Valar Geekolous ICPC Team Notebook (2017-18)

Contents

1 Template

1.1 template

```cpp
//In The Name Of God
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef unsigned long long ull;
typedef vector<string> vs;
typedef pair<int, int> ii;
typedef pair<int, ii> iii;
typedef pair<double, double> dd;
typedef pair<dd, double> ddd;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<dd> vdd;
typedef vector<double> vd;
typedef vector<vd> vvd;
typedef vector<vvi> vvvi;
typedef vector<vvvi> vvvvi;
typedef vector<ii> vii;
typedef vector<iii> viii;
typedef vector<vii> vvii;
typedef vector<vvii> vvvii;
typedef vector<vector<viii>> vvviii;
typedef vector<vector<iii>> vviii;
typedef set<int> si;
typedef vector<si> vsi;

#define inf 1000000000
#define eps 1e-9
#define pi acos(-1.0) // alternative #define pi (2.0 * acos(0.0))
#define F first
#define S second
#define pb push_back

int main() {
    ios::sync_with_stdio(0);


    return 0;
}
```

2 Mathematics

2.1 LIS

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

/* Finds longest strictly increasing subsequence. O(n log k) algorithm. */
void find_lis(vector<int> &a, vector<int> &b){
    vector<int> p(a.size());
    int u, v;

    if (a.empty()) return;
    b.push_back(0);

    for (size_t i = 1; i < a.size(); i++){
        // If next element a[i] is greater than last element of
        // current longest subsequence a[b.back()], just push it at back of "b" and continue
        if (a[b.back()] < a[i]){
            p[i] = b.back();
            b.push_back(i);
            continue;
        }

        // Binary search to find the smallest element referenced by b which is just bigger than a[i]
        // Note : Binary search is performed on b (and not a).
        // Size of b is always <=k and hence contributes O(log k) to complexity.
        for (u = 0, v = b.size()-1; u < v;){
            int c = (u + v) / 2;
            if (a[b[c]] < a[i]) u=c+1; else v=c;
        }

        // Update b if new value is smaller then previously referenced value
        if (a[i] < a[b[u]]){
            if (u > 0) p[i] = b[u-1];
            b[u] = i;
```

```cpp
        }
    }

    for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = v;
}

/* Example of usage: */
#include <cstdio>
int main(){
    vector<int> seq; // seq : Input Vector
    vector<int> lis;                              // lis : Vector containing indexes of longest
          subsequence
    int tmp;
    while(cin >> tmp && tmp != -100000)
        seq.push_back(tmp);
    find_lis(seq, lis);

    //Printing actual output
    for (size_t i = 0; i < lis.size(); i++)
        printf("%d ", seq[lis[i]]);
    printf("\n");

    return 0;
}
```

2.2 miller rabin prime check

```python
def miller_rabin(n, k):
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

2.3 fast power Cpp

```cpp
long long fast_pow(int a,int n)
{
    long long result = 1;
    long long power = n;
    long long value = a;
    while(power>0)
    {
        if(power&1)
            {result = result*value;
            result = result%1000000007;}
        value = value*value;
        value = value%1000000007;
        power /= 2;
        //power >>= 1;
    }
    return result;
}
```

2.4 classical DP

````
# Classical Dynamic Programming Problems

### 1. Max 1D Range Sum

**Kadane's Algorithm:**

* O(n)

```cpp
int kadane(int a[], int n) { // n is size of array a
    int sum, ans = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
        ans = max(ans, sum);
        if (sum < 0) sum = 0;
    }
    return ans;
}
```

### 2. Max 2D Range Sum
**Using Kadane's Algorithm Over 2D**: `n` and `m` are dimensions of array `a`.
* O(n^3)

```cpp
for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) {
// input a[i][j] if needed
if (j > 0) a[i][j] += a[i][j - 1]; // only add columns of this row i
}

maxSubRect = -inf; // the lowest possible number
for (int l = 0; l < n; l++) for (int r = l; r < ; r++) {
    subRect = 0;
    for (int row = 0; row < n; row++) {
        // Max 1D Range Sum on columns of this row i
        if (l > 0) subRect += a[row][r] - a[row][l - 1];
        else       subRect += a[row][r];
        // Kadane s algorithm on rows
        if (subRect < 0) subRect = 0;
        maxSubRect = max(maxSubRect, subRect);
    }
}
```

### 3. Longest Increasing Subsequence (LIS)

* O(n log k): `n` is the size of the array and `k` is the size of the LIS

```cpp
vi seq;
// input seq
vi l(seq.size(), 0), l_index(seq.size(), 0), suc(seq.size(), -1);
int lis = 0, lis_end = 0;
for (int i = 0; i < seq.size(); i++) {
    int pos = lower_bound(l.begin(), l.begin() + lis, seq[i]) - l.begin();
    l[pos] = seq[i];
    l_index[pos] = i;
    suc[i] = pos ? l_index[pos - 1] : -1;
    if (pos + 1 > lis) {
        lis = pos + 1;
        lis_end = i;
    }
}
// the lis length is in lis
```

* O(n^2)

```cpp
vi seq;
// input seq
vi lis(seq.size(), 1), suc(seq.size(), -1);
for (int i = 1; i < n; i++)
        for (int j = 0; j < i; j++)
                if (seq[i] > seq[j] && lis[i] <= lis[j]) {
                        lis[i] = lis[j] + 1;
            suc[i] = j;
        }
// the answer is in largest value of lis
```

**Reconstruct the LIS**: using a stack

```cpp
stack<int> s;
int i;
for (i = lis_end; suc[i] >= 0; i = suc[i])
    s.push(seq[i]);

cout << seq[i];
while (!s.empty()) {
    cout << ', ' << s.top();
    s.pop();
}
cout << endl;
```

### 4. 0-1 Knapsack (Subset Sum)
````

````markdown
* Both versions are O(nW): `n` is the number of items and `W` is the size of knapsack

**Top-Down version (faster then Bottom-Up):**

```cpp
// globals:
int n; // number of items
vi W, V;
// W array holds weights of items
// V array holds values of items
vvi memo;
// memo array is used to memorize states

// value function returns the most value which holds into w from id to end
int value(int id, int w) {
    if (id == n || w == 0) return 0;
    if (memo[id][w] != -1) return memo[id][w];
    if (W[id] > w)         return memo[id][w] = value(id + 1, w);
    return memo[id][w] = max(value(id + 1, w), V[id] + value(id + 1, w - W[id]));
}

// inside main():
W.resize(n + 1, 0);
V.resize(n + 1, 0);
memo.resize(n + 1, vi(MW, 0));

// the answer is in value(0, MW); MW = size of knapsack
```

**Bottom-UP version:**

```cpp
int n; // number of items
// input n

vi W(n + 1, 0), V(n + 1, 0);
// W holds weights of items
// V holds values of items

// input W and V

vvi dp(n + 1, vi(MW, 0)); // dp is used to memorize the states
for (i = 0; i <= N;  i++) dp[i][0] = 0;
for (w = 0; w <= MW; w++) dp[0][w] = 0;

for (i = 1; i <= N; i++)
for (w = 1; w <= MW; w++) {
    if (W[i] > w) dp[i][w] = dp[i - 1][w];
    else          dp[i][w] = max(dp[i - 1][w], V[i] + dp[i - 1][w - W[i]]);
}
// the answer is in dp[n][MW]
```

### 5. Coin Change (CC)

* O(nV): `n` is the number of coin types and `V` is amount of money

**General version: Find the minimum number of coins needed**

```cpp
vi coinValue(n, 0), // holds the value of coins
   memo(V + 1, inf); // used to memorize the states

int change(int value) {
    if (value == 0) return 0;
    if (value < 0) return inf;
    if (memo[value] < inf) return memo[value];
    for (int i = 0; i < n; i++)
        memo[value] = min(memo[value], change(value - coinValue[i]))
    return memo[value] += 1;
}

// the answer is in change(V)
```

**Variant: Find the number of possible ways to get value V**

```cpp
// globals:
int n; // number of coin types
vi coinValue; // holds the value of coins
vvi memo; // used to memorize the states

int ways(int type, int value) {
    if (value == 0)              return 1;
    if (value < 0 || type == n)  return 0;
    if (memo[type][value] != -1) return memo[type][value];
    return memo[type][value] = ways(type + 1, value) + ways(type, value - coinValue[type]);
}

// inside main():
int V; // the value of money
// input V

// coinValue.clear(); if multiple testcases
coinValue.resize(n, 0);

// input coinValue

// memo.clear(); if multiple testcases
memo.resize(n, vi(V + 1, -1));

// the answer is in ways(0, V); V is the value of money
```

### 6. Traveling Salesman Problem

* O(n^2 * 2^n): feasible only with n <= 16

```cpp
int start; // initialize before function

int tsp(int pos, int bitmask) { // bitmask stores the visited coordinates
    if (bitmask == (1 << (n + 1)) - 1)
    return dist[pos][start]; // return trip to close the loop
    if (memo[pos][bitmask] != -1)
    return memo[pos][bitmask];

    int ans = inf;
    for (int nxt = 0; nxt <= n; nxt++) // O(n) here
    if (nxt != pos && !(bitmask & (1 << nxt))) // if coordinate nxt is not visited yet
        ans = min(ans, dist[pos][nxt] + tsp(nxt, bitmask | (1 << nxt)));
    return memo[pos][bitmask] = ans;
}

// the answer is in tsp(start, 1 << start); start is the index of starting node
```
````

2.5 Combinatorics

````markdown
# Combinatorics

### Fibonacci numbers
`fib(0) = 0` and `fib(1) = 1`
`fib(n) = fib(n - 2) + fib(n - 1)` for `n > 2`

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

* O(n) Algorithm using recursion

```cpp
vector<long long> fibseq;

ll fib(int n) {
        if (n < fibseq.size())
                return fibseq[n];
        if (n < 2)
                return n;
        fibseq.push_back(fib(n - 2) + fib(n - 1));
        return fibseq[n];
}
```

**QFibonacci**

* O(lg n) using matrix
* `qfib(n).first` is equal to `fib(n)`
* `ii` refers to `pair<long long, long long>`

```cpp
ii qfib(ll n) {
        if (n == 0)
                return ii(0, 1);
        ii fib = qfib(n / 2);
        ll c = fib.first * ((((fib.second * 2) % mod) - fib.first + mod) % mod);
        c %= mod;
        ll d = (fib.first * fib.first) % mod + (fib.second * fib.second) % mod;
        d %= mod;
        if (n % 2 == 0)
                return ii(c, d);
        return ii(d, (c + d) % mod);
}
```

**Binet's formula**

* O(1)
* Not precise for `n > 75` where `len(fib(n)) ~ 15`

```cpp
````
````

````
double phi = (1 + sqrt(5)) / 2;

double binets_fib(int n) {
    double binets_fib(int n) {
    return (pow(phi, n) - pow(-phi, -n)) / sqrt(5);
}

// round to binets_fib to nearest integer
// (long long)(binets_fib(n) + 0.5)
```

### Binomial Coefficients

**Recursive formula:**

`C(n, 0) = C(n, n) = 1`
`C(n, k) = C(n - 1, k - 1) + C(n - 1, k)` for `n > k > 0`

**Pascal's Triangle:** This triangle uses above formula

```
n=0           1
n=1         1   1
n=2       1   2   1
n=3     1   3   3   1
         \/  \/   \/
n=4   1  4   6    4   1
```

### Catalan Numbers

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

**General formula:**

`cat(0) = 1`
`cat(n) = C(2n, n) / (n + 1)`

**Recursive formula:**

`cat(n) = ((2n * (2n - 1)) / (n + 1) * n) * cat(n - 1)`

**Properties:**
* `cat(n)` counts the number of distinct binary trees with
`n` vertices, e.g.:

```
n = 3 -> cat(3) = 5

    *   *   *   *   *
   /   /   / \    \   \
  *   *   *   *    *   *
 /     \          /     \
*       *        *       *
```

* `cat(n)` counts the number of expressions containing n pairs of parentheses which are correctly
      matched, e.g. for `n = 3`, we have: `()()()`, `()(())`, `(())()`, `((()))`, and `(()())`.

* `cat(n)` counts the number of different ways `n + 1` factors can be completely parenthe-sized, e.g.
      for `n = 3` and `3 + 1 = 4` factors: `{a, b, c, d}` , we have: `(ab)(cd)`, `a(b(cd))`, `((ab)c)d
      `, `(a(bc))(d)`, and `a((bc)d)`.

* `cat(n)` counts the number of ways a convex polygon of `n + 2` sides can be triangulated.

* `cat(n)` counts the number of monotonic paths along the edges of an `n    n` grid, which do not pass
       above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in
      the upper right corner, and consists entirely of edges pointing rightwards or upwards.
````

2.6 Distance On Sphere

```
double getDistOnSphere (double plat , double plong ,
                        double qlat , double qlong , double radius = 6378.0){
    double pi = 3.141592653589793;
    plat *= (pi /180.0);
    plong *= (pi /180.0);
    qlat *= (pi /180.0);
    qlong *= (pi /180.0);
    long double Distance = acos( sin(plat) * sin(qlat) +
        cos(plat) * cos(qlat) * cos(plong) * cos(qlong) +
        cos(plat) * cos(qlat) * sin(plong) * sin(qlong)
       ) * 6378.0; // 6378 = R Of Earth
     return Distance;
}

double gcDistance(double pLat, double pLong,
    double qLat, double qLong, double radius) {
    pLat *= PI / 180; pLong *= PI / 180; // convert degree to radian
    qLat *= PI / 180; qLong *= PI / 180;
    return radius * acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
    cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) + sin(pLat)*sin(qLat));
}
or
acos(cos(qlat) * cos(plat) * cos(plong - qlong) + sin(qlat) * sin(plat));
```

2.7 fast matrix multipication and fibo

```
#define MAX_N 2                                     // increase this if needed
struct Matrix { ll mat[MAX_N][MAX_N]; };      // to let us return a 2D array

Matrix matMul(Matrix a, Matrix b) {             // O(n^3), but O(1) as n = 2
  Matrix ans; int i, j, k;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      for (ans.mat[i][j] = k = 0; k < MAX_N; k++) {
        ans.mat[i][j] += (a.mat[i][k] % MOD) * (b.mat[k][j] % MOD);
        ans.mat[i][j] %= MOD;              // modulo arithmetic is used here
      }
  return ans;
}

Matrix matPow(Matrix base, int p) {  // O(n^3 log p), but O(log p) as n = 2
  Matrix ans; int i, j;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      ans.mat[i][j] = (i == j);                    // prepare identity matrix
  while (p) {       // iterative version of Divide & Conquer exponentiation
    if (p & 1)                      // check if p is odd (the last bit is on)
      ans = matMul(ans, base);                              // update ans
    base = matMul(base, base);                          // square the base
    p >>= 1;                                              // divide p by 2
  }
  return ans;
}


[[1 , 1] , [1 , 0]] ** n = [[fib(n+1) , fib(n)], [fib(n) , fib(n-1)]]
// FIBONACCI FORMULA
```

2.8 finding circle of formula

```
ii floydCycleFinding(int x0) { // function int f(int x) is defined earlier
    int tortoise = f(x0), hare = f(f(x0)); // f(x0) is the node next to x0
    while (tortoise != hare) {
        tortoise = f(tortoise);
        hare = f(f(hare));
    }
    int mu = 0;
    hare = x0;
    while (tortoise != hare) {
        tortoise = f(tortoise);
        hare = f(hare);
        mu++;
    }
    int lambda = 1;
    hare = f(tortoise);
    while (tortoise != hare) {
        hare = f(hare);
        lambda++;
    }
    return ii(mu, lambda); // mu is the start of circle, lambda is lengh of circle
}
```

2.9 gaussian elimination (n ** 3)

```
#define MAX_N 100   // adjust this value as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

// input: N, Augmented Matrix Aug, output: Column vector X, the answer
ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) { // O(N^3)
  int i, j, k, l; double t; ColumnVector X;
  for (j = 0; j < N - 1; j++) { // the forward elimination phase
    l = j;
    for (i = j + 1; i < N; i++) // which row has largest column value
```
```

````
      if (fabs(Aug.mat[i][j]) > fabs(Aug.mat[l][j]))
        l = i; // remember this row l
    // swap this pivot row, reason: to minimize floating point error
    for (k = j; k <= N; k++)     // t is a temporary double variable
      t = Aug.mat[j][k], Aug.mat[j][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
    for (i = j + 1; i < N; i++) // the actual forward elimination phase
      for (k = N; k >= j; k--)
        Aug.mat[i][k] -= Aug.mat[j][k] * Aug.mat[i][j] / Aug.mat[j][j];
  }

  for (j = N - 1; j >= 0; j--) { // the back substitution phase
    for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] * X.vec[k];
    X.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j]; // the answer is here
  }
  return X;
}
````

2.10 josephus

````
// Complete Search:
// Use list<int> or vector<int> to simulate the process

// Special Case k = 2 (skipping rule = 2)
// if n = 1b1b2b3..bn (in binary format) then the surviver is b1b2b3..bn1

// General Case:
// n = number of men
// k = skipping step
// Output: 0-based index of the surviver (add 1 if you want it to become 1-based)
int josephus(int n, int k) {
  if (n == 1)
    return 0;
  return (josephus(n - 1, k) + k) % n;
}
````

2.11 number theory

````
# Number Theory

### Prime Numbers

**Sieve of Eratosthenes**: to generate list of prime numbers
* O(n log log n): `n = 1e7`
**Prime checker**
* O(1) for `n <= sieve_size` and O(sqrt(n) / ln sqrt(n)) for bigger `n`s.


```cpp
#include <bitset>

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

 // create list of primes in [0..upperbound]
void sieve(ll 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(i);
    }
}

bool isPrime(ll n) {
    if (n <= sieve_size) return bs[n];
    for (int i = 0; i < primes.size(); i++)
        if (n % primes[i] == 0) return false;
    return true;
} // note: only work for n <= (last prime in vi "primes")^2

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

### GCD and LCM

* O(log10 n): `n = max(a, b)`

```cpp
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
int lcm(int a, int b) {
    return a * (b / gcd(a, b));
}
```

### Factorial

* O(n)

```cpp
// ll can hold up to fact(20); for beyond use Java BigInteger
ll fact(int n) {
    return n == 0 ? 1 : n * fact(n - 1);
}
```

### Prime-Power Factorization

* O(sqrt(n) / ln sqrt(n))

```cpp
// needs sieve of eratosthenes
vi primeFactors(ll n) {
    vi factors;
    ll PF_idx = 0, PF = primes[PF_idx];
    while (PF * PF <= n) {
        while (n % PF == 0) {
            n /= PF;
            factors.push_back(PF);
        }
        PF = primes[++PF_idx];
    }
    if (n != 1) factors.push_back(n); // special case if n is a prime
    return factors;
}

// inside int main(), assuming sieve(1000000) has been called before
vi r = primeFactors(2147483647);
```

### Functions involving prime numbers

* `numPF(n)`: Count the number of prime factors of `n`

```cpp
ll numPF(ll n) {
    ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
    while (PF * PF <= n) {
        while (n % PF == 0) {
            n /= PF;
            ans++;
        }
        PF = primes[++PF_idx];
    }
    if (n != 1) ans++;
    return ans;
}
```

* `numDiffPF(n)`: Count the number of *different* prime factors of `n`

```cpp
ll numDiffPF(ll n) {
    ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
    while (PF * PF <= n) {
        int power = 0;
        while (n % PF == 0) {
            n /= PF;
            power++;
        }
        if (power)
            ans++;
        PF = primes[++PF_idx];
    }
    if (n != 1) ans++;
    return ans;
}
```

* `numDiv(n)`: Count the number of *divisors* of `n`

```cpp
ll numDiv(ll n) {
    ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
    while (PF * PF <= n) {
        ll power = 0;
        while (n % PF == 0) {
            n /= PF;
            power++;
````
````

````
        }
        ans *= power + 1;
        PF = primes[++PF_idx];
    }
    if (n != 1) ans *= 2; // last factor has pow = 1, we add 1 to it
    return ans;
}
```

* `sumDiv(n)`: Sum of divisors of `n`

```cpp
ll sumDiv(ll n) {
    ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
    while (PF * PF <= n) {
        ll power = 0;
        while (n % PF == 0) {
            n /= PF;
            power++;
        }
        ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
        PF = primes[++PF_idx];
    }
    if (n != 1) ans *= ((ll)pow((double)n, 2.0) - 1) / (n - 1);  // last
    return ans;
}
```

* `EulerPhi(n)`: Count the number of positive integers < `n` that are relatively prime to `n`

The formula is: `phi(n) = n * PI (1 - 1/PF) for PF = prime factors of n`

```cpp
ll EulerPhi(ll n) {
    ll PF_idx = 0, PF = primes[PF_idx], ans = n;
    while (PF * PF <= n) {
        if (n % PF == 0) ans -= ans / PF;
        while (n % PF == 0) n /= PF;
        PF = primes[++PF_idx];
    }
    if (n != 1) ans -= ans / n;
    return ans;
}
```

### Modified Sieve

Used when there are many numbers to determine `numDiffPF` for them

```cpp
vi numDiffPF(MAX_N, 0);

for (int i = 2; i < MAX_N; i++)
    if (numDiffPF[i] == 0) // i is a prime number
        for (int j = i; j < MAX_N; j += i)
            numDiffPF[j]++; // increase the values of multiples of
```

### Extended Euclid: Solving Linear Diophantine Equation

Suppose we have `ax + by = c` and `d = gcd(a, b)`.
If `d | c` is not true then there is no integral solutions.
Otherwise the first solution `(x0, y0)` can be found using **Extended Euclid**.
Then other solutions can be derived from `x = x0 + (b/d)n` and `y = y0 - (a/d)n`.

```cpp
// store x, y, and d as global variables
void extendedEuclid(int a, int b) {
    if (b == 0) {
        x = 1;
        y = 0;
        d = a;
        return;
    } // base case
    extendedEuclid(b, a % b); // similar as the original gcd
    int x1 = y;
    int y1 = x - (a / b) * y;
    x = x1;
    y = y1;
}
```
````

2.12 longest common subsequence

```cpp
/*
Calculates the length of the longest common subsequence of two vectors.
Backtracks to find a single subsequence or all subsequences. Runs in
O(m*n) time except for finding all longest common subsequences, which
may be slow depending on how many there are.
*/

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;

typedef int T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

void backtrack(VVI& dp, VT& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) return;
  if(A[i-1] == B[j-1]) { res.push_back(A[i-1]); backtrack(dp, res, A, B, i-1, j-1); }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrack(dp, res, A, B, i, j-1);
    else backtrack(dp, res, A, B, i-1, j);
  }
}

void backtrackall(VVI& dp, set<VT>& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) { res.insert(VI()); return; }
  if(A[i-1] == B[j-1])
  {
    set<VT> tempres;
    backtrackall(dp, tempres, A, B, i-1, j-1);
    for(set<VT>::iterator it=tempres.begin(); it!=tempres.end(); it++)
    {
      VT temp = *it;
      temp.push_back(A[i-1]);
      res.insert(temp);
    }
  }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrackall(dp, res, A, B, i, j-1);
    if(dp[i][j-1] <= dp[i-1][j]) backtrackall(dp, res, A, B, i-1, j);
  }
}

VT LCS(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);

  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }

  VT res;
  backtrack(dp, res, A, B, n, m);
  reverse(res.begin(), res.end());
  return res;
}

set<VT> LCSall(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);
  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  set<VT> res;
  backtrackall(dp, res, A, B, n, m);
  return res;
}

int main()
{
  int a[] = { 0, 5, 5, 2, 1, 4, 2, 3 }, b[] = { 5, 2, 4, 3, 2, 1, 2, 1, 3 };
  VI A = VI(a, a+8), B = VI(b, b+9);
  VI C = LCS(A, B);
```
```

```
    for(int i=0; i<C.size(); i++) cout << C[i] << " ";
    cout << endl << endl;

    set <VI> D = LCSall(A, B);
    for(set<VI>::iterator it = D.begin(); it != D.end(); it++)
    {
      for(int i=0; i<(*it).size(); i++) cout << (*it)[i] << " ";
      cout << endl;
    }
  }
```

3 Graph

3.1 articulation poing and bridges O(E + V)

```
// this alg finds bridges and articulation points of graph.
// removal of articulation vertex or bridge edge cause graph to become two disconnected part.


// fill these global vars as shown below in the main before calling this func.
// bridges are not saved and are shown in the function, process them as you want.
//articulation points are in articulation_vertex in main.

vi dfs_low, dfs_num, dfs_parent, articulation_vertex;
vvii AdjList;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == 0) { // a tree edge
            dfs_parent[v.first] = u;
            if (u == dfsRoot) rootChildren++; // special case if u is a root
            articulationPointAndBridge(v.first);
            if (dfs_low[v.first] >= dfs_num[u]) // for articulation point
                articulation_vertex[u] = true;
            if (dfs_low[v.first] > dfs_num[u]) // found a bridge
                ;// bridges are here, use them as you want.
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
        }
        else if (v.first != dfs_parent[u]) // a back edge and not direct cycle
            dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
    } }


// inside int main()
// n is number of verteces.

    dfsNumberCounter = 0; dfs_num.assign(n, 0); dfs_low.assign(n, 0);
    dfs_parent.assign(n, 0); articulation_vertex.assign(n, 0);
    for (int i = 0; i < n; i++)
        if (dfs_num[i] == 0) {
            dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
            articulation_vertex[dfsRoot] = (rootChildren > 1); } // special case
```

3.2 belman ford sssp and neg circle detection O(E * V)

```
// belman ford alg is used for finding single source shortest path for small graphs with O(E * V)
// it is also used to detect negative circle in graph.
// s is the source.


    vi dist(n, inf); dist[s] = 0; // holds the distance
    for(int i = 0; i < n - 1; i++) {
        bool up = false; // used to prune before finishing when no update is neccessary.
        for (int j = 0; j < n; j++)
            for (auto &e : AdjList[j]) if(dist[e.first] > e.second + dist[j]) {
                    up = true; dist[e.first] = e.second + dist[j];
                }
        if(!up) break;
    }

    // this part is used to detect negative circle, if hasNegCircle is true, graph has negative circle
        .

    bool hasNegCircle = false;
    for(int j = 0; j < n && !hasNegCircle; j++) for(auto &e : AdjList[j])
            if(dist[e.first] > e.second + dist[j]) {
                hasNegCircle = true;
                break;
            }
```

3.3 bipartite graph check O(E + V)

```
// s holds the starting point of colering.
// a graph is bipartite if its set of verteces V can be partitioned into two disjoind set so every
      edge in graph
// is from one set to another one. (Tree is a bipartite graph, bipartite graph has no odd circles)
// answer is in isBipartite. AdjList is adjacency list representation of the graph.
// inside main()

    queue<int> q; q.push(s);
    vi color(V, inf); color[s] = 0;
    bool isBipartite = true;
    while (!q.empty() & isBipartite) {
        int u = q.front(); q.pop();
        for (int j = 0; j < (int) AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            if (color[v.first] == inf) {
                color[v.first] = 1 - color[u];
                q.push(v.first);
            } else if (color[v.first] == color[u]) {
                isBipartite = false;
                break;
            }
        }
    }
```

3.4 dag special algorithms

```
// 1. Single Source shortest(longest) path on DAG:
// in order to find sssp of DAG just find one valid topological sort of the DAG (there is allways at
      least one
// valid topological sort in dag), then relax all outgoing edges of these vertices base on found Top
      sort
// complexity of this alg is O(V + E) of finding top sort
void topSort(vi &order, int v, vi &vis, vvi &graph) {
    vis[v] = 1;
    for(auto &e : graph[v]) if(!vis[e]) topSort(order, e, vis, graph);
    order.push_back(v);
}

// inside main
// graph is adjList.
vi order, vis(n, 0), dis(n, inf); dis[s] = 0; // n is number of vertices. dis holds distance of every
      vertec from s.
for(int i = 0; i < n; i++) if(!vis[i])
    topSort(order, i, vis, graph);
for(int i = order.size() - 1; i >= 0; i--) for(auto &e : graph[order[i]]) {
    if(dis[order[i]] != inf) // after this alg if vertec i is not connected to s dis[i] is inf.
        dis[e.first] = dis[order[i]] + e.second;
}




// 2. counting paths on DAG:
// this alg is for finding number of paths from a source vertex to other vertices.

void topSort(vi &order, int v, vi &vis, vvi &graph) {
    vis[v] = 1;
    for(auto &e : graph[v]) if(!vis[e]) topSort(order, e, vis, graph);
    order.push_back(v);
}
// in main()
// graph holds the DAG.
vi order, vis(n, 0), ways(n, 0);
ways[s] = 1; // setting starting point's number of paths to 1. s is the starting point.
for(int i = 0; i < n; i++) if(!vis[i]) // in case that graph is not garaunted to be connected.
        topSort(order, 0, vis, graph); // finding a valid top sort.
for(int i = order.size() - 1; i >= 0; i--) { // topSort alg stores the order in reverse order.
    int n = order[i];
    for(auto &e : graph[n]) ways[e] += ways[n];
}
```

3.5 dijkstra sssp O((E + V) * log(v))

```
// dijkstra alg for finding SSSP on graph.
// shortest path of all verteces from s is in dist.
// AdjList holds adjacency representation of graph.

        vi dist(n, inf); dist[s] = 0;
    priority_queue<ii, vector<ii>, greater<ii> > pq;
    pq.push(ii(0, s));
    while (!pq.empty()) {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if (d > dist[u]) continue; // this is a very important check
        for (int j = 0; j < (int) AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
```

3.6 edge property check O(E + V)

```
// we consider three states for running dfs: 1. unvisited, visited(visited and completed), explored(
      visited but not completed)
// Graph Edges are classified into three types:
// 1) tree edge: explored to unvisited.
// 2) back edge: explored to explored, an edge which goes back to a vertecs that is not completed yet.
       Found Circle Here.
// 3) forward/Cross edges: explored to visited, goes to a vertecs which is completly visited.


vi dfs_num, dfs_parent; // fill dfs_num in main with size of verteces and value of 0, allocate size of
       verteces
// for dfs_parent, in main call this func for every unvisited verteces.
vvii AdjList; // holds edges.

// function is raw and should be filled with desiered actions.

void graphCheck(int u) {
    dfs_num[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == 0) { // Tree Edge, EXPLORED->UNVISITED
            dfs_parent[v.first] = u; // parent of this children is me
            graphCheck(v.first);
        }
        else if (dfs_num[v.first] == 1) { // EXPLORED->EXPLORED
            if (v.first == dfs_parent[u]); // biconditional edge, usually is not considered as circle

            else // back edge, circle
                ;
        }
        else if (dfs_num[v.first] == 2) // EXPLORED->VISITED, forward edge
        ;
    }
    dfs_num[u] = 2; // Complete
}
```

3.7 eulerian graph check and tour printing

```
// to check an undirected graph to see if it is eulerian: check if all its vertices have even degrees
      then it is eulerian.
// an undirected graph has an euler path if all except two vertices have even degrees, start from one
      odd and finish in another one.


// before running below alg, make sure the given graph is an eulerian graph --> vertices have even
      degrees.
// graph is adjacency list where the second attrubute in edge info is 1 (this edge can still be used)
// or 0 (this edge can no longer be used).

list<int> cyc; // holds the path (tour) after running alg.

void EulerTour(list<int>::iterator i, int u) {
    for (int j = 0; j < (int)graph[u].size(); j++) {
        ii &v = graph[u][j];
        if (v.second) {
            v.second = 0;
            for (int k = 0; k < (int)graph[v.first].size(); k++) {
                ii &uu = graph[v.first][k];
                if (uu.first == u && uu.second) {
                    uu.second = 0;
                    break;
                }
            }
            EulerTour(cyc.insert(i, u), v.first);
        }
    }
}

// in the main
cyc.clear();
EulerTour(cyc.begin, start); // start is any vertices.
```

3.8 finding SCC O(V + E)

```
// finding Strongly Connected Components.
// numSCC holds the number of strongly connected components
// fill AdjList with the graph before running tarjanSCC alg.
//
// IMP: graph must not have self loops, if it does duplicate scc happens

vi dfs_num, dfs_low, S, visited;
int dfsNumberCounter, numSCC;
vvii AdjList;

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    S.push_back(u); // stores u in a vector based on order of visitation
    visited[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == 0)
            tarjanSCC(v.first);
        if (visited[v.first])
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }
    if (dfs_low[u] == dfs_num[u]) { // if this is a root (start) of an SCC
        printf("SCC %d:", ++numSCC); // printing this SCC, you can manipulate it as you want
        while (1) {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            printf(" %d", v);
            if (u == v) break; }
        printf("\n");
    } }

// inside int main()
int main() {
    dfs_num.assign(V, 0); dfs_low.assign(V, 0); visited.assign(V, 0);
    dfsNumberCounter = numSCC = 0;
    for (int i = 0; i < V; i++)
        if (dfs_num[i] == 0)
            tarjanSCC(i);
}
```

3.9 floyd warshal apsp and variants O(V ** 3)

```
// floyd warshal alg is used for findign shortest path between any two verteces, feasible for V <=
      400.


// inside int main()
// let p be a 2D parent matrix, where p[i][j] is the last vertex before j
// on a shortest path from i to j, i.e. i -> ... -> p[i][j] -> j

for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
        p[i][j] = i; // initialize the parent matrix

for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) // this time, we need to use if statement
            if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
                AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
                p[i][j] = p[k][j];
            }



// this func is used to print the shortest path using p vector after running the alg.
void printPath(int i, int j) {
    if (i != j) printPath(i, p[i][j]);
    printf(" %d", j);
}

// VARIANTS :
```

```
// 1. findig circle (cheapest circle or detecting negative circle):
// just set AdjMat[i][i] = inf before running the alg, after running the alg if AdjMat[i][i] is no
// longer inf then there is a circle, if it is negative then there is a negative circle in graph
// to find cheapest circle check all adjMat[i][i] for the minimum one.


// 2.transitive closure.
// to see if there is any path from i to j transitive closure alg is used.
for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
                for (int j = 0; j < V; j++)
                        AdjMat[i][j] |= (AdjMat[i][k] & AdjMat[k][j]);


// 3.Minmax
// just replace + operator with max operator --> max(AdjMat[i][k], AdjMat[k][j])


// 4. check if i and j belong to the same SCC, after finding transitive closure, if AdjMat[i][j] &&
      AdjMat[j][i]
// is true then i and j belong to the same SCC.

// 5. find number of paths from i to j, just modify floyd warshals so instead of
// AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j] we have AdjMat[i][j] = AdjMat[i][k] * AdjMat[k][j].
// after running floyds if AdjMat[k][k] is greater that 0 then for each i and j such that
// AdjMat[i][k] > 0 && AdjMat[k][j] is true there are infinit number of paths between i and j
// because there is a cirlce for k then you can allways go back to k and reach j again.
// in main AdjMat is 0 for all i and j except for ones that has an edge.
```

3.10 lowest common ancestor O(V + E)

```
#define MAX_N 1000

vvi children;

int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;

void dfs(int cur, int depth) {
  H[cur] = idx;
  E[idx] = cur;
  L[idx++] = depth;
  for (int i = 0; i < children[cur].size(); i++) {
    dfs(children[cur][i], depth+1);
    E[idx] = cur; // backtrack to current node
    L[idx++] = depth;
  }
}

void buildRMQ() {
  idx = 0;
  memset(H, -1, sizeof H);
  dfs(0, 0); // we assume that the root is at index 0
}

// inside main():
// H[u] < H[v] (swap u and v otherwise)
LCA(u, v) = E[RMQ(H[u], H[v])]
```

3.11 max flow dinic O(v ** 2 * E)

```
// to find minimum cut, run flow, the max flow is the value of minimum cut, to find edges
// we put all reachable vertexes from source with positive weight to S components and all others to C
// all edges connecting S vertexes to C are in minimum edges vertexes set.

// Dinic network max flow algorithm, runs in O(V^2*E) time.
// efficient for graph with lots of edges.
// if verteces have capacity as well as edges, simply devide each vertex to two vertex with an edge
      between them
// equal to capacity of the vertex

vi dist, work;
int s, t, n; //fill s, t, n in main ---> s is start, t is destination and n is number of nodes in
      graph.
vvi rem, graph; //fill graph in main. graph is adjList. also fill rem where it keeps capacity of edjes
       in n * n space. edges must be bidictional ?
//it is possible to use rem to construct the path. if there was a path from i to j then rem[j][i] > 0
//if rem[j][i] = 0 before running Dinic's so it can change with questions...

bool dinic_bfs() {
    dist.clear(); dist.resize(n, -1); dist[s] = 0;
    queue<int> queue1; queue1.push(s);
    while(!queue1.empty()) {
        int u = queue1.front(); queue1.pop();
        for(auto &e : graph[u]) {
            if(dist[e] != -1 || rem[u][e] <= 0) continue;
            dist[e] = dist[u] + 1;
            queue1.push(e);
        }
    }
    return (dist[t] != -1);
}

int dinic_dfs(int u, int f) {
    if(u == t) return f;
    for(int &i = work[u]; i < graph[u].size(); i++) {
        int v = graph[u][i];
        if(rem[u][v] <= 0) continue;
        if(dist[u] + 1 == dist[v]) {
            int df = dinic_dfs(v, min(f, rem[u][v]));
            if(df > 0) {
                rem[v][u] += df;
                rem[u][v] -= df;
                return df;
            }
        }
    }
    return 0;
}

int maxFlow() {
    int result = 0;
    while(dinic_bfs()) {
        work.clear(); work.resize(n, 0);
        while(int d = dinic_dfs(s, inf)) result += d;
    }
    return result;
}
```

3.12 max flow edmonskarp (V * E ** 2)

```
// NOTE: edges must be bidirectional --> fi there is an edge from i to j, there must be an edge from j
      to i

// to find minimum cut, run flow, the max flow is the value of minimum cut, to find edges
// we put all reachable vertexes from source with positive weight to S components and all others to C
// all edges connecting S vertexes to C are in minimum edges vertexes set.

//Edmon's karp algo will find network max flow in O(V * E ^ 2). it is easier to code than dinic and
      good for graphs
//with not lots of edges.
//like dinic it is possible to construct the path using res 2D arr, if there is a path from i to j
      then res[j][i] > 0.

// if verteces have capacity as well as edges, simply devide each vertex to two vertex with an edge
      between them
// equal to capacity of the vertex


int s, t, f, mf; //s is start node, t is destination, fill s and t in main. mf will hold the max flow.
vvi graph, res;//graph is adjList fill it in main, res is a n * n 2D vec with capacity of each edge.
vi p;

void augment(int v, int minEdge) {
    if(v == s) {
        f = minEdge; return;
    }
    else if(p[v] != -1) {
        augment(p[v], min(minEdge, res[p[v]][v]));
        res[p[v]][v] -= f; res[v][p[v]] += f;
    }
}

void edmonsKarp(int n) { // n is the graph size. answer is in mf after calling this method.
    mf = 0;
    while(1) {
        f = 0;
        vector<bool> vis(n, false); vis[s] = true;
        p.clear(); p.resize(n, -1);
        queue<int> queue1; queue1.push(s);
        while(!queue1.empty()) {
            int u = queue1.front(); queue1.pop();
            if(u == t) break;
            for(auto &e : graph[u]) {
                if(res[u][e] > 0 && !vis[e]) vis[e] = true, queue1.push(e), p[e] = u;
            }
        }
        augment(t, inf);
```

```
        if(f == 0) break;
        mf += f;
    }
}
```

3.13 minimum cut using network flow O(v ** 2 * E)

```
// minimum cut is problem of minimizing the amount of capacity of edges that are going to be removed
      in order to
// the max flow from source to destination is 0.

// the idea of alg : after running max flow alg, run dfs from source and use edges with positive
      weight to traverse
// the graph, all edges from visited verteces to unvisited ones are edges to be removed for minimum
      cut.


vi dist, work;
int s, t, n; // s -> source, t -> destination, n -> number of verteces.
vvi rem, graph; // rem is n * n vec with capacity of edges, graph is adjList representation of graph.

bool dinic_bfs() {
    dist.clear(); dist.resize(n, -1); dist[s] = 0;
    queue<int> queue1; queue1.push(s);
    while(!queue1.empty()) {
        int u = queue1.front(); queue1.pop();
        for(auto &e : graph[u]) {
            if(dist[e] != -1 || rem[u][e] <= 0) continue;
            dist[e] = dist[u] + 1;
            queue1.push(e);
        }
    }
    return (dist[t] != -1);
}

int dinic_dfs(int u, int f) {
    if(u == t) return f;
    for(int &i = work[u]; i < graph[u].size(); i++) {
        int v = graph[u][i];
        if(rem[u][v] <= 0) continue;
        if(dist[u] + 1 == dist[v]) {
            int df = dinic_dfs(v, min(f, rem[u][v]));
            if(df > 0) {
                rem[v][u] += df;
                rem[u][v] -= df;
                return df;
            }
        }
    }
    return 0;
}

int maxFlow() {
    int result = 0;
    while(dinic_bfs()) {
        work.clear(); work.resize(n, 0);
        while(int d = dinic_dfs(s, inf)) result += d;
    }
    return result;
}

void dfs(vi &vis, int a) {
    vis[a] = 1;
    for(auto &e : graph[a]) if(!vis[e] && rem[a][e]) dfs(vis, e);
}


// inside main
int mx = maxFlow();
vi vis(n, s); dfs(vis, s); // running dfs from source.
for(int i = 0; i < vis.size(); i++) if(vis[i]) {
    for(auto &ee : graph[i])
       if(!vis[ee]); // (i + 1) to (ee + 1) is an edge in minimum cut.
}
```

3.14 MST O(E * log(V))

```
// kruskal alg for finding minimum spanning tree.
// kruskal func will return the amount of MST, you can modify it to get the real MSP representation.

//Variants:
// 1. for finding maximum spanning tree just multiply weights of edges by -1 and run the alg.

// 2. for finding minimum spanning forest just consider this : every time an edge is taken number of
      components decreases by one, so run the alg and take edges while number of components are less
      that the desired number of components.

// 3. in order to find minimum spanning subgraph, just add fixed edges to ufds and run the alg.

// 4. in order to find second best spanning tree, first find MST, after that for each edge in MST
      temporary set it off so the kruskal alg wont select it, do this for every edge in MST and the
      best answer is second best spanning tree.

// 5. minmax is problem of finding minimum of maximum of edge weights among all possible pahts between
       i, j. in order to find min max (or maxmin) run the Kruskal alg and save the MST representation
      then traversing from i to j and finding the maximum weight is the answer.


vector<int> ufds;
vector<pair<int, ii>> graph;

void buildUfds(int n) {
    ufds.clear(); ufds.resize(n);
    for(int i = 0; i < n; i++) ufds[i] = i;
}

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

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

void joinSets(int i, int j) {
    int a = findSet(i), b = findSet(j);
    if (a < b) ufds[a] = b;
    else ufds[b] = a;
}

int kruskal() {
    int cost = 0;
    for(int i = 0; i < graph.size(); i++) {
        pair<int, ii> fr = graph[i];
        if(!isSameSet(fr.second.first, fr.second.second)) {
            cost += fr.first;
            joinSets(fr.second.second, fr.second.first);
        }
    }
    return cost;
}

// in main:
// sort edge list base on weight assending then call buildUfds(n) where n is number of verteces before
       calling kruskal alg.
```

3.15 min vertex cover on tree O(V)

```
// minimum vertex cover is the problem of finding minimum number of vertices such that each edge of
      tree is incident to at least one vertex of selected set.

// answer is in min(MVC(root, false), MVC(root, true))
int MVC(int v, int flag) { // Minimum Vertex Cover
    int ans = 0;
    if (memo[v][flag] != -1)
        return memo[v][flag]; // reserve memo in main, memo.resize(n, vi(2, -1)), where n is number of
              vertices.
    else if (leaf[v]) // leaf[v] is true if v is a leaf, false otherwise
        ans = flag;
    else if (flag == 0) { // if v is not taken, we must take its children
        // Note:  C h i l d r e n  is an Adjacency List that contains the directed version of the tree
        // (parent points to its children; but the children does not point to parents)
        ans = 0;
        for (int j = 0; j < (int)Children[v].size(); j++)
            ans += MVC(Children[v][j], 1);
    }
    else if (flag == 1) {
        ans = 1;
        for (int j = 0; j < (int)Children[v].size(); j++)
            ans += min(MVC(Children.[v][j], 1), MVC(Children[v][j], 0));
    }
    return memo[v][flag] = ans;
}
```

3.16 BFS (sssp) O(E + V)

```
// this BFS modified alg will find shortest path from s to every other verteces in O(E + V)
// AdjList holds the adjacency list representation of graph.
// p holds parent of each vertex in shortest path, it is possible to print the path with p vector.
// distances are in dist.


 // inside int main()
 vi dist(n, inf); dist[s] = 0; // distance from source s to s is 0
 queue<int> q; q.push(s);
 vi p;
 while (!q.empty()) {
     int u = q.front(); q.pop();
     for (int j = 0; j < (int) AdjList[u].size(); j++) {
         ii v = AdjList[u][j];
         if (dist[v.first] == inf) {
             dist[v.first] = dist[u] + 1;
             p[v.first] = u;
             q.push(v.first);
         }
     }
 }
```

3.17 top sort kahn's (V + E)

```
// kahn's alg for finding a valid topological sort.
// in TopSort 'u' comes before 'v' if edge u -> v exists in DAG.

// n is number of vertexes. indegree stores number of incoming edges to i'th vertex.
// fill n and indegree before defination of priority_queue.
// ts holds the Topsort

vi indegree(n, 0), vis(n, 0);
//calculate indegree before running alg.

vi ts;
std::priority_queue<int, std::vector<int>, std::greater<int> > pQueue;
for(int i = 0; i < n; i++) if(!indegree[i]) pQueue.push(i);
        while(!pQueue.empty()) {
            int top = pQueue.top();
            pQueue.pop();
            vis[top] = 1;
            ts.push_back(top);
            for(auto &e : graph[top]) {
                if(vis[e]) continue;
                indegree[e]--;
                if(!indegree[e]) pQueue.push(e);
            }
        }
```

3.18 max bipartite matching

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

#include <vector>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

3.19 min cost matching (V ** 3)

```
///////////////////////////////////////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
///////////////////////////////////////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
        Lmate[i] = j;
        Rmate[j] = i;
        mated++;
        break;
      }
    }
  }

  VD dist(n);
  VI dad(n);
  VI seen(n);

  // repeat until primal solution is feasible
  while (mated < n) {

    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;
```

```
    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {

      // find closest
      j = -1;
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;

      // termination condition
      if (Rmate[j] == -1) break;

      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }

    // update dual variables
    for (int k = 0; k < n; k++) {
      if (k == j || !seen[k]) continue;
      const int i = Rmate[k];
      v[k] += dist[k] - dist[j];
      u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j];

    // augment along path
    while (dad[j] >= 0) {
      const int d = dad[j];
      Rmate[j] = Rmate[d];
      Lmate[Rmate[j]] = j;
      j = d;
    }
    Rmate[j] = s;
    Lmate[s] = j;

    mated++;
  }

  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}
```

3.20 min cost max flow O(V ** 6)

```
// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972).  This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
  }

  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best]) best = k;
      }
      s = best;
    }

    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};

// BEGIN CUT
// The following code solves UVA problem #10594: Data Flow

int main() {
  int N, M;

  while (scanf("%d%d", &N, &M) == 2) {
    VVL v(M, VL(3));
    for (int i = 0; i < M; i++)
      scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
    L D, K;
    scanf("%Ld%Ld", &D, &K);

    MinCostMaxFlow mcmf(N+1);
    for (int i = 0; i < M; i++) {
```

```
      mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
      mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
    }
    mcmf.AddEdge(0, 1, D, 0);

    pair<L, L> res = mcmf.GetMaxFlow(0, N);

    if (res.first == D) {
      printf("%Ld\n", res.second);
    } else {
      printf("Impossible.\n");
    }
  }

  return 0;
}

// END CUT
```

4 Data Structure

4.1 seg tree

```
int arr[N]; // contains numbers for building segTree
int tree[MAX]; // tree representation. MAX = 4 * n
int lazy[MAX]; // array for labeling to update

void build_tree(int node, int a, int b) {
        if(a > b) return; // Out of range

        if(a == b) { // Leaf node
                tree[node] = arr[a]; // Init value
                return;
        }

        build_tree(node*2, a, (a+b)/2); // Init left child
        build_tree(node*2+1, 1+(a+b)/2, b); // Init right child

        tree[node] = max(tree[node*2], tree[node*2+1]); // to change max seg_tree to for example sum
              query
        // change max operator to + operator tree[node] = tree[node*2] + tree[node*2+1];
}

 // Increment elements within range [i, j] with value value
void update_tree(int node, int a, int b, int i, int j, int value) { // update will add value to range,
        to set range to sth
    // you should change this.

        if(lazy[node] != 0) { // This node needs to be updated
                tree[node] += lazy[node]; // Update it

                if(a != b) {
                        lazy[node*2] += lazy[node]; // Mark child as lazy
                lazy[node*2+1] += lazy[node]; // Mark child as lazy
                }

                lazy[node] = 0; // Reset it
        }

        if(a > b || a > j || b < i) // Current segment is not within range [i, j]
                return;

        if(a >= i && b <= j) { // Segment is fully within range
                tree[node] += value;

                if(a != b) { // Not leaf node
                        lazy[node*2] += value;
                        lazy[node*2+1] += value;
                }

                return;
        }

        update_tree(node*2, a, (a+b)/2, i, j, value); // Updating left child
        update_tree(1+node*2, 1+(a+b)/2, b, i, j, value); // Updating right child

        tree[node] = max(tree[node*2], tree[node*2+1]); // to change max seg_tree you must change max
              operator
}

 // Query tree to get max element value within range [i, j]
int query_tree(int node, int a, int b, int i, int j) {

        if(a > b || a > j || b < i) return -inf; // Out of range

        if(lazy[node] != 0) { // This node needs to be updated
                tree[node] += lazy[node]; // Update it

                if(a != b) {
                        lazy[node*2] += lazy[node]; // Mark child as lazy
                        lazy[node*2+1] += lazy[node]; // Mark child as lazy
                }

                lazy[node] = 0; // Reset it
        }

        if(a >= i && b <= j) // Current segment is totally within range [i, j]
                return tree[node];

        int q1 = query_tree(node*2, a, (a+b)/2, i, j); // Query left child
        int q2 = query_tree(1+node*2, 1+(a+b)/2, b, i, j); // Query right child

        int res = max(q1, q2); // final result, to change max seg_tree you must change max operator

        return res;
}

int main() {
        for(int i = 0; i < N; i++) arr[i] = 1;

        build_tree(1, 0, N-1);

        memset(lazy, 0, sizeof lazy);

        update_tree(1, 0, N-1, 0, 6, 5); // Increment range [0, 6] by 5. here 0, N-1 represent the
              current range.
        update_tree(1, 0, N-1, 7, 10, 12); // Incremenet range [7, 10] by 12. here 0, N-1 represent
              the current range.
        update_tree(1, 0, N-1, 10, N-1, 100); // Increment range [10, N-1] by 100. here 0, N-1
              represent the current range.

        cout << query_tree(1, 0, N-1, 0, N-1) << endl; // Get max element in range [0, N-1]
}
```

4.2 ufds

```
class UFDS {
public:
    vector<int> p, rank, setSizes;
    int numSets;

    UFDS(int n) {
        numSets = n;
        rank.assign(n, 0); p.assign(n, 0);
        for (int i = 0; i < n; i++) p[i] = i;
        setSizes.assign(N, 1);
    }
    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)) {
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y]) {
                setSizes[x] += setSizes[y]; p[y] = x;
            }
            else {
                setSizes[y] += setSizes[x]; p[x] = y;
                if (rank[x] == rank[y]) rank[y]++;
            }
            numSets--;
        }
    }
    int setSize(int i) { return setSizes[findSet(i)]; }
    void clear(){ p.clear(); rank.clear(); setSizes.clear(); }
};
```

5 Geometry

5.1 circles

```
// 2D Objects: Circles
// Circle centered at coordinate (a, b) in a 2D Euclidean space with radius
// r is the set (x  a)^2 + (y  b)^2 = r^2.

// test point p relation to circle with c as center and r radius
```

```cpp
// returns 0 -> inside, 1 -> border, 2 -> outside
int insideCircle(dd p, dd c, double r) {
    double dx = p.first - c.first, dy = p.second - c.second;
    double Euc = dx * dx + dy * dy, rSq = r * r;
    return rSq - Euc > eps ? 0 : (fabs(rSq - Euc) < eps ? 1 : 2);
}

double arcLen(double r, double theta){ return (theta / 360.0) * (2 * pi * r); }

// uses: degToRad -> points
// Chord of a circle is defined as a line segment whose endpoints lie on the circle
double chordLen(double r, double theta){ return 2 * r * sin(degToRad(theta) / 2); }

// Sector of a circle is defined as a region of the circle enclosed
// by two radius and an arc lying between the two radius.
double sectorArea(double r, double theta){ return (theta / 360.0) * (pi * r * r); }

// uses: degToRad -> points AND sectorArea -> circles
// Segment of a circle is defined as a region of the circle enclosed
// by a chord and an arc lying between the chord s endpoints
double segmentArea(double r, double theta){
    return sectorArea(r, theta) - (r * r * sin(degToRad(theta)) / 2);
}

// to get the other center, reverse p1 and p2
// Determines the location of the centers (c1 and c2) of the two possible circles
// Given 2 points on the circle (p1 and p2) and radius r of the corresponding circle
bool circle2PtsRad(dd p1, dd p2, double r, dd &c) {
    double &x1 = p1.first, &y1 = p1.second, &x2 = p2.first, &y2 = p2.second;
    double d2 = pow((x1 - x2), 2) +  pow((y1 - y2), 2);
    double det = r * r / d2 - 0.25;
    if (det < 0.0) return false;

    double h = sqrt(det);
    c.first = (x1 + x2) / 2 + (y1 - y2) * h;
    c.second = (y1 + y2) / 2 + (x2 - x1) * h;
    return true; // to get the other center, reverse p1 and p2
}

dd c1, c2;
void circleIntersect(double r1, double r2){
    // R equals the distance of two circle centers
    // NOTE: edit following line
    double R = ;
    // (x1, y1) & (x2, y2) are cordinates of 1st & 2nd circle center respectively
    // NOTE: edit following line
    double x1 = , y1 = , x2 = , y2 = ;

    double co1 = (r1 * r1 - r2 * r2) / (2 * R * R);
    double co2 = sqrt(2 * (r1 * r1 + r2 * r2) / (R * R) -
                     pow((r1 * r1 - r2 * r2), 2) / pow(R, 4) - 1) / 2;

    c1.first = (x1 + x2) / 2 + co1 * (x2 - x1) + co2 * (y2 - y1);
    c1.second = (y1 + y2) / 2 + co1 * (y2 - y1) + co2 * (x1 - x2);


    c2.first = (x1 + x2) / 2 + co1 * (x2 - x1) - co2 * (y2 - y1);
    c2.second = (y1 + y2) / 2 + co1 * (y2 - y1) - co2 * (x1 - x2);
}
```

5.2 lines

```cpp
// 1D OBJECTS: LINES
// Euclidean equation ax + by + c = 0. implementation: ddd(dd(a, b), c)
// Subsequent functions in this subsection assume that this linear equation has b = 1 for
// non vertical lines and b = 0 for vertical lines unless otherwise stated.

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(dd p1, dd p2, ddd &l) {
    double &a = l.first.first, &b = l.first.second, &c = l.second;
    double &x1 = p1.first, &y1 = p1.second, &x2 = p2.first;

    if (fabs(x1 - x2) < eps) // vertical line is fine
        a = 1.0, b = 0.0, c = -x1;

    else {
        a = -(y1 - p2.second) / (x1 - x2);
        b = 1.0; // IMPORTANT: we fix the value of b to 1.0
        c = -(a * x1) - y1;
    }
}

// convert point and gradient/slope to line. y = mx + c --> ax + by + c = 0
void pointSlopeToLine(dd p, double m, ddd &l) {
    double &a = l.first.first, &b = l.first.second, &c = l.second;
    a = -m, b = 1, c = -((a * p.first) + (b * p.second));
}

// check coefficients a & b
bool areParallel(ddd l1, ddd l2) {
    return (fabs(l1.first.first - l2.first.first) < eps) &&
    (fabs(l1.first.second-l2.first.second) < eps);
}

// uses: areParallel -> lines
// also check coefficient c. NOTE: uses areParallel!!
bool areSame(ddd l1, ddd l2) {
    return areParallel(l1 ,l2) && (fabs(l1.second - l2.second) < eps);
}

// uses: areParallel -> lines
// returns true (+ intersection point) if two lines are intersect
bool areIntersect(ddd l1, ddd l2, dd &p) {
    double &a1 = l1.first.first, &b1 = l1.first.second, &c1 = l1.second;
    double &a2 = l2.first.first, &b2 = l2.first.second, &c2 = l2.second;
    double &x = p.first, &y = p.second;
    if (areParallel(l1, l2)) return false; // no intersection

    // solve system of 2 linear algebraic equations with 2 unknowns
    x = (b2 * c1 - b1 * c2) / (a2 * b1 - a1 * b2);

    // special case: test for vertical line to avoid division by zero
    y = -(fabs(b1) > eps ? a1 * x + c1 : a2 * x + c2);
    return true;
}

// line segment p-q intersect with line A-B.
dd lineIntersectSeg(dd p, dd q, dd A, dd B) {
    double px = p.first, py = p.second, qx = q.first, qy = q.second;
    double a = B.second - A.second, b = A.first - B.first;
    double c = B.first * A.second - A.first * B.second;
    double u = fabs(a * px + b * py + c);
    double v = fabs(a * qx + b * qy + c);
    return dd((px * v + qx * u) / (u + v), (py * v + qy * u) / (u + v));
}
```

5.3 points

```cpp
// unit conversion
double degToRad(double theta){ return theta * pi / 180.0; }
double radToDeg(double theta){ return theta * 180.0 / pi; }

// implementation: dd
// testing equality in dd points
bool areEqualPoints(dd p1, dd p2){
    return (fabs(p1.first - p2.first) < eps && fabs(p1.second - p2.second) < eps);
}

//distance of points (Euclidean distance)
double dist(dd p1, dd p2){
    // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
    return hypot(p1.first - p2.first, p1.second - p2.second);
}

// uses: degToRad -> points
// rotate p by theta degrees CCW(counter clock wise) w.r.t origin (0, 0)
dd rotate(dd p, double theta) {
    double rad = degToRad(theta);
    return dd(p.first * cos(rad) - p.second * sin(rad),
              p.first * sin(rad) + p.second * cos(rad));
}
```

5.4 polygon

```cpp
// POLYGONS

// Implementation
// 3 points, entered in counter clockwise order, 0-based indexing
//
// vdd P;
// P.push_back(point(1, 1)); // P0
// P.push_back(point(3, 3)); // P1
// P.push_back(point(9, 1)); // P2
// P.push_back(P[0]); // important: loop back

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double polygonPerimeter(const vdd &P) {
    double result = 0.0;
    for (int i = 0; i < P.size() - 1; ++i) // remember that P[0] = P[n-1]
        result += dist(P[i], P[i + 1]);
    return result;
```

```
}

// returns the area, which is half the determinant
double polygonArea(const vdd &P) {
    double result = 0.0, x1, y1, x2, y2;
    for (int i = 0; i < (int)P.size() - 1; i++) {
        x1 = P[i].first, x2 = P[i + 1].first;
        y1 = P[i].second, y2 = P[i + 1].second;
        result += (x1 * y2 - x2 * y1);
    }
    return fabs(result) / 2.0;
}

// uses: ccw, cross, toVec -> vectors
// note: ccw func must change inorder to accept collinear lines (> -eps)
// returns true if all three consecutive vertices of P form the same turns
bool isConvex(const vdd &P) {
    int sz = (int)P.size();
    if (sz <= 3) return false;
    bool isLeft = ccw(P[0], P[1], P[2]);

    for (int i = 1; i < sz - 1; i++) // then compare with the others
        if (ccw(P[i], P[i + 1], P[(i + 2) == sz ? 1 : i + 2]) != isLeft)
            return false; // different sign -> this polygon is concave
    return true; // this polygon is convex
}

// uses: ccw, cross, toVec, angle, norm_sq -> vectors AND radToDeg -> points
// returns true if point pt is in either convex/concave polygon P
bool inPolygon(dd pt, const vdd &P) {
    if ((int)P.size() == 0) return false;

    double sum = 0; // assume the first vertex is equal to the last vertex
    for (int i = 0; i < (int)P.size() - 1; i++) {
        if (ccw(pt, P[i], P[i + 1])) // left turn/ccw
            sum += degToRad(angle(P[i], pt, P[i + 1]));
        else // right turn/cw
            sum -= degToRad(angle(P[i], pt, P[i + 1]));
    }
    return fabs(fabs(sum) - 2 * pi) < eps;
}

// uses: collinear, corss, toVec -> vectors, dist -> points
// returns true if point pt is on convex/concave polygon P
bool onPolygon(dd pt, const vdd &P){
    int sz = (int)P.size();
    if (sz <= 3) return false;

    for (int i = 0; i < sz - 1; ++i)
        if(collinear(P[i], P[i + 1], pt)) {
            if (fabs(dist(P[i], P[i+1]) - (dist(P[i], pt) + dist(pt, P[i+1]))) < eps)
                return true;
        }

    return false;
}

// uses cross, toVec -> vectos AND lineIntersectSeg -> lines
// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
// to get the right cut just call the function with a, b reversed
vdd cutPolygon(dd a, dd b, const vdd &Q) {
    vdd P;
    for (int i = 0; i < (int)Q.size(); i++) {
        double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
        if (i != (int)Q.size() - 1)
            left2 = cross(toVec(a, b), toVec(a, Q[i+1]));

        if (left1 > -eps) P.push_back(Q[i]); // Q[i] is on the left of ab
        if (left1 * left2 < -eps) // edge (Q[i], Q[i+1]) crosses line ab
            P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
    }
    if (!P.empty() && P.back() != P.front())
        P.push_back(P.front()); // make P  s first point = P  s last point
    return P;
}

// CONVEX HALL
// uses: cross -> vectors
// IMPORTANT!: the first point does not have to be replicated as the last point
vdd convexHull(vdd P){
    int n = P.size(),k = 0; vdd H(2*n);
    sort(P.begin(), P.end());
    for(int i = 0; i < n; i++){
        while(k >= 2 && cross(H[k - 2], H[k - 1] , P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    for(int i = n - 2, t = k + 1; i >= 0; i--){
        while(k >= t && cross(H[k - 2], H[k - 1] , P[i]) <= 0) k--;
        H[k++] = P[i];
    }
    H.resize(k); return H;
}


//for finding the centroid of a polygon
point compute2DPolygonCentroid(const std::vector<point> vertices) {
    point centroid;
    double signedArea = 0.0;
    double x0 = 0.0; // Current vertex X
    double y0 = 0.0; // Current vertex Y
    double x1 = 0.0; // Next vertex X
    double y1 = 0.0; // Next vertex Y
    double a = 0.0;  // Partial signed area
    for (int i = 0; i < vertices.size() - 1; ++i) {

        x0 = vertices[i].x;
        y0 = vertices[i].y;
        x1 = vertices[i + 1].x;
        y1 = vertices[i + 1].y;
        a = x0 * y1 - x1 * y0;
        signedArea += a;
        centroid.x += (x0 + x1) * a;
        centroid.y += (y0 + y1) * a;

    }
    x0 = vertices.back().x;
    y0 = vertices.back().y;
    x1 = vertices.front().x;
    y1 = vertices.front().y;
    a = x0 * y1 - x1 * y0;
    signedArea += a;
    centroid.x += (x0 + x1) * a;
    centroid.y += (y0 + y1) * a;

    signedArea *= 0.5;
    centroid.x /= (6.0 * signedArea);
    centroid.y /= (6.0 * signedArea);

    return centroid;
}
```

5.5 triangle

```
// 2D Objects: Triangles

// returns if 3 sides a, b, c can form a triangle
// overload: canFormTriangle(dist(a, b), dist(b, c), dist(c, a)) if a, b, c are coordinates
bool canFormTriangle(double a, double b, double c){
    return ((a + b > c) && (a + c > b) && (b + c > a));
}

// H e r o n s Formula. For calculation of triangle area using 3 sides a, b, c
double triangleArea(double a, double b, double c){
    double s = (a + b + c) / 2;
    return (sqrt(s) * sqrt(s - a) * sqrt(s - b) * sqrt(s - c));
}

// uses: triangleArea -> triangles
// returns radius of Incircle of triangle with 3 sides: a, b, c
// overload: rInCircle(dist(a, b), dist(b, c), dist(c, a)) if a, b, c are coordinates
double rInCircle(double a, double b, double c) {
    return triangleArea(a, b, c) / (0.5 * (a + b + c));
}

// uses: triangleArea, rInCircle -> triangles
// returns true if triangle with sides a, b, c has incircle
// overload: hasInCircle(dist(a, b), dist(b, c), dist(c, a)) if a, b, c are coordinates
bool hasInCircle(double a, double b, double c){
    return fabs(rInCircle(a, b, c)) >= eps;
}

// returns center of inCircle. NOTE: incircle existance maybe needs to be checked
dd inCircle(dd p1, dd p2, dd p3) {
    dd ctr;
    double a = dist(p2, p3), b = dist(p1, p3), c = dist(p1, p2);
    ctr.first = (a * p1.first + b * p2.first + c * p3.first) / (a + b + c);
    ctr.second = (a * p1.second + b * p2.second + c * p3.second) / (a + b + c);
    return ctr;
}

// uses: triangleArea -> triangles
// returns radius of Circumcircle of triangle with 3 sides: a, b, c
// overload: rCircumCircle(dist(a, b), dist(b, c), dist(c, a)) if a, b, c are coordinates
double rCircumCircle(double a, double b, double c) {
    return a * b * c / (4.0 * triangleArea(a, b, c));
}
```

```
// uses: triangleArea, rCircumCircle -> triangles
// returns true if triangle with sides a, b, c has circumCircle
// overload: hasInCircumCircle(dist(a, b), dist(b, c), dist(c, a)) if a, b, c are coordinates
bool hasInCircumCircle(double a, double b, double c){
    return fabs(rCircumCircle(a, b, c)) >= eps;
}

// returns center of circumCircle. NOTE: circumCircle existance maybe needs to be checked
dd circumCircle(dd p1, dd p2, dd p3){
    double x1 = p1.first, y1 = p1.second, x2 = p2.first, y2 = p2.second;
    double x3 = p3.first, y3 = p3.second;

    double a1 = 2 * (x2 - x1), b1 = 2 * (y2 - y1), c1 = x2 * x2 + y2 * y2 - x1 * x1 - y1 * y1;
    double a2 = 2 * (x3 - x1), b2 = 2 * (y3 - y1), c2 = x3 * x3 + y3 * y3 - x1 * x1 - y1 * y1;
    double d = a1 * b2 - b1 * a2;

    dd ctr;
    ctr.first = (c1 * b2 - b1 * c2) / d;
    ctr.second = (a1 * c2 - c1 * a2) / d;
    return ctr;
}

// Cosine Formula
// uses: radToDeg -> points
// returns angle between sides in DEG: a, b in triangle with 3rd side c
double CosineFormula(double a, double b, double c){
    double theta = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b));
    return radToDeg(theta);
}
```

5.6 vector

```
// 1D OBJECTS: VECTORS
// Vectors are represented with two members: The x and y magnitude of the vector. implementation: dd(x
    , y)
// The magnitude of the vector can be scaled if needed.

// converts 2 points to vector a->b
dd toVec(dd a, dd b) { return dd(b.first - a.first, b.second - a.second); }

// nonnegative s = [<1 (shorter).. 1 (same).. >1 (longer)]
dd scale(dd v, double s) { return dd(v.first * s, v.second * s); }

// translate(move) p according to v
dd translate(dd p, dd v) { return dd(p.first + v.first , p.second + v.second); }

// dot product
double dot(dd a, dd b) { return (a.first * b.first + a.second * b.second); }

// norm of vector
double norm_sq(dd v) { return v.first * v.first + v.second * v.second; }

// uses: toVec, dot, norm_sq, translate, scale -> vectors AND dist -> points
// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (by reference)
double distToLine(dd p, dd a, dd b, dd &c) {
    // formula: c = a + u * ab
    dd ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u)); // translate a to c
    return dist(p, c); // Euclidean distance between p and c
}

// uses: toVec, dot, norm_sq, translate, scale, distToLine -> vectors AND dist -> points
// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (by reference)
double distToLineSegment(dd p, dd a, dd b, dd &c) {
    dd ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) { // closer to a
        c = dd(a.first, a.second);
        return dist(p, a); // Euclidean distance between p and a
    }
    if (u > 1.0){ // closer to b
        c = dd(b.first, b.second);
        return dist(p, b); // Euclidean distance between p and b
    }
    return distToLine(p, a, b, c); // run distToLine as above
}

// uses: pointSlopeToLine, areIntersect, areParallel -> lines
// returns just the closest point from p to the line l
// the closest point is stored in the 3rd parameter (by reference)
void closestPoint(ddd l, dd p, dd &ans) {
    ddd perpendicular; // perpendicular to l and pass through p
    if (fabs(l.first.second) < eps) { // special case 1: vertical line
        ans.first = -(l.second); ans.second = p.second; return;
    }
    if (fabs(l.first.first) < eps) { // special case 2: horizontal line
        ans.first = p.first; ans.second = -(l.second); return;
    }

    pointSlopeToLine(p, 1 / l.first.first, perpendicular); // normal line
    // intersect line l with this perpendicular line
    // the intersection point is the closest point
    areIntersect(l, perpendicular, ans);
}

// uses: pointSlopeToLine, areIntersect, areParallel, toVec, translate -> vectors
// returns the reflection of point p on the line l
// the reflection point is stored in the 3rd parameter (by reference)
void reflectionPoint(ddd l, dd p, dd &ans) {
    dd b;
    closestPoint(l, p, b); // similar to distToLine
    dd v = toVec(p, b); // create a vector
    ans = translate(translate(p, v), v); // translate p twice
}

// uses toVec, norm_sq -> vectors AND radToDeg -> points
// NOTE!: returns angle aob in DEG
double angle(dd a, dd o, dd b) {
    dd oa = toVec(o, a), ob = toVec(o, b);
    double ans =  acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
    return radToDeg(ans);
}

// cross product
double cross(dd a, dd b) { return a.first * b.second - a.second * b.first; }

double cross(dd O, dd A, dd B) {
    return (A.first - O.first) * (B.second - O.second) -
           (A.second - O.second) * (B.first - O.first);
}

// uses: cross, toVec -> vectors
// returns true if point r is on the left side of line pq
// The left turn test is more famously known as the CCW (Counter Clockwise) Test.
bool ccw(dd p, dd q, dd r) { return cross(toVec(p, q), toVec(p, r)) > eps; }

// uses: corss, toVec -> vectors
// returns true if point r is on the same line as the line pq
bool collinear(dd p, dd q, dd r) { return fabs(cross(toVec(p, q), toVec(p, r))) < eps; }
```

6 String Processing

6.1 kmp

```
string T, P; // T = text, P = pattern
int n, m; // size of text, size of pattern
vi b;

void kmpPreprocess() {
        b.assign(n + 1, 0);
            int i = 0, j = -1; b[0] = -1;
                while (i < m) {
                            while (j >= 0 && P[i] != P[j]) j = b[j];
                                  i++; j++;
                                          b[i] = j;
                                              }
}

void kmpSearch() {
        int i = 0, j = 0;
            while (i < n) {
                        while (j >= 0 && T[i] != P[j]) j = b[j]; // different, reset j using b
                                i++; j++;
                                        if (j == m) {
                                                    printf("P is found at index %d in T\n", i - j)
                                                          ;
                                                                j = b[j]; // prepare j for the
                                                                      next possible match
                                                                          }
                                          }
}
                                        }
            }
}
                }
}
```

6.2 longest common prefix and applications

```cpp
// O(n)
// finding common prefix between each subsequence sufix of a string in suffix array.
vi Phi, PLCP, LCP; // LCP is the longest common prefix, resize each with size of string before calling

// needs suffix array before running.
void computeLCP() {
    int i, L;
    Phi[SA[0]] = -1;
    for (i = 1; i < n; i++)
        Phi[SA[i]] = SA[i-1];
    for (i = L = 0; i < n; i++) {
        if (Phi[i] == -1) { PLCP[i] = 0; continue; }
        while (T[i + L] == T[Phi[i] + L]) L++;
        PLCP[i] = L;
        L = max(L-1, 0);
    }
    for (i = 0; i < n; i++)
        LCP[i] = PLCP[SA[i]];
}

// Apclications.

// 1. Finding the Longest Repeated Substring
// longest repeated substring of a string is the maximum number in longest common prefix array.

// 2.Finding the Longest Common Substring
// suppose we have just two strings:
// append each one with a low assci char then join them together --> A = abc and B = cgf
// A --> abc# and B --> cgf$ --> abc#cgf$
// now construct suffix array and LCP.
// then longest common substring is the maximum number in the LCP array where for example
// sufix i and i - 1 belong to different strings.
// if suffix i belong to A then SA[i] < length of A.
// this is extendable to more than two strings.

// ***** sometimes there are duplicate answers with this longest common substring
// algorithm, so check string you are printing with the prevoius one, if same do not print.
```

6.3 suffix array construction

```cpp
// O(nlogn)
string T; // the input string
int n; // n is size of input string
vi RA, tempRA, SA, tempSA, c; // RA is rank array, SA is suffix array, c must have max(300, n)
      elements.

void countingSort(int k) {
    int i, sum, maxi = max(300, n);
    fill(c.begin(), c.end(), 0);
    for (i = 0; i < n; i++)
        c[i + k < n ? RA[i + k] : 0]++;
    for (i = sum = 0; i < maxi; i++)
        swap(sum, c[i]), sum += c[i];
    for (i = 0; i < n; i++)
        tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
    SA = tempSA;
}

void constructSA() {
    int i, k, r;
    for (i = 0; i < n; i++) RA[i] = T[i];
    for (i = 0; i < n; i++) SA[i] = i;
    for (k = 1; k < n; k <<= 1) {
        countingSort(k);
        countingSort(0);
        tempRA[SA[0]] = r = 0;
        for (i = 1; i < n; i++)
            tempRA[SA[i]] =
                (RA[SA[i]] == RA[SA[i - 1]] && RA[SA[i] + k] == RA[SA[i - 1] + k]) ? r : ++r;
        RA = tempRA;
        if (RA[SA[n - 1]] == n - 1) break;
    }
}

// resize the arrays in main, set n
```

6.4 dates

```cpp
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```