**ICPC REFERENCE NOTEBOOK**

**V\_R\_Stalwarts**

**[University of Petroleum and Energy Studies, Dehradun]**

**MACROS**

**#define** s(n) scanf("%d",&n)

**#define** sc(n) scanf("%c",&n)

**#define** sl(n) scanf("%ld",&n)

**#define** sll(n) scanf("%lld",&n)

**#define** sf(n) scanf("%lf",&n)

**#define** ssp(n) scanf("%[^\n]%\*c",n)

**#define** prt(x) printf("%d\n",x);

**#define** plt(x) printf("%lld\n",x);

**#define** INF 0x3f3f3f3f

**#define** EPS 1e-12

**#define** bitcount \_\_builtin\_popcount

**#define** gcd \_\_gcd

**#define** forall(i,a,b) **for**(**int** i=a;i<b;i++)

**#define** foreach(v, c) **for**( **typeof**( (c).begin()) v = (c).begin(); v != (c).end(); ++v)

**#define** all(a) a.begin(), a.end()

**#define** rall(a) a.rbegin(),a.rend()

**#define** in(a,b) ( (b).find(a) != (b).end())

**#define** pb push\_back

**#define** fill(a,v) memset(a, v, **sizeof** a)

**#define** sz(a) ((**int**)(a.size()))

**#define** mp make\_pair

**#define** dot(a,b) ((conj(a)\*(b)).X)

**#define** cross(a,b) ((conj(a)\*(b)).imag())

**#define** normalize(v) ((v)/length(v))

**#define** rotate(p,about,theta) ((p-about)\*exp(point(0,theta))+about)

**#define** pointEqu(a,b) (comp(a.X,b.X)==0 && comp(a.Y,b.Y)==0)

**#define** maX(a,b) ( (a) > (b) ? (a) : (b))

**#define** miN(a,b) ( (a) < (b) ? (a) : (b))

**#define** checkbit(n,b) ( (n >> b) & 1)

**#define** strjoin( x, y ) x **##** y

**#define** DREP(a) sort(all(a)); a.erase(unique(all(a)),a.end())

**#define** INDEX(arr,ind) (lower\_bound(all(arr),ind)-arr.begin())

**typedef** pair<**int**, **int**> ii;

**typedef** vector<ii> vii;

**typedef** vector<**int**> vi;

**typedef** stringstream ss;

**typedef** vector<string> vs;

**typedef** vector<**double**> vd;

**typedef** vector<vector<**int**> > vvi;

**typedef** **long** **long** ll;

**typedef** **long** **double** ld;

//g++ abc.cpp –o abc -DDEBUG

**#ifdef** DEBUG

**#define** trace3(x,y,z) cerr<<\_\_FUNCTION\_\_<<":"<<\_\_LINE\_\_<<": "#x" = "<<x<<" | "#y" = "<<y<<" | "#z" = "<<z<<endl;

**#else**

**#define** trace3(x,y,z)

**#endif**

**FUNCTIONS**

**FastRead**

//ios::sync\_with\_stdio(false);

//#include<cstdio>

**inline** **long** **long** **int** **frl**()

{

**register** **long** **long** **int** c=**getchar**();

**long** **long** **int** x=0LL, neg=0LL; //Bool neg is better?

**for**(; ((c<48LL || c>57LL) && c != '-' && c!=EOF); c = **getchar**());

**if**(c==EOF) **return** EOF;

**if**(c=='-')

{

neg = 1LL;

c = **getchar**();

}

**for**(; c>47LL && c<58LL ; c = **getchar**()) {

x = (x<<1LL) + (x<<3LL) + c - 48LL;

}

**if**(neg) x = -x;

**return** x;

}

**Power/GCD/LCM**

ll power(ll a, ll b) {

ll r = 1;

**while**(b) {

**if**(b & 1) r = r \* a ;

a = a \* a;

b >>= 1;

}

**return** r;

}

ll **gcd**(ll a,ll b){

**return** b==0?a:gcd(b,a%b);

}

ll **lcm**(ll a,ll b){

**return** (a/gcd(a,b))\*b;

nCr

**long** **long** C(**int** n, **int** r)

{

**if**(r>n) **return** 0;

**if**(r > n / 2) r = n - r; // because C(n, r) == C(n, n - r)

**long** **long** ans = 1;

**int** i;

**for**(i = 1; i <= r; i++)

{

ans \*= n - r + i;

ans /= i;

}

**return** ans;

}

**Tobinary**

**int** tobinary(bitset<8>x)

{

string mystring=x.to\_string<**char**,std::string::traits\_type,std::string::allocator\_type>();

**return** atoi(mystring.c\_str());

}

**ab mod T**

/\* Iterative Function to calculate (x^n)%p in O(logy) \*/

**int** **power**(**int** x, **unsigned** **int** y, **int** p)

{

**int** res = 1; // Initialize result

x = x % p; // Update x if it is more than or equal to p

**while** (y > 0)

{

// If y is odd, multiply x with result

**if** (y & 1)

res = (res\*x) % p;

// y must be even now

y = y>>1; // y = y/2

x = (x\*x) % p;

}

**return** res;

}

**Modulo Arithmetic**

**#include**<iostream>

**#include**<cstdio>

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

**int** **fast\_pow**(**long** **long** base, **long** **long** n,**long** **long** M)

{

**if**(n==0)

**return** 1;

**if**(n==1)

**return** base;

**long** **long** halfn=fast\_pow(base,n/2,M);

**if**(n%2==0)

**return** ( halfn \* halfn ) % M;

**else**

**return** ( ( ( halfn \* halfn ) % M ) \* base ) % M;

}

**int** **findMMI\_fermat**(**int** n,**int** M)

{ **return** fast\_pow(n,M-2,M);

}

**int** **main**()

{

**long** **long** fact[100001];

fact[0]=1;

**int** i=1;

**int** MOD=1000000007;

**while**(i<=100000)

{

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

i++;

}

**while**(1)

{

**int** n,r;

printf("Enter n: ");

scanf(" %d",&n);

printf("Enter r: ");

scanf(" %d",&r);

**long** **long** numerator,denominator,mmi\_denominator,ans;

numerator=fact[n];

denominator=(fact[r]\*fact[n-r])%MOD;

mmi\_denominator=findMMI\_fermat(denominator,MOD);

ans=(numerator\*mmi\_denominator)%MOD;

printf("%lld\n",ans);

}

**return** 0;

}

**Base Conversion**

**int** a,b; **char** sa[10000]; **char** sb[10000];

**void** rev(**char** s[]) {

**int** l=strlen(s);

**for**(**int** i=0; i<l-1-i; i++) swap(s[i],s[l-1-i]);

}

**void** multi(**char** s[], **int** k) {

**int** i, c=0, d;

**for**(i=0;s[i];i++)

{

d=(s[i]-’0’)\*k+c;

c=d/b; d%=b;

s[i]=’0’+d;

}

**while**(c)

{

s[i]=’0’+(c%b); i++;

c/=b;

}

s[i]=’\0’;

}

**void** add(**char** s[], **int** k) {

**int** i, c=k, d;

**for**(i=0;s[i];i++)

{

d=(s[i]-’0’)+c;

c=d/b; d%=b;

72

s[i]=’0’+d;

}

**while**(c)

{

s[i]=’0’+(c%b); i++;

c/=b;

}

s[i]=’\0’;

}

**void** trans(**char** s[]) {

**int** i;

**for**(i=0;s[i];i++)

{

**char**& c=s[i];

**if**(c>=’A’ && c<=’Z’) c=’0’+10+(c-’A’);

**if**(c>=’a’ && c<=’z’) c=’0’+36+(c-’a’);

}

}

**void** itrans(**char** s[]) {

**int** i;

**for**(i=0;s[i];i++)

{

**char**& c=s[i]; **int** d=c-’0’;

**if**(d>=10 && d<=35) c=’A’+(d-10);

**if**(d>=36) c=’a’+(d-36);

}

}

**int** main() {

**int** q; cin>>q;

**int** i,j;

**while**(q)

{

q--;

cin>>a>>b>>sa; sb[0]=’0’; sb[1]=’\0’;

cout<<a<<" "<<sa<<endl;

trans(sa);

**for**(i=0;sa[i];i++)

{

multi(sb, a);

add(sb, sa[i]-’0’);

}

rev(sb);

itrans(sb);

cout<<b<<" "<<sb<<endl;

cout<<endl;

}

**return** 0;

}

**Outline: O(n log n) algorithm for The Longest Increasing Subseq**

set<**int**> st;

set<**int**>::iterator it;

...

st.clear();

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

{

st.insert(a[i]); it=st.find(a[i]);

it++; **if**(it!=st.end()) st.erase(it);

}

cout<<st.size()<<endl;

Outline: O(nm) algorithm **for** the LCS with O(n) sapce

**int** m[2][1000]; // instead of [1000][1000]

**for**(i=M; i>=0; i--)

{

ii = i&1;

**for**(j=N; j>=0; j--)

{

**if**(i==M || j==N) { m[ii][j]=0; **continue**; }

**if**(s1[i]==s2[j]) m[ii][j] = 1+m[1-ii][j+1];

**else** m[ii][j] = max(m[ii][j+1], m[1-ii][j]);

}

}

cout<<m[0][0]; // if you want m[x][y], write m[x&1][y];

**Manacher’s Algo O(2n)**

// Transform S into T.

// For example, S = "abba", T = "^#a#b#b#a#$".

// ^ and $ signs are sentinels appended to each end to avoid bounds checking

string preProcess(string s) {

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

**if** (n == 0) **return** "^$";

string ret = "^";

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

ret += "#" + s.substr(i, 1);

ret += "#$";

**return** ret;

}

string longestPalindrome(string s) {

string T = preProcess(s);

**int** n = T.length();

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

**int** C = 0, R = 0;

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

**int** i\_mirror = 2\*C-i; // equals to i' = C - (i-C)

P[i] = (R > i) ? min(R-i, P[i\_mirror]) : 0;

// Attempt to expand palindrome centered at i

**while** (T[i + 1 + P[i]] == T[i - 1 - P[i]])

P[i]++;

// If palindrome centered at i expand past R,

// adjust center based on expanded palindrome.

**if** (i + P[i] > R) {

C = i;

R = i + P[i];

}

}

// Find the maximum element in P.

**int** maxLen = 0;

**int** centerIndex = 0;

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

**if** (P[i] > maxLen) {

maxLen = P[i];

centerIndex = i;

}

}

**delete**[] P;

**return** s.substr((centerIndex - 1 - maxLen)/2, maxLen);

}

**KMP O(n)**

**#include** <cstdio>

**#include** <cstring>

**#include** <time.h>

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

**#define** MAX\_N 100010

**char** T[MAX\_N], P[MAX\_N]; // T = text, P = pattern

**int** b[MAX\_N], n, m; // b = back table, n = length of T, m = length of P

**void** **naiveMatching**() {

**for** (**int** i = 0; i < n; i++) { // try all potential starting indices

**bool** found = **true**;

**for** (**int** j = 0; j < m && found; j++) // use boolean flag `found'

**if** (i + j >= n || P[j] != T[i + j]) // if mismatch found

found = **false**; // abort this, shift starting index i by +1

**if** (found) // if P[0 .. m - 1] == T[i .. i + m - 1]

printf("P is found at index %d in T\n", i);

} }

**void** **kmpPreprocess**() { // call this before calling kmpSearch()

**int** i = 0, j = -1; b[0] = -1; // starting values

**while** (i < m) { // pre-process the pattern string P

**while** (j >= 0 && P[i] != P[j]) j = b[j]; // if different, reset j using b

i++; j++; // if same, advance both pointers

b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4

} } // in the example of P = "SEVENTY SEVEN" above

**void** **kmpSearch**() { // this is similar as kmpPreprocess(), but on string T

**int** i = 0, j = 0; // starting values

**while** (i < n) { // search through string T

**while** (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j using b

i++; j++; // if same, advance both pointers

**if** (j == m) { // a match found when j == m

printf("P is found at index %d in T\n", i - j);

j = b[j]; // prepare j for the next possible match

} } }

**int** **main**() {

strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN");

strcpy(P, "SEVENTY SEVEN");

n = (**int**)strlen(T);

m = (**int**)strlen(P);

//if the end of line character is read too, uncomment the line below

//T[n-1] = 0; n--; P[m-1] = 0; m--;

printf("T = '%s'\n", T);

printf("P = '%s'\n\n", P);

naiveMatching();

printf("KMP\n");

kmpPreprocess();

kmpSearch();

printf("String Library\n");

**char** \*pos = strstr(T, P);

**while** (pos != NULL) {

printf("P is found at index %d in T\n", pos - T);

pos = strstr(pos + 1, P);

}

**return** 0;

}

**Subset Sum Problem O(sum\*n)**

// Returns true if there is a subset of set[] with sun equal to given sum

**bool** isSubsetSum(**int** set[], **int** n, **int** sum)

{

// The value of subset[i][j] will be true if there is a

// subset of set[0..j-1] with sum equal to i

**bool** subset[sum+1][n+1];

// If sum is 0, then answer is true

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

subset[0][i] = **true**;

// If sum is not 0 and set is empty, then answer is false

**for** (**int** i = 1; i <= sum; i++)

subset[i][0] = **false**;

// Fill the subset table in bottom up manner

**for** (**int** i = 1; i <= sum; i++)

{

**for** (**int** j = 1; j <= n; j++)

{

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

**if** (i >= set[j-1])

subset[i][j] = subset[i][j] ||

subset[i - set[j-1]][j-1];

}

}

/\* // uncomment this code to print table

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

{

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

printf ("%4d", subset[i][j]);

printf("\n");

} \*/

**return** subset[sum][n];

}

**int** main()

{

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

**int** sum = 9;

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

**if** (isSubsetSum(set, n, sum) == **true**)

printf("Found a subset with given sum");

**else**

printf("No subset with given sum");

**return** 0;

}

**Prime Sieve of Eratosthenes**

**#include** <bitset>

**#include** <cmath>

**#include** <cstdio>

**#include** <map>

**#include** <vector>

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

**typedef** **long** **long** ll;

**typedef** vector<**int**> vi;

**typedef** map<**int**, **int**> mii;

ll \_sieve\_size;

bitset<10000010> bs; // 10^7 should be enough for most cases

vi primes; // compact list of primes in form of vector<int>

// first part

**void** **sieve**(ll upperbound) { // create list of primes in [0..upperbound]

\_sieve\_size = upperbound + 1; // add 1 to include upperbound

bs.set(); // set all bits to 1

bs[0] = bs[1] = 0; // except index 0 and 1

**for** (ll i = 2; i <= \_sieve\_size; i++) **if** (bs[i]) {

// cross out multiples of i starting from i \* i!

**for** (ll j = i \* i; j <= \_sieve\_size; j += i) bs[j] = 0;

primes.push\_back((**int**)i); // also add this vector containing list of primes

} } // call this method in main method

**bool** **isPrime**(ll N) { // a good enough deterministic prime tester

**if** (N <= \_sieve\_size) **return** bs[N]; // O(1) for small primes

**for** (**int** i = 0; i < (**int**)primes.size(); i++)

**if** (N % primes[i] == 0) **return** **false**;

**return** **true**; // it takes longer time if N is a large prime!

} // note: only work for N <= (last prime in vi "primes")^2

// second part

vi **primeFactors**(ll N) { // remember: vi is vector of integers, ll is long long

vi factors; // vi `primes' (generated by sieve) is optional

ll PF\_idx = 0, PF = primes[PF\_idx]; // using PF = 2, 3, 4, ..., is also ok

**while** (N != 1 && (PF \* PF <= N)) { // stop at sqrt(N), but N can get smaller

**while** (N % PF == 0) { N /= PF; factors.push\_back(PF); } // remove this PF

PF = primes[++PF\_idx]; // only consider primes!

}

**if** (N != 1) factors.push\_back(N); // special case if N is actually a prime

**return** factors; // if pf exceeds 32-bit integer, you have to change vi

}

// third part

ll **numPF**(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

**while** (N != 1 && (PF \* PF <= N)) {

**while** (N % PF == 0) { N /= PF; ans++; }

PF = primes[++PF\_idx];

}

**if** (N != 1) ans++;

**return** ans;

}

ll **numDiffPF**(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

**while** (N != 1 && (PF \* PF <= N)) {

**if** (N % PF == 0) ans++; // count this pf only once

**while** (N % PF == 0) N /= PF;

PF = primes[++PF\_idx];

}

**if** (N != 1) ans++;

**return** ans;

}

ll **sumPF**(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

**while** (N != 1 && (PF \* PF <= N)) {

**while** (N % PF == 0) { N /= PF; ans += PF; }

PF = primes[++PF\_idx];

}

**if** (N != 1) ans += N;

**return** ans;

}

ll **numDiv**(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1; // start from ans = 1

**while** (N != 1 && (PF \* PF <= N)) {

ll power = 0; // count the power

**while** (N % PF == 0) { N /= PF; power++; }

ans \*= (power + 1); // according to the formula

PF = primes[++PF\_idx];

}

**if** (N != 1) ans \*= 2; // (last factor has pow = 1, we add 1 to it)

**return** ans;

}

ll **sumDiv**(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1; // start from ans = 1

**while** (N != 1 && (PF \* PF <= N)) {

ll power = 0;

**while** (N % PF == 0) { N /= PF; power++; }

ans \*= ((ll)**pow**((**double**)PF, power + 1.0) - 1) / (PF - 1); // formula

PF = primes[++PF\_idx];

}

**if** (N != 1) ans \*= ((ll)**pow**((**double**)N, 2.0) - 1) / (N - 1); // last one

**return** ans;

}

ll **EulerPhi**(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = N; // start from ans = N

**while** (N != 1 && (PF \* PF <= N)) {

**if** (N % PF == 0) ans -= ans / PF; // only count unique factor

**while** (N % PF == 0) N /= PF;

PF = primes[++PF\_idx];

}

**if** (N != 1) ans -= ans / N; // last factor

**return** ans;

}

**int** **main**() {

// first part: the Sieve of Eratosthenes

sieve(10000000); // can go up to 10^7 (need few seconds)

printf("%d\n", isPrime(2147483647)); // 10-digits prime

printf("%d\n", isPrime(136117223861LL)); // not a prime, 104729\*1299709

// second part: prime factors

vi res = primeFactors(2147483647); // slowest, 2147483647 is a prime

**for** (vi::iterator i = res.begin(); i != res.end(); i++) printf("> %d\n", \*i);

res = primeFactors(136117223861LL); // slow, 2 large pfactors 104729\*1299709

**for** (vi::iterator i = res.begin(); i != res.end(); i++) printf("# %d\n", \*i);

res = primeFactors(142391208960LL); // faster, 2^10\*3^4\*5\*7^4\*11\*13

**for** (vi::iterator i = res.begin(); i != res.end(); i++) printf("! %d\n", \*i);

//res = primeFactors((ll)(1010189899 \* 1010189899)); // "error"

//for (vi::iterator i = res.begin(); i != res.end(); i++) printf("^ %d\n", \*i);

// third part: prime factors variants

printf("numPF(%d) = %lld\n", 50, numPF(50)); // 2^1 \* 5^2 => 3

printf("numDiffPF(%d) = %lld\n", 50, numDiffPF(50)); // 2^1 \* 5^2 => 2

printf("sumPF(%d) = %lld\n", 50, sumPF(50)); // 2^1 \* 5^2 => 2 + 5 + 5 = 12

printf("numDiv(%d) = %lld\n", 50, numDiv(50)); // 1, 2, 5, 10, 25, 50, 6 divisors

printf("sumDiv(%d) = %lld\n", 50, sumDiv(50)); // 1 + 2 + 5 + 10 + 25 + 50 = 93

printf("EulerPhi(%d) = %lld\n", 50, EulerPhi(50)); // 20 integers < 50 are relatively prime with 50

**return** 0;

}

**Segment Tree**

**#include** <cmath>

**#include** <cstdio>

**#include** <vector>

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

**typedef** vector<**int**> vi;

**class** SegmentTree { // the segment tree is stored like a heap array

**private**: vi st, A; // recall that vi is: typedef vector<int> vi;

**int** n;

**int** **left** (**int** p) { **return** p << 1; } // same as binary heap operations

**int** **right**(**int** p) { **return** (p << 1) + 1; }

**void** **build**(**int** p, **int** L, **int** R) { // O(n log n)

**if** (L == R) // as L == R, either one is fine

st[p] = L; // store the index

**else** { // recursively compute the values

build(left(p) , L , (L + R) / 2);

build(right(p), (L + R) / 2 + 1, R );

**int** p1 = st[left(p)], p2 = st[right(p)];

st[p] = (A[p1] <= A[p2]) ? p1 : p2;

} }

**int** **rmq**(**int** p, **int** L, **int** R, **int** i, **int** j) { // O(log n)

**if** (i > R || j < L) **return** -1; // current segment outside query range

**if** (L >= i && R <= j) **return** st[p]; // inside query range

// compute the min position in the left and right part of the interval

**int** p1 = rmq(left(p) , L , (L+R) / 2, i, j);

**int** p2 = rmq(right(p), (L+R) / 2 + 1, R , i, j);

**if** (p1 == -1) **return** p2; // if we try to access segment outside query

**if** (p2 == -1) **return** p1; // same as above

**return** (A[p1] <= A[p2]) ? p1 : p2; } // as as in build routine

**int** **update\_point**(**int** p, **int** L, **int** R, **int** idx, **int** new\_value) {

// this update code is still preliminary, i == j

// must be able to update range in the future!

**int** i = idx, j = idx;

// if the current interval does not intersect

// the update interval, return this st node value!

**if** (i > R || j < L)

**return** st[p];

// if the current interval is included in the update range,

// update that st[node]

**if** (L == i && R == j) {

A[i] = new\_value; // update the underlying array

**return** st[p] = L; // this index

}

// compute the minimum pition in the

// left and right part of the interval

**int** p1, p2;

p1 = update\_point(left(p) , L , (L + R) / 2, idx, new\_value);

p2 = update\_point(right(p), (L + R) / 2 + 1, R , idx, new\_value);

// return the pition where the overall minimum is

**return** st[p] = (A[p1] <= A[p2]) ? p1 : p2;

}

**public**:

**SegmentTree**(**const** vi &\_A) {

A = \_A; n = (**int**)A.size(); // copy content for local usage

st.assign(4 \* n, 0); // create large enough vector of zeroes

build(1, 0, n - 1); // recursive build

}

**int** **rmq**(**int** i, **int** j) { **return** rmq(1, 0, n - 1, i, j); } // overloading

**int** **update\_point**(**int** idx, **int** new\_value) {

**return** update\_point(1, 0, n - 1, idx, new\_value); }

};

**int** **main**() {

**int** arr[] = { 18, 17, 13, 19, 15, 11, 20 }; // the original array

vi A(arr, arr + 7); // copy the contents to a vector

SegmentTree st(A);

printf(" idx 0, 1, 2, 3, 4, 5, 6\n");

printf(" A is {18,17,13,19,15, 11,20}\n");

printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // answer = index 2

printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // answer = index 5

printf("RMQ(3, 4) = %d\n", st.rmq(3, 4)); // answer = index 4

printf("RMQ(0, 0) = %d\n", st.rmq(0, 0)); // answer = index 0

printf("RMQ(0, 1) = %d\n", st.rmq(0, 1)); // answer = index 1

printf("RMQ(0, 6) = %d\n", st.rmq(0, 6)); // answer = index 5

printf(" idx 0, 1, 2, 3, 4, 5, 6\n");

printf("Now, modify A into {18,17,13,19,15,100,20}\n");

st.update\_point(5, 100); // update A[5] from 11 to 100

printf("These values do not change\n");

printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // 2

printf("RMQ(3, 4) = %d\n", st.rmq(3, 4)); // 4

printf("RMQ(0, 0) = %d\n", st.rmq(0, 0)); // 0

printf("RMQ(0, 1) = %d\n", st.rmq(0, 1)); // 1

printf("These values change\n");

printf("RMQ(0, 6) = %d\n", st.rmq(0, 6)); // 5->2

printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // 5->4

printf("RMQ(4, 5) = %d\n", st.rmq(4, 5)); // 5->4

**return** 0;

}

**Fenwick Tree**

**#include** <iostream>

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

**#define** LOGSZ 17

**int** tree[(1<<LOGSZ)+1];

**int** N = (1<<LOGSZ);

// add v to value at x

**void** **set**(**int** x, **int** v) {

**while**(x <= N) {

tree[x] += v;

x += (x & -x);

}

}

// get cumulative sum up to and including x

**int** get(**int** x) {

**int** res = 0;

**while**(x) {

res += tree[x];

x -= (x & -x);

}

**return** res;

}

// get largest value with cumulative sum less than or equal to x;

// for smallest, pass x-1 and add 1 to result

**int** **getind**(**int** x) {

**int** idx = 0, mask = N;

**while**(mask && idx < N) {

**int** t = idx + mask;

**if**(x >= tree[t]) {

idx = t;

x -= tree[t];

}

mask >>= 1;

}

**return** idx;

}

**String Trie**

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <string.h>

**#include** <stdbool.h>

**#define** ARRAY\_SIZE(a) **sizeof**(a)/**sizeof**(a[0])

// Alphabet size (# of symbols)

**#define** ALPHABET\_SIZE (26)

// Converts key current character into index

// use only 'a' through 'z' and lower case

**#define** CHAR\_TO\_INDEX(c) ((**int**)c - (**int**)'a')

// trie node

**struct** TrieNode

{

**struct** TrieNode \*children[ALPHABET\_SIZE];

// isLeaf is true if the node represents

// end of a word

**bool** isLeaf;

};

// Returns new trie node (initialized to NULLs)

**struct** TrieNode \***getNode**(**void**)

{

**struct** TrieNode \*pNode = NULL;

pNode = (**struct** TrieNode \*)**malloc**(**sizeof**(**struct** TrieNode));

**if** (pNode)

{

**int** i;

pNode->isLeaf = **false**;

**for** (i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

}

**return** pNode;

}

// If not present, inserts key into trie

// If the key is prefix of trie node, just marks leaf node

**void** **insert**(**struct** TrieNode \*root, **const** **char** \*key)

{

**int** level;

**int** length = strlen(key);

**int** index;

**struct** TrieNode \*pCrawl = root;

**for** (level = 0; level < length; level++)

{

index = CHAR\_TO\_INDEX(key[level]);

**if** (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

// mark last node as leaf

pCrawl->isLeaf = **true**;

}

// Returns true if key presents in trie, else false

**bool** **search**(**struct** TrieNode \*root, **const** **char** \*key)

{

**int** level;

**int** length = strlen(key);

**int** index;

**struct** TrieNode \*pCrawl = root;

**for** (level = 0; level < length; level++)

{

index = CHAR\_TO\_INDEX(key[level]);

**if** (!pCrawl->children[index])

**return** **false**;

pCrawl = pCrawl->children[index];

}

**return** (pCrawl != NULL && pCrawl->isLeaf);

}

**int** **main**()

{

// Input keys (use only 'a' through 'z' and lower case)

**char** keys[][8] = {"the", "a", "there", "answer", "any",

"by", "bye", "their"};

**char** output[][32] = {"Not present in trie", "Present in trie"};

**struct** TrieNode \*root = getNode();

// Construct trie

**int** i;

**for** (i = 0; i < ARRAY\_SIZE(keys); i++)

insert(root, keys[i]);

// Search for different keys

printf("%s --- %s\n", "the", output[search(root, "the")] );

printf("%s --- %s\n", "these", output[search(root, "these")] );

printf("%s --- %s\n", "their", output[search(root, "their")] );

printf("%s --- %s\n", "thaw", output[search(root, "thaw")] );

**return** 0;

}

**Suffix Array**

// Suffix array construction in O(L logˆ2 L) time. Routine for

// computing the length of the longest common prefix of any two

// suffixes in O(log L) time.

//

// INPUT: string s

//

// OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)

// of substring s[i...L-1] in the list of sorted suffixes.

// That is, if we take the inverse of the permutation suffix[],

// we get the actual suffix array.

**#include** <vector>

**#include** <iostream>

**#include** <string>

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

**struct** SuffixArray {

**const** **int** L;

string s;

vector<vector<**int**> > P;

vector<pair<pair<**int**,**int**>,**int**> > M;

**SuffixArray**(**const** string &s) : L(s.length()), s(s), P(1, vector<**int**>(L, 0)), M(L) {

**for** (**int** i = 0; i < L; i++) P[0][i] = **int**(s[i]);

**for** (**int** skip = 1, level = 1; skip < L; skip \*= 2, level++) {

P.push\_back(vector<**int**>(L, 0));

**for** (**int** i = 0; i < L; i++)

M[i] = make\_pair(make\_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);

sort(M.begin(), M.end());

**for** (**int** i = 0; i < L; i++)

P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;

}

}

vector<**int**> **GetSuffixArray**() { **return** P.back(); }

// returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]

**int** **LongestCommonPrefix**(**int** i, **int** j) {

**int** len = 0;

**if** (i == j) **return** L - i;

**for** (**int** k = P.size() - 1; k >= 0 && i < L && j < L; k--) {

**if** (P[k][i] == P[k][j]) {

i += 1 << k;

j += 1 << k;

len += 1 << k;

}

}

**return** len;

}

};

// BEGIN CUT

// The following code solves UVA problem 11512: GATTACA.

**#define** TESTING

**#ifdef** TESTING

**int** **main**() {

**int** T;

cin >> T;

**for** (**int** caseno = 0; caseno < T; caseno++) {

string s;

cin >> s;

SuffixArray array(s);

vector<**int**> v = array.GetSuffixArray();

**int** bestlen = -1, bestpos = -1, bestcount = 0;

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

**int** len = 0, count = 0;

**for** (**int** j = i+1; j < s.length(); j++) {

**int** l = array.LongestCommonPrefix(i, j);

**if** (l >= len) {

**if** (l > len) count = 2; **else** count++;

len = l;

}

}

**if** (len > bestlen || len == bestlen && s.substr(bestpos, bestlen) > s.substr(i, len)) {

bestlen = len;

bestcount = count;

bestpos = i;

}

}

**if** (bestlen == 0) {

cout << "No repetitions found!" << **endl**;

} **else** {

cout << s.substr(bestpos, bestlen) << " " << bestcount << **endl**;

}

}

}

**#else**

// END CUT

**int** main() {

// bobocel is the 0’th suffix

// obocel is the 5’th suffix

// bocel is the 1’st suffix

// ocel is the 6’th suffix

// cel is the 2’nd suffix

// el is the 3’rd suffix

// l is the 4’th suffix

SuffixArray suffix("bobocel");

vector<**int**> v = suffix.GetSuffixArray();

// Expected output: 0 5 1 6 2 3 4

// 2

**for** (**int** i = 0; i < v.size(); i++) cout << v[i] << " ";

cout << endl;

cout << suffix.LongestCommonPrefix(0, 2) << endl;

}

// BEGIN CUT

**#endif**

// END CUT

**Dijkstra’s Algo**

// Implementation of Dijkstra’s algorithm using adjacency lists

// and priority queue for efficiency.

// Running time: O(|E| log |V|)

**#include** <queue>

**#include** <cstdio>

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

**const** **int** INF = 2000000000;

**typedef** pair<**int**, **int**> PII;

**int** **main**() {

**int** N, s, t;

scanf("%d%d%d", &N, &s, &t);

vector<vector<PII> > edges(N);

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

**int** M;

scanf("%d", &M);

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

**int** vertex, dist;

scanf("%d%d", &vertex, &dist);

edges[i].push\_back(make\_pair(dist, vertex)); // note order of arguments here

}

}

// use priority queue in which top element has the "smallest" priority

priority\_queue<PII, vector<PII>, greater<PII> > Q;

vector<**int**> dist(N, INF), dad(N, -1);

Q.push(make\_pair(0, s));

dist[s] = 0;

**while** (!Q.empty()) {

PII p = Q.top();

Q.pop();

**int** here = p.second;

**if** (here == t) **break**;

**if** (dist[here] != p.first) **continue**;

**for** (vector<PII>::iterator it = edges[here].begin(); it != edges[here].end(); it++)

{

**if** (dist[here] + it->first < dist[it->second])

{

dist[it->second] = dist[here] + it->first;

dad[it->second] = here;

Q.push(make\_pair(dist[it->second], it->second));

}

}

}

printf("%d\n", dist[t]);

**if** (dist[t] < INF)

**for** (**int** i = t; i != -1; i = dad[i])

printf("%d%c", i, (i == s ? f\nf : f f));

**return** 0;

}

/\*

Sample input:

5 0 4

2 1 2 3 1

2 2 4 4 5

3 1 4 3 3 4 1

2 0 1 2 3

2 1 5 2 1

Expected:

5

4 2 3 0

\*/

**Prim & Kruskal MST**

**#include** <algorithm>

**#include** <cstdio>

**#include** <vector>

**#include** <queue>

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

**typedef** pair<**int**, **int**> ii;

**typedef** vector<**int**> vi;

**typedef** vector<ii> vii;

// Union-Find Disjoint Sets Library written in OOP manner, using both path compression and union by rank heuristics

**class** UnionFind { // OOP style

**private**:

vi p, rank, setSize; // remember: vi is vector<int>

**int** numSets;

**public**:

**UnionFind**(**int** N) {

setSize.assign(N, 1); numSets = N; rank.assign(N, 0);

p.assign(N, 0); **for** (**int** i = 0; i < N; i++) p[i] = i; }

**int** **findSet**(**int** i) { **return** (p[i] == i) ? i : (p[i] = findSet(p[i])); }

**bool** **isSameSet**(**int** i, **int** j) { **return** findSet(i) == findSet(j); }

**void** **unionSet**(**int** i, **int** j) {

**if** (!isSameSet(i, j)) { numSets--;

**int** x = findSet(i), y = findSet(j);

// rank is used to keep the tree short

**if** (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }

**else** { p[x] = y; setSize[y] += setSize[x];

**if** (rank[x] == rank[y]) rank[y]++; } } }

**int** **numDisjointSets**() { **return** numSets; }

**int** **sizeOfSet**(**int** i) { **return** setSize[findSet(i)]; }

};

vector<vii> AdjList;

vi taken; // global boolean flag to avoid cycle

priority\_queue<ii> pq; // priority queue to help choose shorter edges

**void** **process**(**int** vtx) { // so, we use -ve sign to reverse the sort order

taken[vtx] = 1;

**for** (**int** j = 0; j < (**int**)AdjList[vtx].size(); j++) {

ii v = AdjList[vtx][j];

**if** (!taken[v.first]) pq.push(ii(-v.second, -v.first));

} } // sort by (inc) weight then by (inc) id

**int** **main**() {

**int** V, E, u, v, w;

/\*

// Graph in Figure 4.10 left, format: list of weighted edges

// This example shows another form of reading graph input

5 7

0 1 4

0 2 4

0 3 6

0 4 6

1 2 2

2 3 8

3 4 9

\*/

**freopen**("in\_03.txt", "r", stdin);

scanf("%d %d", &V, &E);

// Kruskal's algorithm merged with Prim's algorithm

AdjList.assign(V, vii());

vector< pair<**int**, ii> > EdgeList; // (weight, two vertices) of the edge

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

scanf("%d %d %d", &u, &v, &w); // read the triple: (u, v, w)

EdgeList.push\_back(make\_pair(w, ii(u, v))); // (w, u, v)

AdjList[u].push\_back(ii(v, w));

AdjList[v].push\_back(ii(u, w));

}

sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)

// note: pair object has built-in comparison function

**int** mst\_cost = 0;

UnionFind UF(V); // all V are disjoint sets initially

**for** (**int** i = 0; i < E; i++) { // for each edge, O(E)

pair<**int**, ii> front = EdgeList[i];

**if** (!UF.isSameSet(front.second.first, front.second.second)) { // check

mst\_cost += front.first; // add the weight of e to MST

UF.unionSet(front.second.first, front.second.second); // link them

} } // note: the runtime cost of UFDS is very light

// note: the number of disjoint sets must eventually be 1 for a valid MST

printf("MST cost = %d (Kruskal's)\n", mst\_cost);

// inside int main() --- assume the graph is stored in AdjList, pq is empty

taken.assign(V, 0); // no vertex is taken at the beginning

process(0); // take vertex 0 and process all edges incident to vertex 0

mst\_cost = 0;

**while** (!pq.empty()) { // repeat until V vertices (E=V-1 edges) are taken

ii front = pq.top(); pq.pop();

u = -front.second, w = -front.first; // negate the id and weight again

**if** (!taken[u]) // we have not connected this vertex yet

mst\_cost += w, process(u); // take u, process all edges incident to u

} // each edge is in pq only once!

printf("MST cost = %d (Prim's)\n", mst\_cost);

**return** 0;

}

**DFS Depth First Search**

**#include**<iostream>

**#include** <list>

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

**class** Graph

{

**int** V; // No. of vertices

list<**int**> \*adj;// Pointer to an array containing adj lists

**void** **DFSUtil**(**int** v, **bool** visited[]); // A func used by DFS

**public**:

**Graph**(**int** V); // Constructor

**void** **addEdge**(**int** v, **int** w);// function to add an edge to graph

**void** **DFS**(); // prints DFS traversal of the complete graph

};

**Graph::Graph**(**int** V)

{

**this**->V = V;

adj = **new** list<**int**>[V];

}

**void** **Graph::addEdge**(**int** v, **int** w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

**void** **Graph::DFSUtil**(**int** v, **bool** visited[])

{

visited[v] = **true**;

cout << v << " ";

// Recur for all the vertices adjacent to this vertex

list<**int**>::iterator i;

**for**(i = adj[v].begin(); i != adj[v].end(); ++i)

**if**(!visited[\*i])

DFSUtil(\*i, visited);

}

// The function to do DFS traversal. It uses recursive DFSUtil()

**void** **Graph::DFS**()

{

// Mark all the vertices as not visited

**bool** \*visited = **new** **bool**[V];

**for** (**int** i = 0; i < V; i++)

visited[i] = **false**;

// Call the recursive helper function to print DFS traversal

// starting from all vertices one by one

**for** (**int** i = 0; i < V; i++)

**if** (visited[i] == **false**)

DFSUtil(i, visited);

}

**int** **main**()

{

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal\n";

g.DFS();

**return** 0;

}

**BFS Breadth First Search**

**#include** <algorithm>

**#include** <cstdio>

**#include** <vector>

**#include** <queue>

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

**int** V, E, a, b, s;

vector<vii> AdjList;

vi p; // addition: the predecessor/parent vector

**void** **printPath**(**int** u) { // simple function to extract information from `vi p'

**if** (u == s) { printf("%d", u); **return**; }

printPath(p[u]); // recursive call: to make the output format: s -> ... -> t

printf(" %d", u); }

**int** **main**() {

/\*

// format: list of unweighted edges

// This example shows another form of reading graph input

13 16

0 1 1 2 2 3 0 4 1 5 2 6 3 7 5 6

4 8 8 9 5 10 6 11 7 12 9 10 10 11 11 12

\*/

**freopen**("in\_04.txt", "r", stdin);

scanf("%d %d", &V, &E);

AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList

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

scanf("%d %d", &a, &b);

AdjList[a].push\_back(ii(b, 0));

AdjList[b].push\_back(ii(a, 0));

}

// as an example, we start from this source

s = 5;

// BFS routine

// inside int main() -- we do not use recursion, thus we do not need to create separate function!

vi dist(V, 1000000000); dist[s] = 0; // distance to source is 0 (default)

queue<**int**> q; q.push(s); // start from source

p.assign(V, -1); // to store parent information (p must be a global variable!)

**int** layer = -1; // for our output printing purpose addition of

**bool** isBipartite = **true**; //one boolean flag, initially true

**while** (!q.empty()) {

**int** u = q.front(); q.pop();

**if** (dist[u] != layer) printf("\nLayer %d: ", dist[u]);

layer = dist[u];

printf("visit %d, ", u);

**for** (**int** j = 0; j < (**int**)AdjList[u].size(); j++) {

ii v = AdjList[u][j]; // for each neighbors of u

**if** (dist[v.first] == 1000000000) {

dist[v.first] = dist[u] + 1; // v unvisited + reachable

p[v.first] = u; // addition: the parent of vertex v->first is u

q.push(v.first); // enqueue v for next step

}

**else** **if** ((dist[v.first] % 2) == (dist[u] % 2))// same parity

isBipartite = **false**;

} }

printf("\nShortest path: ");

printPath(7), printf("\n");

printf("isBipartite? %d\n", isBipartite);

**return** 0;

}

**Lowest Common Ancestor LCA**

**const** **int** max\_nodes, log\_max\_nodes;

**int** num\_nodes, log\_num\_nodes, root;

vector<**int**> children[max\_nodes]; // children[i] contains the children of node i

**int** A[max\_nodes][log\_max\_nodes+1]; // A[i][j] is the 2ˆj-th ancestor of node i, or -1 if that

ancestor does **not** exist

**int** L[max\_nodes]; // L[i] is the distance between node i and the root

// floor of the binary logarithm of n

**int** **lb**(**unsigned** **int** n)

{

**if**(n==0)

**return** -1;

**int** p = 0;

**if** (n >= 1<<16) { n >>= 16; p += 16; }

**if** (n >= 1<< 8) { n >>= 8; p += 8; }

**if** (n >= 1<< 4) { n >>= 4; p += 4; }

**if** (n >= 1<< 2) { n >>= 2; p += 2; }

**if** (n >= 1<< 1) { p += 1; }

**return** p;

}

**void** **DFS**(**int** i, **int** l)

{

L[i] = l;

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

DFS(children[i][j], l+1);

}

**int** **LCA**(**int** p, **int** q)

{

// ensure node p is at least as deep as node q

**if**(L[p] < L[q])

swap(p, q);

// "binary search" for the ancestor of node p situated on the same level as q

**for**(**int** i = log\_num\_nodes; i >= 0; i--)

**if**(L[p] - (1<<i) >= L[q])

p = A[p][i];

**if**(p == q)

**return** p;

// "binary search" for the LCA

**for**(**int** i = log\_num\_nodes; i >= 0; i--)

**if**(A[p][i] != -1 && A[p][i] != A[q][i])

{

p = A[p][i];

q = A[q][i];

}

**return** A[p][0];

}

**int** **main**(**int** argc,**char**\* argv[])

{

// read num\_nodes, the total number of nodes

log\_num\_nodes=lb(num\_nodes);

**for**(**int** i = 0; i < num\_nodes; i++)

{

**int** p;

// read p, the parent of node i or -1 if node i is the root

A[i][0] = p;

**if**(p != -1)

children[p].push\_back(i);

**else**

root = i;

}

// precompute A using dynamic programming

**for**(**int** j = 1; j <= log\_num\_nodes; j++)

**for**(**int** i = 0; i < num\_nodes; i++)

**if**(A[i][j-1] != -1)

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

**else**

A[i][j] = -1;

// precompute L

DFS(root, 0);

**return** 0;

}

**Java**

**StringBuilder Use of Functions**

**import** java.lang.StringBuilder;

**public** **class** String\_builder {

**public** **static** **void** main(String[] args) {

// Create a new StringBuilder.

StringBuilder builder1 = **new** StringBuilder();

// Loop and append values.

**for** (**int** i = 0; i < 5; i++) {

builder1.append("abc ");

}

// Convert to string.

String result = builder1.toString();

System.***out***.println(result);

// " INSERT"

StringBuilder builder2 = **new** StringBuilder("abc");

// Insert this substring at position 2.

builder2.insert(2, "xyz");

System.***out***.println(builder2);//abxyzc

// INDEX-OF

StringBuilder builder3 = **new** StringBuilder("abc");

// Try to find this substring.

**int** result1 = builder3.indexOf("bc");

System.***out***.println(result1);// 1

// This substring does not exist.

**int** result2 = builder3.indexOf("de");

System.***out***.println(result2);// -1

// DELETE

StringBuilder builder4 = **new** StringBuilder("carrot");

// Delete characters from index 2 to index 5.

builder4.delete(2, 5);

System.***out***.println(builder4);// cat

// REPLACE

StringBuilder b = **new** StringBuilder("abc");

// Replace second character with "xyz".

b.replace(1, 2, "xyz");

System.***out***.println(b);// axyzc

// SUBSTRING

StringBuilder builder = **new** StringBuilder();

builder.append("Forest");

String firstTwo = builder.substring(0, 2);

System.***out***.println(firstTwo);// Fo

// REVERSE

StringBuilder builder5 = **new** StringBuilder();

builder5.append("abc");

builder5.reverse();

System.***out***.println(builder5);

}

}

**Decimal Formatter**

**import** java.util.Scanner;

**import** java.util.Formatter;

**public** **class** readdouble {

**public** **static** **void** main(String args[]){

Scanner ob=**new** Scanner(System.***in***);

**float** s = ob.nextFloat();

Formatter fmt = **new** Formatter();

fmt = **new** Formatter();

fmt.format("%2.3f",s);

System.***out***.println(fmt);}}

**Fast Read**

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.util.Scanner;

**import** java.util.StringTokenizer;

**import** java.lang.\*;

**public** **class** fastread {

**static** **class** FastReader {

BufferedReader br;

StringTokenizer st;

**public** FastReader() {

br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

}

String next() {

**while** (st == **null** || !st.hasMoreElements()) {

**try** {

st = **new** StringTokenizer(br.readLine());

} **catch** (IOException e) {

e.printStackTrace();

}

}

**return** st.nextToken();

}

**int** nextInt() {

**return** Integer.*parseInt*(next());

}

**long** nextLong() {

**return** Long.*parseLong*(next());

}

**double** nextDouble() {

**return** Double.*parseDouble*(next());

}

String nextLine() {

String str = "";

**try** {

str = br.readLine();

} **catch** (IOException e) {

e.printStackTrace();

}

**return** str;

}

}

**public** **static** **void** main(String[] args) {

FastReader s = **new** FastReader();

**int** t = s.nextInt();

**while** (t-- > 0) {

}}}

**Factorial Using BigInteger**

**public** **static** **void** calculateFactorial(**int** n) {

BigInteger result = BigInteger.***ONE***;

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

result = result.multiply(BigInteger.*valueOf*(i));

}

System.***out***.println(n + "! = " + result);}}

**BigInteger Functions**

**import** java.math.BigInteger; **public** **class** BigIntegerDemo { **public** **static** **void** main(String[] args) {

BigInteger b1 = **new** BigInteger("987654321987654321000000000");

BigInteger b2 = **new** BigInteger("987654321987654321000000000");

BigInteger product = b1.multiply(b2);

BigInteger division = b1.divide(b2);

System.out.println("product = " + product);

System.out.println("division = " + division);

**int** a, b;

BigInteger A, B;

a = 54;

b = 23;

A = BigInteger.valueOf(54);

B = BigInteger.valueOf(37);

A = **new** BigInteger(“54”);

B = **new** BigInteger(“123456789123456789”);

A = BigInteger.ONE;

**int** c = a + b;

BigInteger C = A.add(B); // Other similar function are subtract() multiply(), divide(), remainder(), mod()

String str = “123456789”;

BigInteger C = A.add(**new** BigInteger(str));

**int** val = 123456789;

BigInteger C = A.add(BigIntger.valueOf(val));

//Extraction of value from BigInteger:

**int** x = A.intValue();// value should be in limit of int x

**long** y = A.longValue(); // value should be in limit of long y

String z = A.toString();

// Comparison

**if** (a < b) {} // For primitive int

**if** (A.compareTo(B) < 0) {} // For BigInteger

// Equality

**if** (A.equals(B)) {} // A is equal to B }}

**Check Prime for BigIntegers**

**import** java.util.\*;

**import** java.math.\*;

**class** CheckPrimeTest

{

//Function to check and return prime numbers

**static** **boolean** checkPrime(**long** n)

{

// Converting long to BigInteger

BigInteger b = **new** BigInteger(String.*valueOf*(n));

**return** b.isProbablePrime(1);

// returns whether prime or not

**return** Long.*parseLong*(b.nextProbablePrime().toString());

// returns next probable prime, long

}

// Driver method

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

**long** n = 13;

System.***out***.println(*checkPrime*(n));

}}

**Stack Queue**

**class** ch2\_04\_stack\_queue {

**public** **static** **void** main(String[] args) {

Stack<Character> s = **new** Stack<Character>();

// Queue is abstract, must be instantiated with LinkedList

// (special case for Java Queue)

Queue<Character> q = **new** LinkedList<Character>();

Deque<Character> d = **new** LinkedList<Character>();

System.***out***.println(s.isEmpty()); // currently s is empty, true

System.***out***.println("==================");

s.push('a'); //push b,c

// stack is LIFO, thus the content of s is currently like this: System.***out***.println(s.peek()); // output 'c'

s.pop(); // pop topmost

System.***out***.println(s.peek()); // output 'b'

**while** (!s.isEmpty()) { // stack s still has 2 more items

q.offer(s.peek()); // enqueue 'b', and then 'a' (the method name in Java Queue for push/enqueue operation is 'offer')

s.pop(); }

q.offer('z'); // add one more item

System.***out***.println(q.peek()); // prints 'b'

// in Java, it is harder to see the back of the queue...

// output 'b', 'a', then 'z' (until queue is empty), according to the insertion order above

System.***out***.println("==================");

**while** (!q.isEmpty()) {

System.***out***.printf("%c\n", q.peek()); // take the front first

q.poll(); // before popping (dequeue-ing) it

}

System.***out***.println("==================");

d.addLast('a');

d.addLast('b');

d.addLast('c');

System.***out***.printf("%c - %c\n", d.getFirst(), d.getLast()); // prints 'a - c'

d.addFirst('d');

System.***out***.printf("%c - %c\n", d.getFirst(), d.getLast()); // prints 'd - c'

d.pollLast();

System.***out***.printf("%c - %c\n", d.getFirst(), d.getLast()); // prints 'd - b'

d.pollFirst();

System.***out***.printf("%c - %c\n", d.getFirst(), d.getLast());}} // prints 'a - b'

**KMP**

**import** java.util.\*;

**class** ch6\_02\_kmp {

**char**[] T, P; // T = text, P = pattern

**int** n, m; // n = length of T, m = length of P

**int** [] b; // b = back table

**void** naiveMatching() {

**for** (**int** i = 0; i < n; i++) { // try all potential starting indices

Boolean found = **true**;

**for** (**int** j = 0; j < m && found; j++) // use boolean flag `found'

**if** (i + j >= n || P[j] != T[i + j]) // if mismatch found

found = **false**; // abort this, shift starting index i by +1

**if** (found) // if P[0 .. m - 1] == T[i .. i + m - 1]

System.***out***.printf("P is found at index %d in T\n", i);

} }

**void** kmpPreprocess() { // call this before calling kmpSearch()

**int** i = 0, j = -1; b[0] = -1; // starting values

**while** (i < m) { // pre-process the pattern string P

**while** (j >= 0 && P[i] != P[j]) j = b[j]; // if different, reset j using b

i++; j++; // if same, advance both pointers

b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4

} } // in the example of P = "SEVENTY SEVEN" above

**void** kmpSearch() { // this is similar as kmpPreprocess(), but on string T

**int** i = 0, j = 0; // starting values

**while** (i < n) { // search through string T

**while** (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j using b

i++; j++; // if same, advance both pointers

**if** (j == m) { // a match found when j == m

System.***out***.printf("P is found at index %d in T\n", i - j);

j = b[j]; // prepare j for the next possible match

} } }

**void** run() {

String Tstr = "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN";

String Pstr = "SEVENTY SEVEN";

T = **new** String(Tstr).toCharArray();

P = **new** String(Pstr).toCharArray();

n = T.length;

m = P.length;

System.***out***.println(T);

System.***out***.println(P);

System.***out***.println();

System.***out***.printf("Naive Mathing\n");

naiveMatching();

System.***out***.println();

System.***out***.printf("KMP\n");

b = **new** **int**[100010];

kmpPreprocess();

kmpSearch();

System.***out***.println();

System.***out***.printf("String Library\n");

**int** pos = Tstr.indexOf(Pstr);

**while** (pos != -1) {

System.***out***.printf("P is found at index %d in T\n", pos);

pos = Tstr.indexOf(Pstr, pos + 1);

}

System.***out***.println();

}

**public** **static** **void** main(String[] args)

{

**new** ch6\_02\_kmp().run();

}

}