

8th Marathon of Parallel Programming

WSCAD-SSC/SBAC-PAD-2013

October 24th, 2013.

Rules

For all problems, read carefully the input and output session. For all problems, a sequential implementation is given, and it is against the output of those implementations that the output of your programs will be compared to decide if your implementation is correct. You can modify the program in any way you see fit, except when the problem description states otherwise. You must upload a compressed file with your source code, the *Makefile* and an execution script. The program to execute should have the name of the problem. You can submit as many solutions to a problem as you want. Only the last submission will be considered. The *Makefile* must have the rule *all*, which will be used to compile your source code before submit. The execution script runs your solution the way you design it – it will be inspected not to corrupt the target machine.

All teams have access to the target machine during the marathon. Your execution may have concurrent process from other teams. Only the judges have access to a non-concurrent cluster.

The execution time of your program will be measured running it with *time* program and taking the real CPU time given. Each program will be executed at least twice with the same input and only the smaller time will be taken into account. The sequential program given will be measured the same way. You will earn points in each problem, corresponding to the division of the sequential time by the time of your program (*speedup*). The team with the most points at the end of the marathon will be declared the winner.

This problem set contains 6 problems; pages are numbered from 1 to 19.

Problem A

Longest Common Subsequence

A common subsequence of two given sequences may be informally defined as a series of symbols in which all elements are contained in both sequences and appear in the same order. For instance, both “ose” and “os” are common subsequences of “house” and “browse”. A Longest Common Subsequence (LCS) of two sequences is a subsequence that has maximum length. Finding a longest common subsequence of two distinct sequences poses a very important challenge to various areas where computer science is applied, namely genetics and speech recognition.

Given a sequence of symbols $S=\langle a_0, a_1, \dots, a_n \rangle$, a subsequence S' of S is obtained by removing zero or more symbols from S . For example, given $K=\langle a, b, c, d, e \rangle$, $K'=\langle b, d, e \rangle$ is a subsequence of K . A longest common subsequence of two sequences X and Y is a subsequence of both X and Y with maximum length.

The length of a longest common subsequence, the Levenshtein distance, is a string metric for measuring the difference between two sequences.

The length $c[m, n]$ of a LCS of two sequences $A=\langle a_0, a_1, \dots, a_n \rangle$ and $B=\langle b_0, b_1, \dots, b_n \rangle$ may be defined recursively in the following manner:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j, \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise} \end{cases}$$

From this definition, a dynamic programming algorithm can be derived directly.

Write a parallel program to calculate the size of the Longest Common Subsequence between two sequences. Your solution must use dynamic programming to build a score matrix (as in the provided sequential solution).

Input

The program must read two sequences from different files, containing letters and numbers. The first file has the A sequence (*fileA.in*). The second file has the B sequence (*fileB.in*).

The sequences must be read from the correct input files.

Output

The program will print the size of the Longest Common Subsequence. You do not need to print the score matrix at the end but the matrix must be built in memory and it will be used to verify solutions (use the provided debug function to print out the matrix during development).

The output must be written to the standard output.

Example

<i>fileA</i> Input File heagawghee
<i>fileB</i> Input File pawheae
Output 5

<pre> typedef unsigned short mtype; mtype ** allocatescoreMatrix(int sizeA, int sizeB) { int i; //Allocate memory for LCS score matrix mtype ** scoreMatrix = (mtype **) malloc((sizeB + 1) * sizeof(mtype *)); for (i = 0; i < (sizeB + 1); i++) scoreMatrix[i] = (mtype *) malloc((sizeA + 1) * sizeof(mtype)); return scoreMatrix; } void initScoreMatrix(mtype ** scoreMatrix, int sizeA, int sizeB) { int i, j; //Fill first line of LCS score matrix with zeroes for (j = 0; j < (sizeA + 1); j++) scoreMatrix[0][j] = 0; //Do the same for the first column for (i = 1; i < (sizeB + 1); i++) scoreMatrix[i][0] = 0; } int LCS(mtype ** scoreMatrix, int sizeA, int sizeB, char * seqA, char * seqB) { int i, j; for (i = 1; i < sizeB + 1; i++) { for (j = 1; j < sizeA + 1; j++) { if (seqA[j - 1] == seqB[i - 1]) { /* if elements in both sequences match, the corresponding score will be the score from previous elements + 1 */ scoreMatrix[i][j] = scoreMatrix[i - 1][j - 1] + 1; } else { /* else, pick the maximum value (score) from left and upper elements */ scoreMatrix[i][j] = max(scoreMatrix[i-1][j], scoreMatrix[i][j-1]); } } } return scoreMatrix[sizeB][sizeA]; } void freescoreMatrix(mtype **scoreMatrix, int sizeB) { int i; for (i = 0; i < (sizeB + 1); i++) free(scoreMatrix[i]); free(scoreMatrix); } </pre>	<pre> int main(int argc, char ** argv) { // sequence pointers for both sequences char *seqA, *seqB; // sizes of both sequences int sizeA, sizeB; //read both sequences seqA = read_seq("fileA.in"); seqB = read_seq("fileB.in"); //find out sizes sizeA = strlen(seqA); sizeB = strlen(seqB); // allocate LCS score matrix mtype ** scoreMatrix = allocatescoreMatrix(sizeA, sizeB); //initialize LCS score matrix initScoreMatrix(scoreMatrix, sizeA, sizeB); //fill up the rest of the matrix and return final score (element locate at the last line and column) mtype score = LCS(scoreMatrix, sizeA, sizeB, seqA, seqB); /* if you wish to see the entire score matrix, for debug purposes, define DEBUGMATRIX. */ #ifdef DEBUGMATRIX printMatrix(seqA, seqB, scoreMatrix, sizeA, sizeB); #endif //print score printf("Score: %d\n", score); //free score matrix freescoreMatrix(scoreMatrix, sizeB); return EXIT_SUCCESS; } </pre>
---	--

Problem B

Blum Blum Shub Random Number Generator

Repeatable sequences of random numbers are useful in debugging and to provide reproducibility to experiments. They are also important to guarantee consistency in executions of randomized algorithms, such as Monte Carlo computations. Those sequences are usually provided to programmers in the form of “generators”, a stream that provides a random number upon each request.

A random number generator provides new numbers based on its current state. The initial state is given by a seed value. A generator f generates i th random number at the i th request in function of the $(i - 1)$ th state, which is updated to the i th state, used in the same fashion by later requests.

The Blum Blum Shub random number generator was proposed in 1986 by Leonel Blum, Manuel Blum and Michael Shub. It is advised to be used in cryptography, but not in simulations, because the required modulus operations are slow. In what follows, its seed is denoted by x_0 . Number x_i , the i th number on the sequence, is obtained by performing:

$$x_i = (x_{i-1})^2 \bmod M$$

where $M = pq$ is the product of two primes. It is possible to generate the i th number (thus setting generator to the i th state) directly from the seed by Euler's Theorem:

$$x_i = \left(x_0^{2^i \bmod \lambda(M)} \right) \bmod M$$

where $\lambda(M) = \text{lcm}((p - 1), (q - 1))$, function lcm returning the least common multiplier between two integers.

You should write a program that generates in parallel the N first random numbers of a seeded Blum Blum Shub generator.

Input

It is a line in the form:

$$\langle \text{seed} \rangle \ \langle p \rangle \ \langle q \rangle \ \langle N \rangle$$

Where $\langle \text{seed} \rangle$ is the initial state value x_0 , numbers $\langle p \rangle$ and $\langle q \rangle$ are the primes used to calculate M and $\langle N \rangle$ is amount of random numbers that shall be generated by the Blum

Blum Shub generator from x_0 .

The input must be read from the standard input.

Output

It contains just one line, with all generated numbers separated by space.

The output must be written to the standard output.

Example

Input	Output for the input
3 11 19 9	9 81 82 36 42 92 104 157 196

```

// Calculates Greatest Common Divisor.
template <typename T>
// T models integer
T gcd (T x, T y)
{
    while (true) {
        if (x == T (0)) return y ;
        y %= x ;
        if (y == T (0)) return x ;
        x %= y ;
    }
}

// Calculates Least Common Multiple.
// uses eq: lcm (x, y) * gcd (x, y) == xy
template <typename T>
T lcm (T x, T y)
{
    T tmp (gcd (x, y));
    return tmp ? (x * y / tmp) : T (0) ;
}

// Implementation of Blum Blum Shub generator for integers.
// Reference:
// Blum, Lenore; Blum, Manuel; Shub, Mike (1 May 1986) .
// "A Simple Unpredictable Pseudo-Random Number Generator".
// SIAM Journal on Computing 15 (2): 364&#x2013;383.
class BlumBlumShub {
    size_t seed, p, q, x ;

    // Sets state.
    inline
    void set_state (size_t seed, size_t p, size_t q)
    {
        this->seed = seed ;
        this->p = p ;
        this->q = q ;
        this->x = seed ;
    }

public:
    // constructor
    BlumBlumShub (size_t seed, size_t p, size_t q)
    {
        set_state (seed, p, q) ;
    }
}

// default constructor
BlumBlumShub ()
{
    set_state (size_t (3), size_t (11), size_t (19)) ;
}

// copy-constructor
BlumBlumShub (const BlumBlumShub& g)
{
    set_state (g.seed, g.p, g.q) ;
}

// assignment operator
BlumBlumShub& operator= (const BlumBlumShub& g)
{
    if (this != &g) new (this) BlumBlumShub (g) ;
    return *this ;
}

// Generates next random number and updates state.
size_t operator() (void)
{
    x = (x * x) % (p * q) ;
    return x ;
}

// May be useful to jump to a desired distance...
// (uses Euler's Theorem)
size_t operator() (const size_t i)
{
    size_t e (pow (2, i)) ;
    e %= lcm (p - 1, q - 1) ;
    x = size_t (pow (seed, e)) ;
    x %= p * q ;
    return x ;
}
} ;


// Fills range [i,j) with random numbers generated by f.
template <typename R>
// R models generator function
void generate (R& f, size_t* i, size_t* j)
{
    for ( ; i != j ; ++ i) *i = f () ;
}

```

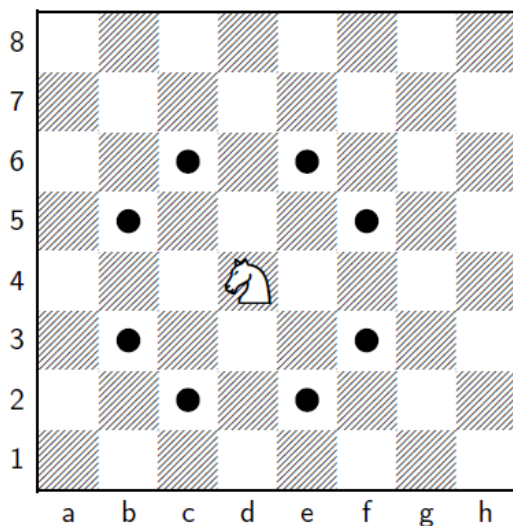
Problem C

The Knight's Tour Problem

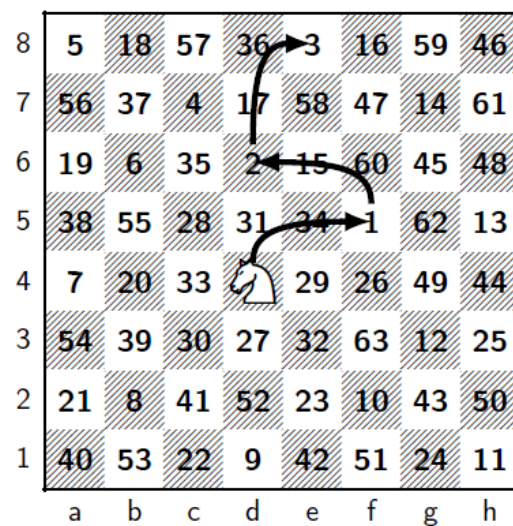
The *Knight's Tour Problem* is a classical mathematical problem which dates back to the 9th century.

In the chess game, a knight () may only make moves which simultaneously shift one square along one axis and two along the other (as illustrated in Figure C1a). A knight's tour on a chessboard (or any other grid of any size) is a sequence of moves by a knight such that each square of the board is visited exactly once.

Write a parallel program that finds a knight's tour starting at a given position, in a square grid of size $s \times s$.



(a) Possible moves of a knight.



(b) A possible knight's tour with its 3 first steps highlighted.

Figure C1. One of the possible knight's tours starting at position d4.

Input

The input contains only one test case. The first (and only) line contains three integers: the size $s \geq 5$ of the side of the square, and a pair of coordinates x and y with the initial position of the knight. To simplify, assume that each axis is numbered from 0 up to $s-1$.

The input must be read from the standard input.

Output

The output must be an $s \times s$ table, where each cell contains the number of knight movements to reach it.

The output must be written to the standard output.

Example

Input	Output for the input
5 0 0	<pre>0 13 18 7 24 5 8 1 12 17 14 19 6 23 2 9 4 21 16 11 20 15 10 3 22</pre>

```

const int move[8][2]={1,2, 2,1, 2,-1, 1,-2, -1,-2, -2,-1, -2,1, -1,2};

class tour {
vector< vector< int > > board;
int sx, sy, size;

public:
bool findtour(tour& , int );

// Constructor
tour(int s = 5, int startx = 0, int starty = 0)
:sx(startx), sy(starty), size(s)
{
// Get the board to size x size
board.resize(size);
for(int i = 0; i < size; ++i)
board[i].resize(size);

// Fill the board with -1s
for(int i = 0; i < size; ++i)
for(int j = 0; j < size; ++j)
board[i][j] = -1;

// Move 0
board[sx][sy] = 0;

// Solve the problem
if(!findtour(*this, 0))
cout << "No solutions found\n";
}

// Copy constructor
tour(const tour& T): sx(T.sx), sy(T.sy), size(T.size) {
this->board.resize(size);
for(int i = 0; i < size; ++i)
board[i].resize(size);

// Copy the board
for(int i = 0; i < size; ++i)
for(int j = 0; j < size; ++j)
board[i][j] = T.board[i][j];
}

// Function to output class to ostream
friend ostream& operator<<
(std::ostream& os, const tour& T);
};

```

```

std::ostream& operator<<(std::ostream& os, const tour& T) {
int size = T.size;

for(int i = 0; i < size; ++i) {
for(int j = 0; j < size; ++j)
os << setw(2) << T.board[i][j] << " ";
os << endl;
}

return os;
}

// A recursive function to find the knight tour.
bool tour::findtour(tour& T, int imove) {
if(imove == (size*size - 1)) return true;

// make a move
int cx = T.sx;
int cy = T.sy;
int cs = T.size;

for (int i = 0; i < 8; ++i) {
int tcx = cx + move[i][0];
int tcy = cy + move[i][1];
if (
// Is this a valid move?
(tcx >= 0) && (tcy >= 0) && (tcx < cs) && (tcy < cs) &&
// Has this place been visited yet?
(T.board[tcx][tcy] == -1)
) {
tour temp(T);
temp.board[tcx][tcy] = imove+1;
temp.sx = tcx;
temp.sy = tcy;
if(findtour(temp, imove+1)) {
cout << temp << endl;
exit(0);
}
}
}
return false;
}
}

```

Problem D

Selection

The problem consists in selecting the k^{th} -largest element from an unsorted list. One obvious solution is to first sort the elements of the list and then pick out the k^{th} position. If you need to put out different k^{th} items multiple times, it might be worthwhile to do the sorting. If, however, you only need to this once, or if the list is updated between selections, you can use an algorithm with a better asymptotic complexity than sorting, namely $O(n)$.

The proposed Selection algorithm to accomplish this task recursively divides the original list in such a way that the search only occurs in subsequences. The central point of this algorithm is in finding the median of a list (the item in the middle). The recursive serial algorithm for selection can be described with five steps:

1. If the size of the data set to be used is less than some constant small size, Q , sort the data and return the k^{th} element; otherwise, subdivide the data set into chunks of size Q and whatever is left over.
2. Sort each chunk and find the median of the medians found in the previous step, creating a new array of medians.
3. Recursively call the selection routine to find the median of the medians in the array of medians found in the previous step.
4. Partition the original data set into three subsequences: those whose elements are less than the median of medians, those that are equal to the median of medians and those that are greater than the median of medians.
5. Determine which subsequence contains the k^{th} element, from the sizes of the three subsequences, and recursively call the selection routine on that subsequence. If the k^{th} element is not in the subsequence of smaller or larger items, it must be in the subsequence equal to the median of medians, so just return the medians of medians value.

According to some authors, the value of Q can be any integer greater than or equal to 5. In the example below this value is set to 5. In the example, S is an array of integers of

size N .

There are four support functions in the algorithm:

- **SortSelect5()**: sorts each chunk of $Q = 5$ elements and returns the median of the sorted data (*not shown*).
- **SortLessThanQ()**: if there is a leftover chunk of less than Q elements (*lastNum*), the median of this chunk is found by calling this function (*not shown*).
- **CountAndMark()**: takes an array of data to be partitioned (S), the array that will hold the notation of which partition the corresponding element from S will be assigned ($Marks$), the number of elements in the first two parameters arrays (num), the value of the median that determines the three partitions ($median$), and the leg array to hold the counts of the number of elements that are less than, equal to, or greater than $median$.
- **ArrayPack()**: takes a list of elements (S) and an array of items, noting which partition the element from S should be assigned ($Marks$). Everywhere the $Marks$ element and the $scanSym$ match, the corresponding element of S is packed into $sPack$ (*tip*: is possible to do this operation with a prefix scan of the $Marks$ array).

Input

The input contains only one test case. The first line contains two integers that correspond respectively to k (the position to be selected) and N (the size of the list). The next lines contain the numbers of the unsorted list, one number in each line. Your program must accept lists up to 500,000,000 of integer elements.

The input must be read from the standard input.

Output

The output contains the k^{th} -largest element from the unsorted list.

The output must be written to the standard output.

Example

Input	Output for the input
6 10 2 1 8 3 9 6 4 7 0 5	5

<pre> int SortlessThanQ(int S[], int num, int k) { switch (num) { case 1: return S[0]; break; case 2: return SortSelect2(S, k); break; case 3: return SortSelect3(S, k); break; case 4: return SortSelect4(S, k); break; case 5: return SortSelect5(S, k); break; default: printf("SortlessThanQ Error: num not in [1..4] num = %d\n", num); return -1; } } int SequentialSelect(int *S, int num, int k) { if (num <= 0) return SortlessThanQ(S, num, k); int cNum = num/Q + 1; int *Medians = new int[cNum]; int i = 0; for (int j = 0; j < num/Q; j++) { Medians[j] = SortSelect5(&S[i], 3); // find medians of subsequences i += Q; } int lastNum = num - (Q * (num / Q)); if (lastNum) { int lastQ = Q * (num / Q); Medians[cNum-1] = SortlessThanQ(&S[lastQ], lastNum, (lastNum+1)/2); } else cNum--; int M = SequentialSelect(Medians, cNum, (cNum+1)/2); delete Medians; int leg[3] = {0, 0, 0}; int *marks = new int[num]; CountAndMark(S, marks, num, M, leg); if (leg[0] >= k) { int *sPack = new int[leg[0]]; ArrayPack(S, sPack, num, marks, 0); delete marks; M = SequentialSelect(sPack, leg[0], k); delete sPack; return M; } else if ((leg[0] + leg[1]) >= k) { delete marks; return M; } else { int *sPack = new int[leg[2]]; ArrayPack(S, sPack, num, marks, 2); delete marks; M = SequentialSelect(sPack, leg[2], k-(leg[0]+leg[1])); delete sPack; return M; } } </pre>	<pre> } } #define swap(A,B) {int t; t = A; A = B; B = t;} int SortSelect2 (int S[], int k) { if (S[0] > S[1]) swap(S[0], S[1]); if (k == 1) return S[0]; else return S[1]; } int SortSelect3 (int S[], int k) { if (S[0] > S[1]) swap(S[0], S[1]); if (S[1] > S[2]) swap(S[1], S[2]); if (S[0] > S[2]) swap(S[0], S[2]); return S[k-1]; } int SortSelect4 (int S[], int k) { if (S[0] > S[1]) swap(S[0], S[1]); if (S[1] > S[2]) swap(S[1], S[2]); if (S[0] > S[2]) swap(S[0], S[2]); if (S[2] > S[3]) swap(S[2], S[3]); if (S[1] > S[3]) swap(S[1], S[3]); if (S[0] > S[3]) swap(S[0], S[3]); return S[k-1]; } int SortSelects (int S[], int k) { if (S[0] > S[1]) swap(S[0], S[1]); if (S[1] > S[2]) swap(S[1], S[2]); if (S[0] > S[2]) swap(S[0], S[2]); if (S[2] > S[3]) swap(S[2], S[3]); if (S[1] > S[3]) swap(S[1], S[3]); if (S[0] > S[3]) swap(S[0], S[3]); if (S[3] > S[4]) swap(S[3], S[4]); if (S[2] > S[4]) swap(S[2], S[4]); if (S[1] > S[4]) swap(S[1], S[4]); if (S[0] > S[4]) swap(S[0], S[4]); if (S[4] > S[5]) swap(S[4], S[5]); if (S[3] > S[6]) swap(S[3], S[6]); if (S[2] > S[6]) swap(S[2], S[6]); if (S[1] > S[6]) swap(S[1], S[6]); if (S[0] > S[6]) swap(S[0], S[6]); return S[k-1]; } void CountAndMark(int S[], int Marks[], int num, int median, int leg[3]) { for (int i = 0; i < num; i++) { if (S[i] == median) {Marks[i] = 1; leg[1]++;} else if (S[i] < median) {Marks[i] = 0; leg[0]++;} else {Marks[i] = 2; leg[2]++;} } } void ArrayPack(int S[], int sPack[], int num, int Marks[], int scansym) { int i, j=0; for (i = 0; i < num; i++) if (Marks[i] == scansym) sPack[j++] = S[i]; } </pre>
--	--

Problem E

Black-Scholes¹

A *European call option* gives its holder the opportunity to purchase from the writer an asset at an agreed expiry time T at an agreed exercise price E . Given a time t , we will let $S(t)$ denote the asset value at time t , so $S(T)$ is the value of the asset at the expiry time. The final payoff to the purchaser is $\max\{S(T) - E, 0\}$, because

- if $S(T) > E$, the option will be exercised for a profit of $S(T) - E$, whereas
- if $S(T) \leq E$, the option will not be exercised.

In 1973, Robert C. Merton published a paper presenting a mathematical model which can be used to calculate a rational price for trading options. In that same year, options were first traded in the open market. Since then, the demand for option contracts has grown to the point that trading options typically far outstrips that for the underlying assets. Merton's work expanded on that of two other researchers, Fischer Black and Myron Scholes, and the pricing model became known as the *Black-Scholes model*. The model depends on a constant σ representing how volatile the market is for the given asset, as well as the continuously compounded interest rate r .

Write a parallel version² of this problem that uses the Monte Carlo technique to calculate the Black-Scholes pricing model to a set of assets.

Input

The input contains only one test case. The first line contains only one integer: the number of assets N ($1 \leq N \leq 32.768$).

The input must be read from the standard input.

Output

The output contains the asset price, each one in separate line.

The output must be written to the standard output.

¹ *Homework 1: A Parallel Monte Carlo Simulation for Black-Scholes Option Valuation*. URL: <http://www.cs.berkeley.edu/~yelick/cs194f07/hw/hw1/> at May, 2013.

² The serial version came from Intel® Development Forum – Brazil 2012.

Example

Input	Output for the input
10	17.07 7.19 25.40 1.23 5.97 11.77 22.74 0.01 1.88 8.17

<pre> #define max #define max(a,b) (((a) > (b)) ? (a) : (b)) #endif const int RAND_N = 1 << 18; static const float RISKFREE = 0.06; static const float VOLATILITY = 0.10; union FPARRAY { float *SPData; double *DPData; }; template<class BaseType> BaseType cdfnorminv(BaseType p) { const BaseType a1 = 2.50662823884; const BaseType a2 = -18.61500062529; const BaseType a3 = 41.39119773534; const BaseType a4 = -25.44106049637; const BaseType b1 = -8.4735109309; const BaseType b2 = 23.08336743743; const BaseType b3 = -21.06224101826; const BaseType b4 = 3.13082909833; const BaseType c1 = 0.337475482272615; const BaseType c2 = 0.976169019091719; const BaseType c3 = 0.16079771491821; const BaseType c4 = 2.76438810333863E-02; const BaseType c5 = 3.8405729373609E-03; const BaseType c6 = 3.951896511919E-04; const BaseType c7 = 3.21767881768E-05; const BaseType c8 = 2.888167364E-07; const BaseType c9 = 3.960315187E-07; BaseType y, z; if (p <= 0 p >= 1.0) { printf("Warning: bad parameter\n"); } y = p - 0.5; if (fabs(y) < 0.42) { z = y * y; z = y * (((a4 * z + a3) * z + a2) * z + a1) / (((b4 * z + b3) * z + b2) * z + b1) * z + 1; } else { if (y > 0) z = log(-log(1.0 - p)); else z = log(-log(p)); z = c1 + z * (c2 + z * (c3 + z * (c4 + z * (c5 + z * (c6 + z * (c7 + z * (c8 + z * (c9))))))))); if (y < 0) z = -z; } return z; } </pre>	<pre> void Montecarlo(float *h_CallResult, float *h_CallConfidence, float *S, float *X, float *T, int OPT_N) { float l_Random[RAND_N]; for (int k = 0; k < RAND_N; k++) l_Random[k] = cdfnorminv<float>((k + 1.0) / (RAND_N + 1.0)); for (int opt = 0; opt < OPT_N; opt++) { float Vbysqrt = VOLATILITY * sqrt(T[opt]); float Mubyt = (RISKFREE - 0.5 * VOLATILITY * VOLATILITY) * T[opt]; float Sval = S[opt]; float Xval = X[opt]; float val = 0.0; val2 = 0.0; for (int pos = 0; pos < RAND_N; pos++) { float callValue = max(0.0, Sval * exp(Mubyt + Vbysqrt * l_Random[pos]) - Xval); val += callValue; val2 += callValue * callValue; } float exprt = expf(-RISKFREE * T[opt]); h_CallResult[opt] = exprt * val / (float) RAND_N; float stdev = sqrtf(((float) RAND_N * val2 - val * val) / ((float) RAND_N * (float) (RAND_N - 1))); h_CallConfidence[opt] = (float) (exprt * 1.96 * stdev / sqrtf((float) RAND_N)); } //end of for int main(int argc, char* argv[]) { FPARRAY CallResultParallel, CallConfidence, StockPrice, OptionStrike, OptionYears; int i, mem_size; int OPT_N; scanf("%d", &OPT_N); mem_size = sizeof(float) * OPT_N; ... // malloc commands for (i = 0; i < OPT_N; i++) { CallResultParallel.SPData[i] = 0.0; CallConfidence.SPData[i] = -1.0; StockPrice.SPData[i] = RandFloat(5.0f, 50.0f); OptionStrike.SPData[i] = RandFloat(10.0f, 25.0f); OptionYears.SPData[i] = RandFloat(1.0f, 5.0f); } Montecarlo(CallResultParallel.SPData, CallConfidence.SPData, StockPrice.SPData, OptionStrike.SPData, OptionYears.SPData, OPT_N); // results in CallResultParallel.SPData[i] for (i = 0; i < OPT_N; i++) printf("%5.2f\n", CallResultParallel.SPData[i]); ... // free commands return 0; } </pre>
--	---

Problem F

Gauss

The solution of sets of linear equations has applications on electrical circuit analysis, simulations of network traffic, simulations of neurons and in various other fields such as chemistry, physics, biology, economics, engineering and social sciences. In this exercise you must solve a system of linear equations given by $Ax=b$, where $A=\{A_{11}, A_{12}, \dots, A_{nn}\}$ is a matrix of size $n \times n$, and $x=\{x_1, x_2, \dots, x_n\}$ and $b=\{b_1, b_2, \dots, b_n\}$ vectors of length n .

To solve a linear system, the program first runs the Gaussian elimination algorithm, which transforms the matrix A into an upper triangular matrix. To this end, for each line i , the algorithm transform in zero all elements from column i for the lines $i+1$ to n . This is accomplished by summing each element from row i to each elements of each row j below it, multiplied by the factor $-A_{ji}/A_{ii}$.

Once transformed into an upper triangular matrix, one can solve the linear system by evaluating the value $x_n = b_n/A_{nn}$. Then we can calculate x_{n-1} and so on.

Your program should generate the linear system using the function `generateLinearSystem(int n, float *A, float *b, int nS)`, which generates nS matrices of size $n \times n$ and a vector b , filled with n values 1. In addition, each solution found is tested using the function `testLinearSystem(float *A, float *b, float *x, int n, int nS)`, which replaces the solutions x of each of the nS linear systems, checking if $Ax=b$.

Input

The input contains only one test case. The first line contains two integers: the size of each linear system N and the number of linear systems to be solved nS , respectively ($N \geq 1.000$; $nS \geq 30$)

The input must be read from the standard input.

Output

The output shows the amount of errors found during the test phase for each line linear system.

The output must be written to the standard output.

Example

Input	Output for the input
1000 30	Errors=0

<pre> int testlinearSystem(float *A, float *b, float *x, int n, int ns) { int i, j; for (i = 0; i < n; i++) { float sum = 0; for (j = 0; j < n; j++) sum += A[i * n + j] * x[j]; if (abs(sum - b[i]) >= 0.001) { return 1; } } return 0; } void generatelinearSystem(int n, float *A, float *b, int ns) { int i, j; for (i = 0; i < n; i++) { for (j = 0; j < n; j++) A[i * n + j] = (1.0 * n + (rand() % n)) / (i + j + 1); } for (i = 0; i < n; i++) b[i] = 1.; } void solvelinearSystem(const float *A, const float *b, float *x, int n) { float *Acpy = (float *) malloc(n * n * sizeof(float)); float *bcpy = (float *) malloc(n * sizeof(float)); memcpy(Acpy, A, n * n * sizeof(float)); memcpy(bcpy, b, n * sizeof(float)); int i, j, count; /* Gaussian Elimination */ for (i = 0; i < (n - 1); i++) { for (j = (i + 1); j < n; j++) { float ratio = Acpy[j * n + i] / Acpy[i * n + i]; for (count = i; count < n; count++) { </pre>	<pre> Acpy[j * n + count] -= (ratio * Acpy[i * n + count]); } bcpy[j] -= (ratio * bcpy[i]); } } /* Back-substitution */ x[n - 1] = bcpy[n - 1] / Acpy[(n - 1) * n + n - 1]; for (i = (n - 2); i >= 0; i--) { float temp = bcpy[i]; for (j = (i + 1); j < n; j++) { temp -= (Acpy[i * n + j] * x[j]); } x[i] = temp / Acpy[i * n + i]; } } int main(int argc, char **argv) { int n, ns; scanf("%d %d", &n, &ns); int i, nneros = 0; float *A = (float *) malloc(ns * n * n * sizeof(float)); float *b = (float *) malloc(ns * n * sizeof(float)); float *x = (float *) malloc(ns * n * sizeof(float)); for (i = 0; i < ns; i++) generatelinearSystem(n, &A[i * n], &b[i * n], ns); for (i = 0; i < ns; i++) solvelinearSystem(&A[i * n], &b[i * n], &x[i * n], n); for (i = 0; i < ns; i++) nneros += testlinearSystem(&A[i * n], &b[i * n], &x[i * n], n, ns); printf("Errors=%d\n", nneros); return EXIT_SUCCESS; } </pre>
---	--