10th Marathon of Parallel Programming

SBAC-PAD – 2015

October 19th, 2015.

Rules for Local Contest

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 script 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 *Local 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 environment.

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 three times with the same input and the mean 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 highest points at the end of the marathon will be declared the winner.

This problem set contains 7 problems; pages are numbered from 1 to 21.

Problem A

Lossless Text Compression

Translating text characters to binary prefix codes is a common way to perform lossless compression of text – *i.e.*, to produce a smaller representation of the data that can be used to reconstruct the original input without any loss. Prefix codes are codifications whose main property is that there is no code word in the system that is a prefix of any other code word in the same system. This trait ensures that the original data can be reconstructed straightforward through a "single pass" on the compressed text, without further parsing.

In this context, the proposed problem can be defined as follows. Given a set of symbols (in this case, text characters) and their weights (usually proportional the number of times it occurs on some input text), find a set of binary prefix codes (one for each symbol) whose expected codeword length is minimum. More formally, given the symbol alphabet $A = \{a_1, a_2, \ldots, a_n\}$ and the set of positive symbol weights $W = \{w_1, w_2, \ldots, w_n\}$, such that $weight(a_i) = w_i$, find a tuple of codewords $C = (c_1, c_2, \ldots, c_n)$, where $c_i = codeword(a_i)$. The goal, thus, is to obtain a C such that, for any other code T, $L(C) < T(C)$, where $L(C) = \sum_{i=1}^{n} w_i \times length(c_i)$ is the weighted path length of code C.

You shall parallelize *Huffman Coding*, an algorithm that generates minimum binary prefix codes for a list of characters and their associated frequencies on some input text. The technique works by creating a binary tree of nodes where each node can be either a leaf node (corresponding to a symbol and its associated frequency) or an internal node (corresponding to a sum of frequencies). Then, the codes are obtained by cumulatively labeling each link between nodes with 0 or 1 in a depth-first visit on the nodes of this tree. The algorithm is detailed next.

Initially, there are only n leaf nodes, one for each symbol, which contains the symbol itself and its weight (frequency). The construction algorithm uses a priority queue sorted by the symbols' weight, where the node with lowest weight is given highest priority:

1. Add all leaves to the priority queue.
2. While there is more than one node in the queue:
    a. Remove the two nodes of highest priority (lowest weight) from the queue;
    b. Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' weights;
    c. Add the new node to the queue.

3. The remaining node is the root node and the tree is complete.

After the tree is assembled, in order to get the binary codes one shall perform a depth-first tree traversal in pre-order. To each leaf corresponds a binary code, built going from the root to it, adding a "0" at each time the path goes by a left child and an "1" when it goes by a right child.

Figure A1 shows an assembled Huffman Tree, with the paths labeled. A given code is delivered by concatenating the zeros and ones from a leaf to the root.

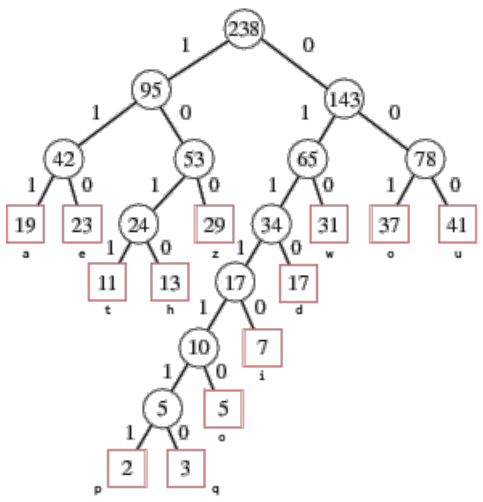

Figure A1. A Huffman Tree. The leaf nodes are represented by pink squares and the numbers inside it represent the frequency of a symbol, which is placed under it. The circles are the intermediary nodes and represent a sum of frequencies. Each edge is labeled with 0 or 1, corresponding to a left or right child (the concatenation of the labels in a given branch is the binary prefix code for a given tree)[1].

Important. Although more than one correct solution exists, your parallel version shall provide the same output as the provided sequential solution.

Input

The first line of the input contains the number of lines of the input text, say N ($0 < N \leq 10^6$). Then, the following N lines are strings of at most 256 ASCII characters that serve as symbols. The only restriction on the characters to appear is that they are readable and

[1] Figure from http://mathworld.wolfram.com/HuffmanCoding.html

no other spacing is present other than the space character.

Blank lines may appear and are valid.

The input must be read from the standard input.

Output

It is also a sequence of lines. Each line corresponds to a symbol and shows its frequency (number of time it appears) on the input text and the correspondent binary code. The format is "`X Y Z`", where `X` is the symbol, `Y` is the frequency, and `Z` is the binary prefix code. For instance, one line could be "`h 13 0110`", meaning the symbol `h` appears `13` times on the input text and has prefix code `0110`. There is only one invariant: if the input text contains the space character it must appear as the word "`space`" on the output instead of the space character.

The output must be written to the standard output.

Example

Input	Output for the input

`2`
`this is an example of a huffman tree`
`j'aime aller sur le bord de l'eau les jeudis`
	`' 2 110111`
	`a 7 1110`
	`b 1 1111000`
	`d 3 10110`
	`e 11 100`
	`f 3 10111`
	`h 2 111101`
	`i 4 11111`
	`j 2 01000`
	`l 6 1100`
	`m 3 11010`
	`n 2 01001`
	`o 2 01010`
	`p 1 1111001`
	`r 4 0110`
	`s 5 1010`
	`space 15 00`
	`t 2 01011`
	`u 4 0111`
	`x 1 110110`

```
class HuffmanTree {

    enum { left = 0, right = 1 } ;

    public :
    Weight weight = Weight {0} ;
    Symbol symbol = Symbol { } ;
    Vector <HuffmanTree> children ;

    HuffmanTree () { }

    HuffmanTree (const HuffmanTree& e) {
        symbol   = e.symbol   ;
        weight   = e.weight   ;
        children = e.children ;
    }

    HuffmanTree& operator= (const HuffmanTree& e) {
        symbol   = e.symbol   ;
        weight   = e.weight   ;
        children = e.children ;
        return *this ;
    }


    HuffmanTree (Symbol s, Weight w) {
        symbol = s ;
        weight = w ;
    }

    HuffmanTree (HuffmanTree a, HuffmanTree b) {
        weight = a.weight + b.weight ;
        children.push_back (a) ; children.push_back (b) ;
        symbol = a.symbol + b.symbol ;
    }

    HuffmanTree left_subtree  () { return children[left ]   ; }
    HuffmanTree right_subtree () { return children[right]   ; }
    bool        is_leaf       () { return children.empty () ; }

    friend bool operator== (HuffmanTree a, HuffmanTree b) {
        return a.symbol == b.symbol && a.weight == b.weight ;
    }

    friend bool operator< (HuffmanTree a, HuffmanTree b) {
        return a.weight == b.weight ? a.symbol < b.symbol : a.weight < b.weight ;
    }

    friend bool operator> (HuffmanTree a, HuffmanTree b) {
        return b < a ;
    }

    friend bool operator<= (HuffmanTree a, HuffmanTree b) {
        return ! (a > b) ;
    }

    friend bool operator>= (HuffmanTree a, HuffmanTree b) {
        return ! (a < b) ;
    }

} ;

HuffmanTree extract (PriorityQueue <HuffmanTree>& p){
    HuffmanTree r ;
    r = p.top () ;
    p.pop () ;
    return r ;
}

Map <Symbol, Code> walk ( HuffmanTree h, Code c = Code (), Map <Symbol, Code> m = Map <Symbol, Code> () ) {
    if (h.is_leaf ()) {
        m.insert ({h.symbol, c}) ;
        return m ;
    }
    m = walk (h.left_subtree  (), c + Code ("0"), m) ;
    m = walk (h.right_subtree (), c + Code ("1"), m) ;
    return m ;
}

PriorityQueue <HuffmanTree> make_queue (Map <Symbol, Weight> m) {
    PriorityQueue <HuffmanTree> q ;
    for (auto i : m)
        q.push (HuffmanTree (i.first, i.second)) ;
    return q ;
}

HuffmanTree huffman (Map <Symbol, Weight> m) {
    assert (! m.empty ()) ; // precondition
    PriorityQueue <HuffmanTree> p = make_queue (m) ;
    HuffmanTree a, b;
    do {
        a = extract (p) ;
        if (p.empty ( )) break ;
        b = extract (p) ;
        p.push (HuffmanTree (a, b)) ;
    } while (true) ;
    return a ;
}

template <typename OutStream>
int write ( OutStream&           out, Map <Symbol, Weight> w, Map <Symbol, Code  > c ) {
    for (auto i : w)
        out << i.first << " " << i.second << " " << c[i.first] << std::endl;
    return 0 ;
}

template <typename I, typename M> bool found (I i, const M& m) {
    return i != std::end (m) ;
}

template <typename Pair> Map <Symbol, Weight> insert_or_add ( Map <Symbol, Weight> m , Pair p ) {
    auto i = m.find (p.first) ;
    if (found (i, m)) i -> second += p.second ;
    else  m.insert (p) ;
    return m ;
}

Map <Symbol, Weight> map_weighted_union ( Map <Symbol, Weight> a , Map <Symbol, Weight> b ) {
    for (auto e : b) a = insert_or_add (a, e) ;
    return a ;
}

template <typename InStream> String read_line (InStream& in){
    String s ;
    std::getline (in, s) ;
    return s ;
}

Symbol to_symbol (char c){
    if (c == ' ') return Symbol ("space") ;
    return Symbol ({c}) ;
}

Map <Symbol, Weight> insert_or_increment( Map <Symbol, Weight> m, Symbol  s ){
    auto i = m.find (s) ;
    if (found (i, m))  ++ (i -> second) ;
    else  m.insert ({s, Weight (1)}) ;
    return m ;
}

template <typename InStream> Map <Symbol, Weight> read_entry (InStream& in) {
    Map <Symbol, Weight> m ;
    for (char c : read_line (in))
        m = insert_or_increment (m, to_symbol (c)) ;
    return m ;
}
```

Problem B

N-Body

In physics, the N-Body problem consists in simulating the gravitational interaction between N particles (bodies) in a system and predicting how the system would evolve in a time frame. In this application the initial position and mass for each particle is randomly generated. The application will compute gravitational forces, positions and velocity of each particle in each time step of the simulation.

We are not interested in finding out different algorithms for computing the N-Body simulation. We just want to focus on obtaining a parallel version of the given code. Therefore, it is not allowed to change the computation method used in this problem.

Input

Input data contains the quantity of particles to be considered ($0 < N \leq 2^{15}$) and the number of time steps to be simulated ($0 < S \leq 100$), separated by a line break.

The input must be read from the standard input.

Output

Output data contains the coordinates of each particle in the $\mathbb{R}^3$. X, Y and Z coordinates are separated by one space and coordinates for each particle are separated by line breaks.

The output must be written to the standard output.

Example

Input	Output for the input

`4`
`10`
	`0.38986 0.38878 0.70927`
	`0.72951 0.07322 0.13581`
	`0.28209 0.51066 0.99480`
	`0.46793 0.72547 0.13771`

```c
typedef struct {
    double x, y, z;
    double mass;
    } Particle;
typedef struct {
    double xold, yold, zold;
    double fx, fy, fz;
    } ParticleV;

int main() {
    double time;
    Particle  * particles;   /* Particles */
    ParticleV * pv;          /* Particle velocity */
    int         npart, i, j;
    int         cnt;         /* number of times in loop */
    double      sim_t;       /* Simulation time */
    int tmp;
    tmp = fscanf(stdin,"%d\n",&npart);
    tmp = fscanf(stdin,"%d\n",&cnt);
/* Allocate memory for particles */
    particles = (Particle *) malloc(sizeof(Particle)*npart);
    pv = (ParticleV *) malloc(sizeof(ParticleV)*npart);
/* Generate the initial values */
    InitParticles( particles, pv, npart);
    sim_t = 0.0;

    while (cnt--) {
      double max_f;
      /* Compute forces (2D only) */
      max_f = ComputeForces( particles, particles, pv, npart );
      /* Once we have the forces, we compute the changes in position */
      sim_t += ComputeNewPos( particles, pv, npart, max_f);
    }
    for (i=0; i<npart; i++)
      fprintf(stdout,"%.5lf %.5lf %.5lf\n", particles[i].x, particles[i].y,
particles[i].z);
    return 0;
}

void InitParticles( Particle particles[], ParticleV pv[], int npart ) {
    int i;
    for (i=0; i<npart; i++) {
   particles[i].x          = Random();
   particles[i].y          = Random();
   particles[i].z          = Random();
   particles[i].mass = 1.0;
   pv[i].xold      = particles[i].x;
   pv[i].yold      = particles[i].y;
   pv[i].zold      = particles[i].z;
   pv[i].fx        = 0;
   pv[i].fy        = 0;
   pv[i].fz        = 0;
    }
}
```

```c
double ComputeForces( Particle myparticles[], Particle others[], ParticleV pv[], int npart ) {
  double max_f;
  int i;
  max_f = 0.0;
  for (i=0; i<npart; i++) {
    int j;
    double xi, yi, mi, rx, ry, mj, r, fx, fy, rmin;
    rmin = 100.0;
    xi   = myparticles[i].x;
    yi   = myparticles[i].y;
    fx   = 0.0; fy   = 0.0;
    for (j=0; j<npart; j++) {
      rx = xi - others[j].x;
      ry = yi - others[j].y;
      mj = others[j].mass;
      r  = rx * rx + ry * ry;
      /* ignore overlap and same particle */
      if (r == 0.0) continue;
      if (r < rmin) rmin = r;
      r  = r * sqrt(r);
      fx -= mj * rx / r;
      fy -= mj * ry / r;
    }
    pv[i].fx += fx;
    pv[i].fy += fy;
    fx = sqrt(fx*fx + fy*fy)/rmin;
    if (fx > max_f) max_f = fx;
  }
  return max_f;
}

double ComputeNewPos( Particle particles[], ParticleV pv[], int npart, double max_f) {
  int i;
  double a0, a1, a2;
  static double dt_old = 0.001, dt = 0.001;
  double dt_new;
  a0        = 2.0 / (dt * (dt + dt_old));
  a2        = 2.0 / (dt_old * (dt + dt_old));
  a1        = -(a0 + a2);
  for (i=0; i<npart; i++) {
    double xi, yi;
    xi               = particles[i].x;
    yi               = particles[i].y;
    particles[i].x = (pv[i].fx - a1 * xi - a2 * pv[i].xold) / a0;
    particles[i].y = (pv[i].fy - a1 * yi - a2 * pv[i].yold) / a0;
    pv[i].xold     = xi;
    pv[i].yold     = yi;
    pv[i].fx       = 0;
    pv[i].fy       = 0;
  }
  dt_new = 1.0/sqrt(max_f);
  /* Set a minimum: */
  if (dt_new < 1.0e-6) dt_new = 1.0e-6;
  /* Modify time step */
  if (dt_new < dt) {
    dt_old = dt;
    dt     = dt_new;
  }
  else if (dt_new > 4.0 * dt) {
    dt_old = dt;
    dt    *= 2.0;
  }
  return dt_old;
}
```

Problem C

LU Decomposition[2]

It is very common to use LU decomposition for solving square systems of linear equations. It factors a matrix as a product of two other matrices: L, which is a *lower* triangular matrix, and U, which is an *upper* triangular matrix:

$$A = L.U$$

In the lower triangular matrix all elements above the diagonal are zero, in the upper triangular matrix, all the elements below the diagonal are zero.

The good solution to this problem is pivoting A, which means rearranging the rows of A, prior to the LU decomposition, in a way that the largest element of each column gets onto the diagonal of A. Rearranging the rows means to multiply A by a permutation matrix P:

$$P.A = L.U$$

Your task is to improve performance of the source-code using parallel strategies. We are not interesting in finding out which decomposition is better; therefore is not allowed to change the LU decomposition.

Input

The input file contains only one test case. The first line contains the size of a square matrix ($0 < N \leq 10^4$). Next N lines are the rows of the matrix, N real numbers per row.

The input must be read from the standard input.

Output

The output must have three matrices in the following order: P, L and U. There must be a

[2] The source and description came from http://rosettacode.org/wiki/LU_decomposition and https://en.wikipedia.org/wiki/LU_decomposition .

blank line between matrices. For each matrix, there are *N* lines and *N* elements per line.

The output must be written to the standard output.

Example 1

Input

```
3
1 3 5
2 4 7
1 1 0
```

Output for the input

```
0.00000 1.00000 0.00000
1.00000 0.00000 0.00000
0.00000 0.00000 1.00000

1.00000 0.00000 0.00000
0.50000 1.00000 0.00000
0.50000 -1.00000 1.00000

2.00000 4.00000 7.00000
0.00000 1.00000 1.50000
0.00000 0.00000 -2.00000
```

Example 2

Input

```
3
35.70631 76.06820 38.30932
37.57327 32.90746 151.85924
89.49273 39.05083 108.00119
```

Output for the input

```
0.00000 0.00000 1.00000
1.00000 0.00000 0.00000
0.00000 1.00000 0.00000

1.00000 0.00000 0.00000
0.39899 1.00000 0.00000
0.41985 0.27298 1.00000

89.49273 39.05083 108.00119
0.00000 60.48748 -4.78160
0.00000 0.00000 107.82054
```

```
typedef double **mat;

void mat_del(mat x) {
  free(x[0]);
  free(x);
}

mat mat_new(int n) {
  int i;
  mat x = malloc(sizeof(double*) * n);
  x[0]  = malloc(sizeof(double) * n * n);
  if(x[0] == NULL)
    perror("memory other failure");

  for(i=0; i<n; i++) {
    x[i] = x[0] + n * i;
  }
  mat_zero(x,n);

  return x;
}

mat mat_mul(const mat a, const mat b, int n) {
  int i,j,k;
  mat c = mat_new(n);
  for(i=0; i<n; i++) {
    for(k=0; k<n; k++) {
      for(j=0; j<n; j++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return c;
}

void mat_pivot(mat a, mat p, int n) {
  int i,j,k;
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
      p[i][j] = (i == j);
    }
  }
  for(i=0; i<n; i++) {
    int max_j = i;
    for(j=i; j<n; j++) {
      if (fabs(a[j][i]) > fabs(a[max_j][i]))
        max_j = j;
    }
    if (max_j != i) {
      for(k=0; k<n; k++) {
        double tmp;
        tmp = p[i][k];
        p[i][k] = p[max_j][k];
        p[max_j][k] = tmp;
      }
    }
  }
}
```

```
void mat_LU(mat A, mat L, mat U, mat P, int n) {
  int i,j,k;
  mat_zero(L,n);
  mat_zero(U,n);
  mat_pivot(A, P, n);

  mat Aprime = mat_mul(P, A, n);

  for(i=0; i<n; i++) {
    L[i][i] = 1;
  }
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
      double s;
      if (j <= i) {
        s=0;
        for(k=0; k<j; k++) {
          s += L[j][k] * U[k][i];
        }
        U[j][i] = Aprime[j][i] - s;
      }
      if (j >= i) {
        s=0;
        for(k=0; k<i; k++) {
          s += L[j][k] * U[k][i];
        }
        L[j][i] = (Aprime[j][i] - s) / U[i][i];
      }
    }
  }
  mat_del(Aprime);
}

int main() {
  int n;
  mat A, L, P, U;

  fscanf(stdin,"%d",&n);
  A = mat_new(n);
  L = mat_new(n);
  P = mat_new(n);
  U = mat_new(n);

  mat_gen(A,n);

  mat_LU(A, L, U, P, n);

  mat_show(P,"P",0,n);
  printf("\n");
  mat_show(L,"L",0,n);
  printf("\n");
  mat_show(U,"U",0,n);

  mat_del(A);
  mat_del(L);
  mat_del(U);
  mat_del(P);

  return 0;
}
```

Problem D

The K-Means Clustering Problem

Clustering analysis plays an important role in different fields, including data mining, pattern recognition, image analysis and bioinformatics. In this context, a widely used and studied clustering approach is the K-Means clustering. Formally, the K-Means clustering problem can be defined as follows. Given a set of n points in a real d-dimensional space, the problem is to partition these n points into k partitions, so as to minimize the mean squared distance from each point to the center of the partition it belongs to. Figure D1 illustrates an instance of this problem.

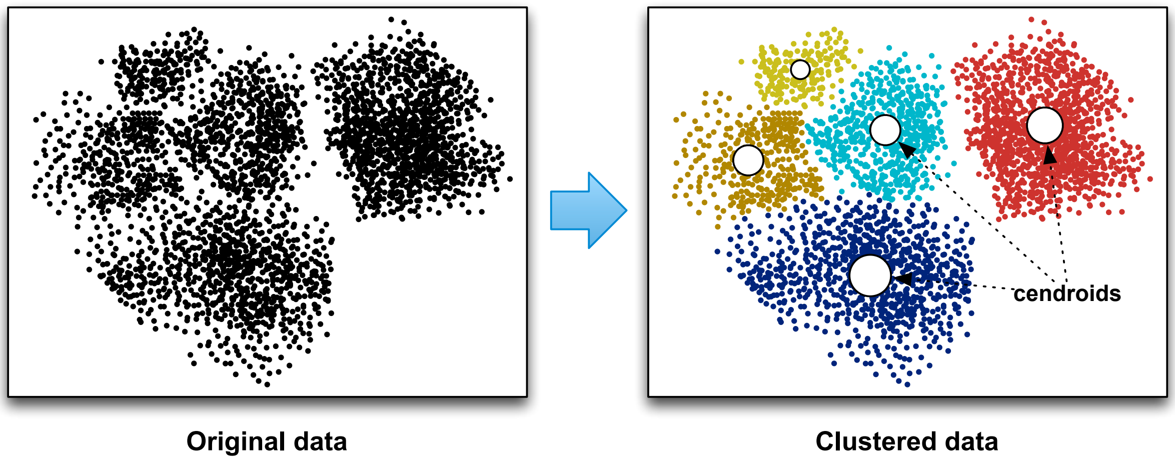

Figure D1. An example of K-Means with 5 partitions.

The general solution to this problem has been shown to be NP-Hard, even with just two clusters (and an arbitrary number of dimensions) or on the plane (for an arbitrary number of clusters). For this reason, several distinct heuristics and approximation algorithms have been proposed to address the K-Means clustering problem. One of the most widely employed is Lloyd's algorithm, also known as K-Means algorithm. Such heuristic is based on an iterative strategy that finds a local minimum solution for the problem.

The algorithm takes as input the set of data points, the number of partitions, and the minimum accepted distance between each point and the partition's center (centroids). Upon completion, the algorithm returns the partitions themselves. Initially, data points are evenly and randomly distributed among the k partitions, and the initial centroids are computed. Then, the data points are re-clustered into partitions taking into account the minimum Euclidean distance between them and the centroids – points are assigned to the nearest partition. Next, the centroid of each partition is recalculated taking the mean of all points in the partition, and the whole procedure is repeated until no centroid is changed and every point is farther than the minimum accepted distance.

Write a parallel version of the sequential program which outputs the clustering solution found. Pay special attention to the parallelization strategy: since this is not an exact algorithm, the order of execution might influence the results. The only acceptable output for your program is one that is identical to the sequential version's output.

Input

The program must read 5 input parameters. The parameters will be given in the following order: number of points ($0 < P \leq 2^{16}$), dimensions ($0 < D \leq 2^{4}$), number of clusters ($0 < C \leq 2^{10}$), minimum distance between points and centroids ($0 < M \leq 10$), and the seed for the random number generator ($0 < S \leq 2^{16}$).

The input must be read from the standard input.

Output

For each point, the program should print a line containing its cluster number on the final solution.

The output must be written to the standard output.

Example

Input	Output for the input

`20 2 4 0.0 321`	`1`
	`0`
	`3`
	`2`
	`3`
	`1`
	`0`
	`3`
	`0`
	`2`
	`3`
	`2`
	`2`
	`0`
	`1`
	`2`
	`1`
	`1`
	`2`
	`0`

```c
typedef float* vector_t;

int npoints;
int dimension;
int ncentroids;
float mindistance;
int seed;
vector_t *data;
int *map;
vector_t *centroids;
int *dirty;
int too_far;
int has_changed;

float v_distance(vector_t a, vector_t b) {
   int i;
   float distance = 0;
   for (i = 0; i < dimension; i++)
        distance +=  pow(a[i] - b[i], 2);
   return sqrt(distance);
}

static void populate(void) {
   int i, j;
   float tmp;
   float distance;
   too_far = 0;
   for (i = 0; i < npoints; i++) {
        distance = v_distance(centroids[map[i]], data[i]);
        /* Look for closest cluster. */
        for (j = 0; j < ncentroids; j++) {
                /* Point is in this cluster. */
                if (j == map[i]) continue;
                tmp = v_distance(centroids[j], data[i]);
                if (tmp < distance) {
                        map[i] = j;
                        distance = tmp;
                        dirty[j] = 1;
                }
        }
        /* Cluster is too far away. */
        if (distance > mindistance)
                too_far = 1;
   }
}

static void compute_centroids(void) {
   int i, j, k;
   int population;
   has_changed = 0;
   /* Compute means. */
   for (i = 0; i < ncentroids; i++) {
        if (!dirty[i]) continue;
        memset(centroids[i], 0, sizeof(float) * dimension);
        /* Compute cluster's mean. */
        population = 0;
        for (j = 0; j < npoints; j++) {
                if (map[j] != i)
                        continue;
                for (k = 0; k < dimension; k++)
                        centroids[i][k] += data[j][k];
                population++;
        }
        if (population > 1) {
                for (k = 0; k < dimension; k++)
                        centroids[i][k] *= 1.0/population;
        }
        has_changed = 1;
   }
   memset(dirty, 0, ncentroids * sizeof(int));
}

int* kmeans(void) {
   int i, j, k;
   too_far = 0;
   has_changed = 0;
   if (!(map  = calloc(npoints, sizeof(int)))) exit (1);
   if (!(dirty = malloc(ncentroids*sizeof(int)))) exit (1);
   if (!(centroids = malloc(ncentroids*sizeof(vector_t)))) exit (1);
   for (i = 0; i < ncentroids; i++)
        centroids[i] = malloc(sizeof(float) * dimension);
   for (i = 0; i < npoints; i++) map[i] = -1;
   for (i = 0; i < ncentroids; i++) {
        dirty[i] = 1;
        j = randnum() % npoints;
        for (k = 0; k < dimension; k++)
                centroids[i][k] = data[j][k];
        map[j] = i;
   }
   /* Map unmapped data points. */
   for (i = 0; i < npoints; i++)
        if (map[i] < 0) map[i] = randnum() % ncentroids;

   do   {/* Cluster data. */
        populate();
        compute_centroids();
   } while (too_far && has_changed);

   for (i = 0; i < ncentroids; i++)
        free(centroids[i]);
   free(centroids);
   free(dirty);
   return map;
}
```

Problem E

Raytracer[3]

Ray tracing is a well-known technique to generate images. It simulates the light ray in a 3D scene colliding it with virtual objects and creating the final 2D image. The image represents the viewport that could be either a camera or a user.

There are two basics algorithms for ray tracing. The forwarding ray tracing creates the image by traveling the light rays from the source (simulation of photons). This is what really happens in the real world: from the Sun to your eyes.

The backward ray tracing (also called eye tracing) creates the image simulating a ray from the source (you eye) to the objects, in a reverse direction from what happens in real life. Figure E1 shows an example for the backward ray tracing.

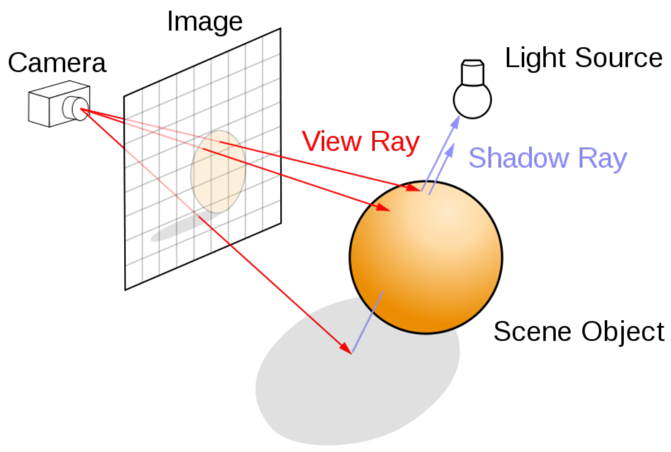

Figure E1. The ray-tracing algorithm builds an image by extending rays into a scene.

Write a parallel version for the serial ray tracer that comes with this problem. Be carefully because the final image must be the same for serial and parallel versions.

Input

The input file contains only one test case. The first line contains the width ($0 < W \leq 7680$) and height ($0 < H \leq 4320$) of an image. The second line has two integers: the number of spheres ($0 < S \leq 1024$) and the number of lights ($0 < L \leq 1024$) in the scene.

[3] The source and description came from http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell and https://en.wikipedia.org/wiki/Ray_tracing_(graphics) .

The next S lines contain the description of spheres, one per line, as follows: the position X_s, Y_s, and Z_s of a sphere ($|X_s|$, $|Y_s|$, $|Z_s| \leq 10^5$), its radius ($0 < r_s \leq 10^5$), the RGB surface color ($0.0 \leq R_s$, G_s, $B_s \leq 1.0$), the reflection factor ($0.0 \leq F_s \leq 1.0$) and the transparency factor ($0.0 \leq T_s \leq 1.0$).

The next L lines contain the description of lights, one per line, as follows: the position X_L, Y_L, and Z_L of a light ($|X_L|$, $|Y_L|$, $|Z_L| \leq 10^5$), its radius ($0 < r_L \leq 10^5$), the RGB surface color ($0.0 \leq R_L$, G_L, $B_L \leq 1.0$), the reflection factor ($0.0 \leq F_L \leq 1.0$), the transparency factor ($0.0 \leq T_L \leq 1.0$) and the emission color ($C_L > 0.0$). A light is a specialized sphere.

The input must be read from the standard input.

Output

The output is a text file using PPM format[4].

The output must be written to a file named *image.ppm* *. You can open this image in your preferred image viewer.*

Example

Input

```
1024 768
3 1
0.0 0.0      -20.0 4.0     1.0  0.32 0.36 1.0 0.5
5.0 0.0      -25.0 3.0     0.15 0.30 0.97 0.0 0.0
0.0 -10004.0 -20.0 10000.0 0.20 0.20 0.20 0.0 0.0
0.0 20.0 -30.0 3.0 0.0 0.0 0.0 0.0 0.0 3
```

[4] More information about this format can be found at http://netpbm.sourceforge.net/doc/ppm.html .

```
Vec3f trace(const Vec3f &rayorig,const Vec3f &raydir,const std::vector<Sphere> &spheres,const int
&depth) {
    float tnear = INFINITY;
    const Sphere* sphere = NULL;
    for (unsigned i = 0; i < spheres.size(); ++i) {
        float t0 = INFINITY, t1 = INFINITY;
        if (spheres[i].intersect(rayorig, raydir, t0, t1)) {
            if (t0 < 0) t0 = t1;
            if (t0 < tnear) {
                tnear = t0;
                sphere = &spheres[i];
            }
        }
    }
    if (!sphere) return Vec3f(2);
    Vec3f surfaceColor = 0; // color of the ray/surfaceof the object intersected by the ray
    Vec3f phit = rayorig + raydir * tnear; // point of intersection
    Vec3f nhit = phit - sphere->center; // normal at the intersection point
    nhit.normalize(); // normalize normal direction
    float bias = 1e-4; // add some bias to the point from which we will be tracing
    bool inside = false;
    if (raydir.dot(nhit) > 0) nhit = -nhit, inside = true;
    if ((sphere->transparency > 0 || sphere->reflection > 0) && depth < MAX_RAY_DEPTH) {
        float facingratio = -raydir.dot(nhit);
        float fresneleffect = mix(pow(1 - facingratio, 3), 1, 0.1);
        Vec3f refldir = raydir - nhit * 2 * raydir.dot(nhit);
        refldir.normalize();
        Vec3f reflection = trace(phit + nhit * bias, refldir, spheres, depth + 1);
        Vec3f refraction = 0;
        if (sphere->transparency) {
            float ior = 1.1, eta = (inside) ? ior : 1 / ior; // are we inside or outside the surface?
            float cosi = -nhit.dot(raydir);
            float k = 1 - eta * eta * (1 - cosi * cosi);
            Vec3f refrdir = raydir * eta + nhit * (eta *  cosi - sqrt(k));
            refrdir.normalize();
            refraction = trace(phit - nhit * bias, refrdir, spheres, depth + 1);
        }
        surfaceColor = (
            reflection * fresneleffect +
            refraction * (1 - fresneleffect) * sphere->transparency) * sphere->surfaceColor;
    }
    else {
        for (unsigned i = 0; i < spheres.size(); ++i) {
            if (spheres[i].emissionColor.x > 0) {
                Vec3f transmission = 1;
                Vec3f lightDirection = spheres[i].center - phit;
                lightDirection.normalize();
                for (unsigned j = 0; j < spheres.size(); ++j) {
                    if (i != j) {
                        float t0, t1;
                        if (spheres[j].intersect(phit + nhit * bias, lightDirection, t0, t1)) {
                            transmission = 0;
                            break;
                        }
                    }
                }
                surfaceColor += sphere->surfaceColor * transmission *
                std::max(float(0), nhit.dot(lightDirection)) * spheres[i].emissionColor;
            }
        }
    }

    return surfaceColor + sphere->emissionColor;
}
```

```
void render(Vec3f* image, unsigned width, unsigned height, const std::vector<Sphere> &spheres)
{
    Vec3f *pixel = image;
    float invWidth = 1 / float(width), invHeight = 1 / float(height);
    float fov = 30, aspectratio = width / float(height);
    float angle = tan(M_PI * 0.5 * fov / 180.);
    // Trace rays
    for (unsigned y = 0; y < height; ++y) {
        for (unsigned x = 0; x < width; ++x, ++pixel) {
            float xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
            float yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
            Vec3f raydir(xx, yy, -1);
            raydir.normalize();
            *pixel = trace(Vec3f(0), raydir, spheres, 0);
        }
    }
}

//[comment]
// In the main function, we will create the scene which is composed of some spheres
// and some light (which is also a sphere). Then, once the scene description is complete
// we render that scene, by calling the render() function.
//[/comment]
int main(int argc, char **argv)
{
    unsigned width, height;
    unsigned s, l;
    scanf("%u %u\n", &width, &height);
    scanf("%u %u\n", &s, &l);
    std::vector<Sphere> spheres;
    // spheres
    for(int i=0; i<s; i++) {
      float x, y, z, r, refl, transp;
      scanf("%f %f %f %f ", &x, &y, &z, &r);
      Vec3f position(x, y, z);
      scanf("%f %f %f %f %f\n", &x, &y, &z, &refl, &transp);
      RGB color(x, y, z);
      // position, radius, surface color, reflectivity, transparency, emission color
      spheres.push_back(Sphere(position, r, color, refl, transp, ec));
    }
    // lights
    for(int i=0; i<l; i++) {
      float x, y, z, r, refl, transp, ec;
      scanf("%f %f %f %f ", &x, &y, &z, &r);
      Vec3f position(x, y, z);
      scanf("%f %f %f %f %f %f\n", &x, &y, &z, &refl, &transp, &ec);
      RGB color(x, y, z);
      // position, radius, surface color, reflectivity, transparency, emission color
      spheres.push_back(Sphere(position, r, color, refl, transp, ec));
    }
    Vec3f *image = new Vec3f[width * height];

    render(image, width, height, spheres);

    save(image, width, height);

          delete image;

    return 0;
}
```

Problem F

Karatsuba Multiplication[5]

There is some different ways to multiply two large numbers. The simplest technique is learned in school and can be called "brute-force". In 1962, Karatsuba and Ofman discovered that a multiplication of two n-digit numbers could be done with a bit complexity of less than n^2 using identities the form:

$$(a + b \cdot 10^n)(c + d \cdot 10^n) = ac + [(a + b)(c + d) - ac - bd] \cdot 10^n + bd \cdot 10^{2n}$$

A recursively algorithm for this technique has $\Theta(n^{\log_2 3})$ elementary steps.

Write a parallel version for the Karatsuba multiplication for large natural numbers.

Input

The input file contains only one test case. The test case has two natural large numbers, each one in a different line ($0 \leq A, B < 10^{2^{22}}$).

The input must be read from the standard input.

Output

The output has the multiplication result between those two numbers.

The output must be written to the standard output.

Example

Input	Output for the input

`1234567890`
`9876543210`
	`12193263111263526900`

[5] The source and description came from http://mathworld.wolfram.com/KaratsubaMultiplication.html and http://www.cburch.com/proj/karat/ .

```c
#define MAX_DIGITS 2097152
#define KARAT_CUTOFF 4

int             a[MAX_DIGITS]; // first multiplicand
int             b[MAX_DIGITS]; // second multiplicand
int             r[6 * MAX_DIGITS]; // result goes here

int
main() {
  int             d_a; // length of a
  int             d_b; // length of b
  int             d; // maximum length
  int             i; // counter

  getNum(a, &d_a);
  getNum(b, &d_b);

  if(d_a < 0 || d_b < 0) {
    printf("0\n");
    exit(0);
    return 0;
  }

  i = (d_a > d_b) ? d_a : d_b;
  for(d = 1; d < i; d *= 2);
  for(i = d_a; i < d; i++) a[i] = 0;
  for(i = d_b; i < d; i++) b[i] = 0;

  karatsuba(a, b, r, d); // compute product w/o regard to carry
  doCarry(r, 2 * d); // now do any carrying

  printNum(r, 2 * d);

  return 0;
}

void
karatsuba(int *a, int *b, int *ret, int d) {
  int             i;
  int             *ar = &a[0]; // low-order half of a
  int             *al = &a[d/2]; // high-order half of a
  int             *br = &b[0]; // low-order half of b
  int             *bl = &b[d/2]; // high-order half of b
  int             *asum = &ret[d * 5]; // sum of a's halves
  int             *bsum = &ret[d * 5 + d/2]; // sum of b's halves
  int             *x1 = &ret[d * 0]; // ar*br's location
  int             *x2 = &ret[d * 1]; // al*bl's location
  int             *x3 = &ret[d * 2]; // asum*bsum's location

  if(d <= KARAT_CUTOFF) {
    gradeSchool(a, b, ret, d);
    return;
  }
```

```c
  for(i = 0; i < d / 2; i++) {
    asum[i] = al[i] + ar[i];
    bsum[i] = bl[i] + br[i];
  }

  karatsuba(ar, br, x1, d/2);
  karatsuba(al, bl, x2, d/2);
  karatsuba(asum, bsum, x3, d/2);

  for(i = 0; i < d; i++) x3[i] = x3[i] - x1[i] - x2[i];
  for(i = 0; i < d; i++) ret[i + d/2] += x3[i];
}

void
gradeSchool(int *a, int *b, int *ret, int d) {
  int             i, j;

  for(i = 0; i < 2 * d; i++) ret[i] = 0;
  for(i = 0; i < d; i++) {
    for(j = 0; j < d; j++) ret[i + j] += a[i] * b[j];
  }
}

void
doCarry(int *a, int d) {
  int             c;
  int             i;

  c = 0;
  for(i = 0; i < d; i++) {
    a[i] += c;
    if(a[i] < 0) {
      c = -(-(a[i] + 1) / 10 + 1);
    } else {
      c = a[i] / 10;
    }
    a[i] -= c * 10;
  }
  if(c != 0) fprintf(stderr, "Overflow %d\n", c);
}
```

Problem G

Bitonic Sort[6]

Bitonic sort is another comparison-based sorting algorithm. It divides the sequence into two bitonic sequences: given an a_i element of a sequence with n elements ($i < n$):

$$a_0 \leq a_1 \leq \ldots \leq a_{i-1} \leq a_i \geq a_{i+1} \geq \ldots \geq a_{n-2} \geq a_{n-1}$$

For each generated bitonic sequence, it runs again dividing into more two bitonic sub-sequences. The algorithm stops when all the elements are in their final position. Figure G1 shows the comparison between numbers and the bitonic sub-sequence for each step.

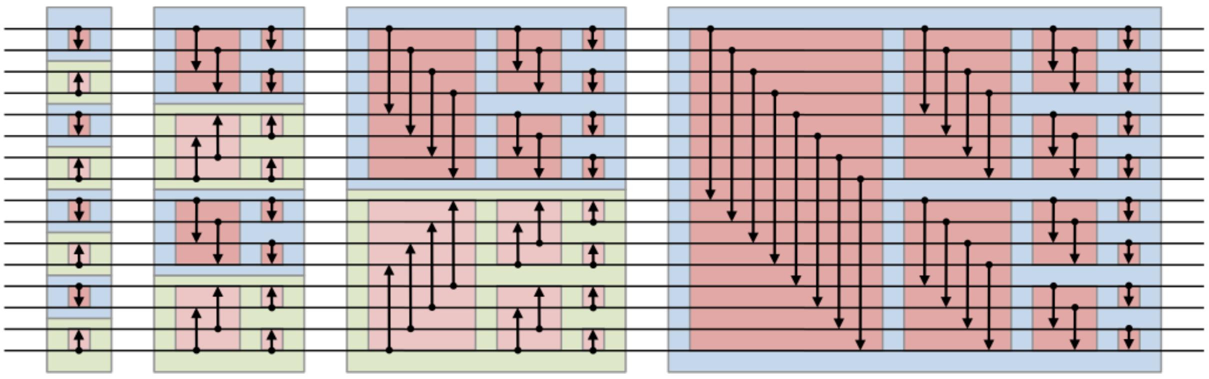

Figure G1. Bitonic sorting with 16 elements[7].

Write a parallel version of the Bitonic Sort algorithm.

Input

The input file contains only one test case. The first line contains the total number of keys (N) to be sorted ($N \leq 2^k$, $0 \leq k \leq 30$). The following lines contain N keys, each key in a separate line. A key is a seven-character string made up of printable characters (0x21 to 0x7E – ASCII) not including the space character (0x20 ASCII).

The input must be read from a file named sort.in

[6] The source and description came from http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm and https://www.cs.duke.edu/courses/fall08/cps196.1/Pthreads/bitonic.c .
[7] Figure from https://en.wikipedia.org/wiki/Bitonic_sorter .

Output

The output file contains the sorted keys. Each key must be in a separate line.

The output must be written to a file named *sort.out*

Example

Input	Output for the input
`8`	`1234567`
`SINAPAD`	`CTDeWIC`
`SbacPad`	`MPP2015`
`Wscad15`	`SINAPAD`
`Sinapad`	`SbacPad`
`1234567`	`Sinapad`
`WEAC-15`	`WEAC-15`
`CTDeWIC`	`Wscad15`
`MPP2015`	

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define LENGTH 8

FILE *fin, *fout;
char *strings;
long int N;

unsigned long int powersOfTwo[] =
{1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
  65536,131072,262144,524288,1048576,2097152,4194304,8388608,
  16777216,33554432,67108864,134217728,268435456,536870912,
  1073741824};

#define ASCENDING  1
#define DESCENDING 0

void openfiles() {
  fin = fopen("sort.in", "r+");
  if (fin == NULL ) {
    perror("fopen fin");
    exit(EXIT_FAILURE);
  }

  fout = fopen("sort.out", "w");
  if (fout == NULL ) {
    perror("fopen fout");
    exit(EXIT_FAILURE);
  }
}

void closefiles(void) {
  fclose(fin);
  fclose(fout);
}

void compare(int i, int j, int dir) {
  if (dir==(strcmp(strings+i*LENGTH,strings+j*LENGTH) > 0)) {
    char t[LENGTH];
    strcpy(t, strings+i*LENGTH);
    strcpy(strings+i*LENGTH, strings+j*LENGTH);
    strcpy(strings+j*LENGTH, t);
  }
}

void bitonicMerge(int lo, int cnt, int dir) {
  if (cnt>1) {
    int k=cnt/2;
    int i;
    for (i=lo; i<lo+k; i++)
      compare(i, i+k, dir);
    bitonicMerge(lo, k, dir);
    bitonicMerge(lo+k, k, dir);
  }
}

void recBitonicSort(int lo, int cnt, int dir) {
  if (cnt>1) {
    int k=cnt/2;
    recBitonicSort(lo, k, ASCENDING);
    recBitonicSort(lo+k, k, DESCENDING);
    bitonicMerge(lo, cnt, dir);
  }
}

void BitonicSort() {
  recBitonicSort(0, N, ASCENDING);
}

int main(int argc, char **argv) {

  long int i;

  openfiles();

  fscanf(fin, "%ld", &N);

  if(N > 1073741824 || powersOfTwo[(int)log2(N)] != N) {
    printf("%ld is not a valid number: power of 2 or less than 1073741824!\n",
N);
    exit(EXIT_FAILURE);
  }

  strings = (char*) calloc(N, LENGTH);
  if (strings == NULL ) {
    perror("malloc strings");
    exit(EXIT_FAILURE);
  }

  for (i = 0; i < N; i++)
    fscanf(fin, "%s", strings + (i * LENGTH));

  BitonicSort();

  for (i = 0; i < N; i++)
    fprintf(fout, "%s\n", strings + (i * LENGTH));

  free(strings);
  closefiles();

  return EXIT_SUCCESS;
}
```