Algorithms

11. Divide & Conquer

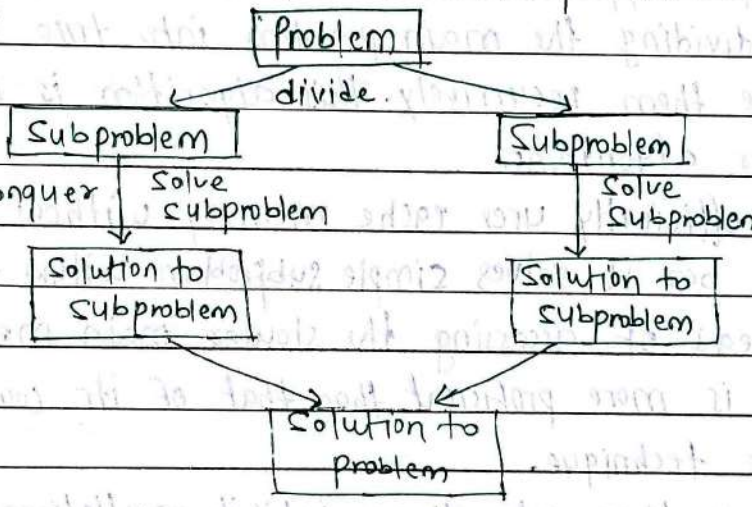
Handwritten by pankaj kumar

3. Divide & Conquer

Date
Page No. 56.

① Introduction

- Divide & Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer strategy.
- Divide : the original problem into a set of subproblems.
- Conquer: Solve every subproblem individually, recursively.
- Combine: Put together the solutions of the subproblems to get the solution to the whole problem.



* Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer strategy.

1. Relational Formula
2. Stopping condition

1. Relational Formula:

It is the formula that we generate from the given technique. After generation of formula we apply D&C strategy. i.e, we break the problem recursively & solve the broken subproblem.

2. Stopping Condition:-
When we break the problem using Divide & Conquer strategy, then we need to know that for How much time, we need to apply divide & conquer. so the condition where the need to stop our recursion steps of D&C is called as stopping condition.

*Advantages of Divide & conquer.
- Divide & conquer tend to successfully solve one of the biggest problems, such as the Tower of Honoi, a mathematical puzzle. it is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide & conquer approach, it has lessened the effort as it works on dividing the main problem into two halves. and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space bcz it solves simple subproblem within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute force technique.
- since these algorithms inhibit parallelism, it does not involve any modification and is handled by system incorporating parallel processing.

* Disadvantages
- since most of its algorithm are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

* Application:
- Merge Sort
- Quick sort
- Closest pair of points
- Strassen's Algorithm
- karatsuba Algo for fast multiplication

(2) Implement Power Function:
- Given two integer, x and n, where n is non-negative, compute the power function

* Example:
pow(-2, 10) = 1024
pow(2, 4) = 16
x n

* Naive Iterative solution

```
int pow(int x, int n){
    int pow = 1;
    for(int i=0; i<n; i++)
        pow = pow * x;
    return pow
}
```

O(n)

* Using divide and conquer.

```
int power(int x, int n){
    if (n==0)
        return 1;
    if (n/2 == 0)
        return power(x, n/2) * power(x, n/2)
    else
        return x * power(x, n/2) * power(x, n/2)
}
```

O(n)

* Optimized divide & conquer.

The problem with the above solution is that the same subproblem is computed twice for each recursive call. We can optimize the above function by computing the solution of the problem only once.

```
int power(int x, 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;
}
```

O(log n)

(3.) Median of two sorted array of same size:

- There are two sorted array A and B of size n each, write an algorithm to find the median of the array obtained after merging the both arrays.

*Example

Input: A[] = {1, 12, 15, 26, 38}
B[] = {2, 13, 17, 30, 45}

Output: 16

Explanation :-

after merging {1, 2, 12, 13, 15, 17, 26, 30, 38, 45}
middle two element are 15 and 17 Average of middle element is $\frac{(15+17)}{2} = 16$

- Since size is 2n (even) there here middle element will be 2

(*) Method 1 (simply count while merging):

- Use the merge procedure of merge sort. keep track of count of element. Take the averge of index (n-1) and n.

```
int getMedian(int arr1[], int arr2[], int n){
    int i=0;
    int j=0;
    int count;
    int m1=-1, m2=-1;
    for(count=0; count<=n; count++){
        if(i==n){
            m1=m2;
            m2=arr2[0];
            break;
        }
        else if(j==n){
            m1=m2;
            m2=arr1[0];
            break;
        }
        if(arr1[i]<=arr2[j]){
            m1=m2;
            m2=arr1[i];
            i++;
        }
        else{
            m1=m2;
            m2=arr2[j];
            j++;
        }
    }
    return (m1+m2)/2;
}
```

handle case when all element of arr1 are smaller than First element of arr2.

When all element of arr2 are smaller than first element of arr1.

Time complexity = O(n)

space complexity = O(1)

Note: doesn't work when array have unequal size

Divide & conquer

✱ Method 2 (By comparing the medians of two arrays)

Algo:

1. Calculate medians m1 and m2 for both arr1 & arr2.
2. if m1 == m2, then we are done return m1 (or m2).
3. if m1 > m2, then median is present in one of below two subarrays.
   a. From first element of arr1 to m1 (arr1[0] / arr1[1] / ... / arr1[n/2])
   b. From m2 to last element of arr2 (arr2[n/2] / ... / arr2[n-1])
4. if m2 > m1, then median is present in one of the below two subarrays.
   a. From m1 to last element of arr1
   b. From first to m2 element of arr2.
5. Repeat the above process until size of both the subarrays becomes 2.
6. if size of both array is 2 then use below formula.

$$\text{median} = (\max(arr1[0], arr2[0]) + \min(arr1[1], arr2[1]))/2$$

Example :-

a1 = {1, 12, 15, 26, 38} | a2 = {2, 13, 17, 30, 45}

here, m1 = 15 & m2 = 17 ∴ a1 = {15, 26, 38}, a2 = {2, 13, 17}

now, m1 = 26 & m2 = 13 ∴ a1 = {15, 26}, a2 = {13, 17}

now median = (max(15, 13) + min(26, 17))/2

= (15 + 17)/2 = 16

Code :-

```
int getMedian (int arr1[], int arr2[], int n){
    if (n<=0) return -1;
    if (n==1) return (arr1+arr2[0])/2;
    if (n==2)
        return (max(arr1[0],arr2[0]) + min(arr1[1], arr2[1]))/2;
```

```
int m1 = median(arr1, n);
int m2 = median(arr2, n);
if (m1==m2) return m1;
if (m1<m2){
        if(n%2==0)
            return getMedian(arr1+n/2-1, arr2, n-n/2+1);
            return getMedian(arr2+n/2, arr2, n-n/2);
}
if(n%2==0)
        return getMedian(arr2+n/2-1, arr1, n-n/2+1);
return getMedian(arr2+n/2, arr1, n-n/2);
}
int median(int arr[], int n){
        if(n%2==0)
            return (arr[n/2]+arr[n/2-1])/2
        else
            return arr[n/2];
}
```

Time Complexity: O(log n)
Space Complexity: O(1)

* Method 3 (By taking union without extra space)

Algo:
1. Take union of two input array arr1[] and arr2[].
2. Sort arr1[] and arr2[] respectively.
3. The median will be the last element of arr1[] + First element of arr2[] divided by 2

Code:

```
int getMedian(int arr1[], int arr2[], int n){
        int j=0, i=n-1;
        while(arr1[i]>arr2[j] && j<n && i>-1)
            swap(arr1[i--], arr2[j++]);
```

```
sort(arr1, arr1+n);
sort(arr2, arr2+n);
return (arr1[n-1]+arr2[0])/2;
}
```

Time complexity: O(n log n)
Space complexity: O(1)

4. Count Inversion in an array

- Two element a[i] & a[j] of an array a[] form an inversion if a[i] > a[j] and i < j

* Example:

Input: a[] = {8, 4, 2, 1}
Output: 6
Explanation Inversion = (8,4), (4,2), (8,2), (8,1), (4,1), (2,1)

* Method 1 (simple):-

Algo:

① Traverse the array from start to end
② For every element, find the count of element smaller than the current element into right side
③ sum the count for every index and return

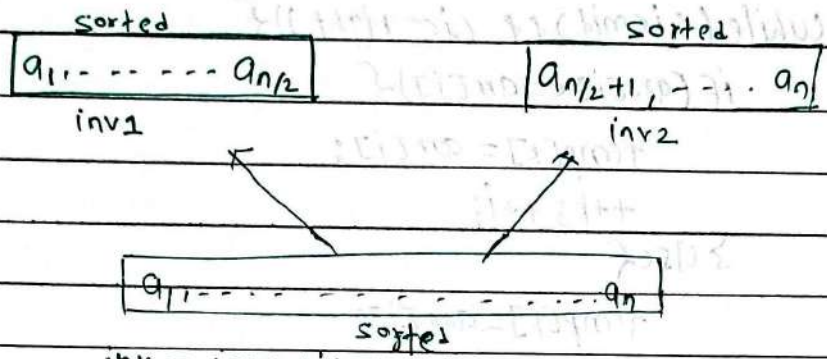
Code:

```
int getInvCount(int arr[], int n){
    int inv_count = 0;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(arr[i] > arr[j])
                inv_count++;
    return inv_count;
}
```

Time: $O(n^2)$ / Space: O(1)

*Method 2 (Enhance Merge Sort):-

- The idea is to use a method similar to the merge sort. Like, divide the given array into two parts. For each left & right half, count the inversions, and at the end, sum up the inversion from both halves to get the resultant inversion.



inv = inv1 + inv2 + inv for mersing

Example:-

(1) 1 3 5 10 | 2 6 8 9 → 1 2 | since A[i] > A[j], all element at index k > i will also be greater than A[j], i.e 3

(2) 1 3 5 10 | 2 6 8 9 → 1 2 3 5 6 | since A[i] > A[j], all element right side the A[i] will be greater in this case it is 1

(3) 1 3 5 10 | 2 6 8 9 → 1 2 3 5 6 8 | 1

(4) 1 3 5 10 | 2 6 8 9 → 1 2 3 5 6 8 9 | 1

(4) 1 3 5 10 | 2 6 8 9 → 1 2 3 5 6 8 9 10 | 0

Total inversion count = 6

- Note:- if the inversion for sorting the left and right array is x and y then total inversion will be.

$$\boxed{Inv = x + y + 6}$$

Code

```
long long merge(long long arr[], int left, int mid, int right){
    int i=left, j=mid, k=0;
    long long invCount=0;
    int temp[(right-left+1)];
    while((i<mid) && (j<=right)){
        if(arr[i]<=arr[j]){
            temp[k]=arr[i];
            ++k; ++i;
        }else{
            temp[k]=arr[j];
            invCount+=(mid-i);
            ++k; ++j;
        }
    }
    while(i<mid){
        temp[k]=arr[i];
        ++k; ++i;
    }
    while(j<=right){
        temp[k]=arr[j];
        ++k; ++j;
    }
    for(i=left; k=0; i<=right; i++, k++){
        arr[i]=temp[k];
    }
    return invCount;
}

long long mergeSort(long long arr[], int left, int right){
    long long invCount=0;
    if(right>left){
        int mid=(right+left)/2;
        invCount=mergeSort(arr, left, mid);
```

```
        invCount += mergeSort(arr, mid+1, right);
        invCount += merge(arr, left, mid+1, right);
    }
    return invCount;
}

long long getInversions(long long arr[], int n){
        return mergeSort(arr, 0, n-1);
}
```

Time: O(NlogN) / Space: O(N)

(5) Closest Pair of Points using Divide & Conquer:-

- We are given an array of n points in the plane, and the problem is to find out the closest pair of point in the array
- This problem arises into number of application ex:- air, traffic
- The distance b/w two points p and q.

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

* Brute Force Approach :-

Compute the distance between each pair and return the smallest. This will take $O(n^2)$.

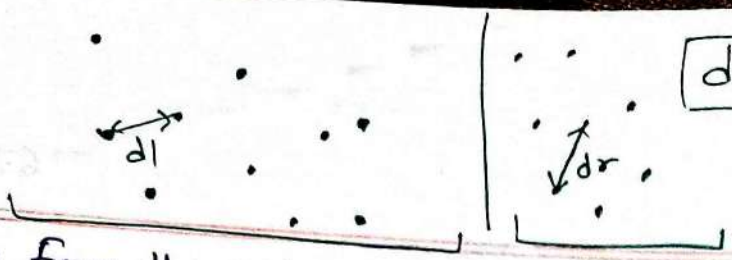
* Using divide & conquer:-

- A pre-processing step, the input array is sorted according to x co-ordinates.

(1) Find middle point in sorted array, i.e $P[n/2]$

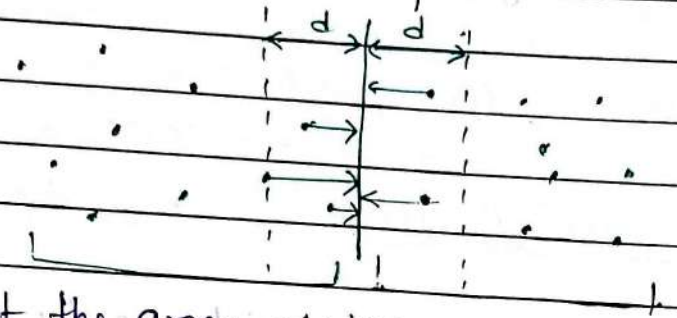
(2) Divide array into two part ($P[0]$ to $P[n/2]$, $P[n/2+1]$ to $P[n-1]$)

(3) Recursively find the smallest distances in both subarrays. let the distance be d_l & d_r, find the min^m of d_l & d_r. let the min^m is d



Date
Page No. 68.

④ From the 3 step, we have an upper bound 'd' of minm distance. Now we need to consider the pairs such that one point in pair is from the left half and other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x co-ordinate 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 co-ordinates it uses O(n) by recursively sorting & merging.

⑥ Find the smallest distance in strip[]. This is tricky. we can find in O(n). it can be proved geometrically that for every point in the strip. we only need to check at most 7 point

⑦ Finally return the minm of d & distance from above step(6).

* Code :—

```
Class Point {
 public:
    int x, y;
};
int compareX(const void *a, const void *b) {
      point *p1 = (point *) a, *p2 = (point *) b;
      return (p1->x - p2->x);
}
int compareY(const void *a, const void *b) {
      point *p1 = (point *) a, *p2 = (point *) b;
      return (p1->y - p2->y);
}
```

```
Float dist(Point p1, Point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*
                (p1.y-p2.y));
}

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 = dis(P[i],P[j]);
    return min;
}

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

Float stripClosest(Point strip[], int size, Float d){
    Float min=d;
    qsort(strip, size, sizeof(Point), compareY);
    for(int i=0; i<size; ++i)
        for(int j=i+1; j<size && (strip[j].y-strip[i].y)
                                <min; ++j)
            if(dist(strip[i], strip[j])<min)
                min = dist(strip[i], strip[j]);
    return min;
}

Float closestUtil(Point P[], int n){
    if(n<=3)
        return(bruteForce(P,n));
    int mid=n/2;
    Point midPoint = P[mid];
    Float dl = closestUtil(P, mid);
```

```
Float dr = closestUtil (P+mid, n-mid);
Float d = min(dl, dr);
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++;
return min(d, stripClosest(strip, j, d));
}
Float closest(Point P[], int n) {
    qsort(P, n, sizeof(point), compareX);
    return closestUtil(P, n);
}

int main() {
    Point P[] = {{2,3}, {12,30}, {40,50}, {5,1}, {12,10}, {3,4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << closest(P, n);
    return 0;
}
```

Output: 1.414214

Time complexity:-

let time complexity is T(n). The above also divide points in two sets and recursively calls for two sets. After dividing, find the strip in O(n), sort the strip in O(nlogn) finally finds the closest points in strip in O(n). ∴ T(n) is

$$T(n) = 2T(n/2) + O(n) + O(n\log n) + O(n)$$

$$T(n) = 2T(n/2) + O(n\log n)$$

$$T(n) = T(n \times \log n \times \log n)$$

$$\boxed{O(n(\log n)^2)}$$

* above approach in O(n log n)

- In the previous approach strip[] was explicitly sorted in every recursive call that made the time complexity O(n (Log n)^2).
- In this, The Idea is to presort all points according to y co-ordinate. Let the sorted array be Py[]. When we make recursive calls, we need to divide points of Py[] also according to vertical line. We can do that by simply processing every point and comparing its x co-ordinate with x-coordinate of the middle line.

Code :-

```
struct Point{
    int x,y;
};

int compareX(const void* a, const void* b){
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x != p2->x)? (p1->x - p2->x) : (p1->y - p2->y);
}

int compareY(const void* a, const void *b){
    point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y != p2->y)? (p1->y - p2->y) : (p1->x - p2->x);
}

float dist(Point p1, Point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y));
}

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;
}
```

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

Float stripclosest(Point strip[], int size, Float d){
        Float min=d;
        for(int i=0; i<size; ++i)
            for(int j=i+1; j<size && (strip[j].y - strip[i].y)
                                    <min; ++j)
                if(dist(strip[i], strip[j])<min)
                    min= dist(strip[i], strip[j]);
        return min;
}

Float ClosestUtil(Point Px[], Point Py[], int n){
        if (n<=3)
            return bruteForce(Px, n);
        int mid=n/2;
        Point midPoint= Px[mid];
        Point Pyl[mid];
        Point Pyr[n-mid];
        int li=0, ri=0;
        for(int i=0; i<n; i++){
            if((Py[i].x< midPoint.x || (Py[i].x== midPoint.x &&
                Py[i].y<midPoint.y)) && li<mid)
                    Pyl[li++]=Py[i];
            else
                Pyr[ri++] =Py[i];
        }
        Float dl= closestUtil(Px, Pyl, mid);
        Float dr= closestUtil(Px+mid, Pyr, n-mid);
        foat d= min(dl, dr);
        Point strip[n];
```

```
int j=0;
for(int i=0; i<n;i++)
    for(abs(Py[i].x-midPoint.x)<d)
        strip[j] = Py[i], j++;
return stripClosest(strip, j, d);
}

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);
    return ClosestUtil(Px, Py, n);
}
```

Time: $T(n)$ = divide + Find strip' + divid Py qeray qeround mid verticle line + Find closest point in strip

$\therefore T(n) = 2T(n/2) + O(n) + O(n) + O(n)$

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

(6) Strassen's Matrix Multiplication:-

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

*Naive Method :-

```
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 :- $O(N^3)$

* Using Divide & Conquer

① Divide matrices A & B in 4 sub-matrices of size N/2 X N/2

② calculate following values recursively.

ae+bg, af+bh, ce+dg & cf+dh.

$(N/2 \times N/2)$ $(N \times N)$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

A B C

- 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

∴ $T(n) = 8T(N/2) + O(N^2)$

- From masters theorem time complexity will be $O(N^3)$ which is same as Naive approach.

* Using Strassen's Method :

- In the above divide & 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. it is similar to divide & conquer, but, the four sub-matrices of result are calculated using following Formula.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} P5+P4+P6-P2 & P1+P2 \\ P3+P4 & P1+P5-P3-P7 \end{bmatrix}$$

where,

$P1 = a(f-h)$	$P3 = (c+d)e$	$P5 = (a+d)(e+h)$
$P2 = (a+b)h$	$P4 = d(g-e)$	$P6 = (b-d)(g+h)$
		$P7 = (a-c)(e+f)$

- Since addition and substraction takes $O(N^2)$ ∴ time complexity
$T(n) = 7T(N/2) + O(N^2)$

- From master's theorem it will be $O(N^{\log 7}) = O(N^{2.8074})$

* Generally Strassan's is not preferred for practical reason—
(1) constant used in strassan's are high & Naive work better
(2) For sparse matrices, there are better method especially for them.
(3) The submatrices in recursion takes extra space
(4) B/c of limited precision of computer arithmetic on non-integer larger error accumulate in strassen's than Naive.

*Code :—

```
#include <bits/stdc++.h>
using namespace std;

int nextpowerof2(int k){
    return pow(2,int(ceil(log2(k))));
}

void display(vector<vector<int>> &matrix, int m, int n){
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(j!=0){
                cout<<" ";
            }
            cout<<matrix[i][j];
        }
        cout<<endl;
    }
}

void add(vector<vector<int>> &A, vector<vector<int>> &B, vector<vector<int>> &C, int size){
    for(int i=0; i<size; i++)
        for(int j=0; j<size; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

```
void sub(vector<vector<int>> &A, vector<vector<int>> &B,
                vector<vector<int>> &C, int size){
    for(int i=0; i<size; i++)
        for(int j=0; j<size; j++)
            C[i][j] = A[i][j] - B[i][j];
}

void strassen(vector<vector<int>> &A, vector<vector<int>> &B,
                vector<vector<int>> &C, int size){
    if(size==1){
        C[0][0] = A[0][0] * B[0][0];
        return;
    }
    else{
        int new_size = size/2;
        vector<int> z(new_size);
        vector<vector<int>>
            a11(new_size,z), a12(new_size,z), a21(new_size,z),
            a22(new_size,z), b11(new_size,z), b12(new_size,z),
            b21(new_size,z), b22(new_size,z), c11(new_size,z),
            c12(new_size,z), c21(new_size,z), c22(new_size,z),
            p1(new_size,z), p2(new_size,z), p3(new_size,z),
            p4(new_size,z), p5(new_size,z), p6(new_size,z),
            p7(new_size,z), aresult(new_size,z) bresult(new_size,z);
        //Dividing matrices into sub matrices:
        for(int i=0; i<new_size; i++){
            for(int j=0; j<new_size; j++){
                a11[i][j] = A[i][j];
                a12[i][j] = A[i][j+new_size];
                a21[i][j] = A[i+new_size][j];
                a22[i][j] = A[i+new_size][j+new_size];
```

```
b11[i][j] = B[i][j];
b12[i][j] = B[i][j+new_size];
b21[i][j] = B[i+new_size][j];
b22[i][j] = B[i+new_size][j+new_size];
}
}
```

//calculating P1 to P7:

```
add(a11, a22, aResult, new_size);
add(b11, b22, bResult, new_size);
Strassen(aResult, bResult, P1, new_size);
```
P1 = (a11 + a22) * (b11 + b22)

```
add(a21, a22, aResult, new_size);
Strassen(aResult, b11, P2, new_size);
```
P2 = (a21 + a22) * b(11)

```
Sub(b12, b22, bResult, new_size);
Strassen(a11, bResult, P3, new_size);
```
P3 = (a11) * (b12 - b22)

```
Sub(b21, b11, bResult, new_size);
Strassen(a22, bResult, P4, new_size);
```
P4 = (a22) * (b21 - b11)

```
add(a11, a12, aResult, new_size);
Strassen(aResult, b22, P5, new_size);
```
P5 = (a11 + a12) * (b22)

```
Sub(a21, a11, aResult, new_size);
Strassen add(b11, b12, bResult, new_size);
Strassen(aResult, bResult, P6, new_size);
```
P6 = (a21 - a11) * (b11 + b12)

```
Sub(a12, a22, aResult, new_size);
add(a21, b22, bResult, new_size);
Strassen(aResult, bResult, P7, new_size);
```
P7 = (a12 - a22) * (b21 + b22)

// Calculating c21, c12, c11 & c22:

```
add(P3, P5, C12, new_size);   // C12 = P3 + P5
add(P2, P4, C21, new_size);   // C21 = P2 + P4
add(P1, P4, aResult, new_size);   // P1 + P4
add(aResult, P7, bResult, new_size);   // P1 + P4 + P7
Sub(bResult, P5, C11, new_size);   // C11 = P1 + P4 + P7 - P5
```

```
add(P1,P3,aResult,new_size); //P1+P3
add(aResult,P6,bResult,new_size); // P1+P3+P6
sub(bResult,P2,C22,new_size);} C22=P1+P3-P2+P6
// Grouping the results obtained in a single matrix:
for(int i=0; i<new_size; i++)
    for(int j=0; j<new_size; j++){
        C[i][j] = C11[i][j];
        C[i][j+new_size] = C12[i][j];
        C[i+new_size][j] = C21[i][j];
        C[i+new_size][j+new_size] = C22[i][j];
    }
}//else end
}

Void Helper(vector<vector<int>>&A, vector<vector<int>>&B,
            int m, int n, int a, int b){
/*Check to see if these matrices are already square and have
*dimension of a power of 2. if not, the matrices must be
*resized and padded with zeros to meet these criteria*/
    int k=max({m,n,a,b});
    int s = nextpowerof2(k);
    vector<int> z(s);
    vector<int>> Aa(s/2), Bb(s/2), Cc(s/2);
    for(unsigned int i=0; i<m; i++)
        for(unsigned int j=0; j<n; j++)
            Aa[i][j] = A[i][j]
    for(unsigned int i=0; i<a; i++)
        for(unsigned int j=0; j<b; j++)
            Bb[i][j]=B[i][j];
    Strassen(Aa,Bb,Cc,s);
```

```
    vector<int> temp1(b);
    vector<vector<int>> c(m, temp1);
    for(unsigned int i=0; i<m; i++)
        for(unsigned int j=0; j<b; j++)
            c[i][j] = c[i][j];
    display(c, m, b);
}

int main(){
    vector<vector<int>> a={{1,2,3},{4,5,6},{7,8,9}};
    vector<vector<int>> b={{9,8,7},{6,5,4},{3,2,1}};
    helper(a, b, 3, 3, 3, 3)
    return 0;
}
```

Output:-

```
30   24   18
84   69   54
138  114  90
```

(7) karatsuba Algorithm for Fast multiplication

- Given two binary strings that represent value of two integers, Find the product of two string.

Ex:-

Input: X = "1100", Y = "1010"
Output: 120

- For simplicity, let the length of two string be same = n.

* Naive Approach:

One by one take all bits of second number & multiply it with all bits of First number. Find add all multiplication. This algorithm takes $O(n^2)$ time.

* Divide and Conquer:

- 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.

$X = X_l * 2^{n/2} + X_r$ [X_l & X_r are left most & right most n/2 bits of X]

$Y = Y_l * 2^{n/2} + Y_r$ [Y_l & Y_r are leftmost & right most n/2 bits of Y]

- The product XY can be written as following.

$$XY = (X_l * 2^{n/2} + X_r) \cdot (Y_l * 2^{n/2} + Y_r)$$
$$= 2^n \cdot X_l Y_l + 2^{n/2} \cdot (X_l Y_r + X_r Y_l) + X_r Y_r$$

- Here, there are four multiplication of size n/2, so we basically divided the problem of size n into four sub-problems of size n/2. but that doesnot help bcz the recurrence will be $T(n) = 4T(n/2) + O(n) = O(n^2)$. The trick part of this algorithm is to change middle two terms to some other form so that only one extra multiplication would be sufficient.

$$X_l Y_r + X_r Y_l = (X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r$$

- so the final value becomes

$$XY = 2^n X_l Y_l + 2^{n/2} * [(X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r] + X_r Y_r$$

- Now Recurrence Relation will be $T(n) = 3T(n/2) + O(n)$
$= O(n^{1.59})$.

Note:- if the length of the inputstring are different and are not even? the we append 0's in the begining. To handle odd length, we put floor(n/2) bits in left half and ceil (n/2) bits in right half. so the expression will be

$$XY = 2^{2\,ceil(n/2)} X_l Y_l + 2^{ceil(n/2)} * [(X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r] + X_r Y_r$$

*code

```
#include<bits/stdc++.h>
using namespace std;
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;
}
string addBitStrings(string first, string second){
    string result;
    int length = makeEqualLength(first, second);
    int carry = 0;
    for(int i=length-1; i>=0; i--){
        int firstBit = first.at(i)-'0';
        int secondBit = second.at(i)-'0';
        int sum = (firstBit^secondBit^carry)+'0';
        result = (char)sum+result;
        carry = (firstBit&secondBit)|(secondBit&carry)|
                (firstBit&carry);
    }
    if (carry) result='1'+result;
    return result;
}
```

```
int multiplySingleBit (string a, string b){
    return (a[0]-'0')*(b[0]-'0'); }

long int multiply (string X, string Y){
    int n = makeEqualLength(X, Y);
    if (n==0) return 0;
    if (n==1) return multiplySingleBit(X, Y);
    int fh = n/2;          //first half floor(n/2)
    int sh = (n-fh);       //second half ceil(n/2)
    string Xl = X.substr(0, fh);
    string Xr = X.substr(fh, sh);
    string Yl = Y.substr(0, fh);
    string Yr = Y.substr(fh, sh);
    long int P1 = multiply(Xl, Yl);
    long int P2 = multiply(Xr, Yr);
    long int P3 = multiply(addBitStrings(Xl, Xr),
                           addBitStrings(Yl, Yr));
    return P1*(1<<(2*sh))+(P3-P1-P2)*(1<<sh)+P2;
}

int main(){
    cout<< multiply("1100","1010");
    cout<< multiply("110","1010");
    cout<< multiply("111","111");
    return 0;
}
```

Output:

120
60
49

Time complexity = $O(n^{\log_2 3}) = O(n^{1.59})$