**TRIPLE\_$**

**Jalpaiguri Government Engineering College**

**Subash Singha Roy, Shivam Gupta, Suman Rana**

**ACM-ICPC Amritapuri Regional Contest**

**Techniques:**

1. For counting problems, try counting number of

incorrect ways instead of correct ways.

2. Prune Infeasible/Inferior Search Space Early

3. Utilize Symmetries

4. Try solving the problem backwards

5. Binary Search the answer

6. Meet in the middle (Solve left half, Solve right

half, combine)

7. Greedy

8. DP

9. Analyse complexity carefully

10. Reduce the problem to some standard problem

11. Add m when doing modular arithmetic.

12. Carefully analyse reasoning behind adding

small details in the Q.

13. Use exponential search in case of unbounded

search.

14. Check for n=1

15. Overflow ll vs int

16. string s; s+=”x”; //correct but s=s+”x”; //incorrect

17. acos(x) returns degree in radian

18. (ll)v.size(); // typecasting to ll for 0 size

**Integer Limits:**

1. Signed ll max LLONG\_MAX(9223372036854775807)

2. Signed ll min LLONG\_MIN(-9223372036854775808)

3.u\_sign ll max ULLONG\_MAX(18446744073709551615)

**Number Theory:**

2.Every even integer greater than 2 can be

expressed as the sum of 2 primes.

3. For rootn prime method, check for 2, 3 then:

for (i=5; i\*i<=n; i=i+6) n%i and n%(i+2)

4. Number of divisors will be prime only if N=p^x

where p is prime.

5. fib(n+m)=fib(n)fib(m+1)+fib(n-1)fib(m)

6. A number is Fibonacci if and only if one or both

of (5\*n2 + 4) or (5\*n2 – 4) is a perfect square

7. every positive Every positive integer can be

written uniquely as a sum of distinct

non-neighbouring Fibonacci numbers.

8. Matrix multiplication

mul[i][j] += a[i][k]\*b[k][j];

9. Root n under mod p exists only if

n^((p-1)/2) % p = 1

10. Divisibility by 4: last 2 digits divisible by 4

11. Divisibility by 8: last 3 digits divisible by 8

12. Divisibility by 3,9: sum of digits divisible by 3,9

13. Divisibility by 11: alternate (+ve,-ve) digit sum

is divisible by 11

14. Divisibility by 12: divisible by 3 and 4

15. Divisibility by 13: alternating sum in blocks of

3 (L to R) div 13

16. Integral solution of ax+by=c exists if gcd(a,b)

divides c

**Number of divisors:**

If N = p1α1 p2α2 ….. pkαk, then

Number of divisors = Number of ways of selecting zero or more objects from the group of identical objects (α1+1)( α2+1)…(αk+1)

This includes 1 and N also.

All divisors excluding 1 and N are called Proper divisors.

**Sum of divisors:**

If N = p1α1 p2α2 ….. pkαk, then sum of divisors of N is

(1+ p1+ p12+…+ p1α1) × (1 + p2+ p22+…+ p2α2) ….. (1 + pk+ pk2+…+ pkαk)

= https://files.askiitians.com/cdn1/images/2014912-151857875-5628-dede.png

* Number of ways of putting N as a product of two natural numbers is

If n is not a perfect square = ½ (a1 + 1)(a2 + 1) ….. (ak +1)

If n is  a perfect square = ½ [(a1 + 1)(a2 + 1) ….. (ak +1) + 1].

**Chinese Remainder Theorem:**

The **Chinese remainder theorem** is a theorem of

number theory , which states that if one knows the

remainders of the Euclidean division of an integer

*n* by several integers, then one can determine

uniquely the remainder of the division of *n* by the

product of these integers, under the condition that

the divisors are pairwise coprime .

1. LL crt(LL num[], LL rem[], LL k)

2. {

3. LL prod = 1;

4. for (int i = 0; i < k; i++)

5. prod \*= num[i];

6. LL result = 0;

7. for (int i = 0; i < k; i++)

8. {

9. LL pp = prod / num[i];

10. LL inv,y;

11. gcde(pp,num[i],&inv,&y);

12. result += rem[i] \* inv \* pp;

13. }

14. return result % prod;

15. }

For combining wrt a large number, use it 2

numbers at a time.

**Wilson’s theorem(doubt):**

((p-1)!)%p=-1

**Number of solutions to a linear eqn:(doubt):**

LL countSol(LL coeff[], LL start, LL end, LL rhs)

{

// Base case

if (rhs == 0)

return 1;

LL result = 0; // Initialize count of solutions

// One by subtract all smaller or equal

coefficiants and recur

for (LL i=start; i<=end; i++)

if (coeff[i] <= rhs)

result += countSol(coeff, i, end, rhs-coeff[i]);

return result;

}

**Geometry:**

1. Number of rectangles of any size in a square of size n × n is https://www.edudose.com/wp-content/uploads/2016/03/permutation-combination-f-19943.png and number of squares of any size ishttps://www.edudose.com/wp-content/uploads/2016/03/permutation-combination-f-19949.png.

2. Number of rectangles of any size in a rectangle size n × p (n < p) is (np/4) (n + 1) (p + 1) and number of squares of any size is https://www.edudose.com/wp-content/uploads/2016/03/permutation-combination-f-19957.png.

3. Area of regular polygon = ½\*perimeter\*apothem

apothem = a segment that joins the polygon's center to the midpoint of any side that is perpendicular to that side.

4. Area of triangle= ½ \* base \* height

5. Area of trapezoid= ½ \* (base1 + base2)\*height

6. Steps to determine area of irregular polygon:

* List the x and y coordinates of each vertex of the polygon in counterclockwise order. Repeat the coordinates of the first point at the bottom of the list.
* Multiply the x coordinate of each vertex by the y coordinate of the next vertex. Add the results.
* Multiply the y coordinate of each vertex by the x coordinate of the next vertex. Again, add these results.
* Subtract the sum of the second products from the sum of the first products.
* Divide this difference by 2 to get the area of the polygon.

7. Area of regular polygon=((s^2)\*n)/(4\*tan(180/n))

where  
*s*  is the length of any side  
*n*  is the number of sides  
*tan*  is the tangent function calculated in degrees

8. Area of regular polygon=((r^2)\*n\*sin(360/n))/2

*r*  is the radius (circumradius)

9. Area of regular polygon=((a^2)\*n\*tan(180/n))

*a* is the length of the apothem (inradius)

10. two straight lines y = m1x + c1 and y = m2x + c2, then the angle between these two lines is given by tan θ = |(m1 – m2)/ (1 + m1m2)|.

11. If the two lines are a1x + b1y + c1 = 0 and a2x +  b2y + c2 = 0, then the formula becomes tan θ = |(a1b2- b1a2)/(a1a2 + b1b2)|

12. Area of triangle= ½ \* a\*b\*sin(C), where C=angle b/w a and b sides

13. Area = | x1 · y2 + x2 · y3 + x3 · y1 − y1 · x2 − y2 ·

x3 − y3 · x1 | / 2

14. Sine rule: a/SinA = b/SinB = c/SinC

15. Cos rule: a^2=b^2+c^2-2\*b\*c\*cosA

16. Circumradius: R=(abc)/4\*area

17. In-radius: r=4RsinA/2SinB/2\*SinC/2

18. Median = ½ \* sqrt(2b^2 + 2c^2 – a^2)

19. Area of quidrilateral= sqrt((s-a)(s-b)(s-c)(s-d)-abcd\*cos^2(phi)), where AB=a, BC=b, CD=c, DA=d, A+C=a\*phi.

Diagonal AC=sqrt((ac+bd)(ad+bc)/(ab+cd)), BD= sqrt((ac+bd)(ab+cd)/(ad+bc))

20. If two chords AB, CD of a circle intersect at a

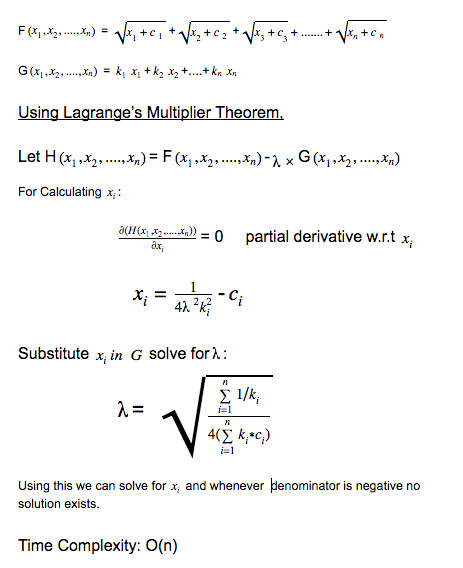
point O (which may lie inside or outside the circle),

then AO.OB=CO.OD

**Trigonometric ratios of half-angles:**

* sin A/2 = √(s-b) (s-c)/bc
* sin B/2 = √(s-b) (s-c)/bc
* sin C/2 = √(s-a) (s-b)/ab
* cos A/2 = √s(s - a)/bc
* cos B/2 = √s(s - b)/ca
* cos C/2 = √s(s - c)/ab
* tan A/2 = √(s - b) (s - c)/s(s - a)
* tan B/2 = √(s - c) (s - a)/s(s - b)
* tan C/2 = √(s - a) (s - b)/s(s - c)

**Lagrange’s Multiplier:**



**Check if Two line segment intersect:**

struct Point {

int x;

int y;

};

// Given three colinear 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 colinear

// 1 --> Clockwise

// 2 --> Counterclockwise

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; // colinear

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

}

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

// and 'p2q2' intersect.

bool doIntersect(Point p1, Point q1, Point p2, Point q2) {

// 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 colinear and p2 lies on segment p1q1

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

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

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

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

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

// p2, q2 and q1 are colinear 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

}

// Driver program to test above functions

int main() {

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

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

doIntersect(p1, q1, p2, q2) ? cout << "Yes\n" : cout << "No\n";

p1 = {10, 0}, q1 = {0, 10};

p2 = {0, 0}, q2 = {10, 10};

doIntersect(p1, q1, p2, q2) ? cout << "Yes\n" : cout << "No\n";

p1 = {-5, -5}, q1 = {0, 0};

p2 = {1, 1}, q2 = {10, 10};

doIntersect(p1, q1, p2, q2) ? cout << "Yes\n" : cout << "No\n";

return 0;

}

**Check recetangle overlap:**

struct Point {

int x, y;

};

bool doOverlap(Point l1, Point r1, Point l2, Point r2) {

// If one rectangle is on left side of other

if (l1.x > r2.x || l2.x > r1.x)

return false;

// If one rectangle is above other

if (l1.y < r2.y || l2.y < r1.y)

return false;

return true;

}

int main() {

Point l1 = {0, 10}, r1 = {10, 0};

Point l2 = {5, 5}, r2 = {15, 0};

if (doOverlap(l1, r1, l2, r2))

printf("Rectangles Overlap");

else

printf("Rectangles Don't Overlap");

return 0;

}

**Circle Intersection:**

int areaOfIntersection(x0, y0, r0, x1, y1, r1) {

var rr0 = r0 \* r0;

var rr1 = r1 \* r1;

var c = Math.sqrt((x1 - x0) \* (x1 - x0) + (y1 - y0) \* (y1 -

y0));

var phi = (Math.acos((rr0 + (c \* c) - rr1) / (2 \* r0 \* c))) \* 2;

var theta = (Math.acos((rr1 + (c \* c) - rr0) / (2 \* r1 \* c))) \* 2;

var area1 = 0.5 \* theta \* rr1 - 0.5 \* rr1 \* Math.sin(theta);

var area2 = 0.5 \* phi \* rr0 - 0.5 \* rr0 \* Math.sin(phi);

return area1 + area2;

}

**Convex Hull(nlogn):**

Point nextToTop(stack < Point > & S) {

Point p = S.top();

S.pop();

Point res = S.top();

S.push(p);

return res;

}

int distSq(Point p1, Point p2) {

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

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

}

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++) {

// Keep removing i while angle of i and i+1 is

same

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 (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();

}

}

**Point in a polygon:**

bool isInside(Point polygon[], int n, Point p) {

if (n < 3) return false;

Point extreme = {

INF,

p.y

};

int count = 0, i = 0;

do {

int next = (i + 1) % n;

if (doIntersect(polygon[i], polygon[next], p,

extreme)) {

if (orientation(polygon[i], p, polygon[next]) ==

0)

return onSegment(polygon[i], p,

polygon[next]);

count++;

}

i = next;

} while (i != 0);

return count & 1; // Same as (count%2 == 1)

}

**Expectations:**

* E(X)= Σ (x1p1, x2p2, …, xnpn), where, x is a random variable with the probability function, f(x), p is the probability of the occurrence, and n is the number of all possible values.
* E(X+Y)=E(X)+E(Y)
* E(XY)=E(X)E(Y)
* E(a+f(X))=a+E(f(X))
* E(aX+b)=aE(X)+b
* E(∑aiXi)=∑ ai E(Xi), Where, ai, (i=1…n) are constants.

**Permutations:**

1.

2. https://www.edudose.com/wp-content/uploads/2016/03/permutation-combination-f-20049.png

* The concept of permutation is used for the arrangement of objects in a specific order i.e. whenever the order is important, permutation is used.
* The total number of permutations on a set of n distinct objects is given by n! and is denoted as nPn = n!
* The total number of permutations on a set of n objects taken r at a time is given by nPr = n!/ (n-r)!
* The number of ways of arranging n objects of which r are the same is given by n!/ r!
* If we wish to arrange a total of n objects, out of which ‘p’ are of one type, q of second type are alike, and r of a third kind are same, then such a computation is done by n!/p!q!r!
* Almost all permutation questions involve putting things in order from a line where the order matters. For example ABC is a different permutation to ACB.
* The number of permutations of n distinct objects when a particular object is not to be considered in the arrangement is given by n-1Pr.
* The number of permutations of n distinct objects when a specific object is to be always included in the arrangement is given by r.n-1Pr-1.
* If we need to compute the number of permutations of n different objects, out of which r have to be selected and each object has the probability of occurring once, twice or thrice… up to r times in any arrangement is given by (n)r.
* Circular permutation is used when some arrangement is to be made in the form of a ring or circle.
* When ‘n’ different or unlike objects are to be arranged in a ring in such a way that the clockwise and anticlockwise arrangements are different, then the number of such arrangements is given by (n – 1)!
* If r things are taken at a time out of n distinct things and arranged along a circle, then the number of ways of doing this is given by nCr(r-1)!.
* If clockwise and anti-clockwise are considered to be the same, then the total number of circular permutations is given by (n-1)!/2.
* If n persons are to be seated around a round table in such a way that no person has similar neighbor then it is given by ½ (n – 1)!
* The number of necklaces formed with n beads of different colors = ½ (n – 1)!
* nP0 =1
* nP1 = n
* nPn = n!/(n-n)! = n! /0! = n!/1 = n!

**Combinatorics:**

* If certain objects are to be arranged in such a way that the order of objects is not important, then the concept of combinations is used.
* The number of combinations of n things taken r (0 < r < n) at a time is given by nCr= n!/r!(n-r)!
* The relationship between combinations and permutations is nCr = nPr/r!
* The number of ways of selecting r objects from n different objects subject to certain condition like:

1. k particular objects are always included =  n-kCr-k

2. k particular objects are never included =  n-kCr

* The number of arrangement of n distinct objects taken r at a time so that k particular objects are

        (i) Always included = n-kCr-k.r!,

        (ii) Never included = n-kCr.r!.

* In order to compute the combination of n distinct items taken r at a time wherein, the chances of occurrence of any item are not fixed and may be one, twice, thrice, …. up to r times is given by **n+r-1Cr**
* If there are m men and n women (m > n) and they have to be seated or accommodated in a row in such a way that no two women sit together then total no. of such arrangements = m+1Cn. m! This is also termed as the Gap Method.
* If we have n different things taken r at a time in form of a garland or necklace, then the required number of arrangements is given by nCr(r-1)!/2.
* If there is a problem that requires n number of persons to be accommodated in such a way that a fixed number say ‘p’ are always together, then that particular set of p persons should be treated as one person. Hence, the total number of people in such a case becomes (n-m+1). Therefore, the total number of possible arrangements is (n-m+1)! m! This is also termed as the String Method.
* Let there be n types of objects with each type containing at least r objects. Then the number of ways of arranging r objects in a row is nr.
* The number of selections from n different objects, taking at least one =  nC1 + nC2 + nC3 + ... + nCn = 2n - 1.
* Total number of selections of zero or more objects from n identical objects isn+1.
* **Selection when both identical and distinct objects are present:**

The number of selections, taking at least one out of a1 + a2 + a3 + ... an + k objects, where a1 are alike (of one kind), a2 are alike (of second kind) and so on ... an are alike (of nth kind), and k are distinct = {[(a1 + 1)(a2 + 1)(a3 + 1) ... (an + 1)]2k} - 1.

* Combination of n different things taken some or all of n things at a time is given by 2n – 1.
* Combination of n things taken some or all at a time when p of the things are alike of one kind, q of the things are alike and of another kind and r of the things are alike of a third kind = [(p + 1) (q + 1)(r + 1)….] – 1.
* The number of ways to select some or all out of (p+q+t) things where p are alike of first kind, q are alike of second kind and the remaining t are different is = (p+1)(q+1)2t – 1.
* Combination of selecting s1 things from a set of n1 objects and s2 things from a set of n2 objects where combination of s1 things and s2 things are independent is given by n1Cs1 x n2Cs2
* Total number of ways in which n identical items which can be distributed among p persons so that each person may get any number of items is n+p-1Cp-1.
* Total number of ways in which n identical items can be distributed among p persons such that each them receive at least one item n-1Cp-1
* **Some results related to nCr**

1. nCr = nCn-r

2. If nCr = nCk, then r = k or n-r = k

3.nCr + nCr-1 = n+1Cr

4.nCr = n/r  n-1Cr-1

5.nCr/nCr-1= (n-r+1)/ r

6. If n is even nCr is greatest for r = n/2

7. If n is odd, nCris greatest for r = (n-1)/2,(n+1)/2

If the number of non negative integral solutions for the equation x1 + x2 + x3 + x4....xr = n isn + r - 1 C r - 1. In this case, value of any variable can be zero.

If the number of positive integral solutions for the equation x1 + x2 + x3 + x4....xr = n is n - 1 C r - 1. In this case, minimum value for any variable is 1.

Permutation Formula

The number of non negative integral solutions for the equation x1 + x2 + x3 + x4....xr = n, given the variable cannot have equal values and minimum value for any variable is 1, is given by,n - 1 P r - 1

Solved Permutation and Combination Problems:

*Example 1:* In how many ways can 10 similar balls be put in 4 distinct boxes?

*Solution:* This question can be understood as finding the number of non-negative integral solutions to the equation w + x + y + z = 10. Using the above mentioned formula, we can get the answer as10 + 4 - 1C4 - 1 = 13C3. Solving this, we get the answer as 286.

*Example 2:* In how many ways can at most 10 similar balls be put in 4 distinct boxes?

*Solution:* This question can be understood as finding the number of non-negative integral solutions to the equation a + b + c + d = 10. So, let us introduce another variable 'e' in the equation to make the total as 10. Now, the equation becomes a + b + c + d + e = 10. Using the above mentioned formula, we can get the answer as10 + 5 - 1C5 - 1 = 14C4. Solving this, we get the answer as 1001. This formula works as the fifth variable taken i.e. 'e' can take any value, so make the sum of all the five items as 10.

*Example 3:* In how many ways can 10 similar balls be put in 4 distinct boxes such that each box contains at least 1 ball?

*Solution:* This question is almost similar to the 1st question with a minor difference. In this question, each box must have at least 1 ball. This question can also be understood as finding the number of positive integral solutions to the equation w + x + y + z = 10, where minimum values of w, x, y and z is 1. So we will give value of 1 to each of these 4 variables. So new equation becomes w + x + y + z = 6. Using the formula given above, we get the answer as 6 + 4 - 1 C 4 - 1 = 9C3= 84. Or we can use the direct formula n - 1 C r - 1. Putting n = 10 and r = 4, we get the answer as 9C3= 84.

**P, Q identical objects and R diff objects:**

Number of ways to select 'zero or more' objects out of 'p + q + r' objects of which p objects are alike of one kind, q objects are alike of second kind and r distinct objects is {(p + 1) (q + 1) 2^r}.

**Formation of Groups:**

* The number of ways in which (m + n) different things can be divided into two groups, one containing m items and the other containing n items is given by

m+nCn = (m+n)!/ m!n!

* In the above case, if m = n i.e. the groups are of same size then the total number of ways of dividing 2n distinct items into two equal groups is given by 2nCn/2!.This can be written as (2n)!/n!n!2!

**Remark:** The result is divided by 2 in order to avoid repetition i.e. false counting.

* The total number of ways of dividing (m + n + p) distinct items into three unequal groups m, n, p is (m + n + p)!/ m!n!p!.
* In the above case, if m = n = p, then the total number of ways reduce to (3n)!/(n!)33!
* The number of ways in which ‘l’ groups of n distinct objects can be formed in such a way that ‘p’ groups are of object n1, q groups of object n2 are given by
* n!/ (n1)!p(n2)!q(p!)(q!)
* If (a + b + c) distinct items are to be divided into 3 groups and then distributed among three persons, then the number of ways of doing this is

(a + b + c)!. 3!/ a!b!c!

**Derangement and Results on Points:**

* If n things are arranged in a row, then the number of ways in which they can be deranged so that r things occupy wrong places while (n-r) things occupy their original places, is

= nCn-r Dr, where

Dr=https://files.askiitians.com/cdn1/images/2014912-15291420-2109-ded.png

* If n things are arranged in a row, the number of ways of deranging them so that none of them occupies its original place, is

= nC0 Dn

= https://files.askiitians.com/cdn1/images/2014912-152959868-7014-ded.png

* If there are n points in plane put of which m (< n) are collinear, then the following results hold good:

1. Total number of different straight lines obtained by joining these n points is

nC2 – mC2 +1

1. Total number of different triangles formed by joining these n points is nC3 – mC3
2. Number of diagonals in polygon of n sides is nC2 – n
3. If m parallel lines in a plane are intersected by a family of other n parallel lines. Then total number of parallelograms so formed is mC2 × nC2
4. Number of triangles formed by joining vertices of convex polygon of n sides is nC3 of which
5. Number of triangles having exactly 2 sides common to the polygon = n
6. Number of triangles having exactly 1 side common to the polygon = n(n-4)
7. Number of triangles having no side common to the polygon = https://files.askiitians.com/cdn1/images/2014912-153937739-5122-dede.png

**Fermat’s Little formula:**

a^(m-1) % m = 1, where m is prime and 1<a<m-1

a^-1=power(a, m-2, m);

This will work when a and m are co-prime.

**Catalan numbers:**

1, 1, 2, 5, 14, 42, 132, 429, 1430,........

c(n) =(1/(n+1)) \* (2nCn);

c(n+1) = Summation(i = 0 to n) [c(i) \* c(n-i)]

**Geometric Progression:**

n-th term = a\*r^(n-1)

a+a\*r+a\*r^2+a\*r^3+…+a\*r^(n-1) = a\*((1-r^n)/(1-r))

**Arithmetic Progression:**

n-th term=a+(n-1)\*d

sum of n-terms = (n/2) \* (2\*a + (n-1)\*d)

**Boost Multiprecision:**

#include <boost/multiprecision/cpp\_int.hpp>

using namespace boost::multiprecision;

int128\_t, int256\_t, int512\_t,int1024\_t,cpp\_int

int128\_t boost\_product(long long A, long long B) {

int128\_t ans = (int128\_t) A \* B;

return ans;

}

**Bitset:**

0-indexed from right

#define M 32

// 00000000000000000000000000000000

bitset<M> bset1;

// 00000000000000000000000000010100

bitset<M> bset2(20);

// 00000000000000000000000000001100

bitset<M> bset3(string("1100"));

set8[1] = 1; set8[4] = set8[1];

*// count function returns number of set bits in bitset*

set8.count();

*// size function returns total number of bits in bitset*

set8.size()

*// test function return 1 if bit is set else returns 0*

set8.test(i)

*// any function returns true, if atleast 1 bit is set*

bset1.any()

*// none function returns true, if none of the bit is set*

bset1.none()

*// sets all bits*

bset.set()

*// makes bset[pos] = b*

bset.set(pos, b)

*// reset all bits*

set8.reset()

*// reset 2nd bit*

set8.reset(2)

*// flips all bits 0<->1*

set8.flip()

*// flips 2nd bit 0<->1*

set8.flip(2)

all logical operators including shifts can be used(&,^ etc)

*// position of first set bit in*

bset.\_Find\_first()

*// Next set bit after index 6 in bset*

bset.\_Find\_next(6)

**Euler’s totient function:**

?(n)=count of numbers in {1,2,3,4,5…n} that are relatively prime with n.

?(n)=n\*Products of all p’s(1-1/p), where p is the prime factors of n.

1. if n is prime ?(n)=n-1

2. ?(n\*m)=?(n)\*?(m), where gcd(n, m)=1

3. ?(n^k)=n^k-n^(k-1), n is prime

4. summation of all ?(p)=n, p is divisor of n

**Policy based data structures:**

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace \_\_gnu\_pbds;

#define fio ios\_base::sync\_with\_stdio(false);cin.tie(NULL);cout.tie(NULL);

typedef tree<*type*, null\_type, less<*type*>, rb\_tree\_tag, tree\_order\_statistics\_node\_update> ordered\_set;

const long double PI=acos(-1.0);

order\_of\_key (k):Number of items strictly smaller than k

\*find\_by\_order(k) : K-th element in a set (zero index).

**Strings(KMP):**

Final algorithm

So we finally can build an algorithm that doesn't perform any string comparisons and only performs O(n) actions.

Here is the final procedure:

We compute the prefix values π[i] in a loop by iterating from i=1 to i=n−1 (π[0] just gets assigned with 0).

To calculate the current value π[i] we set the variable j denoting the length of the best suffix for i−1. Initially j=π[i−1].

Test if the suffix of length j+1 is also a prefix by comparing s[j] and s[i]. If they are equal then we assign π[i]=j+1, otherwise we reduce j to π[j−1] and repeat this step.

If we have reached the length j=0 and still don't have a match, then we assign π[i]=0 and go to the next index i+1.

Implementation

The implementation ends up being surprisingly short and expressive.

vector<int> prefix\_function(string s) {

int n = (int)s.length();

vector<int> pi(n);

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

int j = pi[i-1];

while (j > 0 && s[i] != s[j])

j = pi[j-1];

if (s[i] == s[j])

j++;

pi[i] = j;

}

return pi;

}

This is an online algorithm, i.e. it processes the data as it arrives - for example, you can read the string characters one by one and process them immediately, finding the value of prefix function for each next character. The algorithm still requires storing the string itself and the previously calculated values of prefix function, but if we know beforehand the maximum value M the prefix function can take on the string, we can store only M+1 first characters of the string and the same number of values of the prefix function.

Applications

Search for a substring in a string. The Knuth-Morris-Pratt algorithm

The task is the classical application of the prefix function.

Given a text t and a string s, we want to find and display the positions of all occurrences of the string s in the text t.

For convenience we denote with n the length of the string s and with m the length of the text t.

We generate the string s+#+t, where # is a separator that appears neither in s nor in t. Let us calculate the prefix function for this string. Now think about the meaning of the values of the prefix function, except for the first n+1 entries (which belong to the string s and the separator). By definition the value π[i] shows the longest length of a substring ending in position i that coincides with the prefix. But in our case this is nothing more than the largest block that coincides with s and ends at position i. This length cannot be bigger than n due to the separator. But if equality π[i]=n is achieved, then it means that the string s appears completely in at this position, i.e. it ends at position i. Just do not forget that the positions are indexed in the string s+#+t.

Thus if at some position i we have π[i]=n, then at the position i−(n+1)−n+1=i−2n in the string t the string s appears.

As already mentioned in the description of the prefix function computation, if we know that the prefix values never exceed a certain value, then we do not need to store the entire string and the entire function, but only its beginning. In our case this means that we only need to store the string s+# and the values of the prefix function for it. We can read one character at a time of the string t and calculate the current value of the prefix function.

Thus the Knuth-Morris-Pratt algorithm solves the problem in O(n+m) time and O(n) memory.

Counting the number of occurrences of each prefix

Here we discuss two problems at once. Given a string s of length n. In the first variation of the problem we want to count the number of appearances of each prefix s[0…i] in the same string. In the second variation of the problem another string t is given and we want to count the number of appearances of each prefix s[0…i] in t.

First we solve the first problem. Consider the value of the prefix function π[i] at a position i. By definition it means that in position i the prefix of length π[i] of the string s appears and ends at position i, and there doesn't exists a longer prefix that follows this definition. At the same time shorter prefixes can end at this position. It is not difficult to see, that we have the same question that we already answered when we computed the prefix function itself: Given a prefix of length j that is a suffix ending at position i, what is the next smaller prefix <j that is also a suffix ending at position i. Thus at the position i ends the prefix of length π[i], the prefix of length π[π[i]−1], the prefix π[π[π[i]−1]−1], and so on, until the index becomes zero. Thus we can compute the answer in the following way.

vector<int> ans(n + 1);

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

ans[pi[i]]++;

for (int i = n-1; i > 0; i--)

ans[pi[i-1]] += ans[i];

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

ans[i]++;

Here for each value of the prefix function we first count how many times it occurs in the array π, and then compute the final answers: if we know that the length prefix i appears exactly ans[i] times, then this number must be added to the number of occurrences of its longest suffix that is also a prefix. At the end we need to add 1 to each result, since we also need to count the original prefixes also.

Now let us consider the second problem. We apply the trick from Knuth-Morris-Pratt: we create the string s+#+t and compute its prefix function. The only differences to the first task is, that we are only interested in the prefix values that relate to the string t, i.e. π[i] for i≥n+1. With those values we can perform the exact same computations as in the first task.

The number of different substring in a string

Given a string s of length n. We want to compute the number of different substrings appearing in it.

We will solve this problem iteratively. Namely we will learn, knowing the current number of different substrings, how to recompute this count by adding a character to the end.

So let k be the current number of different substrings in s, and we add the character c to the end of s. Obviously some new substrings ending in c will appear. We want to count these new substrings that didn't appear before.

We take the string t=s+c and reverse it. Now the task is transformed into computing how many prefixes there are that don't appear anywhere else. If we compute the maximal value of the prefix function πmax of the reversed string t, then the longest prefix that appears in s is πmax long. Clearly also all prefixes of smaller length appear in it.

Therefore the number of new substrings appearing when we add a new character c is |s|+1−πmax.

So for each character appended we can compute the number of new substrings in O(n) times, which gives a time complexity of O(n2) in total.

It is worth noting, that we can also compute the number of different substrings by appending the characters at the beginning, or by deleting characters from the beginning or the end.

**Subarray(Kadanes):**

int maxSubArraySum(int a[], int size) {

int max\_so\_far = a[0];

int curr\_max = a[0];

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

curr\_max = max(a[i], curr\_max + a[i]);

max\_so\_far = max(max\_so\_far, curr\_max);

}

return max\_so\_far;

}

int main() {

int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};

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

int max\_sum = maxSubArraySum(a, n);

cout << "Maximum contiguous sum is " << max\_sum;

return 0;

}

**Search for a subarray with a maximum/minimum average:**

This problem lies in finding such a segment a[l,r], such that the average value is maximal:

maxl≤r1r−l+1∑i=lra[i].

Of course, if no other conditions are imposed on the required segment [l,r], then the solution will always be a segment of length 1 at the maximum element of the array. he problem only makes sense, if there are additional restrictions (for example, the length of the desired segment is bounded below).

In this case, we apply the standard technique when working with the problems of the average value: we will select the desired maximum average value by binary search.

To do this, we need to learn how to solve the following subproblem: given the number x, and we need to check whether there is a subarray of array a[] (of course, satisfying all additional constraints of the problem), where the average value is greater than x.

To solve this subproblem, subtract x from each element of array a[]. Then our subproblem actually turns into this one: whether or not there are positive sum subarrays in this array. And we already know how to solve this problem.

Thus, we obtained the solution for the asymptotic O(T(n)logW), where W is the required accuracy, T(n) is the time of solving the subtask for an array of length n (which may vary depending on the specific additional restrictions imposed).

**2-d Kadanes:**

/\* The Key To This Problem:

For every leftBorderIndex and rightBorderIndex of the POSSIBLE maximal rectangle,

we want to know the largest sum we can yield VERTICALLY so that we can

deduce the best topBorderIndex and bottomBorderIndex.

This is why we keep the array of running sums for each row. We are interested

in best vertical value to start and end at (at each attempt of all of the

combinations of leftBorderIndex and rightBorderIndex values).

We know that we can achieve the sum of the Max Contiguous Subarray Sum for the

vertical array if we choose the top bound of topBorderIndex and the lower bound

of bottomBorderIndex.

The video to explain this code is here: https://www.youtube.com/watch?v=-FgseNO-6Gk

\*/

/\*

You can stick this driver function in a class and run the code below

\*/

public static void main(String args[]) {

int matrix[][] = {

{ 6, -5, -7, 4, -4 },

{ -9, 3, -6, 5, 2 },

{ -10, 4, 7, -6, 3 },

{ -8, 9, -3, 3, -7 }

};

Rectangle maxSumRectangle = maxSum(matrix);

System.out.println("Rectangle Sum: " + maxSumRectangle.interiorSum);

System.out.println("Left Index: " + maxSumRectangle.leftBorderIndex);

System.out.println("Right Index: " + maxSumRectangle.rightBorderIndex);

System.out.println("Top Index: " + maxSumRectangle.topBorderIndex);

System.out.println("Bottom Index: " + maxSumRectangle.bottomBorderIndex);

}

private Rectangle maxSum(int matrix[][]) {

/\*

Record the total amount of rows and columns

\*/

int rows = matrix.length;

int cols = matrix[0].length;

/\*

Create an array that will be a "vertical" array and record

the running sums for each row in the each iteration of the

left bound

\*/

int runningRowSums[] = new int[rows];

/\*

This is the max rectangle that we will update along the way

\*/

Rectangle maxRectangle = new Rectangle();

/\*

We will try all left bound plantings from index 0

to index cols - 1

\*/

for (int left = 0; left < cols; left++) {

/\*

We will reset the running row sums all to 0 since

this is a new planting of the left bound

\*/

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

runningRowSums[i] = 0;

}

/\*

For each left bound, we will try all of the right bounds

starting at the left bound we are planted at.

\*/

for (int right = left; right < cols; right++) {

/\*

Add all items in column 'right' to their respective

row's running sum

\*/

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

runningRowSums[i] += matrix[i][right];

}

/\*

Perform Kadane's algorithm on the running sum array

so that we can determine the best top and bottom

bound to choose for the rectangle.

\*/

KadaneResult kadaneResult = kadane(runningRowSums);

/\*

If we notice that this rectangle can achieve a better

maxSum than the best we have done so far then we need

to adjust our new best

\*/

if (kadaneResult.maxSum > maxRectangle.interiorSum) {

/\*

Set a new interiorSum for our maxRectangle

\*/

maxRectangle.interiorSum = kadaneResult.maxSum;

/\*

The left and the right of the maxRectangle become

the 'left' and 'right' where our for loop pointers

are sitting

\*/

maxRectangle.leftBorderIndex = left;

maxRectangle.rightBorderIndex = right;

/\*

Our top and bottom bounds for the max rectangle are

going to be the startIndex and endIndex of the max

subarray region in the 'runningRowSums' sum cache

(respectively).

\*/

maxRectangle.topBorderIndex = kadaneResult.startIndex;

maxRectangle.bottomBorderIndex = kadaneResult.endIndex;

}

}

}

return maxRectangle;

}

/\*

An implementation of Kadane's algorithm that remembers the

start and end of the Max Contiguous Subarray Sum region

in the KadaneResult object returned

This video explains Kadane's algorithm: https://www.youtube.com/watch?v=2MmGzdiKR9Y

\*/

private KadaneResult kadane(int arr[]) {

/\*

The best sum achieved for a region so far

\*/

int bestMaxSoFar = 0;

/\*

maxRegionStart: start index of the max sum region

maxRegionEnd: end index of the max sum region

\*/

int maxRegionStart = -1;

int maxRegionEnd = -1;

int currentStart = 0;

int bestMaxAtThisIndex = 0;

/\*

We will process every

\*/

for (int i = 0; i < arr.length; i++) {

/\*

Add this item to the best subarray achieved at

index i - 1. Then we will decided what to do

after this.

\*/

bestMaxAtThisIndex += arr[i];

/\*

If 'bestMaxAtThisIndex' is < 0 then we will

decide to not extend the best subarray at i - 1.

We will just set the best we can achieve for subarrays

ending at this index i to 0.

The new 'currentStart' to the max subarray region becomes

i + 1

\*/

if (bestMaxAtThisIndex < 0) {

bestMaxAtThisIndex = 0;

currentStart = i + 1;

}

/\*

If the best max (now the best max at this index) beats the

best global max achieved so far then we need to adjust:

'maxRegionStart' becomes the 'currentStart' we were keeping track

of all along.

'maxRegionEnd' becomes the index we are sitting at 'i'.

The 'bestMaxSoFar' becomes the 'bestMaxAtThisIndex'.

\*/

if (bestMaxAtThisIndex > bestMaxSoFar) {

maxRegionStart = currentStart;

maxRegionEnd = i;

bestMaxSoFar = bestMaxAtThisIndex;

}

}

return new KadaneResult(bestMaxSoFar, maxRegionStart, maxRegionEnd);

}

/\*

Holds the result of running Kadan's algorithm

maxSum: the actual sum of the Max Contiguous Subarray Sum region

startIndex: start of Max Contiguous Subarray Sum region

endIndex: end of Max Contiguous Subarray Sum region

\*/

private class KadaneResult {

int maxSum;

int startIndex;

int endIndex;

public KadaneResult(int maxSum, int startIndex, int endIndex) {

this.maxSum = maxSum;

this.startIndex = startIndex;

this.endIndex = endIndex;

}

}

/\*

A rectangle with left, right, top, and bottom bounds. The sum

of all items contained within the rectangle are also recorded

in the 'interiorSum' variable.

\*/

private class Rectangle {

int interiorSum;

int leftBorderIndex;

int rightBorderIndex;

int topBorderIndex;

int bottomBorderIndex;

}

**2-D Prefix Sum:**

void prefixSum2D(int a[][C]) {

int psa[R][C];

psa[0][0] = a[0][0];

// Filling first row and first column

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

psa[0][i] = psa[0][i - 1] + a[0][i];

for (int i = 0; i < R; i++)

psa[i][0] = psa[i - 1][0] + a[i][0];

// updating the values in the cells

// as per the general formula

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

for (int j = 1; j < C; j++)

// values in the cells of new

// array are updated

psa[i][j] = psa[i - 1][j] + psa[i][j - 1] -

psa[i - 1][j - 1] + a[i][j];

}

// displaying the values of the new array

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

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

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

cout << "\n";

}

}

**Range Update and Range Sum:**

ll tree[400005], lazy[400005];

ll query(ll node, ll start, ll end, ll l, ll r) {

if (lazy[node]) {

tree[node] += ((end - start + 1) \* lazy[node]);

if (start != end) {

lazy[2 \* node] += lazy[node];

lazy[2 \* node + 1] += lazy[node];

}

lazy[node] = 0;

}

if (start >= l && end <= r) {

return tree[node];

}

ll mid = (start + end) / 2;

if (r <= mid) {

return query(2 \* node, start, mid, l, r);

} else if (l > mid) {

return query(2 \* node + 1, mid + 1, end, l, r);

}

return query(2 \* node, start, mid, l, r) + query(2 \* node + 1, mid + 1, end, l, r);

}

void update(ll node, ll start, ll end, ll l, ll r, ll val) {

if (lazy[node]) {

tree[node] += ((end - start + 1) \* lazy[node]);

if (start != end) {

lazy[2 \* node] += lazy[node];

lazy[2 \* node + 1] += lazy[node];

}

lazy[node] = 0;

}

if (start >= l && end <= r) {

tree[node] += ((end - start + 1) \* val);

if (start != end) {

lazy[2 \* node] += val;

lazy[2 \* node + 1] += val;

}

return;

}

if (l > end || r < start) {

return;

}

ll mid = (start + end) / 2;

update(2 \* node, start, mid, l, r, val);

update(2 \* node + 1, mid + 1, end, l, r, val);

tree[node] = tree[2 \* node] + tree[2 \* node + 1];

return;

}

**LCA:**

#define MAXN 100000

#define level 18

vector < int > tree[MAXN];

int depth[MAXN];

int parent[MAXN][level];

void dfs(int cur, int prev) {

depth[cur] = depth[prev] + 1;

parent[cur][0] = prev;

for (int i = 0; i < tree[cur].size(); i++) {

if (tree[cur][i] != prev)

dfs(tree[cur][i], cur);

}

}

void precomputeSparseMatrix(int n) {

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

for (int node = 1; node <= n; node++) {

if (parent[node][i - 1] != -1)

parent[node][i] = parent[parent[node][i - 1]][i - 1];

}

}

}

int lca(int u, int v) {

if (depth[v] < depth[u])

swap(u, v);

int diff = depth[v] - depth[u];

for (int i = 0; i < level; i++)

if ((diff >> i) & 1)

v = parent[v][i];

if (u == v)

return u;

for (int i = level - 1; i >= 0; i--)

if (parent[u][i] != parent[v][i]) {

u = parent[u][i];

v = parent[v][i];

}

return parent[u][0];

}

*Calling functions*

memset(parent,-1,sizeof(parent));

depth[0] = 0;

dfs(1,0);

precomputeSparseMatrix(n);

cout<<lca(a, b)<<endl;

**Trie:**

struct TrieNode {

map < char, TrieNode \* > children;

bool endofword;

TrieNode() {

endofword = false;

}

};

void insert(TrieNode \* root, string word) {

TrieNode \* current = root;

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

char ch = word[i];

TrieNode \* node = current - > children[ch];

if (!node) {

node = new TrieNode();

current - > children[word[i]] = node;

}

current = node;

}

current - > endofword = true;

}

bool prefixsearch(TrieNode \* root, string word) {

TrieNode \* current = root;

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

char ch = word[i];

TrieNode \* node = current - > children[ch];

if (!node)

return false;

current = node;

}

return true;

}

int main() {

TrieNode \* root = new TrieNode();

insert(root, "harshita");

insert(root, "harsh");

insert(root, "sharma");

insert(root, "harry");

string p;

cout << "Enter the prefix to be searched :";

cin >> p;

cout << prefixsearch(root, p) << endl;

return 0;

}

**BIT(INV CNT):**

ll BIT[200005];

void update(ll v) {

for (ll x = v; x <= 200000; x += (x & -x)) {

BIT[x]++;

}

return;

}

ll query(ll v) {

ll sum = 0;

for (ll x = v; x > 0; x -= (x & (-x))) {

sum += BIT[x];

}

return sum;

}

// *co-ordinate compression on a[i]*

frr(i, n, 0) {

c += query(a[i]);

update(a[i]);

}

**Bridges:**

const ll maxn = 705;

ll id = 0;

ll vis[maxn], low[maxn], ids[maxn];

vector < ll > adj[maxn];

vector < pr > bridge;

void dfs(ll node, ll parent) {

vis[node] = 1;

id = id + 1;

low[node] = ids[node] = id;

for (ll next: adj[node]) {

if (next == parent) {

continue;

}

if (!vis[next]) {

dfs(next, node);

low[node] = min(low[node], low[next]);

if (ids[node] < low[next]) {

if (node <= next) {

bridge.pb(mp(node, next));

} else {

bridge.pb(mp(next, node));

}

}

} else {

low[node] = min(low[node], ids[next]);

}

}

}

fr(i, 0, n + 1) {

ids[i] = 0;low[i] = 0;adj[i].clear();vis[i] = 0;

}

fr(i, 1, n + 1) {

if (!vis[i]) {

dfs(1, -1);

}

}

**Articulation Point:**

const ll maxn = 10005;

ll id = 0, out = 0;

ll vis[maxn], low[maxn], ids[maxn], isArt[maxn];

vector < ll > adj[maxn];

void dfs(ll root, ll node, ll parent) {

if (parent == root) {

out++;

}

vis[node] = 1;

id = id + 1;

low[node] = ids[node] = id;

for (ll next: adj[node]) {

if (next == parent) {

continue;

}

if (!vis[next]) {

dfs(root, next, node);

low[node] = min(low[node], low[next]);

if (ids[node] <= low[next]) {

isArt[node] = 1;

}

} else {

low[node] = min(low[node], ids[next]);

}

}

}

fr(i, 0, n + 1) {

ids[i] = 0;low[i] = 0;adj[i].clear();vis[i] = 0;isArt[i] = 0;

}

fr(i, 1, n + 1) {

if (!vis[i]) {

out = 0;

dfs(i, i, -1);

if (out > 1) {

isArt[i] = 1;

} else {

isArt[i] = 0;

}

}

}

**SCC:**

#include <bits/stdc++.h>

using namespace std;

#define mod (ll)(998244353)

#define fr(i,a,b) for(ll i=a;i<b;i++)

#define frr(i,a,b) for(ll i=a-1;i>=b;i--)

#define tc(t) int t;cin >>t;while(t--)

ll mult(ll a,ll b, ll p=mod){return ((a%p)\*(b%p))%p;}

ll add(ll a, ll b, ll p=mod){return (a%p + b%p)%p;}

ll fpow(ll n, ll k, ll p = mod) {ll r = 1; for (; k; k >>= 1) {if (k & 1) r = r \* n%p; n = n \* n%p;} return r;}

ll inv(ll a, ll p = mod) {return fpow(a, p - 2, p);}

bool sa(const pair<ll,ll> &a,const pair<ll,ll> &b){return (a.second<b.second);}

bool fd(const pair<ll,ll> &a,const pair<ll,ll> &b){return (a.first>b.first);}

bool sd(const pair<ll,ll> &a,const pair<ll,ll> &b){return (a.second>b.second);}

ll dx[4]={0,0,1,-1};

ll dy[4]={1,-1,0,0};

bool valid(ll x,ll y,ll n,ll m){

if(0<=x && x<n && 0<=y && y<m){

return true;

}

return false;

}

ll id = 0, scc = 0;

ll low[100005], ids[100005], onstack[100005], cost[100005], vis[100005], dp[100005], scc\_cost[100005], scc\_id[100005];

stack < ll > s;

vector < pr > edge;

vector < ll > adj[100005], adj1[100005];

// scc find

void dfs(ll node) {

id = id + 1;

ids[node] = low[node] = id;

onstack[node] = 1;

s.push(node);

for (ll next: adj[node]) {

if (ids[next] == -1) {

dfs(next);

}

if (onstack[next]) {

low[node] = min(low[node], low[next]);

}

}

if (low[node] == ids[node]) {

scc++;

while (!s.empty()) {

ll at = s.top();

s.pop();

onstack[at] = false;

low[at] = ids[node];

scc\_cost[scc] += cost[at];

scc\_id[at] = scc;

if (node == at) {

break;

}

}

}

}

int main() {

ll n, m, x, y, ind;

cin >> n >> m;

fr(i, 1, n + 1) {

cin >> cost[i];

}

fr(i, 1, m + 1) {

cin >> x >> y;

edge.pb(mp(x, y));

adj[x].pb(y);

}

fr(i, 1, n + 1) {

ids[i] = -1;

}

fr(i, 1, n + 1) {

if (ids[i] == -1) {

dfs(i);

}

}

fr(i, 0, edge.size()) {

x = scc\_id[edge[i].ff];

y = scc\_id[edge[i].ss];

if (x != y) {

adj1[x].pb(y);

}

}

rr;

}

**Queries of nCr%p in O(1):**

const int N = 1000001;

ll factorialNumInverse[N + 1];

ll naturalNumInverse[N + 1];

ll fact[N + 1];

void InverseofNumber(ll p) {

naturalNumInverse[0] = naturalNumInverse[1] = 1;

for (int i = 2; i <= N; i++)

naturalNumInverse[i] = naturalNumInverse[p % i] \* (p - p / i) % p;

}

void InverseofFactorial(ll p) {

factorialNumInverse[0] = factorialNumInverse[1] = 1;

for (int i = 2; i <= N; i++)

factorialNumInverse[i] = (naturalNumInverse[i] \* factorialNumInverse[i - 1]) % p;

}

void factorial(ll p) {

fact[0] = 1;

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

fact[i] = (fact[i - 1] \* i) % p;

}

}

ll Binomial(ll N, ll R, ll p) {

ll ans = ((fact[N] \* factorialNumInverse[R]) %

p \* factorialNumInverse[N - R]) % p;

return ans;

}

int main() {

ll p = 1000000007;

InverseofNumber(p);

InverseofFactorial(p);

factorial(p);

cout << Binomial(15, 4, p) << endl;

return 0;

}

**Topological Sort(minimum):**

n=no of nodes, m=no of edges

vector < ll > v, adj[n + 5];

priority\_queue < ll, vector < ll > , greater < ll >> q;

ll indeg[n + 5] = {0};

fr(i, 1, m + 1) {

cin >> x >> y;

adj[x].pb(y);

indeg[y]++;

}

fr(i, 1, n + 1) {

if (indeg[i] == 0) {

q.push(i);

}

}

while (!q.empty()) {

node = q.top();

q.pop();

nc++;

v.pb(node);

for (ll next: adj[node]) {

indeg[next]--;

if (!indeg[next]) {

q.push(next);

}

}

}

if (nc == n) {

fr(i, 0, v.size()) {

cout << v[i] << " ";

}

cout << "\n";

} else {

cout << "Not possible.\n";

}

**M-Candies:**

ll dp[105][100005];

void solve() {

ll n = 0, m = 0, i = 0, j = 0, k = 0, c = 0, l = 0, r = 0, p = 1e9 + 7, q = 0, x = 0, y = 0, z = 0, flag = 0, sum = 0;

ll a[105] = {0};

string s, t;

cin >> n >> k;

dp[0][0] = 1;

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

cin >> a[i];

dp[i][0] = 1;

}

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

for (j = 1; j <= k; j++) {

dp[i][j] = (dp[i - 1][j] + dp[i][j - 1]) % p;

if (j - a[i] - 1 >= 0)

dp[i][j] = (p + dp[i][j] - dp[i - 1][j - a[i] - 1]) % p;

}

}

cout << dp[n][k] << endl;

}

**1-d Knapsack:**

int main() {

int n, W;

cin >> n >> W;

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

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

cin >> wt[i] >> val[i];

int sum\_val = 0;

for (int x: val)

sum\_val += x;

vector < ll > dp(sum\_val + 1, INF);

dp[0] = 0;

for (int item = 0; item < n; ++item)

for (int valrd = sum\_val - val[item]; valrd >= 0; --valrd)

dp[valrd + val[item]] = min(dp[valrd + val[item]], dp[valrd] + wt[item]);

ll answer = 0;

for (int i = 0; i <= sum\_val; ++i)

if (dp[i] <= W)

answer = max(answer, (ll) i);

cout << answer;

}

**DP with Segment tree:**

vector < ll > tree(4 \* 200005, 0);

ll query(ll v, ll l, ll r, ll tl, ll tr) {

if (tl > r or tr < l)

return -1;

if (tl >= l and tr <= r)

return tree[v];

ll tm = (tl + tr) / 2;

return max(query(2 \* v, l, r, tl, tm), query(2 \* v + 1, l, r, tm + 1, tr));

}

void update(ll v, ll tl, ll tr, ll pos, ll new\_val) {

if (tl == tr) {

tree[v] = new\_val;

} else {

ll tm = (tl + tr) / 2;

if (pos <= tm)

update(v \* 2, tl, tm, pos, new\_val);

else

update(v \* 2 + 1, tm + 1, tr, pos, new\_val);

tree[v] = max(tree[v \* 2], tree[v \* 2 + 1]);

}

}

void solve() {

vector < ll > dp(200005, 0);

ll n = 0, m = 0, i = 0, j = 0, k = 0, c = 0, l = 1, r = 0, p = 0, q = 0, x = 0, y = 0, z = 0, flag = 0, sum = 0;

ll a[200005] = {0}, h[200005] = {0};

cin >> n;

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

cin >> h[i];

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

cin >> a[i];

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

x = query(1, 1, h[i], 1, n);

dp[h[i]] = max(dp[h[i]], a[i] + x);

update(1, 1, n, h[i], dp[h[i]]);

}

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

z = max(dp[i], z);

cout << z << endl;

}

**DP- Bitmasking:**

const ll p = 1e9 + 7;

vector < ll > v[25];

ll n;

ll memo[(1 << 22)];

ll dp(ll mask) {

ll x = 0;

x = \_\_builtin\_popcount(mask);

if (x == n)

return 1;

if (memo[mask] != -1)

return memo[mask];

if (memo[mask] == -1)

memo[mask] = 0;

for (auto it: v[x + 1])

if ((mask | (1 << it)) != mask)

memo[mask] = (memo[mask] + dp(mask | (1 << it))) % p;

return memo[mask];

}

void solve() {

vector < pair < ll, ll >> vp;

map < ll, ll > mp;

set < ll > st;

multiset < ll > mst;

ll m = 0, i = 0, j = 0, k = 0, c = 0, l = 0, r = 0, p = 0, q = 0, x = 0, y = 0, z = 0, flag = 0, sum = 0;

ll a[25][25] = {0}, b[300005] = {0}, ans[300005];

string s, t;

memset(memo, -1, sizeof(memo));

cin >> n;

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

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

cin >> a[i][j];

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

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

if (a[i][j] == 1)

v[i + 1].push\_back(j);

cout << dp(0) << endl;

}

**Matrix Exponentiation:**

C++program to find value of f(n) where f(n)

is defined as

F(n) = F(n - 1) + F(n - 2) + F(n - 3), n >= 3

Base Cases:

F(0) = 0, F(1) = 1, F(2) = 1

// A utility function to multiply two matrices

// a[][] and b[][]. Multiplication result is

// stored back in b[][]

void multiply(int a[3][3], int b[3][3]) {

// Creating an auxiliary matrix to store elements

// of the multiplication matrix

int mul[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

mul[i][j] = 0;

for (int k = 0; k < 3; k++)

mul[i][j] += a[i][k] \* b[k][j];

}

}

// storing the multiplication result in a[][]

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

a[i][j] = mul[i][j]; // Updating our matrix

}

// Function to compute F raise to power n-2.

int power(int F[3][3], int n) {

int M[3][3] = {

{1,1,1},

{1,0,0},

{0,1,0}

};

// Multiply it with initial values i.e with

// F(0) = 0, F(1) = 1, F(2) = 1

if (n == 1)

return F[0][0] + F[0][1];

power(F, n / 2);

multiply(F, F);

if (n % 2 != 0)

multiply(F, M);

// Multiply it with initial values i.e with

// F(0) = 0, F(1) = 1, F(2) = 1

return F[0][0] + F[0][1];

}

// Return n'th term of a series defined using below

// recurrence relation.

// f(n) is defined as

// f(n) = f(n-1) + f(n-2) + f(n-3), n>=3

// Base Cases :

// f(0) = 0, f(1) = 1, f(2) = 1

int findNthTerm(int n) {

int F[3][3] = {

{1,1,1},

{1,0,0},

{0,1,0}

};

//Base cases

if (n == 0)

return 0;

if (n == 1 || n == 2)

return 1;

return power(F, n - 2);

}

**Ternary Search:**

ll a[100005], n, k;

ll find(ll l, ll r) {

ll x = 0;

fr(i, 0, n) {

if (a[i] < l) {

x += l - a[i];

}

if (a[i] > r) {

x += a[i] - r;

}

}

return x;

}

ll ternary(ll m) {

ll l = 0, r = max\_diff, ans = inf;

ll m1, m2, x1, x2;

while (l <= r) {

m1 = l + (r - l) / 3;

m2 = r - (r - l) / 3;

x1 = find(m1, m1 + m);

x2 = find(m2, m2 + m);

if (x1 < x2) {

r = m2 - 1;

ans = min(ans, x1);

} else if (x2 < x1) {

l = m1 + 1;

ans = min(ans, x2);

} else {

l = m1 + 1;

r = m2 - 1;

ans = min(ans, x1);

}

}

return ans;

}

int main() {

fio

ll l, r, m, x;

cin >> n >> k;

fr(i, 0, n) {

cin >> a[i];

}

sort(a, a + n);

l = 0;

r = max\_diff;

while (l <= r) {

m = (l + r) / 2;

x = ternary(m);

if (x > k) {

l = m + 1;

} else {

r = m - 1;

}

}

cout << l << "\n";

rr;

}

**Euler Path:**

vector < ll > adj[30], path;

ll in [30], out[30];

void dfs(ll node) {

while (out[node] != 0) {

ll next = adj[node][--out[node]];

dfs(next);

}

path.pb(node);

}

int main() {

fio

tc(t) {

ll n, f = 0, l;

ll st = 0, en = 0;

bool ok = false;

string s;

fr(i, 0, 30) {

adj[i].clear();

out[i] = 0; in [i] = 0;

}

path.clear();

cin >> n;

fr(i, 0, n) {

cin >> s;

l = s.length();

adj[(ll) s[0] - 97].pb((ll) s[l - 1] - 97); in [(ll) s[l - 1] - 97]++;

out[(ll) s[0] - 97]++;

}

fr(i, 0, 26) {

if (((out[i] - in [i]) > 1) || (( in [i] - out[i]) > 1)) {

f = 1;

break;

} else if ((out[i] - in [i]) == 1) {

st++;

} else if (( in [i] - out[i]) == 1) {

en++;

}

}

if ((en == 0 && st == 0) || (en == 1 && st == 1)) {

ok = true;

}

if (!ok || f == 1) {

cout << "The door cannot be opened.\n";

continue;

}

st = 0;

fr(i, 0, 26) {

if ((out[i] - in [i]) == 1) {

st = i;

break;

}

if (out[i] > 0) {

st = i;

}

}

dfs(st);

if (path.size() == n + 1) {

cout << "Ordering is possible.\n";

} else {

cout << "The door cannot be opened.\n";

}

}

rr;

}

**Euler path/circuit: path->every edge exactly once**

Euler path in undirected graph:

All vertices have even degree except or 2 have odd

degrees, and vertex with non zero degree in same

component.

Euler Circuit in undirected graph:

All vertices have even degree and vertex with non

zero degree in same component.

Euler circuit in directed graph:

All no zero degree vertices are a part of a single

strongly connected component and indegree and

outdegree of all vertices is same.

Euler path in directed graph:

At most one vertex has ( out-degree ) − ( in-degree ) =

1, at most one vertex has (in-degree) − (out-degree)

= 1, every other vertex has equal in-degree and

out-degree, and all of its vertices with nonzero

degree belong to a single connected component of

the underlying undirected graph.

**Hierholzer’s algorithm for directed graph:**

void printCircuit(vector < vector < int > > adj) {

unordered\_map < int, int > edge\_count;

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

edge\_count[i] = adj[i].size();

}

if (!adj.size())

return;

stack < int > curr\_path;

vector < int > circuit;

curr\_path.push(0);

int curr\_v = 0;

while (!curr\_path.empty()) {

if (edge\_count[curr\_v]) {

curr\_path.push(curr\_v);

int next\_v = adj[curr\_v].back();

edge\_count[curr\_v]--;

adj[curr\_v].pop\_back();

curr\_v = next\_v;

} else {

circuit.push\_back(curr\_v);

curr\_v = curr\_path.top();

curr\_path.pop();

}

}

for (int i = circuit.size() - 1; i >= 0; i--) {

cout << circuit[i];

if (i)

cout << " -> ";

}

}

**Mo’s Algorithm:**

const int N = 2e5 + 5;

const int M = 1e6 + 5;

struct data {

int l; int r; int idx; long long store\_ans;

};

int n, q, blocksz = 1000;

int a[N];

data queries[N];

long long freq[M];

long long ans = 0;

bool comp(data & d1, data & d2) {

int blocka = d1.l / blocksz;

int blockb = d2.l / blocksz;

if (blocka < blockb)

return true;

else if (blocka == blockb)

return (d1.r < d2.r) ^ (blocka % 2);

else

return false;

}

bool comp2(data & d1, data & d2) {

return d1.idx < d2.idx;

}

void update(long long k, int sign) //Sign 1 = Add, -1

= Remove {

if (sign == 1) {

ans -= freq[k] \* freq[k] \* k;

freq[k]++;

ans += freq[k] \* freq[k] \* k;

} else {

ans -= freq[k] \* freq[k] \* k;

freq[k]--;

ans += freq[k] \* freq[k] \* k;

}

}

void calcmo() {

int moleft = 1;

int moright = 0;

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

int r = queries[i].r;

int l = queries[i].l;

while (moright < r) {

moright++;

update(a[moright], 1);

}

while (moright > r) {

update(a[moright], -1);

moright--;

}

while (moleft < l) {

update(a[moleft], -1);

moleft++;

}

while (moleft > l) {

moleft--;

update(a[moleft], 1);

}

queries[i].store\_ans = ans;

}

}

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

cin >> queries[i].l >> queries[i].r;

queries[i].idx = i;

}

sort(queries + 1, queries + q + 1, comp);

calcmo();

sort(queries + 1, queries + q + 1, comp2);

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

cout << queries[i].store\_ans << endl;

**SOS DP:**

const ll K = 22;

ll dp[(1 << K) + 5][K]; //dp[mask][i] represents the

f(x) for all x - subset of mask(ie some operations over all subset of mask, eg: sum(arr[x]))

//changes in first i elements only

//initialise dp

//memset(dp,0,sizeof(dp));

//dp[mask][0]=f(mask)->no changes

for (ll i = 1; i <= n; i++) {

dp[arr[i]][0] = arr[i];

}

//for optimised approach use i&1,(i+1)&1

for (ll i = 1; i <= K; i++) {

for (ll mask = 0; mask < (1 << (K)); mask++) {

if (mask & (1 << (K - i))) {

//instead of +,anything like &,or

sth..could be there

dp[mask][i] = dp[mask][i - 1] + dp[mask ^ (1 << (K - i))][i - 1];

} else {

dp[mask][i] = dp[mask][i - 1];

}

}

}

**HLD:**

vector < int > parent, depth, heavy, head, pos;

int cur\_pos;

int dfs(int v, vector < vector < int >>

const & adj) {

int size = 1;

int max\_c\_size = 0;

for (int c: adj[v]) {

if (c != parent[v]) {

parent[c] = v, depth[c] = depth[v] + 1;

int c\_size = dfs(c, adj);

size += c\_size;

if (c\_size > max\_c\_size)

max\_c\_size = c\_size, heavy[v] = c;

}

}

return size;

}

int decompose(int v, int h, vector < vector < int >>

const & adj) {

head[v] = h, pos[v] = cur\_pos++;

if (heavy[v] != -1)

decompose(heavy[v], h, adj);

for (int c: adj[v]) {

if (c != parent[v] && c != heavy[v])

decompose(c, c, adj);

}

}

void init(vector < vector < int >>

const & adj) {

int n = adj.size();

parent = vector < int > (n);

depth = vector < int > (n);

heavy = vector < int > (n, -1);

head = vector < int > (n);

pos = vector < int > (n);

cur\_pos = 0;

dfs(0, adj);

decompose(0, 0, adj);

}

int query(int a, int b) {

int res = 0;

for (; head[a] != head[b]; b = parent[head[b]]) {

if (depth[head[a]] > depth[head[b]])

swap(a, b);

int cur\_heavy\_path\_max = segment\_tree\_query(pos[head[b]], pos[b]);

res = max(res, cur\_heavy\_path\_max);

}

if (depth[a] > depth[b])

swap(a, b);

int last\_heavy\_path\_max = segment\_tree\_query(pos[a], pos[b]);

res = max(res, last\_heavy\_path\_max);

return res;

}

**Python Syntax:**

1 ) from fractions import gcd # gcd(a,b)

2) raw\_input() # returns string , int()

3) map(int,raw\_input(),split()) # for full int array

4) sorted(A)

5 ) Fast I/O

from sys import stdin, stdout

stdin.readline() # in place of raw\_input()

stdout.write(str())

6) List functions , let A be list

A.count(‘apple’) # returns count

A.reverse() , len(A)

A.sort()

A.pop(),A.append()

del A[l:r] # deletes from (l,r) inclusive

7) Sets

A={1,2,3,2} # will remove duplicates

A=set(‘vinit’) # A will contain

{‘v’,’i’,’n’,’t’}

a-b,a|b,a&b

8) dictionary

A={‘a’:23,’b’:34}

list(A) # will contain list of keys

A=[‘a’,’b’]

A.get(key\_name) # returns None if not

found

# don’t use A[key\_name]

9) from collections import deque

A=deque(list\_name)

append(),popleft()

10) import bisect

A =[1,3,4,5,5,5,6] # must be sorted

bisect.bisect(A,5) # returns index after all

5’s , i.e 6

bisect.bisect\_left(A,5) #lower\_bound

index , i.e 3

11)

from operator import itemgetter, attrgetter

class node:

def \_\_init\_\_(self, L, R):

self.L = L

self.R = R

def \_\_repr\_\_(self):

return repr((self.L, self.R))

A=sorted(A, key=attrgetter('L'))

12)bin(ans).count("1") #counts number of set bits

in binary representation of ans

13)2d array input

arr=[[] for i in xrange(100005)]

for i in xrange(n):

arr[i]= map(int, raw\_input().split())

14)initialising and creating 2d array

dp=[[0]\*K for i in range(N)]

1d array: a=[0]\*N

15)array of map

arr3=list( {} for i in xrange(1005) )

if x in arr3[i]:

arr3[i][x]+=1

else:

arr3[i][x]=1

16)

Dict functions:clear(),get(key),pop(key),

17. fast pow: pow(x,y) or pow(x, y, mod)

18. Array input:

x = list(map(int, input().split()))

print(x)