9th Marathon of Parallel Programming

WSCAD – 2014

October 9th, 2014.

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 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 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

The Energy Minimization Cooking Problem

There is a very famous restaurant known by its customer service called *The Optimal*. Its Chef is known for being the fastest Chef in the world; so fast that he can actually run the entire restaurant *by itself*, no matter how busy the restaurant is.

One of the features that made this restaurant famous is that the customer can choose when to place the order and also choose when the order must be delivered, i.e., every order that arrives has a deadline attached: the client wants his meal before a time that he sets himself. There is one restriction, though: if a customer places his order after other customers, his deadline must also be after than the ones before.

Obviously each dish has a certain degree of difficulty, that we will call the "volume". Since each order arrives at a certain time, have a certain degree of difficulty and also have a deadline, the Chef must choose how fast he must be in order to respect all its clients' wishes.

The Chef gets tired when he works fast. The fastest he cooks, the more tired he gets.

If, at time *t*, he cooks at speed *s*, the "power" consumed by him can be expressed by the function $P(s(t)) = s(t)^3$, and the total energy of an entire night's work is defined as $E(S) = \int_0^1 P\big(s(t)\big)dt$. He quickly realized that in order to minimize his tiredness he would need to minimize the total energy spent.

One day, reading his favorite publication, the *Proceedings of the Symposium on Foundations of Computer Science*, he learned that the YDS algorithm[1], developed to optimize the energy consumption of a CPU, could be used by him to minimize the energy spent by him. He would only need to know (beforehand) the list of all dishes that will be ordered in one night.

Help the Chef find out the minimum amount of energy that he needs to cook all the dishes, given that he knows everything beforehand.

[1] Frances Yao, Alan Demers, and Scott Shenker. "*A scheduling model for reduced CPU energy*". In: *Proceedings of the Symposium on Foundations of Computer Science*. Ed. by Allan Borodin and Prabbakar Ragbavan. IEEE Computer Society, Oct. 1995, pp. 374–382. doi: 10.1109/SFCS.1995.492493

Input

The input consists of a single test that must be read from standard input. It starts with an integer *n*, the number of orders. Then follows *n* lines that describes the orders. Each order is fully described by 3 integers, r_i, d_i and w_i ($0 \leq r_i, d_i \leq 10^3$, $1 \leq w_i \leq 10^3$), the time in which the order arrives, the order's "deadline" and the order's "volume".

The input must be read from the standard input.

Output

Output a single real number, the minimum energy required for the Chef to cook all the dishes.

The output must be written to the standard output.

Example

Input	Output for the input
3 1 2 1 2 5 2 2 5 3	14.8889

Extra Information

Current dynamic voltage scaling techniques allow the speed of processors to be set dynamically to save energy consumption. This allows the operating system to manage a trade-off between performance and power (energy) consumption.

A theoretical study of speed scaling scheduling was initiated by Yao, Demers, and Shenker. In their model, the power *P*, or energy consumed per unit time, is a convex function of the processor speed *s*(*t*). The total energy consumed by a schedule *S* is $E(S) = \int_0^1 P\big(s(t)\big)dt$, with $P(s(t)) = s(t)^3$. The goal of the min-energy scheduling problem is to find, for any given job set *J*, a feasible schedule that minimizes *E*(*S*). It is assumed the processor *speed may be set at any real value*.

The YDS algorithm is a simple, yet clever, algorithm that can optimally solve the problem for the following scenario. Suppose that we have one CPU and we want to compute the optimal speed of the processor for any given time. Also suppose that each job $j \in J$ has a release time r_j (meaning that the job cannot start before time $t = r_j$), a deadline d_j (meaning that the job cannot end before time $t = d_j$ and that it must run for w_j CPU cycles.

The base of the YDS algorithm is the concept of *critical interval* for *J*, which is an interval *I* in which a group of jobs must be scheduled at maximum, constant speed *g*(*I*) in any optimal schedule for *J*. For any interval *I*, let J_I denote the subset of all jobs in *J* whose intervals are completely contained in *I*. Then $g(i) = \sum_{j_k \in J_I} w_k / |I|$ is the minimum speed needed to execute all jobs J_I respecting their release dates and deadlines. We call I^* the maximum density interval, i.e., the interval achieving maximum *g*(*I*) over all possible intervals *I*. The pseudo-code for the YDS algorithm is presented next.

```
Data: job set J
Result: optimal voltage schedule S for J
repeat
    Select I* = [z, z'] with g(I*) = max g(I);
    Sort jobs j_i ∈ J_I* according to non-decreasing release dates;
    Schedule j_i ∈ J_I* at g(I*) over I* in sorted order;
    J ← J − J_I*;
    foreach j_k ∈ J do
        if d_k ∈ [z, z'] then
            d_k ← z
        else if d_k ≥ z' then
            d_j ← d_k − (z' − z)
        end
        Reset release dates r_k similarly;
    end
until J is empty;
```

The algorithm interactively find the maximum density interval I^* of the set of unscheduled jobs, and execute them in non-decreasing order of their release dates at speed $g(I^*)$; i.e., $s(t) = g(I^*)$, $\forall t \in I^*$. After that, the interval I^* (and its jobs) is removed and the release dates and deadlines of the remaining jobs intersecting (but not included in) I^* are updated.

```cpp
#include<vector>
#include<set>
#include<iostream>
#include<cmath>
#define ALPHA 3
using namespace std;

struct job {
    int r, d, w;
};
struct interval {
    int r, d, time;
    double dens;
    mutable vector<job>jobs_inside;
    bool operator<(const interval& inter) const {
        if(r < inter.r) return true;
        if(r == inter.r && d < inter.d) return true;
        return false;
    }
};

const double energy_consumption(const vector<interval>& sch){
    double energy = 0.0;
    for(size_t i = 0; i < sch.size(); i++){
        double dens = sch[i].dens;
        int inter = sch[i].d - sch[i].r;
        energy += pow(dens, ALPHA)*inter;
    }
    return energy;
}

set<interval> build_intervals(const vector<job> &jobs){
    set<interval> intervals;
    for(size_t i = 0; i < jobs.size(); i++){
        for(size_t j = 0; j < jobs.size(); j++){
            if(jobs[i].r >= jobs[j].d) continue;
            interval inter {jobs[i].r, jobs[i].d, jobs[i].d - jobs[i].r, 0.0};
            intervals.insert(inter);
        }
    }
    return intervals;
}

interval find_maximum_density_interval(const vector<job>& jobs){
    set<interval> intervals = build_intervals(jobs);
    double max_dens = -1.0;
    interval max_interval;
    for(set<interval>::iterator it = intervals.begin(); it!= intervals.end(); ++it){
        int inter_r = it->r, inter_d = it->d;
        double dens = 0.0;
        for(size_t i = 0; i < jobs.size(); i++){
            if(jobs[i].r >= inter_r && jobs[i].d <= inter_d){
                it->jobs_inside.push_back(jobs[i]);
                dens += jobs[i].w;
            }
        }
        dens /= it->time;
        if(dens >= max_dens) {
            max_dens = dens;
            max_interval = *it;
            max_interval.dens = dens;
        }
    }
    return max_interval;
}

void remove_jobs_from_interval(vector<job>& jobs, const interval inter){
    for(size_t i = 0; i < jobs.size(); i++){
        int inter_r = inter.r, inter_d = inter.d;
        if(jobs[i].r >= inter_r && jobs[i].d <= inter_d){
            jobs.erase(jobs.begin() + i);
            i--;
        }
    }
}

void adjust_jobs_given_interval(vector<job>& jobs, const interval inter){
    for(size_t i = 0; i < jobs.size(); i++){
        int inter_r = inter.r, inter_d = inter.d;
        if(jobs[i].r >= inter_r && jobs[i].r <= inter_d && jobs[i].d >= inter_d){
            jobs[i].r = inter_d;
        } else if(jobs[i].r <= inter_r && jobs[i].d >= inter_r && jobs[i].d <= inter_d){
            jobs[i].d = inter_r;
        }
    }
}

vector<interval> yds(vector<job>jobs){
    vector<interval> schedule;
    while(!jobs.empty()) {
        // find maximum density interval
        interval max_interval = find_maximum_density_interval(jobs);

        // put max density interval on the schedule
        schedule.push_back(max_interval);

        // remove max_interval jobs from the total jobs
        remove_jobs_from_interval(jobs, max_interval);

        // adjust jobs
        adjust_jobs_given_interval(jobs, max_interval);
    }
    return schedule;
}

int main(){
    int n;
    vector<job>jobs;

    cin >> n;
    for(int k = 0; k < n; k++){
        int r, d, w;
        cin >> r >> d >> w;
        job j {r, d, w};
        jobs.push_back(j);
    }

    vector<interval> schedule = yds(jobs);
    cout << energy_consumption(schedule) << endl;
    return 0;
}
```

Problem B

The Traveling-Salesman Problem

The Traveling-Salesman Problem (TSP) consists in solving the routing problem of a hypothetical traveling-salesman. Such a route must pass through *n* cities, only once per city, return to the city of origin and have the shortest possible length. It is a very well-studied NP-hard problem. More formally, the problem could be represented as a complete undirected graph $G = (V, E)$, $|V| = n$ where each edge $(i, j) \in E$ has an associated cost $c(i, j) \geq 0$ representing the distance from the city i to j (Figure B1a). The goal is to find a hamiltonian cycle with minimum cost, or a tour with minimum length, that visits each city only once and finishes at the city of departure.

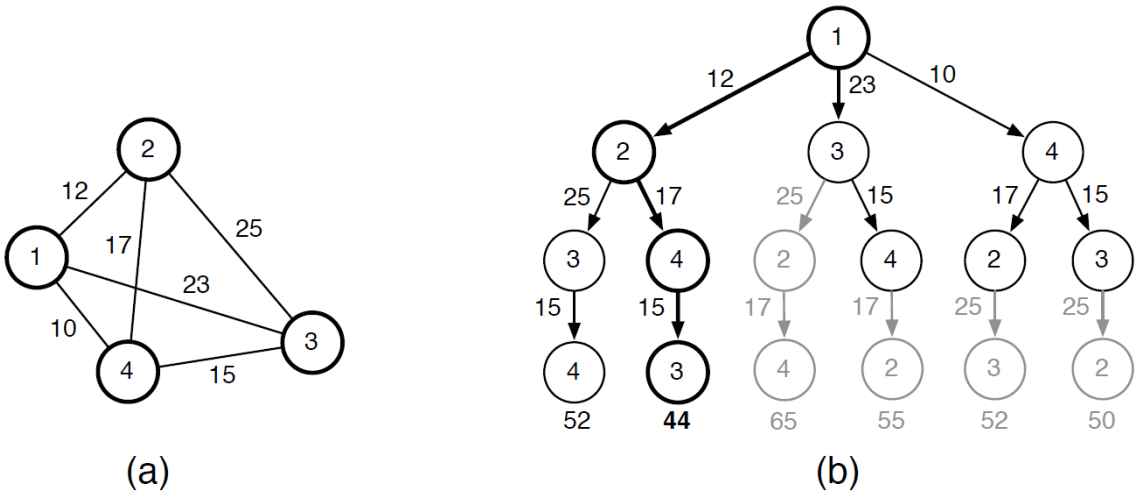


Figure B1. Example of TSP with 4 cities.

There are several different approaches to solve this problem. These solutions normally employ brute force, simple or complex heuristics, approximation algorithms or a mix of them. The provided sequential version of the algorithm deliberately employs a very simple brute force exact algorithm based on a simple greedy heuristic. This algorithm does a depth-first search looking for the shortest path and has complexity O(n!). It does not explore paths that are already known to be longer than the best path found so far, therefore it prunes the search space discarding fruitless branches. Figure B1b shows this behavior. The shaded edges are those that the algorithm does not follow, since a possible solution that includes them would be more costly than the one it has already identified. This simple pruning technique greatly improves the performance of the

algorithm. However, it also introduces irregularities into the search space. The search depth needed to discard one of the branches depends on the order in which the branches were searched.

Write a parallel version of the program which outputs the length of a minimum tour. Your program ***must*** use the same greedy heuristic of the provided example, since here we are interested in the parallelization strategies and not on a better heuristic for the TSP.

Input

The input might contain several problem instances, which is given by the first line of the input. Next, each problem is sequentially given as follows. The first line contains the number of cities (*n*). The following *n* lines are space separated integer pairs describing the Cartesian coordinates of each city.

The input must be read from the standard input.

Output

The output should list the lengths of the shortest paths found for each problem, one distance per line. To avoid rounding problems all the distances considered by the sequential program are truncated. Your program should do the same. Look the provided code for details.

The output must be written to the standard output.

Example

Input	Output for the input
3 5 139 191 92 196 88 106 27 40 130 120 7 94 173 9 10 100 14 188 165 168 67 66 5 60 162 3 198 58 77 198 50 3	420 567 538

```
int min_distance;
int nb_towns;

typedef struct {
    int to_town;
    int dist;
} d_info;

d_info **d_matrix;
int *dist_to_origin;

int present (int town, int depth, int *path) {
    int i;
    for (i = 0; i < depth; i++)
        if (path[i] == town) return 1;
    return 0;
}

void tsp (int depth, int current_length, int *path) {
    int i;
    if (current_length >= min_distance) return;
    if (depth == nb_towns) {
        current_length += dist_to_origin[path[nb_towns - 1]];
        if (current_length < min_distance)
            min_distance = current_length;
    } else {
        int town, me, dist;
        me = path[depth - 1];
        for (i = 0; i < nb_towns; i++) {
            town = d_matrix[me][i].to_town;
            if (!present (town, depth, path)) {
                path[depth] = town;
                dist = d_matrix[me][i].dist;
                tsp (depth + 1, current_length + dist, path);
            }
        }
    }
}

void greedy_shortest_first_heuristic(int *x, int *y) {
    int i, j, k, dist;
    int *tempdist;

    tempdist = (int*) malloc(sizeof(int) * nb_towns);
    for (i = 0; i < nb_towns; i++) {
        for (j = 0; j < nb_towns; j++) {
            int dx = x[i] - x[j];
            int dy = y[i] - y[j];
            tempdist [j] = dx * dx + dy * dy;
        }
        for (j = 0; j < nb_towns; j++) {
            int tmp = INT_MAX;
            int town = 0;
            for (k = 0; k < nb_towns; k++) {
                if (tempdist [k] < tmp) {
                    tmp = tempdist [k];
                    town = k;
                }
            }
            tempdist [town] = INT_MAX;
            d_matrix[i][j].to_town = town;
            dist = (int) sqrt (tmp);
            d_matrix[i][j].dist = dist;
            if (i == 0)
                dist_to_origin[town] = dist;
        }
    }
    free(tempdist);
}

void init_tsp() {
    int i, st;
    int *x, *y;

    min_distance = INT_MAX;

    st = scanf("%u", &nb_towns);
    if (st != 1) exit(1);

    d_matrix = (d_info**) malloc (sizeof(d_info*) * nb_towns);
    for (i = 0; i < nb_towns; i++)
        d_matrix[i] = (d_info*) malloc (sizeof(d_info) * nb_towns);
    dist_to_origin = (int*) malloc(sizeof(int) * nb_towns);

    x = (int*) malloc(sizeof(int) * nb_towns);
    y = (int*) malloc(sizeof(int) * nb_towns);

    for (i = 0; i < nb_towns; i++) {
        st = scanf("%u %u", x + i, y + i);
        if (st != 2) exit(1);
    }

    greedy_shortest_first_heuristic(x, y);

    free(x); free(y);
}

int run_tsp() {
    int i, *path;

    init_tsp();

    path = (int*) malloc(sizeof(int) * nb_towns);
    path[0] = 0;

    tsp (1, 0, path);

    free(path);
    for (i = 0; i < nb_towns; i++)
        free(d_matrix[i]);
    free(d_matrix);
    return min_distance;
}

int main (int argc, char **argv) {
    int num_instances, st;
    st = scanf("%u", &num_instances);
    if (st != 1) exit(1);
    while(num_instances-- > 0)
        printf("%d\n", run_tsp());
    return 0;
}
```

Problem C

Minimum Weight Polygon Decomposition

Every convex polygon, with $2 \times N$ vertexes, can be decomposed into N - 1 quadrangles, by performing N-2 cuts in a straight line between certain pairs of vertexes. Figure C1 illustrates three different decompositions of the same polygon with N=5. The weight of the decomposition is the sum the lengths of all its N-2 cuts. A Minimum Weight Polygon Decomposition is a polygon cut that has the minimum weight.

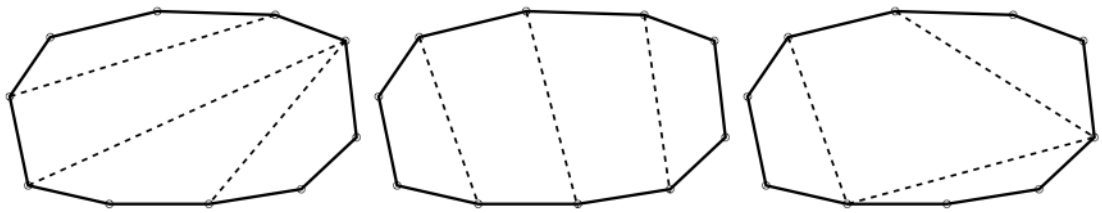


Figure C1. Decomposition examples.

Write a parallel program that finds the weight of a minimum decomposition for each convex polygon in a list.

Input

The input contains list of polygons. The number of polygons in the list is undetermined. For each polygon, a set $(2 \times N) + 1$ lines is provided. The first line of the set contains the value of the integer N ($2 \leq N \leq 500$). The next $2 \times N$ lines contain each a pair of doubles X and Y ($0 \leq X,Y \leq 10000$), with four decimal digits: the coordinates of the $2 \times N$ vertexes of the convex polygon, in a counterclockwise direction.

The input must be read from the standard input.

Output

The solution will print the weight of a minimum polygon decomposition for each polygon of the input, one weight per line. The output of the parallel version must be printed in the same order of the sequential one.

The output must be written to the standard output.

Example

Input	Output for the input
4 5715.7584 3278.6962 3870.5535 4086.7950 3823.2104 4080.7543 3574.4323 170.2905 4521.4796 144.9156 4984.6486 306.2896 5063.1061 347.1661 6099.9959 2095.9358 2 6044.4737 2567.9978 5752.5635 3226.5140 5148.8242 3802.9292 4598.8042 4036.8000	4519.6176 0.0000

```c
#define SQUARE(a) ((a)*(a))
#define MAXX 10000
#define MINX 0

typedef struct polygon_s {
        int size;
        double ** m;
        double ** d;
        double * x;
        double * y;
} polygon_t;

double process(int a, int b, polygon_t * p) {
        int i, j;
        double temp;
        if (p->m[a][b] >= 0) {
                return p->m[a][b];
        }
        p->m[a][b] = (2 * (MAXX - MINX) * (p->size));
        for (i = (a + 1) % (p->size); i != b % (p->size); i = (i + 2) % (p->size)) {
                for (j = (i + 1) % (p->size); j != (b + 1 + (p->size)) % (p->size);
                                j = (j + 2) % (p->size)) {
                        temp = process(a, i, p) + process(i, j, p) + process(j, b, p)
                                        + p->d[a][i] + p->d[i][j] + p->d[j][b];
                        if (p->m[a][b] > temp)
                                p->m[a][b] = temp;
                }
        }
        return p->m[a][b];
}
double * allocate_vector(int size, char name) {
        double * v;
        if ((v = (double *) malloc(sizeof(double) * size)) == NULL ) {
                fprintf(stderr, "Error allocating memory for polygon data (%c). \n", name);
                exit(1);
        }
        return v;
}
double ** allocate_square_matrix(int size, char name) {
        double ** m;
        if ((m = (double **) malloc(sizeof(double *) * size)) == NULL ) {
                fprintf(stderr, "Error allocating memory for polygon data (%c). \n", name);
                exit(1);
        }
        int i;
        for (i = 0; i < size; i++) {
                if ((m[i] = (double *) malloc(sizeof(double) * size)) == NULL ) {
                        fprintf(stderr, "Error allocating memory for polygon data (%c). \n", name);
                        exit(1);
                }
        }
        return m;
}
polygon_t * allocate_polygon_data(int n) {
        polygon_t * p;
        if ((p = (polygon_t *) malloc(sizeof(polygon_t))) == NULL ) {
                fprintf(stderr, "Error allocating memory for polygon data. \n");
                exit(1);
        }
        int size = 2 * n;
        p->x = allocate_vector(size, 'x');
        p->y = allocate_vector(size, 'y');
        p->m = allocate_square_matrix(size, 'm');
        p->d = allocate_square_matrix(size, 'd');
        p->size = size;
        return p;
}
int read_polygon(polygon_t ** p) {
        int n, i, status = 0;
        if ((status = scanf("%d", &n)) == 1) {
                *p = allocate_polygon_data(n);
                n *= 2;
                for (i = 0; i < n; i++) {
                        scanf("%lf %lf", &((*p)->x[i]), &((*p)->y[i]));
                }
        }
        return (status > 0);
}
void init(polygon_t * p) {
        int i, j;
        for (i = 0; i < p->size; i++) {
                for (j = 0; j < p->size; j++) {
                        if ((i + 1) % p->size == j) {
                          p->d[i][j] = 0;
                          p->m[i][j] = 0;
                        } else {
                          p->d[i][j] = sqrt(SQUARE((p->x[i])-(p->x[j]))+SQUARE((p->y[i])-(p->y[j])));
                          p->m[i][j] = -1;
                        }
                }
        }
}
void free_polygon(polygon_t *p) {
        free(p->x); free(p->y);
        int i;
        for (i = 0; i < (p->size); i++) {
                free(p->d[i]); free(p->m[i]);
        }
        free(p->d); free(p->m); free(p);
}
int main() {
        int i;
        double smaller, temp;
        polygon_t * p;

        while (read_polygon(&p)) {
                init(p);

                smaller = (2 * (MAXX - MINX) * (p->size));

                for (i = 0; i < p->size; i++) {
                        temp = process((i + 3) % p->size, i, p)
                                        + p->d[(i + 3) % p->size][i];
                        if (temp < smaller)
                                smaller = temp;
                }

                free_polygon(p);

                printf("%.4f\n", smaller);
        }

        return 0;
}
```

Problem D

A Graph's Maximal Independent Set

An independent set of a graph is any subset of its nodes where no two nodes are adjacent. I.e., for a given undirected graph $G = \{V, E\}$, such that V is its set of n nodes labelled from 1 to n and E is its set of edges – each edge a pair $\langle v, u \rangle$ with $v, u \in V$ –, an independent set $S \subseteq V$ is a set where, for any two $v', u' \in S$, there is neither $\langle v', u' \rangle$ nor $\langle u', v' \rangle$ in E. A *maximal* independent set I of a graph is an independent set that is not a subset of any other independent set. I.e., there is no $v \in V - I$ such that $I \cup \{v\}$ is an independent set. See Figure D1.

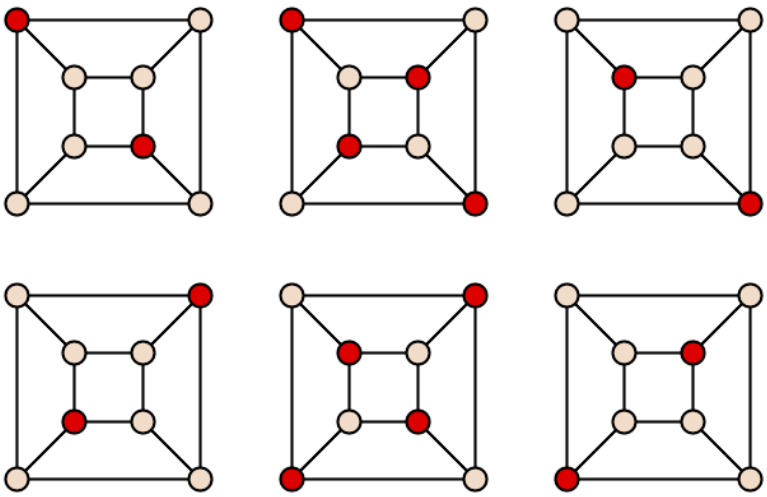


Figure D1. All maximal independent sets of the cube graph (by David Eppstein, public domain)

The given sequential program receives as input an arbitrary undirected graph and outputs one maximal subset using Luby's Method. The method iterates the following steps until the input is empty:

1. Select nodes from the input with probability $1/2^{\sigma}$, where σ is the node's degree.
2. Unselect lowest degree node of two neighbor selected nodes.
3. Move the remaining selected nodes to the maximal independent and remove them and their neighbors from the input.

You have to re-write it in order to obtain speedup.

Input

It is a sequence of lines. The first line contains a pair of integers – separated by space –, the number of nodes and number of edges. The following lines, one per edge, are also space-separated pairs of integers, each representing edges between the correspondent nodes.

Invariants:

- There is no self-edge, i.e., for a given node *x*, the line "`x x`" never appears in the list of edges.
- Each edge of the undirected graph appears only once (one line) on the input file. Its format is "`x y`", where $x < y$.

The input must be read from the standard input.

Output

It contains just one line, with all the nodes in the maximal independent set separated by space.

The output must be written to the standard output.

Example

Input	Output for the input

`8 12`
`1 2`
`1 4`
`1 5`
`2 3`
`2 6`
`3 4`
`3 7`
`4 8`
`5 6`
`5 8`
`6 7`
`7 8`
	`1 3 6 8`

```
#define Set    std::set
#define Vertex std::size_t
#define Edge   std::pair<Vertex, Vertex>

// data structures

struct Graph {

    Set<Vertex> v ;
    Set<Edge>   e ;

    // default constructor
    Graph () {} ;

    // copy constructor
    Graph (const Graph& x) : v (x.v), e (x.e) { } ;

    // assignment
    Graph& operator= (const Graph& x) {
        this->~Graph () ;
        new (this) Graph (x) ;
        return *this ;
    } ;

    // equality comparison
    friend
        bool operator== (const Graph& x, const Graph& y) {
            return (x.v == y.v) && (x.e == y.e) ;
        } ;

    // inequality comparison
    friend
        bool operator!= (const Graph& x, const Graph& y) {
            return ! (x == y) ;
        } ;

    // vertex degree
    std::size_t degree (Vertex i) const
    {
        std::size_t n = 0 ;
        for (const Edge& p : e) {
            if (p.first  == i || p.second == i) ++ n ;
        }
        return n ;
    }

    // vertex degree
    bool are_neighbors (Vertex x, Vertex y) const
    {
        Vertex a = std::min (x, y) ;
        Vertex b = std::max (x, y) ;
        if (e.find (Edge (a, b)) != e.end ()) return true ;
        return false ;
    }
} ;
```

```
// selects node with probability 1/2^n
// using a uniformly random generated natural m
template <typename N, typename M> inline
bool try_select (N n, M m)
{
    return (m % (N (1) << n)) + N (1) <= N (1) ;
}
template <typename S> inline
void set_union (S& x, const S& y) {
    for (const auto& i : y) {
        x.insert (i) ;
    }
    return ;
}
void exclude_neighbors (Set<Vertex>& v, const Graph& g) {
    for (Vertex i : v) for (Vertex j : v) {
        if (g.are_neighbors (i, j)) {
            v.erase (i) ;
        }
    }
    return ;
}
template <typename S> inline
void set_difference (S& x, const S& y) {
    for (const auto& i : y) {
        x.erase (i) ;
    }
    return ;
}
template <typename RandGenerator>
void rand_select (Set<Vertex>& u, const Set<Vertex>& v, const Graph& g, RandGenerator& r) {
    // constant-time clear of u (STL method is linear time)
    Set<Vertex> x ;
    x.swap (u) ;
    for (Vertex i : v) {
        if (try_select (g.degree (i), r ())) {
            u.insert (i) ;
        }
    }
    return ;
}
Set<Vertex> mis (const Graph& g) {
    Set<Vertex> m;      // current mis (empty)
    Set<Vertex> u;// currently selected nodes (empty)
    Set<Vertex> v = g.v;// initial set of vertexes
    std::minstd_rand r;// an STL linear congruential random number gen.
    while (! v.empty ()) {
        do {
            rand_select (u, v, g, r);
        }while (u.empty ());
        set_difference (v, u);
        exclude_neighbors (u, g);
        set_difference (v, neighbors (u, g));
        set_union (m, u);
    }
    return m;
}
int main(int argc, char *argv[]) {
    out(mis(in()));
    return 0;
}
```

Problem E

Enumeration Sort[2]

Enumeration Sort is a method of finding the exact position of each element in a sorted list by comparing and finding the frequency of elements having smaller value (Knuth, 1973). That is if p elements are smaller than a_q, then a_q occupies the (p+1)th position in the sorted list.

Write a parallel version of the Enumeration Sort algorithm.

Input

The input file contains only one test case. The first line contains the total number of keys (N) to be sorted ($1 \leq N \leq 10^{10}$). 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

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
11 SINAPAD SbacPad Wscad14 Sinapad 1234567 LADGRID WEAC-14 CTDeWIC sinaPAD MPP2014 SINApad	1234567 CTDeWIC LADGRID MPP2014 SINAPAD SINApad SbacPad Sinapad WEAC-14 Wscad14 sinaPAD

[2] NPTEL :: Computer Science and Engineering - Parallel Algorithms. URL: http://nptel.ac.in/courses/106106112/13 .

```c
#define LENGTH 8

FILE *fin, *fout;

char *strings;
long int N;

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 enum_sort(char *a, int length, long int size) {
    long int i, j, rank;
    char *tmp = malloc(length);

    /* Enumeration sort */
    for (j = 0; j < size; j++) {
        rank = 0;
        for (i = 0; i < size; i++) {
            if (strcmp(a + (i * length), a + (j * length)) < 0)
                rank++;
        }
        if (j < rank)
            while (rank < size && !strcmp(a + (rank * length), a
+ (j * length)))
                rank++;
        else
            while (rank < j && !strcmp(a + (rank * length), a +
(j * length)))
                rank++;
        if (rank != j) {
            strcpy(tmp, a + (rank * length));
            strcpy(a + (rank * length), a + (j * length));
            strcpy(a + (j * length), tmp);
            j--;
        }
    }
    free(tmp);
}

int main(int argc, char* argv[]) {
    long int i;

    openfiles();

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

    enum_sort(strings, LENGTH, N);

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

    free(strings);
    closefiles();

    return EXIT_SUCCESS;
}
```

Problem F

Dijkstra

Dijkstra's algorithm, conceived by computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs[3].

For a give node in a graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that node and the destination node. Figure F1 shows a graph.

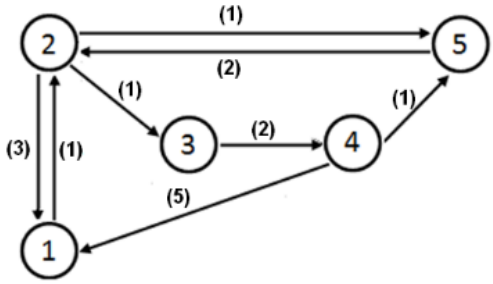
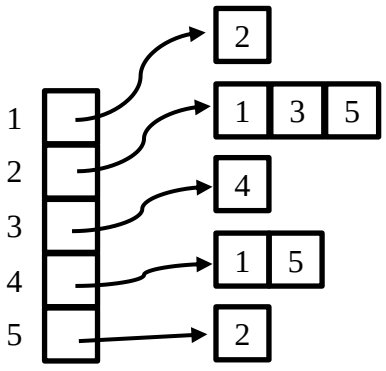


Figure F1. A simple graph.

An adjacency list representation for a graph associates each node in the graph with the collection of its neighboring nodes[4]. I.e, for Figura F1, its representation can be:



This list also can store the weight of each edge or other information that helps the algorithm finding the shortest path.

Write a parallel version of the Dijkstra's algorithm.

[3] Dijkstra's algorithm. URL: http://en.wikipedia.org/wiki/Dijkstra's_algorithm .
[4] Adjacency list. URL: http://en.wikipedia.org/wiki/Adjacency_list .

Input

The input contains 3 integers. The first integer represents the total number of nodes in the graph ($2 \leq V \leq 50$). The second integer represents the average number of outgoing edges per node ($1 \leq E \leq V/2$). The last integer represents the seed for a random number generator ($0 \leq S < 2^{32}$).

The input must be read from the standard input.

Output

The output has only one number. It represents the mean distance from node 0 to all nodes.

The output must be written to the standard output.

Example

Input	Output for the input
`5 2 6`	`14.20`

```c
struct Graph {
        int nNodes;
        int *nEdges;
        int **edges;
        int **w;
};

struct Graph *createRandomGraph(int nNodes, int nEdges, int seed) {
        my_srand(seed);

        struct Graph *graph = (struct Graph *) malloc(sizeof(struct Graph));
        graph->nNodes = nNodes;
        graph->nEdges = (int *) malloc(sizeof(int) * nNodes);
        graph->edges = (int **) malloc(sizeof(int *) * nNodes);
        graph->w = (int **) malloc(sizeof(int *) * nNodes);

        int k, v;
        for (v = 0; v < nNodes; v++) {
                graph->edges[v] = (int *) malloc(sizeof(int) * nNodes);
                graph->w[v] = (int *) malloc(sizeof(int) * nNodes);
                graph->nEdges[v] = 0;
        }

        int source = 0;
        for (source = 0; source < nNodes; source++) {
                int nArestasVertice = (double) nEdges / nNodes
                                * (0.5 + my_rand() / (double) RAND_MAX);
                for (k = nArestasVertice; k >= 0; k--) {
                        int dest = my_rand() % nNodes;
                        int w = 1 + (my_rand() % 10);
                        graph->edges[source][graph->nEdges[source]] = dest;
                        graph->w[source][graph->nEdges[source]++] = w;
                }
        }

        return graph;
}

int *dijkstra(struct Graph *graph, int source) {
        int nNodes = graph->nNodes;
        int *visited = (int *) malloc(sizeof(int) * nNodes);
        int *distances = (int *) malloc(sizeof(int) * nNodes);
        int k, v;
        for (v = 0; v < nNodes; v++) {
                distances[v] = INT_MAX;
                visited[v] = 0;
        }
        distances[source] = 0;
        visited[source] = 1;
        for (k = 0; k < graph->nEdges[source]; k++)
                distances[graph->edges[source][k]] = graph->w[source][k];

        for (v = 1; v < nNodes; v++) {
                int min = 0;
                int minValue = INT_MAX;
                for (k = 0; k < nNodes; k++)
                        if (visited[k] == 0 && distances[k] < minValue) {
                                minValue = distances[k];
                                min = k;
                        }
                visited[min] = 1;
                for (k = 0; k < graph->nEdges[min]; k++) {
                        int dest = graph->edges[min][k];
                        if (distances[dest] > distances[min] + graph->w[min][k])
                                distances[dest] = distances[min] + graph->w[min][k];
                }
        }
        free(visited);
        return distances;
}
int main(int argc, char ** argv) {

        int nNodes;
        int nEdges;
        int seed;
        if (argc == 4) {
                nNodes = atoi(argv[1]);
                nEdges = atoi(argv[2]);
                seed = atoi(argv[3]);
        } else {
                fscanf(stdin, "%d %d %d", &nNodes, &nEdges, &seed);
        }
        assert(nEdges <= nNodes/2);
        nEdges = nNodes * nEdges;

        struct Graph *graph = createRandomGraph(nNodes, nEdges, seed);
        int *dist = dijkstra(graph, 0);

        double mean = 0;
        int v;
        for (v = 0; v < graph->nNodes; v++)
                mean += dist[v];

        fprintf(stdout, "%.2f\n", mean / nNodes);
        return 0;
}
```