Divide & Conquer

1. Write a C program to calculate pow(x,n)

Below solution divides the problem into subproblems of size y/2 and call the subproblems recursively.

```c
#include<stdio.h>

/* Function to calculate x raised to the power y */
int power(int x, unsigned int y)
{
    if( y == 0)
        return 1;
    else if (y%2 == 0)
        return power(x, y/2)*power(x, y/2);
    else
        return x*power(x, y/2)*power(x, y/2);

}

/* Program to test function power */
int main()
{
    int x = 2;
    unsigned int y = 3;

    printf("%d", power(x, y));
    getchar();
    return 0;
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Algorithmic Paradigm: Divide and conquer.

Above function can be optimized to O(logn) by calculating power(x, y/2) only once and storing it.

```c
/* Function to calculate x raised to the power y in O(logn)*/
int power(int x, unsigned int y)
{
    int temp;
    if( y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

Time Complexity of optimized solution: O(logn)

Let us extend the pow function to work for negative y and float x.

```
/* 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));
    getchar();
    return 0;
}
```

2. Median of two sorted arrays

Question: There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays(i.e. array of length 2n). The complexity should be O(log(n))

Median: In probability theory and statistics, a median is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half.
The median of a finite list of numbers can be found by arranging all the numbers from lowest value to highest value and picking the middle one.

For getting the median of input array { 12, 11, 15, 10, 20 }, first sort the array. We get { 10, 11, 12, 15, 20 } after sorting. Median is the middle element of the sorted array which is 12.

There are different conventions to take median of an array with even number of elements, one can take the mean of the two middle values, or first middle value, or second middle value.

Let us see different methods to get the median of two sorted arrays of size n each. Since size of the set for which we are looking for median is even (2n), we are taking average of middle two numbers in all below solutions.

Method 1 (Simply count while Merging)
Use merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n(For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array. See the below implementation.

Implementation:

```
#include <stdio.h>

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int i = 0;  /* Current index of i/p array ar1[] */
    int j = 0; /* Current index of i/p array ar2[] */
    int count;
    int m1 = -1, m2 = -1;

    /* Since there are 2n elements, median will be average
     of elements at index n-1 and n in the array obtained after
     merging ar1 and ar2 */
    for (count = 0; count <= n; count++)
    {
        /*Below is to handle case where all elements of ar1[] are
          smaller than smallest(or first) element of ar2[]*/
        if (i == n)
        {
            m1 = m2;
            m2 = ar2[0];
            break;
        }

        /*Below is to handle case where all elements of ar2[] are
          smaller than smallest(or first) element of ar1[]*/
        else if (j == n)
        {
            m1 = m2;
            m2 = ar1[0];
            break;
        }

        if (ar1[i] < ar2[j])
        {
            m1 = m2;  /* Store the prev median */
            m2 = ar1[i];
            i++;
        }
        else
        {
            m1 = m2;  /* Store the prev median */
            m2 = ar2[j];
            j++;
        }
    }

    return (m1 + m2)/2;
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 12, 15, 26, 38};
    int ar2[] = {2, 13, 17, 30, 45};

    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Method 2 (By comparing the medians of two arrays)
This method works by first getting medians of the two sorted arrays and then comparing them.

Let ar1 and ar2 be the input arrays.

Algorithm:

```
1) Calculate the medians m1 and m2 of the input arrays ar1[]
   and ar2[] respectively.
2) If m1 and m2 both are equal then we are done.
     return m1 (or m2)
3) If m1 is greater than m2, then median is present in one
   of the below two subarrays.
    a)  From first element of ar1 to m1 (ar1[0...|_n/2_|])
    b)  From m2 to last element of ar2  (ar2[|_n/2_|...n-1])
4) If m2 is greater than m1, then median is present in one
   of the below two subarrays.
   a)  From m1 to last element of ar1  (ar1[|_n/2_|...n-1])
   b)  From first element of ar2 to m2 (ar2[0...|_n/2_|])
5) Repeat the above process until size of both the subarrays
```

```
   becomes 2.
6) If size of the two arrays is 2 then use below formula to get
  the median.
    Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2
```

Example:

```
   ar1[] = {1, 12, 15, 26, 38}
   ar2[] = {2, 13, 17, 30, 45}
```

For above two arrays m1 = 15 and m2 = 17

For the above ar1[] and ar2[], m1 is smaller than m2. So median is present in one of the following two subarrays.

```
   [15, 26, 38] and [2, 13, 17]
```

Let us repeat the process for above two subarrays:

```
    m1 = 26 m2 = 13.
```

m1 is greater than m2. So the subarrays become

```
  [15, 26] and [13, 17]
Now size is 2, so median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2
                       = (max(15, 13) + min(26, 17))/2
                       = (15 + 17)/2
                       = 16
```

Implementation:

```
#include<stdio.h>

int max(int, int); /* to get maximum of two integers */
int min(int, int); /* to get minimum of two integeres */
int median(int [], int); /* to get median of a sorted array */

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int m1; /* For median of ar1 */
    int m2; /* For median of ar2 */

    /* return -1  for invalid input */
    if (n <= 0)
        return -1;

    if (n == 1)
        return (ar1[0] + ar2[0])/2;

    if (n == 2)
        return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;
```

```c
        return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;

    m1 = median(ar1, n); /* get the median of the first array */
    m2 = median(ar2, n); /* get the median of the second array */

    /* If medians are equal then return either m1 or m2 */
    if (m1 == m2)
        return m1;

     /* if m1 < m2 then median must exist in ar1[m1....] and ar2[....m
    if (m1 < m2)
    {
        if (n % 2 == 0)
            return getMedian(ar1 + n/2 - 1, ar2, n - n/2 +1);
        else
            return getMedian(ar1 + n/2, ar2, n - n/2);
    }

    /* if m1 > m2 then median must exist in ar1[....m1] and ar2[m2...]
    else
    {
        if (n % 2 == 0)
            return getMedian(ar2 + n/2 - 1, ar1, n - n/2 + 1);
        else
            return getMedian(ar2 + n/2, ar1, n - n/2);
    }
}

/* Function to get median of a sorted array */
int median(int arr[], int n)
{
    if (n%2 == 0)
        return (arr[n/2] + arr[n/2-1])/2;
    else
        return arr[n/2];
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 2, 3, 6};
    int ar2[] = {4, 6, 8, 10};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
      printf("Median is %d", getMedian(ar1, ar2, n1));
    else
     printf("Doesn't work for arrays of unequal size");

    getchar();
    return 0;
}

/* Utility functions */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x > y? y : x;
    return x > y? y : x;
}
```

Time Complexity: O(logn)
Algorithmic Paradigm: Divide and Conquer

Method 3 (By doing binary search for the median):
The basic idea is that if you are given two arrays ar1[] and ar2[] and know the length of each, you can check whether an element ar1[i] is the median in constant time. Suppose that the median is ar1[i]. Since the array is sorted, it is greater than exactly i values in array ar1[]. Then if it is the median, it is also greater than exactly j = n – i – 1 elements in ar2[].
It requires constant time to check if ar2[j] <= ar1[i] <= ar2[j + 1]. If ar1[i] is not the median, then depending on whether ar1[i] is greater or less than ar2[j] and ar2[j + 1], you know that ar1[i] is either greater than or less than the median. Thus you can binary search for median in O(lg n) worst-case time.

For two arrays ar1 and ar2, first do binary search in ar1[]. If you reach at the end (left or right) of the first array and don't find median, start searching in the second array ar2[].

```
1) Get the middle element of ar1[] using array indexes left and right.
   Let index of the middle element be i.
2) Calculate the corresponding index j of ar2[]
     j = n - i - 1
3) If ar1[i] >= ar2[j] and ar1[i] <= ar2[j+1] then ar1[i] and ar2[j]
   are the middle elements.
     return average of ar2[j] and ar1[i]
4) If ar1[i] is greater than both ar2[j] and ar2[j+1] then
     do binary search in left half  (i.e., arr[left ... i-1])
5) If ar1[i] is smaller than both ar2[j] and ar2[j+1] then
     do binary search in right half (i.e., arr[i+1....right])
6) If you reach at any corner of ar1[] then do binary search in ar2[]
```

Example:

```
   ar1[] = {1, 5, 7, 10, 13}
   ar2[] = {11, 15, 23, 30, 45}
```

Middle element of ar1[] is 7. Let us compare 7 with 23 and 30, since 7 smaller than both 23 and 30, move to right in ar1[]. Do binary search in {10, 13}, this step will pick 10. Now compare 10 with 15 and 23. Since 10 is smaller than both 15 and 23, again move to right. Only 13 is there in right side now. Since 13 is greater than 11 and smaller than 15, terminate here. We have got the median as 12 (average of 11 and 13)

Implementation:

```c
#include<stdio.h>

int getMedianRec(int ar1[], int ar2[], int left, int right, int n);

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    return getMedianRec(ar1, ar2, 0, n-1, n);
}

/* A recursive function to get the median of ar1[] and ar2[]
   using binary search */
int getMedianRec(int ar1[], int ar2[], int left, int right, int n)
{
    int i, j;

    /* We have reached at the end (left or right) of ar1[] */
    if (left > right)
        return getMedianRec(ar2, ar1, 0, n-1, n);

    i = (left + right)/2;
    j = n - i - 1;  /* Index of ar2[] */

    /* Recursion terminates here.*/
    if (ar1[i] > ar2[j] && (j == n-1 || ar1[i] <= ar2[j+1]))
    {
        /* ar1[i] is decided as median 2, now select the median 1
           (element just before ar1[i] in merged array) to get the
           average of both*/
        if (i == 0 || ar2[j] > ar1[i-1])
            return (ar1[i] + ar2[j])/2;
        else
            return (ar1[i] + ar1[i-1])/2;
    }

    /*Search in left half of ar1[]*/
    else if (ar1[i] > ar2[j] && j != n-1 && ar1[i] > ar2[j+1])
        return getMedianRec(ar1, ar2, left, i-1, n);

    /*Search in right half of ar1[]*/
    else /* ar1[i] is smaller than both ar2[j] and ar2[j+1]*/
        return getMedianRec(ar1, ar2, i+1, right, n);
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 12, 15, 26, 38};
    int ar2[] = {2, 13, 17, 30, 45};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");

    getchar();
    return 0;
}
```

}

Time Complexity: O(logn)
Algorithmic Paradigm: Divide and Conquer

The above solutions can be optimized for the cases when all elements of one array are smaller than all elements of other array. For example, in method 3, we can change the getMedian() function to following so that these cases can be handled in O(1) time. Thanks to nutcracker for suggesting this optimization.

```
/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
   // If all elements of array 1 are smaller then
   // median is average of last element of ar1 and
   // first element of ar2
   if (ar1[n-1] < ar2[0])
     return (ar1[n-1]+ar2[0])/2;

   // If all elements of array 1 are smaller then
   // median is average of first element of ar1 and
   // last element of ar2
   if (ar2[n-1] < ar1[0])
     return (ar2[n-1]+ar1[0])/2;

   return getMedianRec(ar1, ar2, 0, n-1, n);
}
```

References:

http://en.wikipedia.org/wiki/Median

http://ocw.alfaisal.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/30C68118-E436-4FE3-8C79-6BAFBB07D935/0/ps9sol.pdf ds3etph5wn

Asked by Snehal

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

3. Count Inversions in an array

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.
Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j

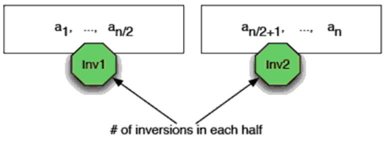
Example:
The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

METHOD 1 (Simple)
For each element, count number of elements which are on right side of it and are smaller than it.

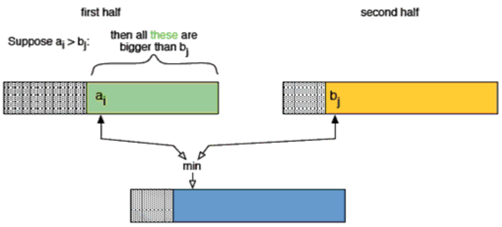
```
int getInvCount(int arr[], int n)
{
  int inv_count = 0;
  int i, j;

  for(i = 0; i < n - 1; i++)
    for(j = i+1; j < n; j++)
      if(arr[i] > arr[j])
        inv_count++;

  return inv_count;
}

/* Driver progra to test above functions */
int main(int argv, char** args)
{
  int arr[] = {1, 20, 6, 4, 5};
  printf(" Number of inversions are %d \n", getInvCount(arr, 5));
  getchar();
  return 0;
}
```

Time Complexity: O(n^2)

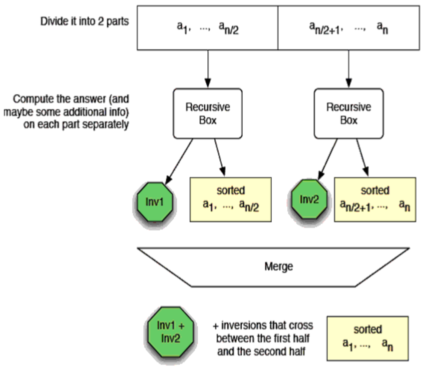
METHOD 2(Enhance Merge Sort)
Suppose we know the number of inversions in the left half and right half of the array (let be inv1 and inv2), what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().



How to get number of inversions in 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]



The complete picture:


Divide it into 2 parts
$a_1, ..., a_{n/2}$
$a_{n/2+1}, ..., a_n$
Compute the answer (and maybe some additional info) on each part separately
Recursive Box
Recursive Box
Inv1
sorted $a_1, ..., a_{n/2}$
Inv2
sorted $a_{n/2+1}, ..., a_n$
Merge
Inv1 + Inv2
+ inversions that cross between the first half and the second half
sorted $a_1, ..., a_n$

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

int  _mergeSort(int arr[], int temp[], int left, int right);
int merge(int arr[], int temp[], int left, int mid, int right);

/* This function sorts the input array and returns the
   number of inversions in the array */
int mergeSort(int arr[], int array_size)
{
    int *temp = (int *)malloc(sizeof(int)*array_size);
    return _mergeSort(arr, temp, 0, array_size - 1);
}

/* An auxiliary recursive function that sorts the input array and
  returns the number of inversions in the array. */
int _mergeSort(int arr[], int temp[], int left, int right)
{
  int mid, inv_count = 0;
  if (right > left)
  {
    /* Divide the array into two parts and call _mergeSortAndCountInv(
       for each of the parts */
    mid = (right + left)/2;

    /* Inversion count will be sum of inversions in left-part, right-p
      and number of inversions in merging */
```

```c
    inv_count  = _mergeSort(arr, temp, left, mid);
    inv_count += _mergeSort(arr, temp, mid+1, right);

    /*Merge the two parts*/
    inv_count += merge(arr, temp, left, mid+1, right);
  }
  return inv_count;
}

/* This funt merges two sorted arrays and returns inversion count in
   the arrays.*/
int merge(int arr[], int temp[], int left, int mid, int right)
{
  int i, j, k;
  int inv_count = 0;

  i = left; /* i is index for left subarray*/
  j = mid;  /* i is index for right subarray*/
  k = left; /* i is index for resultant merged subarray*/
  while ((i <= mid - 1) && (j <= right))
  {
    if (arr[i] <= arr[j])
    {
      temp[k++] = arr[i++];
    }
    else
    {
      temp[k++] = arr[j++];

     /*this is tricky -- see above explanation/diagram for merge()*/
      inv_count = inv_count + (mid - i);
    }
  }

  /* Copy the remaining elements of left subarray
   (if there are any) to temp*/
  while (i <= mid - 1)
    temp[k++] = arr[i++];

  /* Copy the remaining elements of right subarray
   (if there are any) to temp*/
  while (j <= right)
    temp[k++] = arr[j++];

  /*Copy back the merged elements to original array*/
  for (i=left; i <= right; i++)
    arr[i] = temp[i];

  return inv_count;
}

/* Driver progra to test above functions */
int main(int argv, char** args)
{
  int arr[] = {1, 20, 6, 4, 5};
  printf(" Number of inversions are %d \n", mergeSort(arr, 5));
  getchar();
  return 0;
}
```

Note that above code modifies (or sorts) the input array. If we want to count only inversions then we need to create a copy of original array and call mergeSort() on copy.

Time Complexity: O(nlogn)
Algorithmic Paradigm: Divide and Conquer

References:
http://www.cs.umd.edu/class/fall2009/cmsc451/lectures/Lec08-inversions.pdf
http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

4. Check for Majority Element in a sorted array

Question: Write a C function to find if a given integer x appears more than n/2 times in a sorted array of n integers.

Basically, we need to write a function say isMajority() that takes an array (arr[]), array's size (n) and a number to be searched (x) as parameters and returns true if x is a majority element (present more than n/2 times).

Examples:

```
Input: arr[] = {1, 2, 3, 3, 3, 3, 10}, x = 3
Output: True (x appears more than n/2 times in the given array)

Input: arr[] = {1, 1, 2, 4, 4, 4, 6, 6}, x = 4
Output: False (x doesn't appear more than n/2 times in the given array)

Input: arr[] = {1, 1, 1, 2, 2}, x = 1
Output: True (x appears more than n/2 times in the given array)
```

METHOD 1 (Using Linear Search)
Linearly search for the first occurrence of the element, once you find it (let at index i), check element at index i + n/2. If element is present at i+n/2 then return 1 else return 0.

```
/* Program to check for majority element in a sorted array */
# include <stdio.h>
# include <stdbool.h>

bool isMajority(int arr[], int n, int x)
{
  int i;

  /* get last index according to n (even or odd) */
  int last_index = n%2? (n/2+1): (n/2);

  /* search for first occurrence of x in arr[]*/
  for (i = 0; i < last_index; i++)
  {
    /* check if x is present and is present more than n/2 times */
    if (arr[i] == x && arr[i+n/2] == x)
       return 1;
  }
  return 0;
}

/* Driver program to check above function */
int main()
{
   int arr[] ={1, 2, 3, 4, 4, 4, 4};
   int n = sizeof(arr)/sizeof(arr[0]);
   int x = 4;
   if (isMajority(arr, n, x))
     printf("%d appears more than %d times in arr[]", x, n/2);
   else
    printf("%d does not appear more than %d times in arr[]", x, n/2);

   getchar();
   return 0;
}
```

Time Complexity: O(n)

METHOD 2 (Using Binary Search)

Use binary search methodology to find the first occurrence of the given number. The criteria for binary search is important here.

```
/* Program to check for majority element in a sorted array */
# include <stdio.h>;
# include <stdbool.h>

/* If x is present in arr[low...high] then returns the index of
  first occurrence of x, otherwise returns -1 */
int _binarySearch(int arr[], int low, int high, int x);

/* This function returns true if the x is present more than n/2
   times in arr[] of size n */
bool isMajority(int arr[], int n, int x)
{
   /* Find the index of first occurrence of x in arr[] */
   int i = _binarySearch(arr, 0, n-1, x);

   /* If element is not present at all, return false*/
   if (i == -1)
     return false;

   /* check if the element is present more than n/2 times */
   if (((i + n/2) <= (n -1)) && arr[i + n/2] == x)
     return true;
   else
     return false;
}

/* If x is present in arr[low...high] then returns the index of
  first occurrence of x, otherwise returns -1 */
int  _binarySearch(int arr[], int low, int high, int x)
{
  if (high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/

    /* Check if arr[mid] is the first occurrence of x.
        arr[mid] is first occurrence if x is one of the following
        is true:
        (i)  mid == 0 and arr[mid] == x
        (ii) arr[mid-1] < x and arr[mid] == x
     */
    if ( (mid == 0 || x > arr[mid-1]) && (arr[mid] == x) )
      return mid;
    else if (x > arr[mid])
      return _binarySearch(arr, (mid + 1), high, x);
    else
      return _binarySearch(arr, low, (mid -1), x);
  }

  return -1;
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 2, 3, 3, 3, 3, 10};
   int n = sizeof(arr)/sizeof(arr[0]);
   int x = 3;
   if(isMajority(arr, n, x))
     printf("%d appears more than %d times in arr[]", x, n/2);
   else
    printf("%d does not appear more than %d times in arr[]", x, n/2);

   return 0;
}
```

Time Complexity: O(Logn)

Algorithmic Paradigm: Divide and Conquer

Please write comments if you find any bug in the above program/algorithm or a better way to solve the same problem.

5. Maximum and minimum of an array using minimum number of comparisons

Write a C function to return minimum and maximum in an array. You program should make minimum number of comparisons.

First of all, how do we return multiple values from a C function? We can do it either using structures or pointers.

We have created a structure named pair (which contains min and max) to return multiple values.

```
struct pair
{
  int min;
  int max;
};
```

And the function declaration becomes: struct pair getMinMax(int arr[], int n) where arr[] is the array of size n whose minimum and maximum are needed.

METHOD 1 (Simple Linear Search)
Initialize values of min and max as minimum and maximum of the first two elements respectively. Starting from 3rd, compare each element with max and min, and change max and min accordingly (i.e., if the element is smaller than min then change min, else if the element is greater than max then change max, else ignore the element)

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
  int min;
  int max;
};

struct pair getMinMax(int arr[], int n)
{
  struct pair minmax;
  int i;

  /*If there is only one element then return it as min and max both*/
  if (n == 1)
  {
     minmax.max = arr[0];
     minmax.min = arr[0];
     return minmax;
  }

  /* If there are more than one elements, then initialize min
      and max*/
  if (arr[0] > arr[1])
  {
      minmax.max = arr[0];
      minmax.min = arr[1];
  }
  else
  {
      minmax.max = arr[1];
      minmax.min = arr[0];
  }

  for (i = 2; i<n; i++)
  {
    if (arr[i] >  minmax.max)
      minmax.max = arr[i];

    else if (arr[i] <  minmax.min)
      minmax.min = arr[i];
  }

  return minmax;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1000, 11, 445, 1, 330, 3000};
  int arr_size = 6;
  struct pair minmax = getMinMax (arr, arr_size);
  printf("\nMinimum element is %d", minmax.min);
  printf("\nMaximum element is %d", minmax.max);
  getchar();
}
```

Time Complexity: O(n)

In this method, total number of comparisons is 1 + 2(n-2) in worst case and 1 + n – 2 in

best case.
In the above implementation, worst case occurs when elements are sorted in descending order and best case occurs when elements are sorted in ascending order.

METHOD 2 (Tournament Method)
Divide the array into two parts and compare the maximums and minimums of the the two parts to get the maximum and the minimum of the the whole array.

```
Pair MaxMin(array, array_size)
   if array_size = 1
      return element as both max and min
   else if arry_size = 2
      one comparison to determine max and min
      return that pair
   else    /* array_size  > 2 */
      recur for max and min of left half
      recur for max and min of right half
      one comparison determines true max of the two candidates
      one comparison determines true min of the two candidates
      return the pair of max and min
```

Implementation

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
  int min;
  int max;
};

struct pair getMinMax(int arr[], int low, int high)
{
  struct pair minmax, mml, mmr;
  int mid;

  /* If there is only on element */
  if (low == high)
  {
     minmax.max = arr[low];
     minmax.min = arr[low];
     return minmax;
  }

  /* If there are two elements */
  if (high == low + 1)
  {
     if (arr[low] > arr[high])
     {
        minmax.max = arr[low];
        minmax.min = arr[high];
     }
```

```
    else
    {
       minmax.max = arr[high];
       minmax.min = arr[low];
    }
    return minmax;
  }

  /* If there are more than 2 elements */
  mid = (low + high)/2;
  mml = getMinMax(arr, low, mid);
  mmr = getMinMax(arr, mid+1, high);

  /* compare minimums of two parts*/
  if (mml.min < mmr.min)
    minmax.min = mml.min;
  else
    minmax.min = mmr.min;

  /* compare maximums of two parts*/
  if (mml.max > mmr.max)
    minmax.max = mml.max;
  else
    minmax.max = mmr.max;

  return minmax;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1000, 11, 445, 1, 330, 3000};
  int arr_size = 6;
  struct pair minmax = getMinMax(arr, 0, arr_size-1);
  printf("\nMinimum element is %d", minmax.min);
  printf("\nMaximum element is %d", minmax.max);
  getchar();
}
```

Time Complexity: O(n)

Total number of comparisons: let number of comparisons be T(n). T(n) can be written as follows:

Algorithmic Paradigm: Divide and Conquer

```
T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2
T(2) = 1
T(1) = 0
```

If n is a power of 2, then we can write T(n) as:

```
 T(n) = 2T(n/2) + 2
```

After solving above recursion, we get

```
T(n)  = 3/2n -2
```

Thus, the approach does 3/2n -2 comparisons if n is a power of 2. And it does more than 3/2n -2 comparisons if n is not a power of 2.

METHOD 3 (Compare in Pairs)
If n is odd then initialize min and max as first element.
If n is even then initialize min and max as minimum and maximum of the first two elements respectively.
For rest of the elements, pick them in pairs and compare their maximum and minimum with max and min respectively.

```
#include<stdio.h>

/* structure is used to return two values from minMax() */
struct pair
{
  int min;
  int max;
};

struct pair getMinMax(int arr[], int n)
{
  struct pair minmax;
  int i;

  /* If array has even number of elements then
    initialize the first two elements as minimum and
    maximum */
  if (n%2 == 0)
  {
    if (arr[0] > arr[1])
    {
      minmax.max = arr[0];
      minmax.min = arr[1];
    }
    else
    {
      minmax.min = arr[0];
      minmax.max = arr[1];
    }
    i = 2;  /* set the startung index for loop */
  }

   /* If array has odd number of elements then
    initialize the first element as minimum and
    maximum */
  else
  {
    minmax.min = arr[0];
    minmax.max = arr[0];
    i = 1;  /* set the startung index for loop */
  }

  /* In the while loop, pick elements in pair and
     compare the pair with max and min so far */
  while (i < n-1)
  {
    if (arr[i] > arr[i+1])
    {
      if(arr[i] > minmax.max)
        minmax.max = arr[i];
      if(arr[i+1] < minmax.min)
        minmax.min = arr[i+1];
    }
    else
    {
      if (arr[i+1] > minmax.max)
        minmax.max = arr[i+1];
      if (arr[i] < minmax.min)
        minmax.min = arr[i];
    }
    i += 2; /* Increment the index by 2 as two
               elements are processed in loop */
  }

  return minmax;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1000, 11, 445, 1, 330, 3000};
  int arr_size = 6;
  struct pair minmax = getMinMax (arr, arr_size);
  printf("\nMinimum element is %d", minmax.min);
  printf("\nMaximum element is %d", minmax.max);
  getchar();
}
```

Time Complexity: O(n)

Total number of comparisons: Different for even and odd n, see below:

```
       If n is odd:    3*(n-1)/2
       If n is even:   1 Initial comparison for initializing min and max,
                           and 3(n-2)/2 comparisons for rest of the elements
                      =  1 + 3*(n-2)/2 = 3n/2 -2
```

Second and third approaches make equal number of comparisons when n is a power of 2.

In general, method 3 seems to be the best.

Please write comments if you find any bug in the above programs/algorithms or a better way to solve the same problem.

6. Program to count number of set bits in an (big) array

Given an integer array of length N (an arbitrarily large number). How to count number of set bits in the array?

The simple approach would be, create an efficient method to count set bits in a word (most prominent size, usually equal to bit length of processor), and add bits from individual elements of array.

Various methods of counting set bits of an integer exists, see this for example. These methods run at best O(logN) where N is number of bits. Note that on a processor N is fixed, count can be done in O(1) time on 32 bit machine irrespective of total set bits. Overall, the bits in array can be computed in O(n) time, where ‘n’ is array size.

However, a table look up will be more efficient method when array size is large. Storing table look up that can handle 2^{32} integers will be impractical.

The following code illustrates simple program to count set bits in a randomly generated 64 K integer array. The idea is to generate a look up for first 256 numbers (one byte), and break every element of array at byte boundary. A meta program using C/C++ preprocessor generates the look up table for counting set bits in a byte.

The mathematical derivation behind meta program is evident from the following table (Add the column and row indices to get the number, then look into the table to get set bits in that number. For example, to get set bits in 10, it can be extracted from row named as 8 and column named as 2),

```
   0, 1, 2, 3
 0 - 0, 1, 1, 2 -------- GROUP_A(0)
 4 - 1, 2, 2, 3 -------- GROUP_A(1)
 8 - 1, 2, 2, 3 -------- GROUP_A(1)
12 - 2, 3, 3, 4 -------- GROUP_A(2)
16 - 1, 2, 2, 3 -------- GROUP_A(1)
20 - 2, 3, 3, 4 -------- GROUP_A(2)
24 - 2, 3, 3, 4 -------- GROUP_A(2)
28 - 3, 4, 4, 5 -------- GROUP_A(3) ... so on
```

From the table, there is a patten emerging in multiples of 4, both in the table as well as in the group parameter. The sequence can be generalized as shown in the code.

Complexity:

All the operations takes O(1) except iterating over the array. The time complexity is O(n) where ‘n’ is size of array. Space complexity depends on the meta program that generates look up.

Code:

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <time.h>

/* Size of array 64 K */
#define SIZE (1 << 16)

/* Meta program that generates set bit count
   array of first 256 integers */

/* GROUP_A - When combined with META_LOOK_UP
   generates count for 4x4 elements */

#define GROUP_A(x) x, x + 1, x + 1, x + 2

/* GROUP_B - When combined with META_LOOK_UP
   generates count for 4x4x4 elements */

#define GROUP_B(x) GROUP_A(x), GROUP_A(x+1), GROUP_A(x+1), GROUP_A(x+2

/* GROUP_C - When combined with META_LOOK_UP
   generates count for 4x4x4x4 elements */

#define GROUP_C(x) GROUP_B(x), GROUP_B(x+1), GROUP_B(x+1), GROUP_B(x+2

/* Provide appropriate letter to generate the table */

#define META_LOOK_UP(PARAMETER) \
   GROUP_##PARAMETER(0),  \
   GROUP_##PARAMETER(1),  \
   GROUP_##PARAMETER(1),  \
   GROUP_##PARAMETER(2)   \

int countSetBits(int array[], size_t array_size)
{
   int count = 0;

   /* META_LOOK_UP(C) - generates a table of 256 integers whose
      sequence will be number of bits in i-th position
      where 0 <= i < 256
   */

    /* A static table will be much faster to access */
       static unsigned char const look_up[] = { META_LOOK_UP(C) };

    /* No shifting funda (for better readability) */
    unsigned char *pData = NULL;

   for(size_t index = 0; index < array_size; index++)
   {
      /* It is fine, bypass the type system */
      pData = (unsigned char *)&array[index];

      /* Count set bits in individual bytes */
      count += look_up[pData[0]];
      count += look_up[pData[1]];
      count += look_up[pData[2]];
      count += look_up[pData[3]];
   }

   return count;
}
```

```
/* Driver program, generates table of random 64 K numbers */
int main()
{
   int index;
   int random[SIZE];

   /* Seed to the random-number generator */
   srand((unsigned)time(0));

   /* Generate random numbers. */
   for( index = 0; index < SIZE; index++ )
   {
      random[index] = rand();
   }

   printf("Total number of bits = %d\n", countSetBits(random, SIZE));
   return 0;
}
```

Contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given an integer array of length N (an arbitrarily large number). How to count number of set bits in the array?

The simple approach would be, create an efficient method to count set bits in a word (most prominent size, usually equal to bit length of processor), and add bits from individual elements of array.

Various methods of counting set bits of an integer exists, see this for example. These methods run at best O(logN) where N is number of bits. Note that on a processor N is fixed, count can be done in O(1) time on 32 bit machine irrespective of total set bits. Overall, the bits in array can be computed in O(n) time, where 'n' is array size.

However, a table look up will be more efficient method when array size is large. Storing table look up that can handle 2^{32} integers will be impractical.

The following code illustrates simple program to count set bits in a randomly generated 64 K integer array. The idea is to generate a look up for first 256 numbers (one byte), and break every element of array at byte boundary. A meta program using C/C++ preprocessor generates the look up table for counting set bits in a byte.

The mathematical derivation behind meta program is evident from the following table (Add the column and row indices to get the number, then look into the table to get set bits in that number. For example, to get set bits in 10, it can be extracted from row named as 8 and column named as 2),

```
   0, 1, 2, 3
 0 - 0, 1, 1, 2 -------- GROUP_A(0)
 4 - 1, 2, 2, 3 -------- GROUP_A(1)
 8 - 1, 2, 2, 3 -------- GROUP_A(1)
12 - 2, 3, 3, 4 -------- GROUP_A(2)
16 - 1, 2, 2, 3 -------- GROUP_A(1)
20 - 2, 3, 3, 4 -------- GROUP_A(2)
24 - 2, 3, 3, 4 -------- GROUP_A(2)
28 - 3, 4, 4, 5 -------- GROUP_A(3) ... so on
```

From the table, there is a patten emerging in multiples of 4, both in the table as well as in the group parameter. The sequence can be generalized as shown in the code.

Complexity:

All the operations takes O(1) except iterating over the array. The time complexity is O(n) where 'n' is size of array. Space complexity depends on the meta program that generates look up.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Size of array 64 K */
#define SIZE (1 << 16)

/* Meta program that generates set bit count
   array of first 256 integers */

/* GROUP_A - When combined with META_LOOK_UP
   generates count for 4x4 elements */

#define GROUP_A(x) x, x + 1, x + 1, x + 2

/* GROUP_B - When combined with META_LOOK_UP
   generates count for 4x4x4 elements */

#define GROUP_B(x) GROUP_A(x), GROUP_A(x+1), GROUP_A(x+1), GROUP_A(x+2

/* GROUP_C - When combined with META_LOOK_UP
   generates count for 4x4x4x4 elements */

#define GROUP_C(x) GROUP_B(x), GROUP_B(x+1), GROUP_B(x+1), GROUP_B(x+2

/* Provide appropriate letter to generate the table */

#define META_LOOK_UP(PARAMETER) \
   GROUP_##PARAMETER(0),   \
   GROUP_##PARAMETER(1),   \
   GROUP_##PARAMETER(1),   \
   GROUP_##PARAMETER(2)    \

int countSetBits(int array[], size_t array_size)
```

```
{
   int count = 0;

   /* META_LOOK_UP(C) - generates a table of 256 integers whose
      sequence will be number of bits in i-th position
      where 0 <= i < 256
   */

    /* A static table will be much faster to access */
       static unsigned char const look_up[] = { META_LOOK_UP(C) };

    /* No shifting funda (for better readability) */
    unsigned char *pData = NULL;

   for(size_t index = 0; index < array_size; index++)
   {
      /* It is fine, bypass the type system */
      pData = (unsigned char *)&array[index];

      /* Count set bits in individual bytes */
      count += look_up[pData[0]];
      count += look_up[pData[1]];
      count += look_up[pData[2]];
      count += look_up[pData[3]];
   }

   return count;
}

/* Driver program, generates table of random 64 K numbers */
int main()
{
   int index;
   int random[SIZE];

   /* Seed to the random-number generator */
   srand((unsigned)time(0));

   /* Generate random numbers. */
   for( index = 0; index < SIZE; index++ )
   {
      random[index] = rand();
   }

   printf("Total number of bits = %d\n", countSetBits(random, SIZE));
   return 0;
}
```

Contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

8. Find a Fixed Point in a given array

Given an array of n distinct integers sorted in ascending order, write a function that returns a Fixed Point in the array, if there is any Fixed Point present in array, else returns -1. Fixed Point in an array is an index i such that arr[i] is equal to i. Note that integers in array can be negative.

Examples:

```
Input: arr[] = {-10, -5, 0, 3, 7}
Output: 3  // arr[3] == 3

Input: arr[] = {0, 2, 5, 8, 17}
Output: 0  // arr[0] == 0



Input: arr[] = {-10, -5, 3, 4, 7, 9}
Output: -1  // No Fixed Point
```

Asked by rajk

Method 1 (Linear Search)
Linearly search for an index i such that arr[i] == i. Return the first such index found. Thanks to pm for suggesting this solution.

```
int linearSearch(int arr[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(arr[i] == i)
            return i;
    }

    /* If no fixed point present then return -1 */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Fixed Point is %d", linearSearch(arr, n));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Method 2 (Binary Search)
First check whether middle element is Fixed Point or not. If it is, then return it; otherwise

check whether index of middle element is greater than value at the index. If index is greater, then Fixed Point(s) lies on the right side of the middle point (obviously only if there is a Fixed Point). Else the Fixed Point(s) lies on left side.

```c
int binarySearch(int arr[], int low, int high)
{
    if(high >= low)
    {
        int mid = (low + high)/2;  /*low + (high - low)/2;*/
        if(mid == arr[mid])
            return mid;
        if(mid > arr[mid])
            return binarySearch(arr, (mid + 1), high);
        else
            return binarySearch(arr, low, (mid -1));
    }

    /* Return -1 if there is no Fixed Point */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[10] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Fixed Point is %d", binarySearch(arr, 0, n-1));
    getchar();
    return 0;
}
```

Algorithmic Paradigm: Divide & Conquer
Time Complexity: O(Logn)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

9. Find the maximum element in an array which is first increasing and then decreasing

Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.

```
Input: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}
Output: 500


Input: arr[] = {1, 3, 50, 10, 9, 7, 6}
Output: 50
```

```
Corner case (No decreasing part)
Input: arr[] = {10, 20, 30, 40, 50}
Output: 50

Corner case (No increasing part)
Input: arr[] = {120, 100, 80, 20, 0}
Output: 120
```

Method 1 (Linear Search)
We can traverse the array and keep track of maximum and element. And finally return the maximum element.

```c
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{
   int max = arr[low];
   int i;
   for (i = low; i <= high; i++)
   {
       if (arr[i] > max)
          max = arr[i];
   }
   return max;
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 30, 40, 50, 60, 70, 23, 20};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The maximum element is %d", findMaximum(arr, 0, n-1));
   getchar();
   return 0;
}
```

Time Complexity: O(n)

Method 2 (Binary Search)
We can modify the standard Binary Search algorithm for the given type of arrays.
i) If the mid element is greater than both of its adjacent elements, then mid is the maximum.
ii) If mid element is greater than its next element and smaller than the previous element then maximum lies on left side of mid. Example array: {3, 50, 10, 9, 7, 6}
iii) If mid element is smaller than its next element and greater than the previous element then maximum lies on right side of mid. Example array: {2, 4, 6, 8, 10, 3, 1}

```c
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{

   /* Base Case: Only one element is present in arr[low..high]*/
   if (low == high)
     return arr[low];

   /* If there are two elements and first is greater then
      the first element is maximum */
   if ((high == low + 1) && arr[low] >= arr[high])
      return arr[low];

   /* If there are two elements and second is greater then
      the second element is maximum */
   if ((high == low + 1) && arr[low] < arr[high])
      return arr[high];

   int mid = (low + high)/2;   /*low + (high - low)/2;*/

   /* If we reach a point where arr[mid] is greater than both of
     its adjacent elements arr[mid-1] and arr[mid+1], then arr[mid]
     is the maximum element*/
   if ( arr[mid] > arr[mid + 1] && arr[mid] > arr[mid - 1])
      return arr[mid];

   /* If arr[mid] is greater than the next element and smaller than th
    element then maximum lies on left side of mid */
   if (arr[mid] > arr[mid + 1] && arr[mid] < arr[mid - 1])
     return findMaximum(arr, low, mid-1);
   else // when arr[mid] is greater than arr[mid-1] and smaller than a
     return findMaximum(arr, mid + 1, high);
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 3, 50, 10, 9, 7, 6};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The maximum element is %d", findMaximum(arr, 0, n-1));
   getchar();
   return 0;
}
```

Time Complexity: O(Logn)

This method works only for distinct numbers. For example, it will not work for an array like {0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 5, 3, 3, 2, 2, 1, 1}.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

10. Median of two sorted arrays of different sizes

This is an extension of median of two sorted arrays of equal size problem. Here we handle arrays of unequal size also.

The approach discussed in this post is similar to method 2 of equal size post. The basic idea is same, we find the median of two arrays and compare the medians to discard almost half of the elements in both arrays. Since the number of elements may differ here, there are many base cases that need to be handled separately. Before we proceed to complete solution, let us first talk about all base cases.

Let the two arrays be A[N] and B[M]. In the following explanation, it is assumed that N is smaller than or equal to M.

Base cases:
The smaller array has only one element
Case 1: N = 1, M = 1.
Case 2: N = 1, M is odd
Case 3: N = 1, M is even
The smaller array has only two elements
Case 4: N = 2, M = 2
Case 5: N = 2, M is odd
Case 6: N = 2, M is even

Case 1: There is only one element in both arrays, so output the average of A[0] and B[0].

Case 2: N = 1, M is odd
Let B[5] = {5, 10, 12, 15, 20}
First find the middle element of B[], which is 12 for above array. There are following 4 sub-cases.
…**2.1** If A[0] is smaller than 10, the median is average of 10 and 12.
…**2.2** If A[0] lies between 10 and 12, the median is average of A[0] and 12.
…**2.3** If A[0] lies between 12 and 15, the median is average of 12 and A[0].
…**2.4** If A[0] is greater than 15, the median is average of 12 and 15.
In all the sub-cases, we find that 12 is fixed. So, we need to find the median of B[M / 2 – 1], B[M / 2 + 1], A[0] and take its average with B[M / 2].

Case 3: N = 1, M is even
Let B[4] = {5, 10, 12, 15}
First find the middle items in B[], which are 10 and 12 in above example. There are following 3 sub-cases.
…**3.1** If A[0] is smaller than 10, the median is 10.
…**3.2** If A[0] lies between 10 and 12, the median is A[0].
…**3.3** If A[0] is greater than 10, the median is 12.

So, in this case, find the median of three elements B[M / 2 – 1], B[M / 2] and A[0].

Case 4: N = 2, M = 2
There are four elements in total. So we find the median of 4 elements.

Case 5: N = 2, M is odd
Let B[5] = {5, 10, 12, 15, 20}
The median is given by median of following three elements: B[M/2], max(A[0], B[M/2 – 1]), min(A[1], B[M/2 + 1]).

Case 6: N = 2, M is even
Let B[4] = {5, 10, 12, 15}
The median is given by median of following four elements: B[M/2], B[M/2 – 1], max(A[0], B[M/2 – 2]), min(A[1], B[M/2 + 1])

Remaining Cases:
Once we have handled the above base cases, following is the remaining process.
1) Find the middle item of A[] and middle item of B[].
…..**1.1)** If the middle item of A[] is greater than middle item of B[], ignore the last half of A[], let length of ignored part is idx. Also, cut down B[] by idx from the start.
…..**1.2)** else, ignore the first half of A[], let length of ignored part is idx. Also, cut down B[] by idx from the last.

Following is C implementation of the above approach.

```c
// A C program to find median of two sorted arrays of unequal size
#include <stdio.h>
#include <stdlib.h>

// A utility function to find maximum of two integers
int max( int a, int b )
{ return a > b ? a : b; }

// A utility function to find minimum of two integers
int min( int a, int b )
{ return a < b ? a : b; }

// A utility function to find median of two integers
float MO2( int a, int b )
{ return ( a + b ) / 2.0; }

// A utility function to find median of three integers
float MO3( int a, int b, int c )
{
    return a + b + c - max( a, max( b, c ) )
                     - min( a, min( b, c ) );
}

// A utility function to find median of four integers
float MO4( int a, int b, int c, int d )
{
    int Max = max( a, max( b, max( c, d ) ) );
    int Min = min( a, min( b, min( c, d ) ) );
    return ( a + b + c + d - Max - Min ) / 2.0;
}

// This function assumes that N is smaller than or equal to M
float findMedianUtil( int A[], int N, int B[], int M )
{
    // If the smaller array has only one element
    if( N == 1 )
    {
        // Case 1: If the larger array also has one element, simply ca
        if( M == 1 )
            return MO2( A[0], B[0] );

        // Case 2: If the larger array has odd number of elements, the
        // the middle 3 elements of larger array and the only element
        // smaller array. Take few examples like following
        // A = {9}, B[] = {5, 8, 10, 20, 30} and
        // A[] = {1}, B[] = {5, 8, 10, 20, 30}
        if( M & 1 )
            return MO2( B[M/2], MO3(A[0], B[M/2 - 1], B[M/2 + 1]) );

        // Case 3: If the larger array has even number of element, the
        // will be one of the following 3 elements
        // ... The middle two elements of larger array
        // ... The only element of smaller array
        return MO3( B[M/2], B[M/2 - 1], A[0] );
    }

    // If the smaller array has two elements
    else if( N == 2 )
    {
        // Case 4: If the larger array also has two elements, simply c
        if( M == 2 )
            return MO4( A[0], A[1], B[0], B[1] );

        // Case 5: If the larger array has odd number of elements, the
        // will be one of the following 3 elements
        // 1. Middle element of larger array
        // 2. Max of first element of smaller array and element just
        //    before the middle in bigger array
        // 3. Min of second element of smaller array and element just
        //    after the middle in bigger array
        if( M & 1 )
            return MO3 ( B[M/2],
                         max( A[0], B[M/2 - 1] ),
                         min( A[1], B[M/2 + 1] )
                       );

        // Case 6: If the larger array has even number of elements, th
        // median will be one of the following 4 elements
        // 1) & 2) The middle two elements of larger array
        // 3) Max of first element of smaller array and element
        //    just before the first middle element in bigger array
        // 4. Min of second element of smaller array and element
        //    just after the second middle in bigger array
        return MO4 ( B[M/2],
                     B[M/2 - 1],
                     max( A[0], B[M/2 - 2] ),
                     min( A[1], B[M/2 + 1] )
                   );
    }
    int idxA = ( N - 1 ) / 2;
    int idxB = ( M - 1 ) / 2;

     /* if A[idxA] <= B[idxB], then median must exist in
        A[idxA....] and B[....idxB] */
    if( A[idxA] <= B[idxB] )
        return findMedianUtil( A + idxA, N / 2 + 1, B, M - idxA );

    /* if A[idxA] > B[idxB], then median must exist in
       A[...idxA] and B[idxB....] */
    return findMedianUtil( A, N / 2 + 1, B + idxA, M - idxA );
}

// A wrapper function around findMedianUtil(). This function makes
// sure that smaller array is passed as first argument to findMedianUt
float findMedian( int A[], int N, int B[], int M )
{
    if ( N > M )
       return findMedianUtil( B, M, A, N );

    return findMedianUtil( A, N, B, M );
}

// Driver program to test above functions
int main()
{
    int A[] = {900};
    int B[] = {5, 8, 10, 20};

    int N = sizeof(A) / sizeof(A[0]);
    int M = sizeof(B) / sizeof(B[0]);

    printf( "%f", findMedian( A, N, B, M ) );
    return 0;
}
```

Output:

```
10
```

Time Complexity: O(LogM + LogN)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

11. Divide and Conquer | Set 1 (Introduction)

Like Greedy and Dynamic Programming, Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide:* Break the given problem into subproblems of same type.
2. *Conquer:* Recursively solve these subproblems
3. *Combine:* Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

1) Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5) Strassen's Algorithm is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.

6) Cooley–Tukey Fast Fourier Transform (FFT) algorithm is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(nlogn) time.

7) Karatsuba algorithm for fast multiplication it does multiplication of two *n*-digit numbers in at most single-digit multiplications in general (and exactly when *n* is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59{,}049$ and $(2^{10})^2 = 1{,}048{,}576$, respectively.

We will publishing above algorithms in separate posts.

Divide and Conquer (D & C) vs Dynamic Programming (DP)
Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How to choose one of them for a given problem? Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used. For example, Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for calculating nth Fibonacci number, Dynamic Programming should be

preferred (See this for details).

References

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.
http://en.wikipedia.org/wiki/Karatsuba_algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

12. Divide and Conquer | Set 2 (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.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy. In this post, a O(n x (Logn)^2) approach is discussed. We will be discussing a O(nLogn) approach in a separate post.

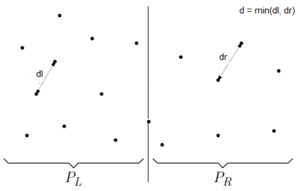
Algorithm
Following are the detailed steps of a O(n (Logn)^2) algortihm.
Input: An array of n points *P[]*
Output: The smallest distance between two points in the given array.

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

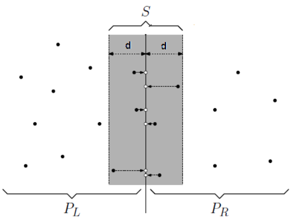
1) Find the middle point in the sorted array, we can take *P[n/2]* as middle point.

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].

3) 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.



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.



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(n^2) 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)

Implementation

Following is C/C++ implementation of the above algorithm.

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

```c
// 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 po
// 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 beween the closest points o
// strip of given size. All points in strip[] are sorted accordint 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 dif
    // 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;
            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 con
// 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
    // 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
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main functin 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 dista
    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;
}
```

Output:

```
The smallest distance is 1.414214
```

Time Complexity Let Time complexity of above algorithm be T(n). Let us assume that

we use a O(nLogn) sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows
T(n) = 2T(n/2) + O(n) + O(nLogn) + O(n)
T(n) = 2T(n/2) + O(nLogn)
T(n) = T(n x Logn x Logn)

Notes
1) Time complexity can be improved to O(nLogn) by optimizing step 5 of the above algorithm. We will soon be discussing the optimized solution in a separate post.
2) The code finds smallest distance. It can be easily modified to find the points with smallest distance.
3) The code uses quick sort which can be O(n^2) in worst case. To have the upper bound as O(n (Logn)^2), a O(nLogn) sorting algorithm like merge sort or heap sort can be used

References:
http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf
http://www.youtube.com/watch?v=vS4Zn1a9KUc
http://www.youtube.com/watch?v=T3T7T8Ym20M
http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

13. Divide and Conquer | Set 3 (Maximum Subarray Sum)

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum. The time complexity of the Naive method is O(n^2).

Using **Divide and Conquer** approach, we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.

1) Divide the given array in two halves
2) Return the maximum of following three
….**a)** Maximum subarray sum in left half (Make a recursive call)

….**b)** Maximum subarray sum in right half (Make a recursive call)
….**c)** Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.

```
// A Divide and Conquer based program for maximum subarray sum problem
#include <stdio.h>
#include <limits.h>

// A utility funtion to find maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// A utility funtion to find maximum of three integers
int max(int a, int b, int c) { return max(max(a, b), c); }

// Find the maximum possible sum in arr[] auch that arr[m] is part of
int maxCrossingSum(int arr[], int l, int m, int h)
{
    // Include elements on left of mid.
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--)
    {
        sum = sum + arr[i];
        if (sum > left_sum)
          left_sum = sum;
    }

    // Include elements on right of mid
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m+1; i <= h; i++)
    {
        sum = sum + arr[i];
        if (sum > right_sum)
          right_sum = sum;
    }

    // Return sum of elements on left and right of mid
    return left_sum + right_sum;
}

// Returns sum of maxium sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
{
   // Base Case: Only one element
   if (l == h)
     return arr[l];

   // Find middle point
   int m = (l + h)/2;

   /* Return maximum of following three possible cases
      a) Maximum subarray sum in left half
      b) Maximum subarray sum in right half
      b) Maximum subarray sum in right half
      c) Maximum subarray sum such that the subarray crosses the midpo
   return max(maxSubArraySum(arr, l, m),
              maxSubArraySum(arr, m+1, h),
              maxCrossingSum(arr, l, m, h));
}

/*Driver program to test maxSubArraySum*/
int main()
{
   int arr[] = {2, 3, 4, 5, 7};
   int n = sizeof(arr)/sizeof(arr[0]);
   int max_sum = maxSubArraySum(arr, 0, n-1);
   printf("Maximum contiguous sum is %d\n", max_sum);
   getchar();
   return 0;
}
```

Time Complexity: maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation.
T(n) = 2T(n/2) + $\Theta(n)$
The above recurrence is similar to Merge Sort and can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(nLogn)$.

The Kadane's Algorithm for this problem takes O(n) time. Therefore the Kadane's algorithm is better than the Divide and Conquer approach, but this problem can be considered as a good example to show power of Divide and Conquer. The above simple approach where we divide the array in two halves, reduces the time complexity from O(n^2) to O(nLogn).

References:
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum.

The time complexity of the Naive method is O(n^2).

Using **Divide and Conquer** approach, we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.

1) Divide the given array in two halves
2) Return the maximum of following three
….**a)** Maximum subarray sum in left half (Make a recursive call)
….**b)** Maximum subarray sum in right half (Make a recursive call)
….**c)** Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.

```c
// A Divide and Conquer based program for maximum subarray sum problem
#include <stdio.h>
#include <limits.h>

// A utility funtion to find maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// A utility funtion to find maximum of three integers
int max(int a, int b, int c) { return max(max(a, b), c); }

// Find the maximum possible sum in arr[] auch that arr[m] is part of
int maxCrossingSum(int arr[], int l, int m, int h)
{
    // Include elements on left of mid.
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--)
    {
        sum = sum + arr[i];
        if (sum > left_sum)
          left_sum = sum;
    }

    // Include elements on right of mid
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m+1; i <= h; i++)
    {
        sum = sum + arr[i];
        if (sum > right_sum)
          right_sum = sum;
    }

    // Return sum of elements on left and right of mid
    return left_sum + right_sum;
}

// Returns sum of maxium sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
```

```
{
   // Base Case: Only one element
   if (l == h)
     return arr[l];

   // Find middle point
   int m = (l + h)/2;

   /* Return maximum of following three possible cases
      a) Maximum subarray sum in left half
      b) Maximum subarray sum in right half
      c) Maximum subarray sum such that the subarray crosses the midpo
   return max(maxSubArraySum(arr, l, m),
              maxSubArraySum(arr, m+1, h),
              maxCrossingSum(arr, l, m, h));
}

/*Driver program to test maxSubArraySum*/
int main()
{
   int arr[] = {2, 3, 4, 5, 7};
   int n = sizeof(arr)/sizeof(arr[0]);
   int max_sum = maxSubArraySum(arr, 0, n-1);
   printf("Maximum contiguous sum is %d\n", max_sum);
   getchar();
   return 0;
}
```

Time Complexity: maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation.
T(n) = 2T(n/2) + $\Theta(n)$
The above recurrence is similar to Merge Sort and can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(nLogn)$.

The Kadane's Algorithm for this problem takes O(n) time. Therefore the Kadane's algorithm is better than the Divide and Conquer approach, but this problem can be considered as a good example to show power of Divide and Conquer. The above simple approach where we divide the array in two halves, reduces the time complexity from O(n^2) to O(nLogn).

References:
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

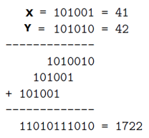
15. Divide and Conquer | Set 4 (Karatsuba algorithm for fast

multiplication)

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is “1100” and second bit string is “1010”, output should be 120.

For simplicity, let the length of two strings be same and be n.

A **Naive Approach** is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes O(n^2) time.



```
   X = 101001 = 41
   Y = 101010 = 42
-------------
      1010010
    101001
+ 101001
-------------
  11010111010 = 1722
```

Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y.

For simplicity let us assume that n is even

$X = Xl*2^{n/2} + Xr$ [Xl and Xr contain leftmost and rightmost n/2 bits of X]

$Y = Yl*2^{n/2} + Yr$ [Yl and Yr contain leftmost and rightmost n/2 bits of Y]

The product XY can be written as following.

$XY = (Xl*2^{n/2} + Xr)(Yl*2^{n/2} + Yr)$

$\quad = 2^{n} XlYl + 2^{n/2}(XlYr + XrYl) + XrYr$

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into for sub-problems of size n/2. But that doesn’t help because solution of recurrence T(n) = 4T(n/2) + O(n) is O(n^2). The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

```
XlYr + XrYl = (Xl + Xr)(Yl + Yr) - XlYl- XrYr
```

So the final value of XY becomes

$XY = 2^{n} XlYl + 2^{n/2} * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr$

With above trick, the recurrence becomes T(n) = 3T(n/2) + O(n) and solution of this recurrence is $O(n^{1.59})$.

What if the lengths of input strings are different and are not even? To handle the different length case, we append 0's in the beginning. To handle odd length, we put *floor(n/2)* bits in left half and *ceil(n/2)* bits in right half. So the expression for XY changes to following.

$$XY = 2^{2ceil(n/2)}\ XlYl + 2^{ceil(n/2)} * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr$$

The above algorithm is called Karatsuba algorithm and it can be used for any base.

Following is C++ implementation of above algorithm.

```
// C++ implementation of Karatsuba algorithm for bit string multiplica
#include<iostream>
#include<stdio.h>

using namespace std;

// FOLLOWING TWO FUNCTIONS ARE COPIED FROM http://goo.gl/q0OhZ
// Helper method: given two unequal sized bit strings, converts them t
// same length by adding leading 0s in the smaller string. Returns the
// the new length
int makeEqualLength(string &str1, string &str2)
{
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2)
    {
        for (int i = 0 ; i < len2 - len1 ; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2)
    {
        for (int i = 0 ; i < len1 - len2 ; i++)
            str2 = '0' + str2;
    }
    return len1; // If len1 >= len2
}

// The main function that adds two bit sequences and returns the addit
string addBitStrings( string first, string second )
{
    string result;  // To store the sum bits

    // make the lengths same before adding
    int length = makeEqualLength(first, second);
    int carry = 0;  // Initialize carry

    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--)
    {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (char)sum + result;
        // boolean expression for 3-bit addition
        carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&c
    }

    // if overflow, then add a leading 1
    if (carry)  result = '1' + result;

    return result;
}

// A utility function to multiply single bits of strings a and b
int multiplyiSingleBit(string a, string b)
{  return (a[0] - '0')*(b[0] - '0');  }

// The main function that multiplies two bit strings X and Y and retur
// result as long integer
long int multiply(string X, string Y)
{
    // Find the maximum of lengths of x and Y and make length
    // of smaller string same as that of larger string
    int n = makeEqualLength(X, Y);

    // Base cases
    if (n == 0) return 0;
    if (n == 1) return multiplyiSingleBit(X, Y);

    int fh = n/2;   // First half of string, floor(n/2)
    int sh = (n-fh); // Second half of string, ceil(n/2)

    // Find the first half and second half of first string.
    // Refer http://goo.gl/lLmgn for substr method
    string Xl = X.substr(0, fh);
    string Xr = X.substr(fh, sh);

    // Find the first half and second half of second string
    string Yl = Y.substr(0, fh);
    string Yr = Y.substr(fh, sh);

    // Recursively calculate the three products of inputs of size n/2
    long int P1 = multiply(Xl, Yl);
    long int P2 = multiply(Xr, Yr);
    long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr

    // Combine the three products to get the final result.
    return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}

// Driver program to test aboev functions
int main()
{
    printf ("%ld\n", multiply("1100", "1010"));
    printf ("%ld\n", multiply("110", "1010"));
    printf ("%ld\n", multiply("11", "1010"));
    printf ("%ld\n", multiply("1", "1010"));
    printf ("%ld\n", multiply("0", "1010"));
    printf ("%ld\n", multiply("111", "111"));
    printf ("%ld\n", multiply("11", "11"));
}
```

Output:

```
120
60
30
10
0
49
9
```

Time Complexity: Time complexity of the above solution is O($n^{1.59}$).

Time complexity of multiplication can be further improved using another Divide and Conquer algorithm, fast Fourier transform. We will soon be discussing fast Fourier transform as a separate post.

Exercise
The above program returns a long int value and will not work for big strings. Extend the above program to return a string instead of a long int value.

References:
Wikipedia page for Karatsuba algorithm
Algorithms 1st Edition by Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani
http://courses.csail.mit.edu/6.006/spring11/exams/notes3-karatsuba
http://www.cc.gatech.edu/~ninamf/Algos11/lectures/lect0131.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is “1100” and second bit string is “1010”, output should be 120.

For simplicity, let the length of two strings be same and be n.

A **Naive Approach** is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes O(n^2) time.

```
    X = 101001 = 41
    Y = 101010 = 42
 -------------
       1010010
     101001
 + 101001
 -------------
   11010111010 = 1722
```

Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y.

For simplicity let us assume that n is even

$X = Xl*2^{n/2} + Xr$ [Xl and Xr contain leftmost and rightmost n/2 bits of X]

$Y = Yl*2^{n/2} + Yr$ [Yl and Yr contain leftmost and rightmost n/2 bits of Y]

The product XY can be written as following.

$XY = (Xl*2^{n/2} + Xr)(Yl*2^{n/2} + Yr)$

$\quad = 2^{n} XlYl + 2^{n/2}(XlYr + XrYl) + XrYr$

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into for sub-problems of size n/2. But that doesn’t help because solution of recurrence T(n) = 4T(n/2) + O(n) is O(n^2). The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

```
XlYr + XrYl = (Xl + Xr)(Yl + Yr) - XlYl- XrYr
```

So the final value of XY becomes

$XY = 2^{n} XlYl + 2^{n/2} * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr$

With above trick, the recurrence becomes T(n) = 3T(n/2) + O(n) and solution of this recurrence is $O(n^{1.59})$.

What if the lengths of input strings are different and are not even? To handle the different length case, we append 0’s in the beginning. To handle odd length, we put *floor(n/2)* bits in left half and *ceil(n/2)* bits in right half. So the expression for XY changes to following.

$XY = 2^{2ceil(n/2)} XlYl + 2^{ceil(n/2)} * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr$

The above algorithm is called Karatsuba algorithm and it can be used for any base.

Following is C++ implementation of above algorithm.

```
// C++ implementation of Karatsuba algorithm for bit string multiplica
#include<iostream>
#include<stdio.h>

using namespace std;

// FOLLOWING TWO FUNCTIONS ARE COPIED FROM http://goo.gl/q0OhZ
// Helper method: given two unequal sized bit strings, converts them t
// same length by adding leading 0s in the smaller string. Returns the
// the new length
```

```
int makeEqualLength(string &str1, string &str2)
{
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2)
    {
        for (int i = 0 ; i < len2 - len1 ; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2)
    {
        for (int i = 0 ; i < len1 - len2 ; i++)
            str2 = '0' + str2;
    }
    return len1; // If len1 >= len2
}

// The main function that adds two bit sequences and returns the addit
string addBitStrings( string first, string second )
{
    string result;  // To store the sum bits

    // make the lengths same before adding
    int length = makeEqualLength(first, second);
    int carry = 0;  // Initialize carry

    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--)
    {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (char)sum + result;

        // boolean expression for 3-bit addition
        carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&c
    }

    // if overflow, then add a leading 1
    if (carry)  result = '1' + result;

    return result;
}

// A utility function to multiply single bits of strings a and b
int multiplyiSingleBit(string a, string b)
{  return (a[0] - '0')*(b[0] - '0');  }

// The main function that multiplies two bit strings X and Y and retur
// result as long integer
long int multiply(string X, string Y)
{
    // Find the maximum of lengths of x and Y and make length
    // of smaller string same as that of larger string
    int n = makeEqualLength(X, Y);

    // Base cases
    if (n == 0) return 0;
    if (n == 1) return multiplyiSingleBit(X, Y);

    int fh = n/2;   // First half of string, floor(n/2)
    int sh = (n-fh); // Second half of string, ceil(n/2)

    // Find the first half and second half of first string.
    // Refer http://goo.gl/lLmgn for substr method
    string Xl = X.substr(0, fh);
    string Xr = X.substr(fh, sh);

    // Find the first half and second half of second string
    string Yl = Y.substr(0, fh);
    string Yr = Y.substr(fh, sh);

    // Recursively calculate the three products of inputs of size n/2
    long int P1 = multiply(Xl, Yl);
    long int P2 = multiply(Xr, Yr);
    long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr

    // Combine the three products to get the final result.
    return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}

// Driver program to test aboev functions
int main()
{
    printf ("%ld\n", multiply("1100", "1010"));
    printf ("%ld\n", multiply("110", "1010"));
    printf ("%ld\n", multiply("11", "1010"));
    printf ("%ld\n", multiply("1", "1010"));
    printf ("%ld\n", multiply("0", "1010"));
    printf ("%ld\n", multiply("111", "111"));
    printf ("%ld\n", multiply("11", "11"));
}
```

Output:

```
120
60
30
10
0
49
9
```

Time Complexity: Time complexity of the above solution is O($n^{1.59}$).

Time complexity of multiplication can be further improved using another Divide and Conquer algorithm, fast Fourier transform. We will soon be discussing fast Fourier transform as a separate post.

Exercise
The above program returns a long int value and will not work for big strings. Extend the above program to return a string instead of a long int value.

References:
Wikipedia page for Karatsuba algorithm
Algorithms 1st Edition by Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani
http://courses.csail.mit.edu/6.006/spring11/exams/notes3-karatsuba
http://www.cc.gatech.edu/~ninamf/Algos11/lectures/lect0131.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

17. Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time)

Given a function 'int f(unsigned int x)' which takes a **non-negative integer** 'x' as input and returns an **integer** as output. The function is monotonically increasing with respect to value of x, i.e., the value of f(x+1) is greater than f(x) for every input x. Find the value 'n' where f() becomes positive for the first time. Since f() is monotonically increasing, values of f(n+1), f(n+2),… must be positive and values of f(n-2), f(n-3), .. must be negative.
Find n in O(logn) time, you may assume that f(x) can be evaluated in O(1) time for any input x.

A **simple solution** is to start from i equals to 0 and one by one calculate value of f(i) for 1, 2, 3, 4 .. etc until we find a positive f(i). This works, but takes O(n) time.

Can we apply Binary Search to find n in O(Logn) time? We can't directly apply Binary Search as we don't have an upper limit or high index. The idea is to do repeated doubling until we find a positive value, i.e., check values of f() for following values until f(i) becomes positive.

```
  f(0)
  f(1)
  f(2)
  f(4)
  f(8)
  f(16)
  f(32)
  ....
  ....
  f(high)
Let 'high' be the value of i when f() becomes positive for first time.
```

Can we apply Binary Search to find n after finding ‘high’? We can apply Binary Search now, we can use ‘high/2′ as low and ‘high’ as high indexes in binary search. The result n must lie between ‘high/2′ and ‘high’.

Number of steps for finding ‘high’ is O(Logn). So we can find ‘high’ in O(Logn) time. What about time taken by Binary Search between high/2 and high? The value of ‘high’ must be less than 2*n. The number of elements between high/2 and high must be O(n). Therefore, time complexity of Binary Search is O(Logn) and overall time complexity is 2*O(Logn) which is O(Logn).

```c
#include <stdio.h>
int binarySearch(int low, int high); // prototype

// Let's take an example function as f(x) = x^2 - 10*x - 20
// Note that f(x) can be any monotonocally increasing function
int f(int x) { return (x*x - 10*x - 20); }

// Returns the value x where above function f() becomes positive
// first time.
int findFirstPositive()
{
    // When first value itself is positive
    if (f(0) > 0)
        return 0;

    // Find 'high' for binary search by repeated doubling
    int i = 1;
    while (f(i) <= 0)
        i = i*2;

    //  Call binary search
    return binarySearch(i/2, i);
}

// Searches first positive value of f(i) where low <= i <= high
int binarySearch(int low, int high)
{
    if (high >= low)
    {
        int mid = low + (high - low)/2; /* mid = (low + high)/2 */

        // If f(mid) is greater than 0 and one of the following two
        // conditions is true:
        // a) mid is equal to low
        // b) f(mid-1) is negative
        if (f(mid) > 0 && (mid == low || f(mid-1) <= 0))
            return mid;

        // If f(mid) is smaller than or equal to 0
        if (f(mid) <= 0)
            return binarySearch((mid + 1), high);
        else // f(mid) > 0
            return binarySearch(low, (mid -1));
    }

    /* Return -1 if there is no positive value in given range */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    printf("The value n where f() becomes positive first is %d",
           findFirstPositive());
    return 0;
}
```

Output:

```
The value n where f() becomes positive first is 12
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

18. Find the minimum element in a sorted and rotated array

A sorted array is rotated at some unknown point, find the minimum element in it.

Following solution assumes that all elements are distinct.

Examples

```
Input: {5, 6, 1, 2, 3, 4}
Output: 1

Input: {1, 2, 3, 4}
Output: 1

Input: {2, 1}
Output: 1
```

A simple solution is to traverse the complete array and find minimum. This solution requires $\Theta(n)$ time.
We can do it in O(Logn) using Binary Search. If we take a closer look at above examples, we can easily figure out following pattern: The minimum element is the only element whose previous element is greater than it. If there is no such element, then there is no rotation and first element is the minimum element. Therefore, we do binary search for an element which is smaller than the previous element. We strongly recommend you to try it yourself before seeing the following C implementation.

```
// C program to find minimum element in a sorted and rotated array
#include <stdio.h>

int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)  return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {3, 4, 5, 1, 2}
    if (mid < high && arr[mid+1] < arr[mid])
```

```
        return arr[mid+1];

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
        return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
        return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] =  {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));

    int arr2[] =  {1, 2, 3, 4};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));

    int arr3[] =  {1};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

    int arr4[] =  {1, 2};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

    int arr5[] =  {2, 1};
    int n5 = sizeof(arr5)/sizeof(arr5[0]);
    printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));

    int arr6[] =  {5, 6, 7, 1, 2, 3, 4};
    int n6 = sizeof(arr6)/sizeof(arr6[0]);
    printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

    int arr7[] =  {1, 2, 3, 4, 5, 6, 7};
    int n7 = sizeof(arr7)/sizeof(arr7[0]);
    printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

    int arr8[] =  {2, 3, 4, 5, 6, 7, 8, 1};
    int n8 = sizeof(arr8)/sizeof(arr8[0]);
    printf("The minimum element is %d\n", findMin(arr8, 0, n8-1));

    int arr9[] =  {3, 4, 5, 1, 2};
    int n9 = sizeof(arr9)/sizeof(arr9[0]);
    printf("The minimum element is %d\n", findMin(arr9, 0, n9-1));

    return 0;
}
```

Output:

```
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
```

How to handle duplicates?

It turned out that duplicates can't be handled in O(Logn) time in all cases. Thanks to Amit Jain for inputs. The special cases that cause problems are like {2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} and {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to go to left half or right half by doing constant number of comparisons at the middle. Following is an implementation that handles duplicates. It may become O(n) in worst case though.

```c
// C program to find minimum element in a sorted and rotated array
#include <stdio.h>

int min(int x, int y) { return (x < y)? x :y; }

// The function that handles duplicates.  It can be O(n) in worst case
int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)  return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {1, 1, 0, 1}
    if (mid < high && arr[mid+1] < arr[mid])
       return arr[mid+1];

    // This case causes O(n) time
    if (arr[low] == arr[mid] && arr[high] == arr[mid])
        return min(findMin(arr, low, mid-1), findMin(arr, mid+1, high)

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
       return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
       return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] =  {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));
    int arr2[] =  {1, 1, 0, 1};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));

    int arr3[] =  {1, 1, 2, 2, 3};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

    int arr4[] =  {3, 3, 3, 4, 4, 4, 4, 5, 3, 3};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

    int arr5[] =  {2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2};
    int n5 = sizeof(arr5)/sizeof(arr5[0]);
    printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));

    int arr6[] =  {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1};
    int n6 = sizeof(arr6)/sizeof(arr6[0]);
    printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

    int arr7[] =  {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2};
    int n7 = sizeof(arr7)/sizeof(arr7[0]);
    printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

    return 0;
}
```

Output:

```
The minimum element is 1
The minimum element is 0
The minimum element is 1
The minimum element is 3
The minimum element is 0
The minimum element is 1
The minimum element is 0
```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

19. Closest Pair of Points | O(nlogn) Implementation

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.

$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$

We have discussed a divide and conquer solution for this problem. The time complexity of the implementation provided in the previous post is O(n (Logn)^2). In this post, we discuss an implementation with time complexity as O(nLogn).

Following is a recap of the algorithm discussed in the previous post.

1) We sort all points according to x coordinates.

2) Divide all points in two halves.

3) Recursively find the smallest distances in both subarrays.

4) Take the minimum of two smallest distances. Let the minimum be d.

5) Create an array strip[] that stores all points which are at most d distance away from the middle line dividing the two sets.

6) Find the smallest distance in strip[].

7) Return the minimum of d and the smallest distance calculated in above step 6.

The great thing about the above approach is, if the array strip[] is sorted according to y coordinate, then we can find the smallest distance in strip[] in O(n) time. In the implementation discussed in previous post, strip[] was explicitly sorted in every recursive call that made the time complexity O(n (Logn)^2), assuming that the sorting step takes O(nLogn) time.
In this post, we discuss an implementation where the time complexity is O(nLogn). The idea is to presort all points according to y coordinates. Let the sorted array be Py[]. When we make recursive calls, we need to divide points of Py[] also according to the vertical line. We can do that by simply processing every point and comparing its x coordinate with x coordinate of middle line.

Following is C++ implementation of O(nLogn) approach.

```
// A divide and conquer program in C++ to find the smallest distance f
// given set of points.

#include <iostream>
#include <float.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

// 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 po
// 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 beween the closest points o
// strip of given size. All points in strip[] are sorted accordint 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

    // Pick all points one by one and try the next points till the dif
    // 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;
            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 Px co
// all points sorted according to x coordinates and Py contains all po
// sorted according to y coordinates
float closestUtil(Point Px[], Point Py[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(Px, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = Px[mid];


    // Divide points in y sorted array around the vertical line.
    // Assumption: All x coordinates are distinct.
    Point Pyl[mid+1];   // y sorted points on left of vertical line
    Point Pyr[n-mid-1];  // y sorted points on right of vertical line
    int li = 0, ri = 0;  // indexes of left and right subarrays
    for (int i = 0; i < n; i++)
    {
      if (Py[i].x <= midPoint.x)
         Pyl[li++] = Py[i];
      else
         Pyr[ri++] = Py[i];
    }

    // 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(Px, Pyl, mid);
    float dr = closestUtil(Px + mid, Pyr, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j] = Py[i], j++;

    // Find the closest points in strip.  Return the minimum of d and
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main functin that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }
    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);

    // Use recursive function closestUtil() to find the smallest dista
    return closestUtil(Px, Py, 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]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}
```

Output:

```
The smallest distance is 1.41421
```

Time Complexity: Let Time complexity of above algorithm be T(n). Let us assume that we use a O(nLogn) sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time. Also, it takes O(n) time to divide the Py array around the mid vertical line. Finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows
T(n) = 2T(n/2) + O(n) + O(n) + O(n)
T(n) = 2T(n/2) + O(n)
T(n) = T(nLogn)

References:
http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf
http://www.youtube.com/watch?v=vS4Zn1a9KUc
http://www.youtube.com/watch?v=T3T7T8Ym20M
http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

20. Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)

Given two square matrices A and B of size n x n each, find their multiplication matrix.

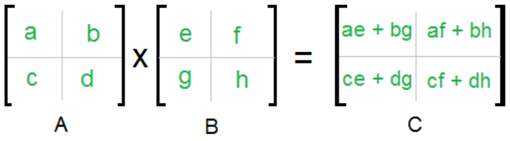
Naive Method
Following is a simple way to multiply two matrices.

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is $O(N^3)$.

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



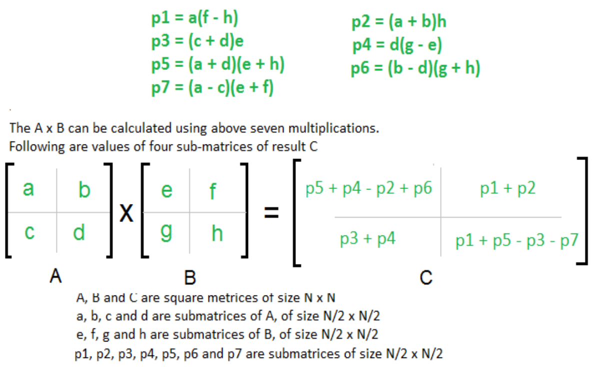
A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

```
T(N) = 8T(N/2) + O(N^2)

From Master's Theorem, time complexity of above method is O(N^3)
which is unfortunately same as the above naive method.
```

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.



p1 = a(f - h)　　p2 = (a + b)h
p3 = (c + d)e　　p4 = d(g - e)
p5 = (a + d)(e + h)　　p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes O(N^2) time. So time complexity can be written as

```
T(N) = 7T(N/2) +  O(N^2)

From Master's Theorem, time complexity of above method is
O(N^Log7) which is approximately O(N^2.8074)
```

Generally Strassen's Method is not preferred for practical applications for following reasons.
1) The constants used in Strassen's method are high and for a typical application Naive method works better.
2) For Sparse matrices, there are better methods especially designed for them.
3) The submatrices in recursion take extra space.
4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method (Source: CLRS Book)

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
https://www.youtube.com/watch?v=LOLebQ8nKHA
https://www.youtube.com/watch?v=QXY4RskLQcI

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

21. Find the number of zeroes

Given an array of 1s and 0s which has all 1s first followed by all 0s. Find the number of 0s. Count the number of zeroes in the given array.

Examples:

```
Input: arr[] = {1, 1, 1, 1, 0, 0}
Output: 2

Input: arr[] = {1, 0, 0, 0, 0}
Output: 4

Input: arr[] = {0, 0, 0}
Output: 3

Input: arr[] = {1, 1, 1, 1}
Output: 0
```

We strongly recommend to minimize the browser and try this yourself in time complexity better than O(n).

A **simple solution** is to traverse the input array. As soon as we find a 0, we return n – index of first 0. Here n is number of elements in input array. Time complexity of this solution would be O(n).

Since the input array is sorted, we can use **Binary Search** to find the first occurrence of 0. Once we have index of first element, we can return count as n – index of first zero.

```c
// A divide and conquer solution to find count of zeroes in an array
// where all 1s are present before all 0s
#include <stdio.h>

/* if 0 is present in arr[] then returns the index of FIRST occurrence
   of 0 in arr[low..high], otherwise returns -1 */
int firstZero(int arr[], int low, int high)
{
    if (high >= low)
    {
        // Check if mid element is first 0
        int mid = low + (high - low)/2;
        if (( mid == 0 || arr[mid-1] == 1) && arr[mid] == 0)
            return mid;

        if (arr[mid] == 1)  // If mid element is not 0
            return firstZero(arr, (mid + 1), high);
        else  // If mid element is 0, but not first 0
            return firstZero(arr, low, (mid -1));
    }
    return -1;
}

// A wrapper over recursive function firstZero()
int countOnes(int arr[], int n)
{
    // Find index of first zero in given array
    int first = firstZero(arr, 0, n-1);

    // If 0 is not present at all, return 0
    if (first == -1)
        return 0;

    return (n - first);
}

/* Driver program to check above functions */
int main()
{
    int arr[] =   {1, 1, 1, 0, 0, 0, 0, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Count of zeroes is %d", countOnes(arr, n));
    return 0;
}
```

Output:

```
Count of zeroes is 5
```

Time Complexity: O(Logn) where n is number of elements in arr[].

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

22. Multiply two polynomials

Given two polynomials represented by two arrays, write a function that multiplies given two polynomials.

Example:

```
Input:  A[] = {5, 0, 10, 6}
        B[] = {1, 2, 4}
Output: prod[] = {5, 10, 30, 26, 52, 24}

The first input array represents "5 + 0x^1 + 10x^2 + 6x^3"
The second array represents "1 + 2x^1 + 4x^2"
And Output is "5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5"
```

We strongly recommend to minimize your browser and try this yourself first.

A simple solution is to one by one consider every term of first polynomial and multiply it with every term of second polynomial. Following is algorithm of this simple method.

```
multiply(A[0..m-1], B[0..n01])
1) Create a product array prod[] of size m+n-1.
2) Initialize all entries in prod[] as 0.
3) Travers array A[] and do following for every element A[i]
...(3.a) Traverse array B[] and do following for every element B[j]
          prod[i+j] = prod[i+j] + A[i] * B[j]
4) Return prod[].
```

The following is C++ implementation of above algorithm.

```cpp
// Simple C++ program to multiply two polynomials
#include <iostream>
using namespace std;

// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *multiply(int A[], int B[], int m, int n)
{
   int *prod = new int[m+n-1];

   // Initialize the porduct polynomial
   for (int i = 0; i<m+n-1; i++)
     prod[i] = 0;

   // Multiply two polynomials term by term

   // Take ever term of first polynomial
   for (int i=0; i<m; i++)
   {
     // Multiply the current term of first polynomial
     // with every term of second polynomial.
     for (int j=0; j<n; j++)
         prod[i+j] += A[i]*B[j];
   }

   return prod;
}

// A utility function to print a polynomial
void printPoly(int poly[], int n)
{
    for (int i=0; i<n; i++)
    {
       cout << poly[i];
       if (i != 0)
        cout << "x^" << i ;
       if (i != n-1)
       cout << " + ";
    }
}

// Driver program to test above functions
int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};

    // The following array represents polynomial 1 + 2x + 4x^2
    int B[] = {1, 2, 4};
    int m = sizeof(A)/sizeof(A[0]);
    int n = sizeof(B)/sizeof(B[0]);

    cout << "First polynomial is \n";
    printPoly(A, m);
    cout << "\nSecond polynomial is \n";
    printPoly(B, n);

    int *prod = multiply(A, B, m, n);

    cout << "\nProduct polynomial is \n";
    printPoly(prod, m+n-1);

    return 0;
}
```

Output

```
First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Product polynomial is
5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5
```

Time complexity of the above solution is O(mn). If size of two polynomials same, then time complexity is O(n^2).

Can we do better?

There are methods to do multiplication faster than $O(n^2)$ time. These methods are mainly based on divide and conquer. Following is one simple method that divides the given polynomial (of degree n) into two polynomials one containing lower degree terms(lower than n/2) and other containing higher degree terns (higher than or equal to n/2)

```
Let the two given polynomials be A and B.
For simplicity, Let us assume that the given two polynomials are of
same degree and have degree in powers of 2, i.e., n = 2^i

The polynomial 'A' can be written as A0 + A1*x^(n/2)
The polynomial 'B' can be written as B0 + B1*x^(n/2)

For example 1 + 10x + 6x^2 - 4x^3 + 5x^4 can be
written as (1 + 10x) + (6 - 4x + 5x^2)*x^2

A * B  = (A0 + A1*x^(n/2)) * (B0 + B1*x^(n/2))
       = A0*B0 + A0*B1*x^(n/2) + A1*B0*x^(n/2) + A1*B1*x^n
       = A0*B0 + (A0*B1 + A1*B0)x^(n/2) + A1*B1*x^n
```

So the above divide and conquer approach requires 4 multiplications and O(n) time to add all 4 results. Therefore the time complexity is T(n) = 4T(n/2) + O(n). The solution of the recurrence is $O(n^2)$ which is same as the above simple solution.

The idea is to reduce number of multiplications to 3 and make the recurrence as T(n) = 3T(n/2) + O(n)

How to reduce number of multiplications?
This requires a little trick similar to Strassen's Matrix Multiplication. We do following 3 multiplications.

```
X = (A0 + A1)*(B0 + B1) // First Multiplication
Y = A0B0  // Second
Z = A1B1  // Third

The missing middle term in above multiplication equation A0*B0 + (A0*B1 +
A1*B0)x^(n/2) + A1*B1*x^n can obtained using below.
A0B1 + A1B0 = X - Y - Z
```

So the time taken by this algorithm is T(n) = 3T(n/2) + O(n)
The solution of above recurrence is $O(n^{Lg3})$ which is better than $O(n^2)$.

We will soon be discussing implementation of above approach.

There is a O(nLogn) algorithm also that uses Fast Fourier Transform to multiply two polynomials (Refer this and this for details)

Sources:
http://www.cse.ust.hk/~dekai/271/notes/L03/L03.pdf

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

23. Find the missing number in Arithmetic Progression

Given an array that represents elements of arithmetic progression in order. One element is missing in the progression, find the missing number.

Examples:

```
Input: arr[]  = {2, 4, 8, 10, 12, 14}
Output: 6

Input: arr[]  = {1, 6, 11, 16, 21, 31};
Output: 26
```

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to linearly traverse the array and find the missing number. Time complexity of this solution is O(n).

We can solve this problem **in O(Logn) time** using Binary Search. The idea is to go to the middle element. Check if the difference between middle and next to middle is equal to diff or not, if not then the missing element lies between mid and mid+1. If the middle element is equal to n/2th term in Arithmetic Series (Let n be the number of elements in input array), then missing element lies in right half. Else element lies in left half.

Following is C implementation of above idea.

```
// A C program to find the missing number in a given
// arithmetic progression
#include <stdio.h>
#include <limits.h>

// A binary search based recursive function that returns
// the missing element in arithmetic progression
int findMissingUtil(int arr[], int low, int high, int diff)
{
    // There must be two elements to find the missing
    if (high <= low)
        return INT_MAX;

    // Find index of middle element
    int mid = low + (high - low)/2;

    // The element just after the middle element is missing.
    // The arr[mid+1] must exist, because we return when
    // (low == high) and take floor of (high-low)/2
    if (arr[mid+1] - arr[mid] != diff)
        return (arr[mid] + diff);

    // The element just before mid is missing
    if (mid > 0 && arr[mid] - arr[mid-1] != diff)
        return (arr[mid-1] + diff);

    // If the elements till mid follow AP, then recur
    // for right half
    if (arr[mid] == arr[0] + mid*diff)
        return findMissingUtil(arr, mid+1, high, diff);

    // Else recur for left half
    return findMissingUtil(arr, low, mid-1, diff);
}

// The function uses findMissingUtil() to find the missing
// element in AP.  It assumes that there is exactly one missing
// element and may give incorrect result when there is no missing
// element or more than one missing elements.
// This function also assumes that the difference in AP is an
// integer.
int findMissing(int arr[], int n)
{
    // If exactly one element is missing, then we can find
    // difference of arithmetic progression using following
    // formula.  Example, 2, 4, 6, 10, diff = (10-2)/4 = 2.
    // The assumption in formula is that the difference is
    // an integer.
    int diff = (arr[n-1] - arr[0])/n;

    // Binary search for the missing number using above
    // calculated diff
    return findMissingUtil(arr, 0, n-1, diff);
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {2, 4, 8, 10, 12, 14};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("The missing element is %d", findMissing(arr, n));
    return 0;
}
```

Output:

```
The missing element is 6
```

Exercise:

Solve the same problem for Geometrical Series. What is the time complexity of your solution? What about Fibonacci Series?

This article is contributed by **Harshit Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

24. Divide and Conquer | Set 6 (Tiling Problem)

Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

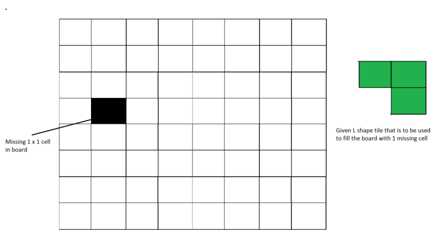

Missing 1 x 1 cell in board
Given L shape tile that is to be used to fill the board with 1 missing cell

Figure 1: An example input

This problem can be solved using Divide and Conquer. Below is the recursive algorithm.

```
// n is size of given square, p is location of missing cell
Tile(int n, Point p)

1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing
   but a tile and can be filled with a single tile.

2) Place a L shaped tile at the center such that it does not cover
   the n/2 * n/2 subsquare that has a missing square. Now all four
   subsquares of size n/2 x n/2 have a missing cell (a cell that doesn't
   need to be filled).  See figure 2 below.

3) Solve the problem recursively for following four. Let p1, p2, p3 and
   p4 be positions of the 4 missing cells in 4 squares.
   a) Tile(n/2, p1)
   b) Tile(n/2, p2)
   c) Tile(n/2, p3)
   d) Tile(n/2, p3)
```

The below diagrams show working of above algorithm

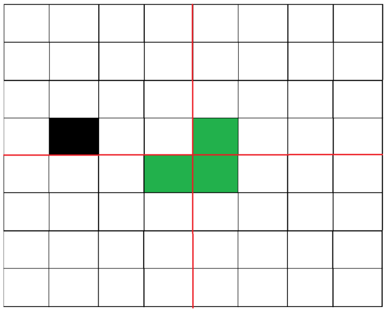


Figure 2: After placing first tile

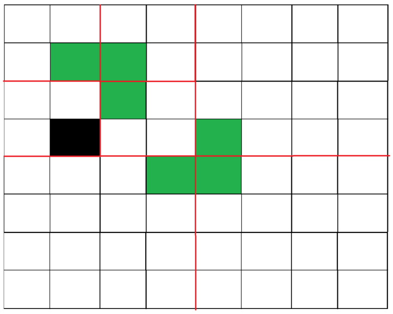


Figure 3: Recurring for first subsquare.

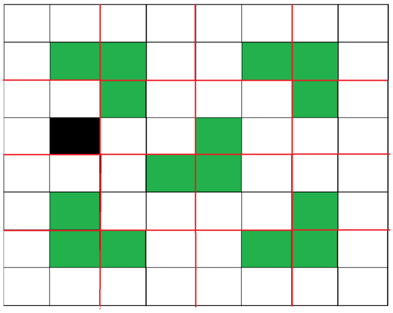


Figure 4: Shows first step in all four subsquares.

Time Complexity:
Recurrence relation for above recursive algorithm can be written as below. C is a constant.
T(n) = 4T(n/2) + C
The above recursion can be solved using Master Method and time complexity is O(n^2)

How does this work?
The working of Divide and Conquer algorithm can be proved using Mathematical Induction. Let the input square be of size 2^k x 2^k where k >=1.
Base Case: We know that the problem can be solved for k = 1. We have a 2 x 2 square with one cell missing.
Induction Hypothesis: Let the problem can be solved for k-1.
Now we need to prove to prove that the problem can be solved for k if it can be solved for k-1. For k, we put a L shaped tile in middle and we have four subsqures with dimension 2^{k-1} x 2^{k-1} as shown in figure 2 above. So if we can solve 4 subsquares, we can solve the complete square.

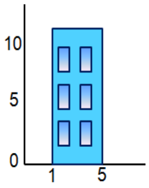
References:
http://www.comp.nus.edu.sg/~sanjay/cs3230/dandc.pdf

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

25. Divide and Conquer | Set 7 (The Skyline Problem)

Given n rectangular buildings in a 2-dimensional city, computes the skyline of these buildings, eliminating hidden lines. The main task is to view buildings from a side and remove all sections that are not visible.

All buildings share common bottom and every **building** is represented by triplet (left, ht, right)



'left': is x coordinated of left side (or wall).
'right': is x coordinate of right side
'ht': is height of building.

For example, the building on right side (the figure is taken from here) is represented as (1, 11, 5)

A **skyline** is a collection of rectangular strips. A rectangular **strip** is represented as a pair (left, ht) where left is x coordinate of left side of strip and ht is height of strip.

Examples:

```
Input: Array of buildings
       { (1,11,5), (2,6,7), (3,13,9), (12,7,16), (14,3,25),
         (19,18,22), (23,13,29), (24,4,28) }
Output: Skyline (an array of rectangular strips)
        A strip has x coordinate of left side and height
        (1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18),
        (22, 3), (25, 0)
The below figure (taken from here) demonstrates input and output.
The left side shows buildings and right side shows skyline.



Consider following as another example when there is only one
building
Input:  {(1, 11, 5)}
Output: (1, 11), (5, 0)
```

A **Simple Solution** is to initialize skyline or result as empty, then one by one add buildings to skyline. A building is added by first finding the overlapping strip(s). If there are no overlapping strips, the new building adds new strip(s). If overlapping strip is found, then height of the existing strip may increase. Time complexity of this solution is O(n^2)

We can find Skyline in Θ(nLogn) time using **Divide and Conquer**. The idea is similar to Merge Sort, divide the given set of buildings in two subsets. Recursively construct skyline for two halves and finally merge the two skylines.

How to Merge two Skylines?
The idea is similar to merge of merge sort, start from first strips of two skylines, compare x coordinates. Pick the strip with smaller x coordinate and add it to result. The height of added strip is considered as maximum of current heights from skyline1 and skyline2.
Example to show working of merge:

```
  Height of new Strip is always obtained by takin maximum of following
     (a) Current height from skyline1, say 'h1'.
     (b) Current height from skyline2, say 'h2'
  h1 and h2 are initialized as 0. h1 is updated when a strip from
  SkyLine1 is added to result and h2 is updated when a strip from
  SkyLine2 is added.

  Skyline1 = {(1, 11),  (3, 13),  (9, 0),  (12, 7),  (16, 0)}
  Skyline2 = {(14, 3),  (19, 18), (22, 3), (23, 13),  (29, 0)}
  Result = {}
  h1 = 0, h2 = 0

  Compare (1, 11) and (14, 3).  Since first strip has smaller left x,
  add it to result and increment index for Skyline1.
  h1 = 11, New Height  = max(11, 0)
  Result =   {(1, 11)}

  Compare (3, 13) and (14, 3). Since first strip has smaller left x,
  add it to result and increment index for Skyline1
  h1 = 13, New Height =  max(13, 0)
  Result =  {(1, 11), (3, 13)}

  Similarly (9, 0) and (12, 7) are added.
  h1 = 7, New Height =  max(7, 0) = 7
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7)}

  Compare (16, 0) and (14, 3). Since second strip has smaller left x,
  it is added to result.
  h2 = 3, New Height =  max(7, 3) = 7
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7)}

  Compare (16, 0) and (19, 18). Since first strip has smaller left x,
  it is added to result.
  h1 = 0, New Height =  max(0, 3) = 3
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 3), (16, 3)}

Since Skyline1 has no more items, all remaining items of Skyline2
are added
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 3), (16, 3),
              (19, 18), (22, 3), (23, 13), (29, 0)}

One observation about above output is, the strip (16, 3) is redundant
(There is already an strip of same height). We remove all redundant
strips.
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 3), (19, 18),
              (22, 3), (23, 13), (29, 0)}

In below code, redundancy is handled by not appending a strip if the
previous strip in result has same height.
```

Below is C++ implementation of above idea.

```
// A divide and conquer based C++ program to find skyline of given
// buildings
#include<iostream>
```

```cpp
#include<iostream>
using namespace std;

// A structure for building
struct Building
{
    int left;  // x coordinate of left side
    int ht;    // height
    int right; // x coordinate of right side
};

// A strip in skyline
class Strip
{
    int left;  // x coordinate of left side
    int ht; // height
public:
    Strip(int l=0, int h=0)
    {
        left = l;
        ht = h;
    }
    friend class SkyLine;
};

// Skyline:  To represent Output (An array of strips)
class SkyLine
{
    Strip *arr;   // Array of strips
    int capacity; // Capacity of strip array
    int n;   // Actual number of strips in array
public:
    ~SkyLine() {  delete[] arr;  }
    int count()  { return n;   }

    // A function to merge another skyline
    // to this skyline
    SkyLine* Merge(SkyLine *other);

    // Constructor
    SkyLine(int cap)
    {
        capacity = cap;
        arr = new Strip[cap];
        n = 0;
    }

    // Function to add a strip 'st' to array
    void append(Strip *st)
    {
        // Check for redundant strip, a strip is
        // redundant if it has same height or left as previous
        if (n>0 && arr[n-1].ht == st->ht)
            return;
        if (n>0 && arr[n-1].left == st->left)
        {
            arr[n-1].ht = max(arr[n-1].ht, st->ht);
            return;
        }

        arr[n] = *st;
        n++;
        ....;
    }

    // A utility function to print all strips of
    // skyline
    void print()
    {
        for (int i=0; i<n; i++)
        {
            cout << " (" << arr[i].left << ", "
                 << arr[i].ht << "), ";
        }
    }
};

// This function returns skyline for a given array of buildings
// arr[l..h].  This function is similar to mergeSort().
SkyLine *findSkyline(Building arr[], int l, int h)
{
    if (l == h)
    {
        SkyLine *res = new SkyLine(2);
        res->append(new Strip(arr[l].left, arr[l].ht));
        res->append(new Strip(arr[l].right, 0));
        return res;
    }

    int mid = (l + h)/2;

    // Recur for left and right halves and merge the two results
    SkyLine *sl = findSkyline(arr, l, mid);
    SkyLine *sr = findSkyline(arr, mid+1, h);
    SkyLine *res = sl->Merge(sr);

    // To avoid memory leak
    delete sl;
    delete sr;

    // Return merged skyline
    return res;
}

// Similar to merge() in MergeSort
// This function merges another skyline 'other' to the skyline
// for which it is called.  The function returns pointer to
// the resultant skyline
SkyLine *SkyLine::Merge(SkyLine *other)
{
    // Create a resultant skyline with capacity as sum of two
    // skylines
    SkyLine *res = new SkyLine(this->n + other->n);

    // To store current heights of two skylines
    int h1 = 0, h2 = 0;

    // Indexes of strips in two skylines
    int i = 0, j = 0;
    while (i < this->n && j < other->n)
    {
        // Compare x coordinates of left sides of two
        // skylines and put the smaller one in result
        if (this->arr[i].left < other->arr[j].left)
        {
            int x1 = this->arr[i].left;
            h1 = this->arr[i].ht;

            // Choose height as max of two heights
            int maxh = max(h1, h2);

            res->append(new Strip(x1, maxh));
            i++;
        }
        else
        {
            int x2 = other->arr[j].left;
            h2 = other->arr[j].ht;
            int maxh = max(h1, h2);
            res->append(new Strip(x2, maxh));
            j++;
        }
    }

    // If there are strips left in this skyline or other
    // skyline
    while (i < this->n)
    {
        res->append(&arr[i]);
        i++;
    }
    while (j < other->n)
    {
        res->append(&other->arr[j]);
        j++;
    }
    return res;
}

// drive program
int main()
{
    Building arr[] = {{1, 11, 5}, {2, 6, 7}, {3, 13, 9},
                      {12, 7, 16}, {14, 3, 25}, {19, 18, 22},
                      {23, 13, 29}, {24, 4, 28}};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Find skyline for given buildings and print the skyline
    SkyLine *ptr = findSkyline(arr, 0, n-1);
    cout << " Skyline for given buildings is \n";
    ptr->print();
    return 0;
}
```

```
Skyline for given buildings is
(1, 11),  (3, 13),  (9, 0),  (12, 7),  (16, 3),  (19, 18),
(22, 3),  (23, 13),  (29, 0),
```

Time complexity of above recursive implementation is same as Merge Sort.

T(n) = T(n/2) + Θ(n)

Solution of above recurrence is Θ(nLogn)

References:
http://faculty.kfupm.edu.sa/ics/darwish/stuff/ics353handouts/Ch4Ch5.pdf
www.cs.ucf.edu/~sarahb/COP3503/Lectures/DivideAndConquer.ppt

This article is contributed **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above