**STRING MATCHING KMP**

#include<iostream>

#include<vector>

#include<cstring>

using namespace std;

vector<int> prefix(string p){

int m=p.length();

vector<int> pi(m);

pi[0]=0;

int k=0;

for(int q=1;q<m;q++){

while(k>0&&p[k]!=p[q]){

k=pi[k-1];

}

if(p[k]==p[q]){

k=k+1;

}

pi[q]=k;

}

return pi;

}

void kmp(string t,string p){

int m=p.length();

int n=t.length();

vector<int> pi=prefix(p);

int q=0;

int flag=0;

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

while(q>0&&p[q]!=t[i]){

q=pi[q-1];

}

if(p[q]==t[i]){

q=q+1;

}

if(q==m){

cout<<"match available"<<endl<<i+1-m<<endl<<i<<endl;

q=pi[q];

flag=1;

}

}

if(flag!=1){

cout<<"No match available";

}

}

int main(){

string t,p;

cin>>t;

cin>>p;

kmp(t,p);

}

**SEGMENT INTERSECTION**

#include <iostream>

using namespace std;

struct point

{

int x;

int y;

};

bool onSegment(point p,point q,point r)

{

if(q.x<=max(p.x,r.x)&&q.x>=min(p.x,r.x)&&

q.y<=max(p.y,r.y)&&q.y>=min(p.y,r.y))

return true;

return false;

}

int orientation(point p,point q,point r)

{

int val=(q.y-p.y)\*(r.x-q.x)-

(q.x-p.x)\*(r.y-q.y);

if(val==0)return 0;

return(val>0)?1:2;

}

bool intersect(point p1,point q1,point p2,point q2)

{

int o1=orientation(p1,q1,p2);

int o2=orientation(p1,q1,q2);

int o3=orientation(p2,q2,p1);

int o4=orientation(p2,q2,q1);

if(o1!=o2&&o3!=o4)

return true;

if(o1==0&&onSegment(p1,p2,q1)) return true;

if(o2==0&&onSegment(p1,q2,q1)) return true;

if(o3==0&&onSegment(p2,p1,q2)) return true;

if(o4==0&&onSegment(p2,q1,q2)) return true;

return false;

}

int main()

{

struct point p1={1,1},q1={10,1};

struct point p2={1,2},q2={10,2};

intersect(p1,q1,p2,q2)?cout<<"Intersecting\n":cout<<"Not Intersecting\n";

return 0;

#include <bits/stdc++.h>

using namespace std;

// A point in 2D plane

struct Point

{

int x, y;

};

// A line segment with left as Point

// with smaller x value and right with

// larger x value.

struct Segment

{

Point left, right;

};

// An event for sweep line algorithm

// An event has a point, the position

// of point (whether left or right) and

// index of point in the original input

// array of segments.

struct Event {

int x, y;

bool isLeft;

int index;

Event(int x, int y, bool l, int i) : x(x), y(y), isLeft(l), index(i) {}

// This is for maintaining the order in set.

bool operator<(const Event& e) const {

if(y==e.y)return x<e.x;

return y < e.y;

}

};

// Given three collinear points p, q, r, the function checks if

// point q lies on line segment 'pr'

bool onSegment(Point p, Point q, Point r)

{

if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&

q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))

return true;

return false;

}

// To find orientation of ordered triplet (p, q, r).

// The function returns following values

// 0 --> p, q and r are collinear

// 1 --> Clockwise

// 2 --> Counterclockwise

int orientation(Point p, Point q, Point r)

{

// See https://www.geeksforgeeks.org/orientation-3-ordered-points/

// for details of below formula.

int val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

if (val == 0) return 0; // collinear

return (val > 0)? 1: 2; // clock or counterclock wise

}

// The main function that returns true if line segment 'p1q1'

// and 'p2q2' intersect.

bool doIntersect(Segment s1, Segment s2)

{

Point p1 = s1.left, q1 = s1.right, p2 = s2.left, q2 = s2.right;

// Find the four orientations needed for general and

// special cases

int o1 = orientation(p1, q1, p2);

int o2 = orientation(p1, q1, q2);

int o3 = orientation(p2, q2, p1);

int o4 = orientation(p2, q2, q1);

// General case

if (o1 != o2 && o3 != o4)

return true;

// Special Cases

// p1, q1 and p2 are collinear and p2 lies on segment p1q1

if (o1 == 0 && onSegment(p1, p2, q1)) return true;

// p1, q1 and q2 are collinear and q2 lies on segment p1q1

if (o2 == 0 && onSegment(p1, q2, q1)) return true;

// p2, q2 and p1 are collinear and p1 lies on segment p2q2

if (o3 == 0 && onSegment(p2, p1, q2)) return true;

// p2, q2 and q1 are collinear and q1 lies on segment p2q2

if (o4 == 0 && onSegment(p2, q1, q2)) return true;

return false; // Doesn't fall in any of the above cases

}

// Find predecessor of iterator in s.

set<Event>::iterator pred(set<Event> &s, set<Event>::iterator it) {

return it == s.begin() ? s.end() : --it;

}

// Find successor of iterator in s.

set<Event>::iterator succ(set<Event> &s, set<Event>::iterator it) {

return ++it;

}

// Returns true if any two lines intersect.

int isIntersect(Segment arr[], int n)

{

unordered\_map<string,int> mp; // to note the pair for which intersection is checked already

// Pushing all points to a vector of events

vector<Event> e;

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

e.push\_back(Event(arr[i].left.x, arr[i].left.y, true, i));

e.push\_back(Event(arr[i].right.x, arr[i].right.y, false, i));

}

// Sorting all events according to x coordinate.

sort(e.begin(), e.end(), [](Event &e1, Event &e2) {return e1.x < e2.x;});

// For storing active segments.

set<Event> s;

int ans=0;

// Traversing through sorted points

for (int i=0; i<2\*n; i++)

{

Event curr = e[i];

int index = curr.index;

// If current point is left of its segment

if (curr.isLeft)

{

// Get above and below points

auto next = s.lower\_bound(curr);

auto prev = pred(s, next);

// Check if current point intersects with

// any of its adjacent

bool flag=false;

if (next != s.end() && doIntersect(arr[next->index], arr[index])){

string s=to\_string(next->index+1)+" "+to\_string(index+1);

if(mp.count(s)==0){mp[s]++;ans++;} //if not already checked we can increase count in map

}

if (prev != s.end() && doIntersect(arr[prev->index], arr[index])){

string s=to\_string(prev->index+1)+" "+to\_string(index+1);

if(mp.count(s)==0){mp[s]++;ans++;} //if not already checked we can increase count in map

}

// if same line segment is there then decrease answer as it got increased twice

if(prev != s.end() && next != s.end() && next->index==prev->index)ans--;

// Insert current point (or event)

s.insert(curr);

}

// If current point is right of its segment

else

{

// Find the iterator

auto it=s.find(Event(arr[index].left.x, arr[index].left.y, true, index));

// Find above and below points

auto next = succ(s, it);

auto prev = pred(s, it);

// If above and below point intersect

if (next != s.end() && prev != s.end())

{ string s=to\_string(next->index+1)+" "+to\_string(prev->index+1);

string s1=to\_string(prev->index+1)+" "+to\_string(next->index+1);

if (mp.count(s)==0&&mp.count(s1)==0&&doIntersect(arr[prev->index], arr[next->index]))

ans++;

mp[s]++;

}

// Remove current segment

s.erase(it);

}

}

//print pair of lines having intersection

for(auto &pr:mp){

cout<<"Line: "<<pr.first<<"\n";

}

return ans;

}

int main() {

Segment arr[] = { {{1, 5}, {4, 5}}, {{2, 5}, {10, 1}},{{3, 2}, {10, 3}},{{6, 4}, {9, 4}},{{7, 1}, {8, 1}}};

int n = sizeof(arr)/sizeof(arr[0]);

cout<<"Number of intersection points: "<<isIntersect(arr, n);

return 0;

}

**RABIN KARP**

/\* Following program is a C++ implementation of Rabin Karp

Algorithm given in the CLRS book \*/

#include <bits/stdc++.h>

using namespace std;

// d is the number of characters in the input alphabet

#define d 256

/\* pat -> pattern

txt -> text

q -> A prime number

\*/

void search(char pat[], char txt[], int q)

{

int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for txt

int h = 1;

// The value of h would be "pow(d, M-1)%q"

for (i = 0; i < M - 1; i++)

h = (h \* d) % q;

// Calculate the hash value of pattern and first

// window of text

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

p = (d \* p + pat[i]) % q;

t = (d \* t + txt[i]) % q;

}

// Slide the pattern over text one by one

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

// Check the hash values of current window of text

// and pattern. If the hash values match then only

// check for characters one by one

if (p == t) {

/\* Check for characters one by one \*/

for (j = 0; j < M; j++) {

if (txt[i + j] != pat[j]) {

break;

}

}

// if p == t and pat[0...M-1] = txt[i, i+1,

// ...i+M-1]

if (j == M)

cout << "Pattern found at index " << i

<< endl;

}

// Calculate hash value for next window of text:

// Remove leading digit, add trailing digit

if (i < N - M) {

t = (d \* (t - txt[i] \* h) + txt[i + M]) % q;

// We might get negative value of t, converting

// it to positive

if (t < 0)

t = (t + q);

}

}

}

/\* Driver code \*/

int main()

{

char txt[] = "GEEKS FOR GEEKS";

char pat[] = "GEEK";

// we mod to avoid overflowing of value but we should

// take as big q as possible to avoid the collison

int q = INT\_MAX;

// Function Call

search(pat, txt, q);

return 0;

}

**CONVEX HULL**

#include <iostream>

#include <stack>

#include <stdlib.h>

using namespace std;

struct point

{

int x,y;

};

point p0;

point nextToTop(stack<point> &S)

{

point p=S.top();

S.pop();

point res=S.top();

S.push(p);

return res;

}

void swap(point &p1,point &p2)

{

point temp=p1;

p1=p2;

p2=temp;

}

int distSq(point p1,point p2)

{

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

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

}

int orientation(point p,point q,point r)

{

int val=(q.y-p.y)\*(r.x-q.x)-

(q.x-p.x)\*(r.y-q.y);

if (val==0) return 0;

return(val>0)?1:2;

}

int compare(const void \*vp1,const void \*vp2)

{

point \*p1=(point\*)vp1;

point \*p2=(point\*)vp2;

int o=orientation(p0,\*p1,\*p2);

if (o==0)

return (distSq(p0,\*p2)>=distSq(p0,\*p1))?-1:1;

return(o==2)?-1:1;

}

void convexHull(point points[],int n)

{

int ymin=points[0].y,min = 0;

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

{

int y=points[i].y;

if ((y < ymin) || (ymin == y &&

points[i].x < points[min].x))

ymin = points[i].y, min = i;

}

swap(points[0], points[min]);

p0 = points[0];

qsort(&points[1], n-1, sizeof(point), compare);

int m = 1;

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

{

while (i < n-1 && orientation(p0, points[i], points[i+1]) == 0)

i++;

points[m] = points[i];

m++;

}

if (m < 3) return;

stack<point> S;

S.push(points[0]);

S.push(points[1]);

S.push(points[2]);

for (int i = 3; i < m; i++)

{

while (S.size()>1 && orientation(nextToTop(S), S.top(), points[i]) != 2)

S.pop();

S.push(points[i]);

}

while (!S.empty())

{

point p = S.top();

cout << "(" << p.x << ", " << p.y <<")" << endl;

S.pop();

}

}

int main()

{

point points[] = {{8,5},{7,1},{6,2},{1,3},

{1,1},{4,5},{3,7},{0,7},{4,8},{2,9}};

int n=sizeof(points)/sizeof(points[0]);

convexHull(points, n);

return 0;

}

**MAX SUB ARRAY SUM**

#include<iostream>

#include<climits>

using namespace std;

struct SubArray{

int sum;

int low;

int high;

};

SubArray cross\_sum(int A[],int l,int mid,int r){

int left\_sum=INT\_MIN;

int sum1=0;

int max\_left;

for(int i=mid;i>=l;i--){

sum1=sum1+A[i];

if(sum1>left\_sum){

left\_sum=sum1;

max\_left=i;

}

}

int right\_sum=INT\_MIN;

int sum2=0;

int max\_right;

for(int i=mid+1;i<=r;i++){

sum2=sum2+A[i];

if(sum2>right\_sum){

right\_sum=sum2;

max\_right=i;

}

}

return {left\_sum+right\_sum, max\_left, max\_right};

}

SubArray max\_subarray\_sum(int A[],int l,int r){

if (l==r){

return {A[l],l,r};

}

int mid=(l+r)/2;

int left\_sub\_sum, right\_sub\_sum, cross\_sub\_sum, left\_low, left\_high, right\_low, right\_high, cross\_low, cross\_high;

SubArray left=max\_subarray\_sum(A,l,mid);

SubArray right=max\_subarray\_sum(A,mid+1,r);

SubArray cross=cross\_sum(A,l,mid,r);

if(left.sum>=right.sum && left.sum>=cross.sum){

return left;

}

else if(right.sum>=left.sum && right.sum>=cross.sum){

return right;

}

else{

return cross;

}

}

int main(){

int n;

cin>>n;

int A[n];

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

cin>>A[i];

}

SubArray result=max\_subarray\_sum(A,0,n-1);

cout<<result.sum<<"\n"<<result.low<<"\n"<<result.high;

}

**MATRIX CHAIN MULTIPLICATION**

#include <iostream>  
#include<climits>  
using namespace std;  
void par(int n,int i,int j,int \*s,char&mat)  
{   if (i==j) {  
        cout<<mat++;  
        return;  
    }  
    cout <<"(";  
    par(n,i,\*((s+i\*n)+j),s,mat);  
    par(n,\*((s+i\*n)+j)+1,j,s,mat);  
    cout<<")";  
}  
   
void operations(int arr[],int n)  
{  
    int m[n][n];  
    int s[n][n];  
    for(int i=1;i<n;i++){  
        m[i][i]=0;}  
    for(int i=2;i<n;i++)  
    {  
        for(int j=1;j<n-i+1;j++)   
        {  
            int k=j+i-1;  
            m[j][k]=INT\_MAX;  
            for(int l=j;l<=k-1;l++)   
            {  
                int p=m[j][l]+m[l+1][k]  
                        + arr[j-1]\*arr[l]\*arr[k];  
                if (p<m[j][k])   
                {  
                    m[j][k]=p;  
                    s[j][k]=l;  
                }  
            }  
        }  
    }  
    char mat='A';   
    cout<<m[1][n-1]<<"\n";  
    par(n,1,n-1,(int\*)s,mat);  
     
}  
int main()  
{  
   int n;  
   cin>>n;  
   int d;  
   cin>>d;  
   int arr[d];  
   for(int i=0;i<d;i++){  
       cin>>arr[i];  
   }  
   operations(arr,d);  
}

  if (i == j)

    {

        return 0;

    }

    if (dp[i][j] != -1)

    {

        return dp[i][j];

    }

    dp[i][j] = INT\_MAX;

    for (int k = i; k < j; k++)

    {

        dp[i][j] = min(

            dp[i][j], matrixChainMemoised(p, i, k)

                     + matrixChainMemoised(p, k + 1, j)

                       + p[i - 1] \* p[k] \* p[j]);

    }

    return dp[i][j];

**KNAPSACK**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int knapsack(int W, vector<int>& wt, vector<int>& val, int n) {

vector<vector<int>> K(n + 1, vector<int>(W + 1));

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

for (int w = 0; w <= W; w++) {

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

K[i][w] = K[i - 1][w];

}

}

// Finding the selected items

vector<int> selectedItems(n, 0);

int i = n, w = W;

while (i > 0 && w > 0) {

if (K[i][w] != K[i - 1][w]) {

selectedItems[i - 1] = 1;

w -= wt[i - 1];

}

i--;

}

cout << "Items selected: ";

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

cout << selectedItems[i];

}

cout << endl;

return K[n][W];

}

int main() {

int W, n;

cout << "Bag Capacity: ";

cin >> W;

cout << "Total items: ";

cin >> n;

vector<int> wt(n), val(n);

cout << "Item Weights: ";

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

cin >> wt[i];

}

cout << "Item values: ";

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

cin >> val[i];

}

int maxValue = knapsack(W, wt, val, n);

cout << "Maximum value: " << maxValue << endl;

return 0;

}

int knapSack(int W, int wt[], int val[], int n)

{

*// Base Case*

**if** (n == 0 || W == 0)

**return** 0;

*// If weight of the nth item is more*

*// than Knapsack capacity W, then*

*// this item cannot be included*

*// in the optimal solution*

**if** (wt[n - 1] > W)

**return** knapSack(W, wt, val, n - 1);

*// Return the maximum of two cases:*

*// (1) nth item included*

*// (2) not included*

**else**

**return** max(

val[n - 1]

+ knapSack(W - wt[n - 1], wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

**LCS**

#include<iostream>

using namespace std;

int LCS(string a,string b,int m,int n){

int p[m+1][n+1];

for(int i=0;i<=m;i++){

for(int j=0;j<=n;j++){

if(i==0||j==0){

p[i][j]=0;

}

else if(a[i-1]==b[j-1]){

p[i][j]=p[i-1][j-1]+1;

}

else{

p[i][j]=max(p[i-1][j],p[i][j-1]);

}

}

}

int i=p[m][n];

char seq[i+1];

seq[i]='\0';

int j=m,k=n;

while(j>0&&k>0){

if(a[j-1]==b[k-1]){

seq[i-1]=a[j-1];

j--;

k--;

i--;

}

else if(p[j-1][k]>=p[j][k-1]){

j--;

}

else{

k--;

}

}

cout<<p[m][n];

cout<<"\n"<<seq;

}

int main(){

string a,b;

cin>>a;

cin>>b;

int m=a.size();

int n=b.size();

LCS(a,b,m,n);

}

int lcs(string X, string Y, int m, int n)

{

    if (m == 0 || n == 0)

        return 0;

    if (X[m - 1] == Y[n - 1])

        return 1 + lcs(X, Y, m - 1, n - 1);

    else

        return max(lcs(X, Y, m, n - 1),

                   lcs(X, Y, m - 1, n));

}

int lcs(char\* X, char\* Y, int m, int n,

        vector<vector<int> >& dp)

{

    if (m == 0 || n == 0)

        return 0;

    if (X[m - 1] == Y[n - 1])

        return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

    if (dp[m][n] != -1) {c

        return dp[m][n];

    }

    return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),

                          lcs(X, Y, m - 1, n, dp));

}

**N QUEENS**

#include<iostream>

#include<cstring>

using namespace std;

bool issafe(int \*\*mat, int r, int c, int n){

for(int i=0;i<r;i++){

if(mat[i][c]==1){

return false;

}}

for(int i=r, j=c;i>=0 && j>=0;i--,j--){ //upper left diagonal

if (mat[i][j]==1){

return false;

}

}

for(int i=r,j=c;i>=0&&j<n;i--,j++){ //upper right diagonal

if(mat[i][j]==1){

return false;

}

}

return true;

}

bool place(int \*\*mat, int r, int n){

if(r==n){

return true;

}

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

if (issafe(mat,r,i,n)){

mat[r][i]=1;

if(place(mat,r+1,n)){

return true;

}

mat[r][i]=0;

}

}

return false;

}

int main(){

int n;

cin>>n;

if(n==1){

cout<<1<<endl<<"Completed";

}

if(n>1 && n<4){

cout<<"Not Possible";

}

int \*\*mat=new int\*[n];

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

mat[i]=new int[n];

for(int j=0;j<n;j++){

mat[i][j]=0;

}

}

if(n>=4){

if(place(mat,0,n)){

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

for(int j=0;j<n;j++){

cout<<mat[i][j]<<" ";

}

cout<<endl;

}

cout<<"Completed";

}

else{

cout<<"Not Completed";

}}

}

**SUBSET SUM**

bool flag = 0;

void PrintSubsetSum(int i, int n, int set[], int targetSum,

                    vector<int>& subset)

{

    // targetSum is zero then there exist a

    // subset.

    if (targetSum == 0) {

        // Prints valid subset

        flag = 1;

        cout << "[ ";

        for (int i = 0; i < subset.size(); i++) {

            cout << subset[i] << " ";

        }

        cout << "]";

        return;

    }

    if (i == n) {

        // return if we have reached at the end of the array

        return;

    }

    // Not considering current element

    PrintSubsetSum(i + 1, n, set, targetSum, subset);

    // consider current element if it is less than or equal

    // to targetSum

    if (set[i] <= targetSum) {

        // push the current element in subset

        subset.push\_back(set[i]);

        // Recursive call for consider current element

        PrintSubsetSum(i + 1, n, set, targetSum - set[i],

                       subset);

        // pop-back element after recursive call to restore

        // subsets original configuration

        subset.pop\_back();

    }

}

// Driver code

int main()

{

    // Test case 1

    int set[] = { 1, 2, 1 };

    int sum = 3;

    int n = sizeof(set) / sizeof(set[0]);

    vector<int> subset;

    cout << "Output 1:" << endl;

    PrintSubsetSum(0, n, set, sum, subset);

    cout << endl;

    flag = 0;

    // Test case 2

    int set2[] = { 3, 34, 4, 12, 5, 2 };

    int sum2 = 30;

    int n2 = sizeof(set) / sizeof(set[0]);

    vector<int> subset2;

    cout << "Output 2:" << endl;

    PrintSubsetSum(0, n2, set2, sum2, subset2);

    if (!flag) {

        cout << "There is no such subset";

    }

    return 0;

}

**ROD CUTTING**

#include *<algorithm>*

#include *<iostream>*

**using** **namespace** **std**;

int cutRod(int prices[], int n)

{

int mat[n + 1][n + 1];

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

**for** (int j = 0; j <= n; j++) {

**if** (i == 0 || j == 0) {

mat[i][j] = 0;

}

**else** {

**if** (i == 1) {

mat[i][j] = j \* prices[i - 1];

}

**else** {

**if** (i > j) {

mat[i][j] = mat[i - 1][j];

}

**else** {

mat[i][j] = max(prices[i - 1]

+ mat[i][j - i],

mat[i - 1][j]);

}

}

}

}

}

**return** mat[n][n];

}

int main()

{

int prices[] = { 1, 5, 8, 9, 10, 17, 17, 20 };

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

cout << "Maximum obtained value is "

<< cutRod(prices, n) << endl;

}