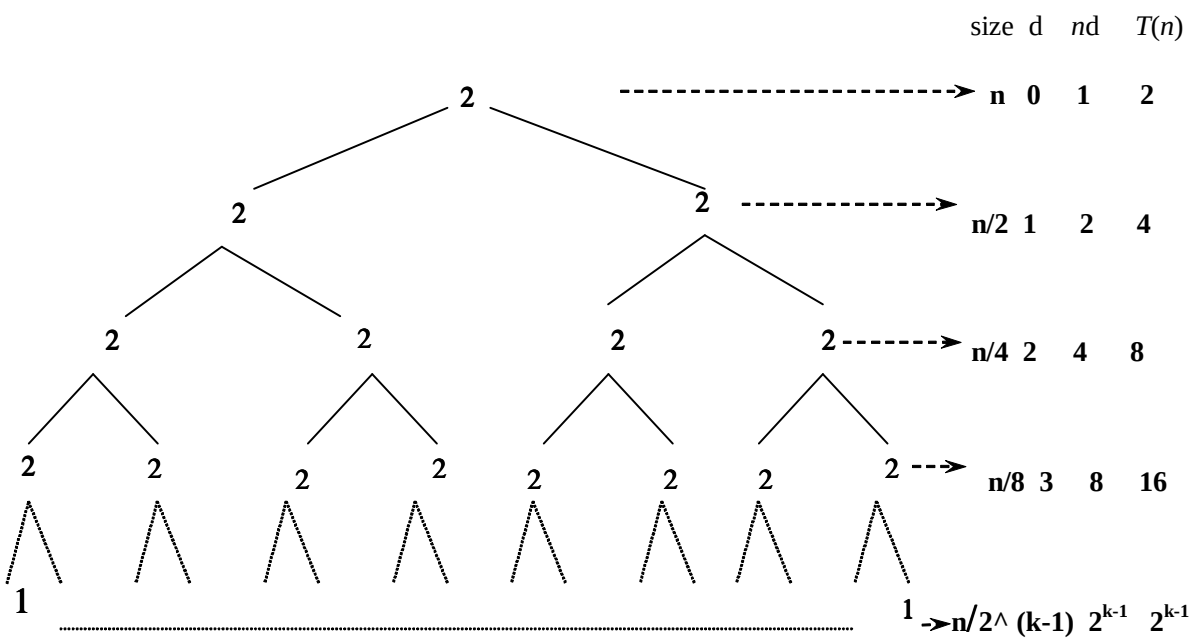


Total = $(2+4+\ldots+2^k) = 2(1+2+\ldots+2^{k-1}) = 2(2^k-1) = 2(n-1)$

Base: n=2 T(n)=2(n-1) = 2(2-1) = 2
Hypothesis: T(n)=2(n-1)
Induction step: T(2n)=2T(n)+2=2(2(n-1)) +2= 4n-2 = 2((2n) -1)

Alternative answer: (Still be fine if take O(1) if n<=2)

$$T(n) = \begin{cases} 1 \text{ if } n \le 2 \\ 2T(n/2) + 2 \text{ otherwise} \end{cases}$$



Total = $2(1 + 2 + \ldots + 2^{k-2}) + 2^{k-1} = 2*(2^{k-1}-1) + 2^{k-1} = 3(n/2) - 2$

a. [3%] Prove the solution using induction

Base: n=2 T(n)=3(2/2)-2=1
Hypothesis: T(n)=3(n/2)-2
Induction step: T(2n)=2T(n)+2=2(3n/2-2)+2=3n-2=3((2n)/2)-2

7. [6%] Find the asymptotic bound of the divide and conquer recurrence T(n) using master method: (1) T(n)=16T(n/4)+n^4+3;

Answer:

$a = 16, b = 4,\ f(n) = n^4 + 3$

$\log_4 16 = 2$

$\frac{n^4+3}{n^2} = n^2 + \frac{3}{n^2} \geq n^2 \in \Omega(n^1)$

$\varepsilon = 1(Case:3)$

$16 f(n/4) = 16((\frac{n}{4})^4 + 3) = \frac{n^4}{16} + 48 \leq c(n^4 + 48)$

$we\_can\_select: c = 1/8$

$(af(n/b) \leq cf(n))\_is\_true ==> when\_n >= 6$

$T(n) = \Theta(n^4 - 1) = \Theta(n^4)$

(8) [9%] Given the following piece of code, prove its correctness based on the loop invariant. (assume X[i] are known).

```
Line 1: A[0]=X[0]; s=X[0]*X[0];
Line 2: for i=1 to n-1 do
Line 3:     s= s + X[i]* X[i]
Line 4:     A[i] = sqrt(s)
Line 5: return array A
```

Answer:
Loop invariant:
At the start of jth iteration of FOR loop,
A[j-1]=sqrt(X[0]*X[0]+X[1]*X[1]+…+X[j-1]*X[j-1])

Initialization:
At the start of execution of FOR loop, A[1-1]=sqrt(X[0]*X[0])=X[0];

Maintenance:
At the start of kth iteration:
A[k-1]=sqrt(X[0]*X[0]+X[1]*X[1]+…+X[k-1]*X[k-1])

At the start of (k+1)th iteration:

A[k]= sqrt(s+X[k]*X[k]) =sqrt(X[0]*X[0]+X[1]*X[1]+…+X[k-1]*X[k-1] +X[k]*X[k])

Termination:
At the start of nth iteration (j=n):
A[n-1]= sqrt(X[0]*X[0]+X[1]*X[1]+…+X[n-1]*X[n-1])

9. [6%] Solve the following recurrence equation using methods of characteristic equation.

T(n)=5T(n-1)-8T(n-2)+4T(n-3) for n>2
T(0)=0
T(1)=1
T(2)=2

Answer:

$r^3 - 5r^2 + 8r - 4 = 0$

$r^3 - r^2 - 4r^2 + 8r - 4 = 0$

$r^2(r-1) - 4(r-1)^2 = 0$

$(r-1)(r^2 - 4r + 4) = 0$

$(r-1)(r-2)(r-2) = 0$

$r = 1; r = 2; r = 2$

$T(n) = c_1 1^n + c_2 2^n + c_3 n 2^n$

$0 = c_1 + c_2$

$1 = c_1 + c_2 2 + c_3 2$

$2 = c_1 + c_2 2^2 + c_3 2 * 2^2$

$c_1 = -2, c_2 = 2, c_3 = -1/2$

$T(n) = -2 + 2^{n+1} - n2^{n-1}$

10. [Extra 8%] Solve the following recurrence equations using recursion tree technique discussed in class. You may make any assumptions about *n*, such as to assume that *n* is an exact power of 2.

T(n)=T(n/2)+T(n/4)+T(n/8)+n

n → n

n/2 n/4 n/8 → n(1/2+1/4+1/8)=(7/8)n

n/4 n/8 n/16 n/8 n/16 n/32 n/16 n/32 n/64 → n(1/4+2/8+3/16+2/32+1/64)= n(49/64)= n (7/8)^2

……

$$\sum_{i=0}^{\log n} (\tfrac{7}{8})^i n = n\Theta(1) = \Theta(n)$$

[Part B]: Divide and Conquer Programming (23%)

1. [13%] Implement the algorithm for finding the closest pair of points in two dimension plane using divide and conquer strategy.

A system for controlling air or sea traffic might need to know which are the two closest vehicles in order to detect potential collisions. This part solves the problem of finding the closest pair of points in a set of points. The set consists of points in R^2 defined by both an x and a y coordinate. The "closest pair" refers to the pair of points in the set that has the smallest Euclidean distance, where Euclidean distance between points $p_1=(x_1,y_1)$ and $p_2=(x_2,y_2)$ is simply $sqrt((x_1-x_2)^2+(y_1-y_2)^2)$. If there are two identical points in the set, then the closest pair distance in the set will obviously be zero.

Input data: n points with coordinates:

X coordinates: p[0].x, p[1].x, p[2].x,…., p[n].x

Y coordinates: p[0].y, p[1].y, p[2].y,…., p[n].y

Output: minimum distance between points p[i] and p[j] (index i and j should be identified)

Input data for test:

```
n = 10000;
for(i=0; i<n; i++)
 {
```

```
p[i].x= n- i;
p[i].y= n- i;
 }
```

Other Input data for test:

```
n = 10000;
for(i=0; i<n; i++)
 {
  p[i].x= i*i;
  p[i].y= i*i;
   }
```

Or:

Input:

If X>0
X=1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, ……,19995, 19997, 19999 (odd numbers)
Y=int($\sqrt{9999^2-(X-10000)^2}$)

Else
X=0, -2, -4, -6, -8, -10, -12, ……, -19996, -19998, -20000 (even number)
Y= int($\sqrt{10000^2-(X+10000)^2}$)

2. (10%) Use the brute-force approach to solve the above problem. Compare the two implemented algorithms in terms of time complexity. Print out the time cost during the execution of these two algorithms.

3. (Extra 10%) apply the close-pair algorithm to the three dimensional case.

Note 1:

Your report of the programming part should follow the following format:

(1) Algorithm description

(2) Major codes

(3) Running results

(4) Report of any bugs

Note 2:

Zip all your files (including the source code, executable code, report (.doc), ….) into one file (using the name: *YourLastName_ ProgrammingLanguageUsed _Assignment2.zip*).
Submit the codes through the digital drop box of blackboard.

Note: Your program only needs to output ONE pair of points with smallest distance. Multiple-pairs (with a same smallest distance) are not required.

Our closest pair point algorithm is to address one pair (with a smallest distance).

If output all the pairs which have a same smallest distance, you can get extra points (5 points).

Suggested Answer:

Using the divide-and-conquer algorithm introduced in the lecture.

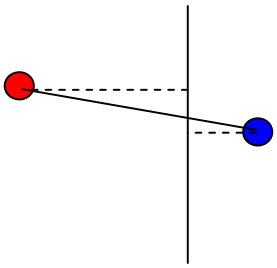
However, you need to consider carefully the base case:

(1) One way to do: (The best way to do)

Make sure you split the group points into two parts approximately half the number of the total points, and each part should contain at least two points. In other words, if the number of points in a part is **2 or 3**, then stop splitting of this part.

(2) Alternative way to do: (More complicated than the first way)

If each part (left and right) has only one point, the shortest distance should be the distance from the red dot to the blue dot.



If one side has only one point and the other side has two points, then the return of the first part will be the distance from the blue point to the mid-line, and the return of the other part will be the distance of two red dots. The mid-band is decided by the blue distance.