> Search if word is present in matrix

```
   bool DFS(int i , int j , int index , string&word , vector<vector<char>>&board){

    if (i<0 || i==board.size() || j<0 || j==board[0].size() ||
word[index]!=board[i][j])
        return false ;

    if (index==word.size()-1)
        return true ;

    char temp = board[i][j] ;
    board[i][j] = ' ' ;

    bool tour = DFS(i-1 , j , index+1 , word , board)
        || DFS(i+1 , j , index+1 , word , board)
        || DFS(i , j+1 , index+1 , word , board)
        || DFS(i , j-1 , index+1 , word , board) ;

    board[i][j] = temp ;
    return tour ;
}


bool exist(vector<vector<char>>& board, string word) {

    for (int i =0 ; i<board.size() ; i++)
        for (int j =0 ; j<board[0].size() ; j++)

            if (board[i][j]==word[0] && DFS(i , j , 0 , word , board))
                return true ;

    return false ;
}
```

> If A is area of triangle, S is semiperimeter

   In-radius, 'r' for any triangle = A/s
   Circumradius, R for any triangle = abc/4A
   ex-radius of the equilateral triangle, r1 = A/(s–a)
   Distance between incentre and circumcentre=sqrt(R2-2rR)

> Inverse modulo, compute nCk modp

  nCk%mod = n!%(mod)/(n-k)!%(mod)k!%mod = n!%(mod) * power(fact(n-k), mod-2)%mod + power(fact(k), mod-2)%mod

  inv(a) = inverse modulo a w.r.t. mod
  inv(d) = power(d,mod-2)%mod

```
  long long int power(long long int A,long long int B)
{
    //base case
    long long result=1;
    if(A==0)
    {
        return 0;
    }
    if(B==0)
    {
        return 1;
    }

    if(B%2==0)
    {
        result  =power(A,B/2);
        result = (result* result )%mod;
    }
    else
    {
        result = ((A %mod)* power(A,B-1)%mod)%mod;
    }


    return result%mod;
}
```

Time Complexities of Sorting Algorithms:

Algorithm	Best	Average	Worst
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Radix Sort	Ω(nk)	θ(nk)	O(nk)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)

1> quick

```
 void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

    int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {

        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}


void quickSort(int arr[], int low, int high)  // low is 0, high is size-1
{
    if (low < high)
```

```
    {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

2> Merge

```
void merge(int *Arr, int start, int mid, int end) {

        int temp[end - start + 1];


        int i = start, j = mid+1, k = 0;


        while(i <= mid && j <= end) {
                if(Arr[i] <= Arr[j]) {
                        temp[k] = Arr[i];
                        k += 1; i += 1;
                }
                else {
                        temp[k] = Arr[j];
                        k += 1; j += 1;
                }
        }


        while(i <= mid) {
                temp[k] = Arr[i];
                k += 1; i += 1;
        }


        while(j <= end) {
                temp[k] = Arr[j];
                k += 1; j += 1;
        }
```

```
        for(i = start; i <= end; i += 1) {
                Arr[i] = temp[i - start]
        }
}


void mergeSort(int *Arr, int start, int end) {

        if(start < end) {
                int mid = (start + end) / 2;
                mergeSort(Arr, start, mid);
                mergeSort(Arr, mid+1, end);
                merge(Arr, start, mid, end);
        }
}
```

2> * Definition for binary tree

```
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
```

4> Definition of trie

```
struct Trie{
    int c=0;
    Trie *pt[26]={NULL};
};
```

3> Definition for singly-linked list.

```
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
```

4> Multi child tree

```
struct Node {
     int val;
     vector<Node*> child;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->val = key;
    return temp;
}

use Node*temp=newNode(value)
```

> Max element in every window

```
#define f first
#define s second

vector<int> Solution::slidingMaximum(const vector<int> &A, int B) {
vector<int> ans;
deque<pair<int, int>> dq;
for (int i = 0; i < A.size(); i++) {

    if (!dq.empty() && dq.front().s == i-B)
        dq.pop_front();

    while (!dq.empty() && dq.back().f < A[i])
        dq.pop_back();
```

```
    dq.push_back({A[i], i});

    if (i-B+1 >= 0) ans.push_back(dq.front().f);
}
return ans;
}
```

> Maximum size rectangle in binay matrix

```
int maxHist(int row[])
{

    stack<int> result;

    int top_val;

    int max_area = 0;

    int area = 0;

    int i = 0;
    while (i < C) {

        if (result.empty() || row[result.top()] <= row[i])
            result.push(i++);

        else {

            top_val = row[result.top()];
            result.pop();
            area = top_val * i;

            if (!result.empty())
                area = top_val * (i - result.top() - 1);
            max_area = max(area, max_area);
        }
    }


    while (!result.empty()) {
        top_val = row[result.top()];
        result.pop();
```

```
        area = top_val * i;
        if (!result.empty())
            area = top_val * (i - result.top() - 1);

        max_area = max(area, max_area);
    }
    return max_area;
}


int maxRectangle(int A[][C])
{

    int result = maxHist(A[0]);


    for (int i = 1; i < R; i++) {

        for (int j = 0; j < C; j++)

             if (A[i][j])
                A[i][j] += A[i - 1][j];

        result = max(result, maxHist(A[i]));
    }

    return result;
}
```

> Reverse linkedlist

```
        Node* current = head;
        Node *prev = NULL, *next = NULL;

        while (current != NULL) {

            next = current->next;


            current->next = prev;

            prev = current;
```

```
            current = next;
        }
        head = prev;
```

> KMP

```
    void computeLPSArray(char* pat, int M, int* lps){


    int len = 0;

    lps[0] = 0;


    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {

            if (len != 0) {
                len = lps[len - 1];


            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}


void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
```

```
    int N = strlen(txt);


    int lps[M];


    computeLPSArray(pat, M, lps);

    int i = 0;
    int j = 0;
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }


        else if (i < N && pat[j] != txt[i]) {

            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}
```

> Z algo

```
   void getZarr(string str, int Z[])
{
    int n = str.length();
    int L, R, k;
```

```
    L = R = 0;
    for (int i = 1; i < n; ++i)
    {

        if (i > R)
        {
            L = R = i;


            while (R<n && str[R-L] == str[R])
                R++;
            Z[i] = R-L;
            R--;
        }
        else
        {

            k = i-L;


            if (Z[k] < R-i+1)
                Z[i] = Z[k];


            else
            {

                L = i;
                while (R<n && str[R-L] == str[R])
                    R++;
                Z[i] = R-L;
                R--;
            }
        }
    }
}



   void search(string text, string pattern)
{

    string concat = pattern + "$" + text;
    int l = concat.length();


    int Z[l];
    getZarr(concat, Z);
```

```
    for (int i = 0; i < l; ++i)
    {
        if (Z[i] == pattern.length())
            cout << "Pattern found at index "
                << i - pattern.length() -1 << endl;
    }
}
```

> Rabin Karp

```
   #define d 256

void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;


    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;


    for (i = 0; i < M; i++)
    {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }


    for (i = 0; i <= N - M; i++)
    {

        if ( p == t )
        {
```

```
            bool flag = true;

            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                {
                  flag = false;
                  break;
                }
                  if(flag)
                  cout<<i<<" ";

            }



            if (j == M)
                cout<<"Pattern found at index "<< i<<endl;
        }

        if ( i < N-M )
        {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;

            if (t < 0)
            t = (t + q);
        }
    }
}
```

> Josephus problem

For a general N, find largest m such that 2^m < N. Take t = N-2^m. Then Safe position is 2*t + 1

> 1D random walk, probability to return to 0 if starting at m, and P(going one step away) if p

((1-p)/p)^m

> when 1 unit is divided into n parts, average of

smallest = 1/n(1/n)
2nd small = 1/n(1/n +1/(n-1))
3rd small = 1/n(1/n + 1/(n-1) +1/(n-2))
largest = 1/n(1/n + ...1/2 + 1)

> for randomwalk, probability to return
1D = 1
2D = 1
3D = 0.239
4D = 0.105