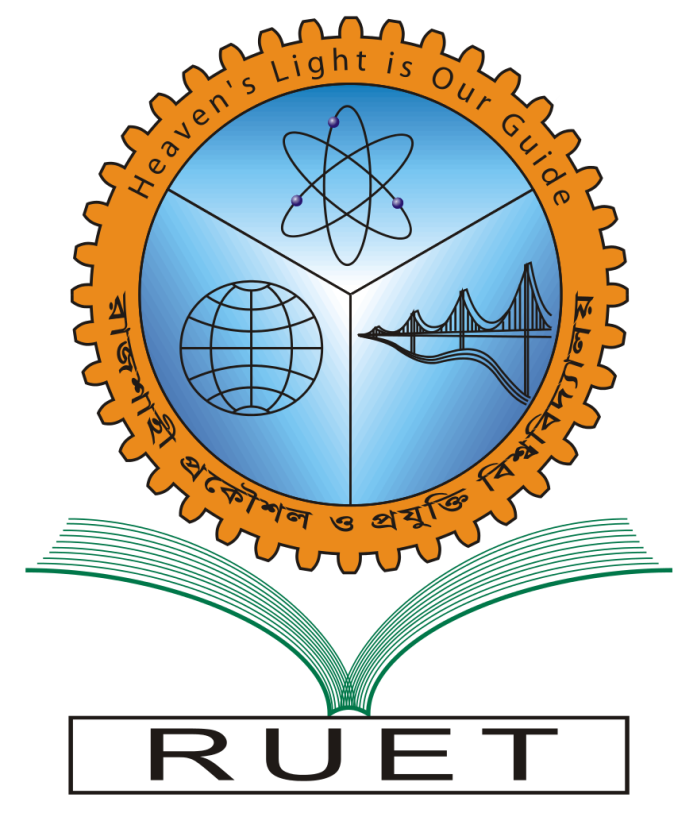
“ Heaven’s Light is Our Guide”

**Rajshahi University of Engineering and Technology**



|  |
| --- |
| Project on  **ALGORITHM ANALYSIS AND DESIGN**  **COURSE NO: 2100** |

|  |  |
| --- | --- |
| **Submitted To:** | **Submitted By:** |
| [**Dr. Engr. Md. Nazrul Islam Monda**l](https://www.linkedin.com/in/dr-engr-md-nazrul-islam-mondal-782605a9)  Prof. & Head, Dept. of CSE, RUET | Md. Tusher Ahamed  Roll: 133069  2nd year Odd Semester  Session: 2013-2014 |

**Maximum Profit Problem**

Suppose that you been offered the opportunity to invest in the Volatile Chemical

Corporation. Like the chemicals the company produces, the stock price of the

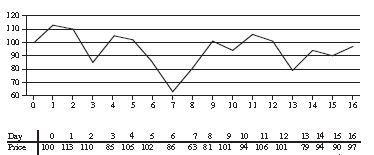
Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit

of stock only one time and then sell it at a later date, buying and selling after the

close of trading for the day. To compensate for this restriction, you are allowed to

learn what the price of the stock will be in the future. Your goal is to maximize

your profit. Figure below shows the price of the stock over a 17-day period .



**Algorithm (Version 1.0):**

Here, DATA is a linear array. Given the number of DAYS and profit in each day. We have to find out how we can maximize the profit if we but it at one day and sell it another.

1. (Insert DAYS and Insert Profits in each day in DATA array)
2. Set i:=0 and repeat steps 3,4 and 5 while i< DAYS *(loop 1 starts here, this loop performs for DAYS times)*
3. Set RESULT:=0 *( Initialization )*
4. Set j:=i+1 and repeat step (a) while j< DAYS *(loop2 starts here, For each DAYS time loop 2 will also performs for DAYS times)*
5. If DATA[i] < DATA[j], then

ANS:=DATA[j] – DATA[i]

if RESULT< ANS then RESULT:= ANS

POS:=j

[End of Loop of step 4]

1. *(This step gives constant time performance for the given data)*
2. If i=0, then MAXRESULT:= RESULT
3. Else if MAXRESULT< RESULT, then

MAXRESULT:= RESULT

BUYDAY:= i

SELLDAY:= POS

[End of Loop of step 2]

1. Return

**Time Complexity (Worst Case Analysis) :**

First loop was carrying out for n (=DAYS) times which has a nested loop that also carries out for ( DAYS – 1) or (n –1) times. So, the worst case time complexity for this particular algorithm will be O(n^2).

**Algorithm (Version 2.0):**

A Simpler approach can be done which will have lesser time complexity and more efficient coding reusability for this kind of problem.

In this algorithm we will use the standard template library found in C++, JAVA or in other languages. This will lessen our effort. We will use the following standard template library and it’s component (e.g. for C++).

<vector> : (vector is just like linear array with some special features)

max\_element( ) = which returns the iterator to the element that has maximum value.

min\_element( ) = which returns the iterator to the element that has minimum value.

<vector> has iterators that works just like pointers and can accessed and changed in such way. We will use iterator MAXPOS1, MAXPOS2, MINPOS1, MINPOS2.

1. (Insert DAYS and insert profit in the <vector> DATA) *(Data is inserted in this step)*
2. *(Using Max element Standard Template Library Function this step is done, which gives log2 n time complexity)*

MAXPOS1:= max\_element ( DATA.begin( ), DATA.end( ) )

MINPOS1:= min\_element ( DATA.begin( ), DATA.end( ) )

MINPOS2 := min\_element( DATA.begin( ), MAXPOS1 )

MAXPOS2 := max\_element( MINPOS1, DATA.end( ) )

1. *(This step gives constant time complexity which is negligible in term of step 2)*
2. If (\*MAXPOS - \*MINPOS2) < (\*MAXPOS2 - \*MINPOS ), then

BUYDAY:= MINPOS - DATA.begin()

SELLDAY := MAXPOS2 - DATA.begin( )

1. Else, then

BUYDAY := MINPOS2 - DATA.begin( )

SELLDAY := MAXPOS- DATA.begin( )

1. (Print BUYDAY and SELLDAY)
2. Return

**Time Complexity (Worst Case Analysis) :**

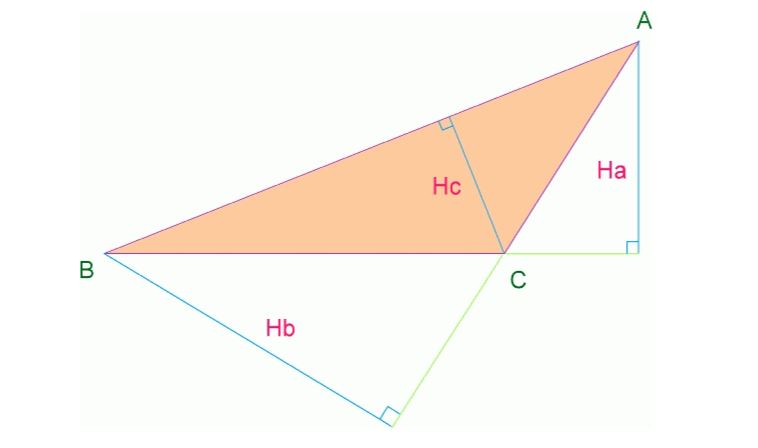
Main advantage for this algorithm is that we didn’t even use any loop inside of that. The min\_element( )/max\_element( ) library function finds the maximum or minimum value using binary search which has time complexity of log2 n. Then a simple difference operation helped us to find our solution pretty easily. So for any number of DAYS we can find the solution in a constant time of log n. Never to say, time complexity of O(log2 n) is far better than O(n^2).

**Problem Name: Height To Area**

**Problem ID:** UVa 10522

**Problem Link:** <https://uva.onlinejudge.org/external/105/10522.pdf>

It’s an easy geometry problem. For any triangle ABC we know that the height from A to the line BC (or it’s extension) is Ha, from B to the line AC (or it’s extension) is Hb and from C to the line AB (or it’s extension) is Hc. Now you are given these three values and you have to figure out the area of the △ABC.



**Input**

At first the input will be an integer n which denotes the number of invalid inputs after which the input will terminate. Then there will be three real numbers

Ha, Hb and Hc per line.

**Output**

For each input block there should be one output line. For valid inputs the line contains the area of the △ ABC up to 3 decimal places after the decimal point and for invalid inputs there will be a line ‘ Theseare invalid inputs!’. After invalid input sets the program will terminate.

**Mathematical Explanation:**

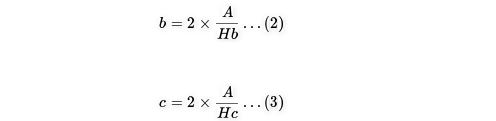
There is no direct formula for this geometry problem. So, we have to derive a generalized formula for this. Given 3 sides Ha, Hb and Hc. We have to calculate the area A.  
If three sides are a, b and c we can have –



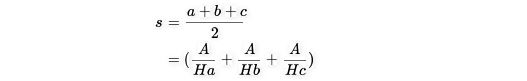
So,



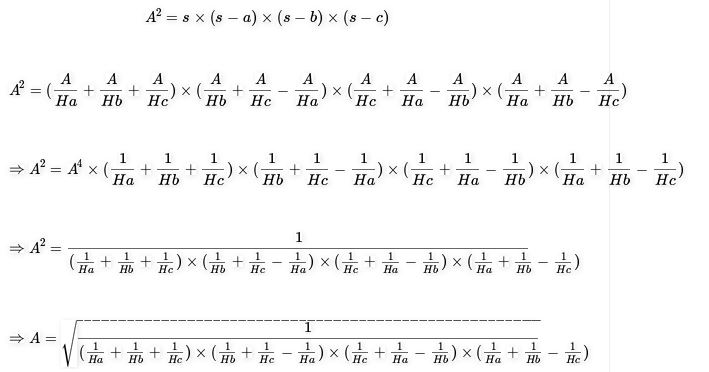
Similarly,



Now,



So according to Veron’s formula we can write,



This is the generalized formula for this problem. Now, the algorithm will be more efficient.

**Algorihtm (version 1.0):**

Here, AREA is the area of the triangle. HA, HB,HC are the heights from three vetices to corresponding bases. And NOII is the the Number OF Invalid INPUTS

1. (Input HA, HB and HC, NOII)
2. Calculate the area using the derived formula *(Step 2 and 3 are carried each time which will be constant, but time complexity will mainly depend on step 4)*

AREA:= sqrt [1/ {(1/HA +1/HB+1/HC)\*(1/HB + 1/HC -1/HA)\*(1/HC + 1/HA-1/HB)\*(1/HA +1/HB- 1/HC)} ]

1. If AREA = VALID, then print AREA
2. Else if AREA = NOT VALID,

Then Print “These are invalid inputs!”

1. Repeat step 1, 2 and 3 until there is maximum NOII number of invalid INPUTS. *(Complexity wholly depends on this step for NOII which give O(NOII) complexity)*
2. Return

**Time Complexity (Worst Case Aalysis) :**

The above algorithm depends on the Number Of Invalid Inputs. SO, considering

N = NOII. The AREA always is calculated in constant time. So, worst case time

complexity for this algorithm will be O(N), where N=Number Of Invalid Inputs.

**Newton’s Formula for Interpolation**

(Forward and Backward Formula)

One of the most important features of **Newton’s formula for interpolation** is that

one can gradually increase the support data without re-computing what is already

computed. Given the set of (n+1) values, (X0,Y0), (X1,Y1), (X2,Y2),…...... ,

(Xm,Ym) of X And Y, it is required to find Yn(x), a polynomial of the nth degree

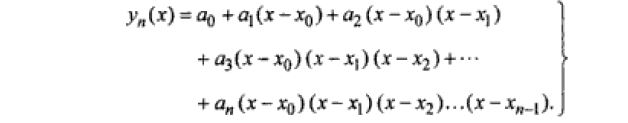
such that Y and Yn(x) agree at the tabulated points.

**Mathematical Explanation:**

Let, the values of x be equidistant. i.e. let,

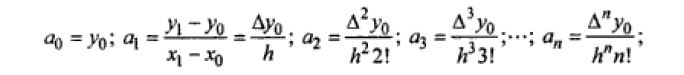


Since, Yn(x) is a polynomial of the nth degree, it may be written as

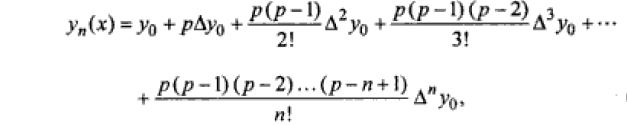


………(1)

Imposing the condition that y and Yn(x) should agree at the set of tabulated points, we obtain



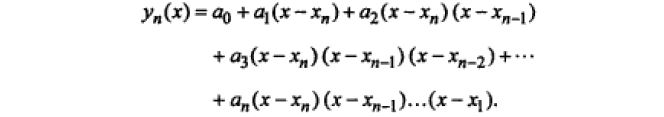
Setting x=x0+ph and substituting for a0, a1,….. an Equation (1) gives



Which is Newton’s forward difference interpolation formula and is useful for

Interpolation near the beginning of a set of tabular values.

Instead of assuming Yn(x) as in equation (1), if we choose it in the form



And then impose the condition that Y and Yn(x) should agree at the tabulated

Points Xn, Xn-1,…., X2, X1, X0, we obtain

...(2)

Where p=(X-Xn)/h Equation (2) is known as the Newton’s backward difference

interpolation formula and uses tabular values left of Yn. This formula is therefore

useful for interpolation near the end of the tabular values.

**Algorithm (version 1.0) :**

Here, X and Y are linear array that store the tabular values x and y. N is the

number of tabular values. Y0 and Yn are Floating point variables and DELY is the

two dimensional linear array that stores the difference table. EXTENSION\_1 is the

process that computes the difference table DELY. FACTORIAL is the function

that determines the factorial of any function This algorithm receives the value of

user inputted XVALUE and automatically takes decision which formula to use

(backward/forward) for computing Yn(x) .

1. (Input tabular values of x and y in X and Y array.)
2. EXTENSION\_1: computes the DELY difference table
3. Set y0=Y[0] and yn=Y[N-1]
4. Input XVALUE, XCOMPARE:=X[ N/2]

Initialize Fx=0

1. *(This step mainly compares each time )*

If XVALUE < XCOMPARE, then

Set P:=(XVALUE-X[0])/(X[1]-X[0] )

Fx:=Fx+y0

*(The main time complexity depends on the loop below which carries out for N-1 times)*

1. Set i:= 0 repeat step a1, a2, a3 and a4 while i < N-1

(a1) Set RESULT:=1

*(For each of N-1 times this loop will go on for calculating RESULT N-1 times)*

(a2) Set j:= 0 repeat this step while j <= i

RESULT:= RESULT\*(P - j)

(a3) RESULT:= (RESULT/ FACTORIAL (i+1)\*DELY[i][0] )

(a4) Fx := Fx + RESULT;

1. *(The main time complexity depends on the loop below which carries out for N-1 times)*

Else if XVALUE > XCOMPARE, then

Set P:= (XVALUE – X[N-1]) / (X[1]- X[0] ) and Fx:= Fx+yn

*(For each of N-1 times this loop will go on for calculating RESULT N-1 times)*

1. Set i:= 0 repeat step b1, b2, b3 and b4 while i < N-1

(b1) Set RESULT:=1

(b2) Set j:= 0 repeat this step while j <= i

RESULT:= RESULT\*(P + j)

j:= j+1 (increment j)

[End of loop (b2) ]

(b3) RESULT:= (RESULT/ FACTORIAL (i+1)\*DELY[i] [ N-2-i ] )

(b4) Fx := Fx + RESULT i := i+1 (increment i)

[End of loop (b) ]

1. Print Fx( the values for Yn(XVALUE) )
2. Return

EXTENSION\_1:

1. Set i:=0 and repeat step while i < N-1
2. Set j:=0 and repeat step while j < N – 1 – i
3. Assign DELY[i] := Y[j+1] – Y[j]
4. Assign Y[j] := Y[j+1] – Y[j]

[End of loop of step 2]

[End of loop of step 1]

1. Return

**Time Complexity (Worst Case Analysis):**

Time complexity of this algorithm depends on N (the number tabular values). There is (N-1) number of difference table for DELY. Two nested loops but both of them are bounded by if-else condition performs the program in O(N^2) in case of worst case.

**Lagrange’s Interpolation Formula**

In numerical analysis, Lagrange polynomials are used for polynomial interpolation.

For a given set of distinct points and numbers, the Lagrangepolynomial is the

Polynomial of the least degree that at each point assumes thecorresponding value

(i.e. the functions coincide at each point).

**Mathematical Explanation:**

Let, Y(x) be continuous and differentiable (n+1) times in the interval (a, b). Given

(n+1) points (x0,y0), (x1,y1), (x2,y2)…..(xn, yn) where the values of x need not

necessarily be equally spaced, we wish to find a polynomial of degree n for any

given value of x. Given a set of *k* + 1 data points

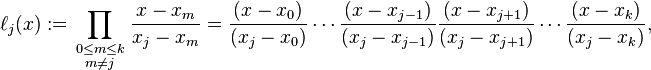
(x_0, y_0),\ldots,(x_j, y_j),\ldots,(x_k, y_k)

where no two x_jare the same, the interpolation polynomial in the Lagrange

form is a linear combination.

L(x) := \sum_{j=0}^{k} y_j \ell_j(x)

of Lagrange basis polynomials



Where 0\le j\le k. Note how, given the initial assumption that no two x_iare the

same, x_j - x_m \neq 0, so this expression is always well-defined.

For all i\neq j, \ell_j(x)includes the term (x-x_i)in the numerator, so the whole

product will be zero at x=x_i:

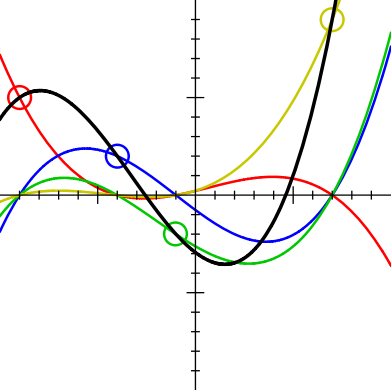
\ell_{j\ne i}(x_i) = \prod_{m\neq j} \frac{x_i-x_m}{x_j-x_m} = \frac{(x_i-x_0)}{(x_j-x_0)} \cdots \frac{(x_i-x_i)}{(x_j-x_i)} \cdots \frac{(x_i-x_k)}{(x_j-x_k)} = 0.

On the other hand,

\ell_i(x_i) := \prod_{m\neq i} \frac{x_i-x_m}{x_i-x_m} = 1

In other words, all basis polynomials are zero at x=x_i, except \ell_i(x), for which it

holds that \ell_i(x_i)=1, because it lacks the (x-x_i)term.



This image shows, for four points ((−9, 5), (−4, 2), (−1, −2), (7, 9)), the (cubic)

interpolation polynomial *L*(*x*) (in black), which is the sum of the *scaled* basis

polynomials y0*ℓ*0(*x*), y1*ℓ*1(*x*), y2*ℓ*2(*x*) and y3*ℓ*3(*x*). The interpolation polynomial

passes through all four control points, and each *scaled* basis polynomial passes

through its respective control point and is 0 where *x* corresponds to the other three

control points.

**Algorithm (version 1.0):**

Here, X and Y are linear array that stores the values of tabular value (points) of x

and y. N is the number of tabular values.

1. (Input tabular values of x and y in X and Y)
2. (User input XVALUE for L(x) polynomial)
3. Initialize SUM:=0 // *Intialization*
4. Set i:=0 and repeat step while i < N *(This First loop will go on for N times)*
5. Set TEMP := 1
6. *(For each of these N time this step of loop will go on for N times also, which ultimately gives the complexity of this algorithm)*

Set j:=0 and repeat step (a) while j < N

1. If i != j, then

Set TEMP := TEMP \* ( (XVALUE – X[j] ) / (X[i] – X[j] ) )

[End of loop of step 6]

1. Set TEMP := TEMP\*Y[i] and SUM+=TEMP

[ End of loop of step 4]

1. Print SUM
2. Return

**Time Complexity (Worst Case Analysis):**

Time complexity for this algorithm depends on the number of tabular inputs N.

Both of the loop perform N times. So, N\*N = N^2 times the constant time statement in step 6(a) performs which is the main logic for this algorithm. So, for the worst case time complexity for this algorithm is O(N^2).

**Jacobi’s Method**

**(Solutions of System of Linear Equations)**

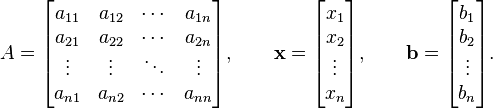
In numerical linear algebra, the Jacobi method (or Jacobi iterative method) is an algorithm for determining the solutions of a diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization. The method is named after Carl Gustav Jacob Jacobi.

**Mathamatical Explanation:**

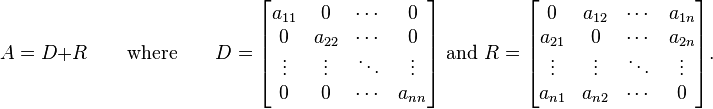
Let

A\mathbf x = \mathbf b

be a square system of *n* linear equations, where:



Then *A* can be decomposed into a diagonal component *D*, and the remainder *R*:



The solution is then obtained iteratively via

 \mathbf{x}^{(k+1)} = D^{-1} (\mathbf{b} - R \mathbf{x}^{(k)}), 

where \mathbf{x}^{(k)}is the *k*th approximation or iteration of \mathbf{x} and \mathbf{x}^{(k+1)}is the next or *k* + 1 iteration of \mathbf{x}. The element-based formula is thus:

 x^{(k+1)}_i  = \frac{1}{a_{ii}} \left(b_i -\sum_{j\ne i}a_{ij}x^{(k)}_j\right),\quad i=1,2,\ldots,n. 

The computation of *xi*(*k*+1) requires each element in **x**(*k*) except itself. Unlike the Gauss–Seidel method, we can't overwrite *xi*(*k*) with *xi*(*k*+1), as that value will be needed by the rest of the computation. The minimum amount of storage is two vectors of size *n*.

**Algorithm (version 1.0):**

Here, VARIABLE, CONSTANT, FVALUE and COEF are linear array. N is the

number of unknowns in the system of linear equation. FUNCTIONVALUE is the

function that computes the value of unnkowns and stores it.

1. (Input number of unknowns in N, coefficient in COEF and constants in CONSTANT)
2. Repeat the steps 3, 4 and 5 till the BREAK statement *(Program will run until there is a break)*
3. FUNCTIONVALUE ( )
4. Set i:=0 repeat step (b) and (c) while i < N *(This step of loop will go on for N times)*
5. Set j:=o repeat step (b1) while j < N *(For each of these N times this step of loop will go on for N times also)*

(b1) If i != j, then

FVALUE[i] := FVALUE [i] – (COEF[i] [j] \* VARIABLE[j] )

FVALAUE [i] := FAVLUE[i] / COEF[i] [i]

[End of loop of step (b) ]

1. FVALUE [i] = FVALUE[i] + (CONSTANT [i] / COEF[i] [i] )

[End of loop of step (a) ]

1. If ERROR ( VARIABLE[0], FVALUE[0]) < 0.0000001 *(Checks error)*

Then BREAK

1. Else
2. Set k:=0 and repeat the steps (b) and (c) while k < N
3. VARIABLE[i] := FVALUE[i]
4. FVALUE[i] := 0

[End of loop of step (a) ]

[End of loop of step 2]

*(So, the time complexity depends on N and number of iterartion for ERROR (in short NOIE) )*

1. Print the VARIABLE array which now have the result.
2. Return

**Time complexity (Worst Case Anaysis) :**

Time complexity for this algorithm depends on the number of unknowns N. In the function FUNCTIONVALUE there is a nested loop both iterating N times. FUNCTIONVAUE function also iterates till the ERROR reaches. So, the time complexity depends on N and number of iterartion for ERROR (in short NOIE). Then, the worst case time complexity for this algorithm will be O(N^2\*NOIE).

**Gauss-Seidal Method**

**(Solutions of System Linear Equations)**

In numerical linear algebra, the “Gauss–Seidel method”, also known as the “Liebmann method” or the “method of successive displacement”, is an iterative method used to solve a linear system of equations. It is named after the German mathematicians Carl Friedrich Gauss and Philipp Ludwig von Seidel, and is similar to the Jacobi method. Though it can be applied to any matrix with non-zero elements on the diagonals, convergence is only guaranteed if the matrix is either diagonally dominant, or symmetric and positive definite.

**Mathematical Explanation:**

The Gauss–Seidel method is an iterative technique for solving a square system of *n* linear equations with unknown **x**:

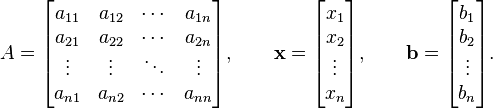
A\mathbf x = \mathbf b.

It is defined by the iteration

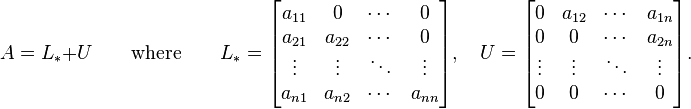
 L_* \mathbf{x}^{(k+1)} = \mathbf{b} - U \mathbf{x}^{(k)}, 

where \mathbf{x}^{(k)}is the *k*th approximation or iteration of \mathbf{x},\,\mathbf{x}^{k+1}is the next or *k* + 1 iteration of \mathbf{x}, and the matrix *A* is decomposed into a lower triangular component L_*, and a strictly upper triangular component *U*:  A = L_* + U .

In more detail, write out *A*, **x** and **b** in their components:



Then the decomposition of *A* into its lower triangular component and its strictly upper triangular component is given by:



The system of linear equations may be rewritten as:

L_* \mathbf{x} = \mathbf{b} - U \mathbf{x} 

The Gauss–Seidel method now solves the left hand side of this expression for **x**, using previous value for **x** on the right hand side. Analytically, this may be written as:

 \mathbf{x}^{(k+1)} = L_*^{-1} (\mathbf{b} - U \mathbf{x}^{(k)}). 

However, by taking advantage of the triangular form of L_*, the elements of **x**(*k*+1) can be computed sequentially using forward substitution:

 x^{(k+1)}_i  = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i}a_{ij}x^{(k+1)}_j - \sum_{j>i}a_{ij}x^{(k)}_j \right),\quad i,j=1,2,\ldots,n. 

The procedure is generally continued until the changes made by an iteration are below some tolerance, such as a sufficiently small residual.

**Algorithm (version 1.0)**

Here, VARIABLE, CONSTANT, FVALUE and COEF are linear array. N is the

number of unknowns in the system of linear equation. FUNCTIONVALUE is the

function that computes the value of unkowns and stores it.

1. (Input number of unknowns in N, coefficient in COEF and constants in CONSTANT)
2. Repeat the steps 3, 4 and 5 till the BREAK statement
3. FUNCTIONVALUE ( )
4. Set i:=0 repeat step (b) and (c) while i < N *(This step of loop will go on for N times)*
5. Set TEMP:=0
6. Set j:=0 repeat step (c1) while j < N *(For each of the N times this step of loop will go on for N times also)*

(c1) If i != j, then

TEMP := TEMP – (COEF[i] [j] \* VARIABLE[j] )

TEMP := TEMP / COEF[i] [i]

[End of loop of step (c) ]

1. TEMP = TEMP + (CONSTANT [i] / COEF[i] [i] )
2. VARIABLE[i] := TEMP

[End of loop of step (a) ]

1. If ERROR ( VARIABLE[0], FVALUE[0]) < 0.0000001 ,

Then BREAK

1. Else
2. Set k:=0 and repeat the steps (b) and (c) while k < N
3. FVALUE [k] := VARIABLE [K}

[End of loop of step (a) ]

[End of loop of step 2]

*(So, the time complexity depends on N and number of iterartion for ERROR (in short NOIE) )*

1. Print the VARIABLE array
2. Return

**Time complexity ( Worst Case Analysis) :**

Time complexity of this algorithm depends on the number of unknowns and the number of iteration for getting closed to the error (Same as Jacobi??). No, In Jacobi’s method the iteration takes much time than in Gauss-Seidal method. But theoretically the complexity is same as Jacobi but this Gauss-Seidal algorithm is better than Jacobi in practical matter.

Then, the worst case time complexity for this algorithm will be O(N^2\*NOIE). Where, NOIE = Number of Iteration for ERROR

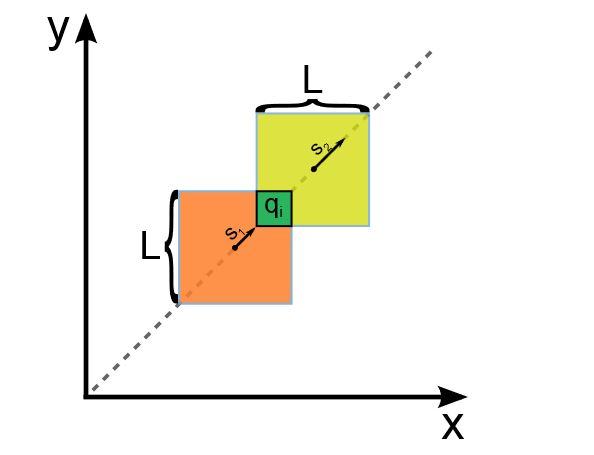
**Problem Name: Sherlock and Moving tiles**

**Problem Source:** HackerRank (Math Domain)

**Problem Link**: <https://www.hackerrank.com/challenges/sherlock-and-moving-tiles>

**Problem Statement:**

Sherlock is given 2 square tiles, initially both of whose sides have length *L* placed in an *x*−*y* plane; so that the left bottom of each square coincides with the origin and their sides are parallel to the axes.



At *t*=0, both squares start moving along line *y*=*x* (along the positive *x* and *y*) with velocities *S*1 and *S*2.

For each query of form *qi*, Sherlock has to report the time at which the overlapping area of tiles is equal to *qi*.

**Note**: Assume all values are in standard units.

**Input Format**   
First line contains integers *L*,*S*1,*S*2. Next line contains *Q*, the number of queries. Each of the next *Q* lines consists of one integer *qi* in one line.

**Constraints**   
1≤*L*,*S*1,*S*2≤109   
1≤*Q*≤105   
1≤*qi*≤*L*2   
*S*1≠*S*2

**Output Format**   
For each query, print the required answer in one line. Your answer will be considered correct if it is at most 0.0001 away from the true answer. See the explanation for more details.

**Algorithm (Version1.0):**

Here, L is the length of sides of the squares. S1 and S2 are velocities of those. Q is the number of query and AREA is the overlapping area qi. We have to determine the TIME.

1. (Input L, S1, S2 and Q)
2. Assign V := AbsoluteValueOf(S1 – S2)
3. Set i:=0 and repeat steps 4 and 5 while i <= Q *(The whole complexity of the program depends on this step, the program will go on for Q times and will terminate after Q times)*
4. (Input AREA{=qi} )
6. If V ! = 0, then

(a1) Assign TIME := ( L – SquareRootOf(AREA) ) \* SquareRootOf (2)

(a2) Assign TIME := TIME / V

(a3) Print TIME

1. Else Print “0.0000000”

[End of loop of step 3]

1. Return

**Time complexity (Worst Case Anaysis):**

Time complexity for this algorithm depends on the number of query Q. For every query the process of determining the TIME gives constant time. As a result the time complexity for the worst Case is the number of Query. So, time complexity is O(Q), where Q = the number of query.

**Problem Name: Prime Words**

**Problem ID:** UVa 10924

**ProblemLink:**https://uva.onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page=show\_problem&problem=1865

**Problem Statement:**

A prime number is a number that has only two divisors: itself and the number one. Examples of prime numbers are: 1, 2, 3, 5, 17, 101 and 10007.

In this problem you should read a set of words, each word is composed only by letters in the range a-z and A-Z. Each letter has a specific value, the letter a is worth 1, letter b is worth 2 and so on until letter z that is worth 26. In the same way, letter A is worth 27, letter B is worth 28 and letter Z is worth 52.

You should write a program to determine if a word is a prime word or not. A word is a prime word if the sum of its letters is a prime number.

**Input**

The input consists of a set of words. Each word is in a line by itself and has L

letters, where 1 <= L <=20. The input is terminated by end of file (EOF).

**Output**

For each word you should print: `It is a prime word.', if the sum of the letters of the word is a prime number, otherwise you should print: `It is not a prime word’.

**Algorithm (version 1.0):**

For this particular problem if we use the Standard Template Library found in Library function of C++ or JAVA, it will be much more easier to anticipate this problem. We are going to use MAPS of the Standard Template Library. MAPs has library function that makes pair that we will use for assigning value to ‘a’=1, ‘B’= 28 etc. and we will also use ITERATOR that works exactly like pointers.

Here, STRING is the given string that we have to find if it’s equivalent word SUM prime or not.

1. Set T := 0
2. Set i := ‘a’ and repeat this process while i < 123*(This step does the Mappinfg of the possible all value of the alphanbet)*

Insert into MAP ( make\_pair ( character equivalent of i, T:=T+1 ) )

[End of loop of step 2]

1. Set j := ‘A’ and repeat this process while j < 91

Insert into MAP ( make\_pair ( character equivalent of j , T:=T+1 ) )

1. Input STRING and repeat the steps 5, 6, 7, 8, 9, 10 and 11 until STRING = EOF
2. Assign LEN := Length Of STRING
3. Set SUM := 0
4. *(This loop determines the main time complexion of the program,the program will go on for LEN times with this step which ultimately determines the complexity)*

Set i := 0 and repeat steps 8 and 9 while i < LEN

1. MAPITER := MAP. Find ( STRING [i] )
2. If MAPITER != End of MAP,

then SUM := SUM + MAP. Second value

[End of loop of step 7]

1. **EXTENSION\_1:**

FUNCTION PRIME\_NOT\_PRIME ( ) : Send value of SUM into PRIME NOT PRIME function

2. If PRIME print “It is a prime word.”
3. Else if NOT PRIME print “It is not a prime word.”

[End of loop of step 4]

1. Return

**EXTENSION\_1 ( PRIME\_NOT\_PRIME function )**

1. If NUM != 0, then
2. Set i := 0 and repeat the step while i < SquareRootOf (NUM)
3. If NUM % i = 0, then RETURN 0

[End of loop of step (a) ]

1. Otherwise Return 1
2. Else, then Return 0

**Time Complexity (Worst Case Analysis):**

Time complexity for this particular problem depends on the Length of the STRING (LEN) .Because the main logic statement performs up to the LEN times. For determining whether the sum of all letters is prime or not, the PRIME NOT PRIME does that work in Square Root Of (SUM=NUM) times. So, for worst case time complexity works in O( LEN + SquareRootOf(NUM) ). But for big length of STRING , ignoring SquareRootOf (NUM) : as it is smaller than LEN, the time complexity becomes O( LEN).

**Problem Name: Word Scramble**

**Problem ID:** UVa 483

**ProblemLink:**<https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=424>

**Problem Statamant:**

Write a program that will reverse the letters in each of a sequence of words while preserving the order of the words themselves.

**Input**

The input file will consist of several lines of several words. Words are contiguous stretches of printable characters delimited by white space.

**Output**

The output will consist of the same lines and words as the input file. However, the letters within each word must be reversed.

**Sample Input**

We're a happy family.

**Sample Output**

er'eW a yppah .ylimaf

**Algorithm (Version 1.0):**

Here, STACK is a stack. STRING is a character array that has to be word scrambled. We will put the result in OUTSTRING.

1. (Create a stack named STACK)
2. User Input STRING and repeat the steps 3, 4, 5, 6 and 7 until STRING = EOF
3. Set LEN:= LengthOf (STRING)
4. Set COUNTLETTER := 0 and POS := 0
5. *(Complexity depends on this step for each step consist of LEN times)*

Set i := 0 and repeat steps 6 while i <= LEN

2. If STRING [i] ! = SPACE and STRING [i] != NULL

Then, PUSH in the STACK (STRING [i] )

1. Else. Then

*(For each of the LEN time POS will be set up and will go on for until STACK is empty, so complexity depends on this step also)*

(b1) Set j:=POS and POP until STACK is empty

OUTSTRING [j] := TOP of STACK

[End of loop of (b1) ]

(b2) POS := i+1

(b3) If STRING [i] != NULL,

Then, OUTSTRING [j] := SPACE

(b4) Else OUTSTRING [j] := NULL

[End of loop of step 5]

1. Print OUTSTRING

[ End of loop of step 2]

1. Return

**Time Complexity (Worst Case Analysis):**

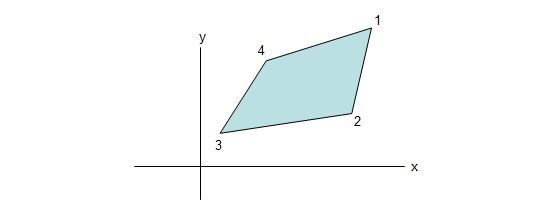
Time complexity for this algorithm depends on the length of the string LEN. In this loop there is a another loop that operates for each iteration until stack is empty but this is in the else condition. So, the time complexity for the worst case is O(LEN\*IUSE). Where IUSE= is Iteration Until Stack is Empty.

**Area of a Polygon**

Given the number of sides and co-ordinate for each of the vertices, we have to find out the Area of that polygon.

**Mathmatical Analysis:**

First, number the vertices in order, going either clockwise or counter-clockwise, starting at any vertex.



The area is then given by the formula

http://www.mathopenref.com/images/coordpolygonarea/eqn2.jpg

Where x1 is the “x” coordinate of vertex 1 and yn is the “y” coordinate of the nth vertex etc. Notice that the in the last term, the expression wraps around back to the first vertex again.

**Algorithm (Version 1.0):**

Here, No of sides of the Polygon is N. POINT is the 2-dimensional array that stores the coordinate of the N number of vertices. We have to figure out the AREA.

1. (Input No of Sides of Polygon as N and N number of point in the POINT array)
2. Set AREA :=0
3. *(This step of loop will go on for N times)*

SET i := 0 and repeat steps 3 and 4 while i < N

1. Set j :=0 and repeat while j < 2 *(For each N times this step will always execute 2 times)*
2. *(This step will give constant complexity)*
3. If j = 0 , then

(a1) POINTMUL := POINT [i] [j] \* POINT [i+1] [j+1]

(a2) AREA := AREA + POINTMUL

1. Else then

(b1) POINTMUL := POINT [i] [j] \* POINT [i+1] [j - 1]

(b2) AREA := AREA – POINTMUL

[End of loop of step 4]

[End of loop of step 3]

1. Print AbsoulteValueOf (AREA)
2. Return

**Time Complexity (Worst Case Analysis):**

Time Complexity of this Algorithm depends on the number of sides of the Polygon. Because the loop of step 3 runs up to N times, where N = number of sides of a Polygonal. There is Nested loop but it only runs for 2 times, a constant time run. The statement under the IF-ELSE condition also gives constant time run. So, the worst case time complexity for this algorithm is O(N). Where, N is the number of sides of the Polygonal.

.