14th Marathon of Parallel Programming

SBAC-PAD & WSCAD – 2019

October 17th, 2019.

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 (zip) with your source code, the *Makefile* and an execution script. The script must 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. The execution script runs your solution the way you design it – it will be inspected not to corrupt the target machine.

All *Local Teams* must use the computers that the organization provides. Only the judges have access to the judge machine.

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 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 17.

General information

Compilation

You must use CC or CXX inside your *Makefile*. Be careful when redefining them! There is a simple *Makefile* inside you problem package that should be modified. Example:

```
FLAGS=-O3
EXEC=sum
CXX=icpc

all: $(EXEC)

$(EXEC):
	$(CXX) $(FLAGS) $(EXEC).cpp -c -o $(EXEC).o
	$(CXX) $(FLAGS) $(EXEC).o -o $(EXEC)
```

Running

You must have an execution script that has the same name of the problem. This script runs your solution the way you design it. There is a simple script inside you problem package that should be modified. Example:

```
$ cat A
#!/bin/bash
# This script runs Problem A
# export OMP_NUM_THREADS=32
# mpiexec –n 32 ./sum
./sum
```

Measure the execution time of your solution using *time* program. Add input/output redirection when collecting time. Use *diff* program to compare the original and your solution results. Example:

```
$ time –p ./A < original_input.txt > my_output.txt
real 4.94
user 0.08
sys 1.56

$ diff my_output.txt original_output.txt

$
```

Problem A

Maximum Sum Subsequence

Given an array sequence [A_1, A_2 …A_n], where n is the total amount of integer values (also the array size), this implementation aims to find the maximum possible sum of increasing subsequence *S* of length *k* such that $S_1 \leq S_2 \leq S_3 \leq S_4 \leq .... \leq S_k$.

Input

The input set contains only one test case. The first line contains one value: the array size (which is also the amount of elements to be read). The second line contains the length of the subsequence *S*, represented by *k*. Then, the last line contains a list of the elements to be inserted into the array (note that the number of elements must be equal to the array size).

The input must be read from the standard input.

Output

The output contains only one line printing the maximum possible sum of increasing subsequence *S*.

The output must be written to the standard output.

Example

Input example 1	Output example 1
8 3 8 5 9 10 5 6 21 8	40

```
int MaxIncreasingSub(int arr[], int n, int k)
{
    int **dp, ans = -1;
    dp = new int *[n];
    for(int i=0; i < n; i++)
        dp[i] = new int[k+1];
    for(int i = 0; i < n; i++){
        for(int j = 0; j < k; j++){
            dp[i][j] = -1;
        }
    }
    for (int i = 0; i < n; i++) {
        dp[i][1] = arr[i];
    }
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[j] < arr[i]) {
                for (int l = 1; l <= k - 1; l++) {
                    if (dp[j][l] != -1) {
                        dp[i][l + 1] =
max(dp[i][l + 1],dp[j][l] + arr[i]);
                    }
                }
            }
        }
    }
    for (int i = 0; i < n; i++) {
        if (ans < dp[i][k])
            ans = dp[i][k];
    }
    return (ans == -1) ? 0 : ans;
}
```

Problem B

Brute-Force Password Cracking

One way to crack a password is through a brute-force algorithm, which tests all possible combinations of a password exhaustively until it finds the correct one.

A common practice in security systems is to store the *hash* of user passwords (instead of the plan text). Hashes map a given x (the password) with variable length N to a given $y = hash(x)$ with a fixed length M, following specific security properties. One of them is: you cannot obtain the value of x from the value of y. In a security system, y is stored instead of x and the comparison is performed directly between the hash of attempted password x' and the stored $y = hash(x)$.

Examples of hash algorithms are MD5 and SHA. For MD5 algorithm, the fixed length of the $hash(x)$ is 128 bits usually presented as 32-hexadecimal characters.

This problem considers the cracking of MD5-hashed password by trying exhaustively all the combinations of possible characters (uppercase and lower-case letters and numeric symbols) and comparing pairs of MD5 hashes. That is, for all combinations of characters x'_i, $hashMD5(x) = hashMD5(x'_i)$?

Input

An input consists of only one case of test. The single line contains a string with 32-hexadecimal characters representing the value of MD5 hash of the password to be cracked.

Consider that the possible passwords used to generate the hashes have the length N, with $1 \leq N \leq 10$.

The input must be read from the standard input.

Output

The output is a single line that contains the password value found.

The output must be written to the standard output.

Example

Input example 1	Output example 1
`7a95bf926a0333f57705aeac07a362a2`	`found: PASS`

```
void print_digest(byte * hash){
    int x;

    for(x = 0; x < MD5_DIGEST_LENGTH; x++)
        printf("%02x", hash[x]);
    printf("\n");
}

/*
 * This procedure generate all combinations of possible
letters
*/
void iterate(byte * hash1, byte * hash2, char *str, int idx,
int len, int *ok) {
    int c;

    // 'ok' determines when the algorithm matches.
    if(*ok) return;
    if (idx < (len - 1)) {
        // Iterate for all letter combination.
        for (c = 0; c < strlen(letters) && *ok==0; ++c)
        {
            str[idx] = letters[c];
            // Recursive call
            iterate(hash1, hash2, str, idx + 1, len,
ok);
        }
    } else {
        // Include all last letters and compare the
hashes.
        for (c = 0; c < strlen(letters) && *ok==0; ++c)
        {
            str[idx] = letters[c];
            MD5((byte *) str, strlen(str), hash2);
            if(strncmp((char*)hash1, (char*)hash2,
MD5_DIGEST_LENGTH) == 0){
                printf("found: %s\n", str);
                *ok = 1;
            }
        }
    }
}

/*
 * Convert hexadecimal string to hash byte.
*/
void strHex_to_byte(char * str, byte * hash){
    char * pos = str;
    int i;

    for (i = 0; i < MD5_DIGEST_LENGTH/sizeof *hash; i++) {
        sscanf(pos, "%2hhx", &hash[i]);
        pos += 2;
    }
}
```

Problem C

HopByte

The HopByte metric measures how much data has to travel through a network (hops × bytes). For an application executed in a parallel platform with a specific mapping of processes to processors, we can compute the HopByte metric as the sum (for all pairs of processes) of the product of the number of bytes exchanged by the distance (number of network hops) between the processors where the processes are mapped.

For our specific case, consider:

- The application as a directed weighted graph $G = (V, E, w)$ with V the set of vertices (processes), E the set of edges (ordered pairs of vertices, or communication between processors), and w: $E \rightarrow \mathbb{N}^+$ the weight of the edges (number of bytes sent from one process to the other).
- The parallel platform as a set of processors P. The network topology of the platform is organized as a tree with height h, where the leaves represent P and the nodes in each level have the same number of children. The distance between processors (in number of hops) is computed as d: $P \times P \rightarrow \mathbb{N}$.
- The mapping of the application processes to processors as δ: $V \rightarrow P$.

In this case, the HopByte (HB) metric is computed as:

$$HB = \sum_{i,j \in V} d\big(\delta(i), \delta(j)\big) w(i,j)$$

Input

The input has three sections to be read with no separation (whitespace, new line, etc).

The first section represents the application. Its first line contains two values: the first is the number of vertices (processors) $|V|$ and the second is the number of edges (communication between processors) $|E|$. The other $|E|$ lines contain three values: two vertices (whose ids are in the interval $[0, |V|[$) composing an edge and the weight of said edge.

The second section represents the network topology of the parallel platform (as a tree). Its

first line contains two values: the first is the number of leaves in the tree (processors) $|P|$ and the second is the height of the tree h. The other h-1 lines contain a single value representing the number of children each node of that level has (number of children of the root node, number of children of the children of the root node, etc.).

The third section represents the mapping of processes to processors. The first line contains two values: the first is the number of processes $|V|$ and the second is the number of processors $|P|$. The following V lines contain the id of a process in the interval $[0, |V|[$ and the id of the processors where it is mapped in the interval $[0, |P|[$.

The input must be read from the standard input.

Output

The output contains only one line printing the computed HopByte metric.

The output must be written to the standard output.

Example

Input example 1	Output example 1
3 4 0 1 40 1 0 20 1 2 30 2 1 50 6 3 2 3 3 6 0 5 1 1 2 3	560

```c
/* Computes the number of hops between two leaf nodes in the
topology */
uint32_t find_hops(uint32_t first_node, uint32_t second_node,
uint32_t** topology, uint32_t height){
    // Number of levels searched to find the common parent
(or common node)
    uint32_t common_level = 0;
    uint32_t current_level = height-1;
    // If the nodes are different, searches for their common
parent
    while(first_node != second_node){
        // Move one level closer to the root
        first_node = topology[current_level][first_node];
        second_node = topology[current_level][second_node];
        ++common_level;
        --current_level;
    }
    //In a tree, we have two hops for each level that we go
in the direction of the root
    return common_level*2;
}

/* Computes the hopbyte metric for the mapping */
uint64_t hopbyte(uint32_t* mapping, uint32_t* application,
uint32_t processes, uint32_t** topology, uint32_t height){
    uint64_t total = 0;

    uint32_t i, j, bytes, hops;
    //For all pairs of processors
    for(i = 0; i < processes; ++i){
        for(j = 0; j < processes; ++j){
            //Gets the number of bytes sent from i to j
            bytes = application[i*processes+j];
            //Gets the number of hops between the processors
where i and j are mapped
            hops = find_hops(mapping[i], mapping[j],
topology, height);
            //Adds their product to the hopbyte total
            total+= bytes*hops;
        }
    }

    return total;
}

// Main program - reads input, computes hopbyte, prints
output
int main (int argc, char* argv[]){

    uint32_t *application, **topology, *mapping;
    uint32_t processes, processors, height;

    //Tries to read the first input (application)
    application = read_application(&processes);
    //Tries to read the second input (topology)
    topology = read_topology(&processors, &height);
    //Tries to read the third input (mapping)
    mapping = read_mapping(processes, processors);

    //Computes and outputs the hopbyte metric of the mapping
    printf("%lu\n",hopbyte(mapping, application, processes,
topology, height));

    return 0;
}
```

Problem D

Galaxy Simulator

From Wikipedia:

> Newton's law of universal gravitation states that every particle attracts every other particle in the universe with a force which is directly proportional to the product of their masses and inversely proportional to the square of the distance between their centers

The Newton's law leads up to a $O(n^2)$ time complexity algorithm, since the force of every particle must act upon all other particles, when one try to simulate a particle system. Such algorithm is too much slow to simulate a large number of particles, such as galaxies that are commonly composed of billions of stars. As an example, the Milky Way galaxy alone is supposed to have between 100 and 400 billion stars.

Barnes-Hut[1] came up with $O(n \log n)$ time complexity algorithm that enables one to approximate particle interaction by assuming that nearby bodies, a group, work as a single large fictitious body in the center of mass of the group. For a 2D galaxy simulation (where all stars share the same plane), the Barnes-Hut algorithm divides the space in four quadrants. Depending on the number of stars, each quadrant can be divided again in four quadrants, and so on. A tree data structure is used to keep all information: each node has four children, each one to represent a quadrant of the two dimensional space associated with the node. The particle interaction uses that tree data structure to compute approximations by considering the center of mass of each quadrant.

Input

An input represents only one test case. The first line contains one integer value N that represents the number of stars. Each one of the following N lines contains five floating-point values that describe the *mass*, the plane position (x, y) and the plane velocity (u, v) of each star. The last line contains the number of time steps that is used into this galaxy simulation.

[1] Josh Barnes and Piet Hut. *A hierarchical o (n log n) force-calculation algorithm*. Nature, 324(6096):446, 1986.

The input must be read from the standard input.

Output

The output contains two lines: the first line contains the total vector velocity (*u*, *v*) considering all stars, in both dimensions; and the second line contains the center (*x*, *y*) of mass considering all stars.

The output must be written to the standard output.

Example

Input	Output for the input
3 1.000000 0.334531 0.532345 -5.192624 -6.640964 1.000000 0.558670 0.453306 3.576856 1.123553 1.000000 0.573755 0.553912 -4.322105 1.478229 1000	-13.109914 -6.611076 0.509590 0.580083

```
void time_step(void) {
  //Allocate memory for root
  root = malloc(sizeof(struct node_t));
  set_node(root);
  root->min_x = 0; root->max_x = 1; root->min_y = 0; root->max_y = 1;

  //Put particles in tree
  for(int i = 0; i < N; i++) {
    put_particle_in_tree(i, root);
  }

  //Calculate mass and center of mass
  calculate_mass(root);
  calculate_center_of_mass_x(root);
  calculate_center_of_mass_y(root);

  //Calculate forces
  update_forces();

  //Update velocities and positions
  for(int i = 0; i < N; i++) {
    double ax = force_x[i]/mass[i];
    double ay = force_y[i]/mass[i];
    u[i] += ax*dt;
    v[i] += ay*dt;
    x[i] += u[i]*dt;
    y[i] += v[i]*dt;

    /* This of course doesn't make any sense physically,
     * but makes sure that the particles stay within the
     * bounds. Normally the particles won't leave the
     * area anyway.
     */
    bounce(&x[i], &y[i], &u[i], &v[i]);
  }

  //Free memory
  free_node(root);
  free(root);
}
void put_particle_in_tree(int new_particle, struct node_t *node) {
  //If no particle is assigned to the node
  if(!node->has_particle) {
    node->particle = new_particle;
    node->has_particle = 1;
  }
  //If the node has no children
  else if(!node->has_children) {
    //Allocate and initiate children
    node->children = malloc(4*sizeof(struct node_t));
    for(int i = 0; i < 4; i++) {
      set_node(&node->children[i]);
    }

    //Set boundaries for the children
    node->children[0].min_x = node->min_x; node->children[0].max_x = (node->min_x + node->max_x)/2;
    node->children[0].min_y = node->min_y; node->children[0].max_y = (node->min_y + node->max_y)/2;

    node->children[1].min_x = (node->min_x + node->max_x)/2; node->children[1].max_x = node->max_x;
    node->children[1].min_y = node->min_y; node->children[1].max_y = (node->min_y + node->max_y)/2;

    node->children[2].min_x = node->min_x; node->children[2].max_x = (node->min_x + node->max_x)/2;
    node->children[2].min_y = (node->min_y + node->max_y)/2; node->children[2].max_y = node->max_y;

    node->children[3].min_x = (node->min_x + node->max_x)/2; node->children[3].max_x = node->max_x;
    node->children[3].min_y = (node->min_y + node->max_y)/2; node->children[3].max_y = node->max_y;

    //Put old particle into the appropriate child
    place_particle(node->particle, node);

    //Put new particle into the appropriate child
    place_particle(new_particle, node);
    //It now has children
    node->has_children = 1;
  }
  //Add the new particle to the appropriate children
  else {
    //Put new particle into the appropriate child
    place_particle(new_particle, node);
  }
}


/*
 * Puts a particle in the right child of a node with children.
 */
void place_particle(int particle, struct node_t *node) {
  if(x[particle] <= (node->min_x + node->max_x)/2 && y[particle] <= (node->min_y + node->max_y)/2) {
    put_particle_in_tree(particle, &node->children[0]);
  } else if(x[particle] > (node->min_x + node->max_x)/2 && y[particle] < (node->min_y + node->max_y)/2) {
    put_particle_in_tree(particle, &node->children[1]);
  } else if(x[particle] < (node->min_x + node->max_x)/2 && y[particle] > (node->min_y + node->max_y)/2) {
    put_particle_in_tree(particle, &node->children[2]);
  } else {
    put_particle_in_tree(particle, &node->children[3]);
  }
}
double calculate_mass(struct node_t *node) {
  if(!node->has_particle) {
    node->total_mass = 0;
  } else if(!node->has_children) {
    node->total_mass = mass[node->particle];
  } else {
    node->total_mass = 0;
    for(int i = 0; i < 4; i++) {
      node->total_mass += calculate_mass(&node->children[i]);
    }
  }
  return node->total_mass;
}
double calculate_center_of_mass_x(struct node_t *node) {
  if(!node->has_children) {
    node->c_x = x[node->particle];
  } else {
    node->c_x = 0;
    double m_tot = 0;
    for(int i = 0; i < 4; i++) {
      if(node->children[i].has_particle) {
        node->c_x += node->children[i].total_mass*calculate_center_of_mass_x(&node->children[i]);
        m_tot += node->children[i].total_mass;
      }
    }
    node->c_x /= m_tot;
  }
  return node->c_x;
}
double calculate_center_of_mass_y(struct node_t *node) {
  if(!node->has_children) {
    node->c_y = y[node->particle];
  } else {
    node->c_y = 0;
    double m_tot = 0;
    for(int i = 0; i < 4; i++) {
      if(node->children[i].has_particle) {
        node->c_y += node->children[i].total_mass*calculate_center_of_mass_y(&node->children[i]);
        m_tot += node->children[i].total_mass;
      }
    }
    node->c_y /= m_tot;
  }
  return node->c_y;
}
```

Problem E

FDM in Heat Equation

The heat equation is solved by derivatives in space and time, as shown below:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right)$$

Some numerical methods solve it based on approximate approach. One of these methods is called Finite Different Method – FDM – which resolves derivatives through finites differences. Thus, the numerical approximate based on FDM is given by:

$$u_{i,j,k}^{t+1} = \frac{\Delta t}{\Delta h^2}\alpha\begin{bmatrix} u_{i+1,j,k}^{t} + u_{i-1,j,k}^{t} + u_{i,j+1,k}^{t} + u_{i,j-1,k}^{t} \\ +u_{i,j,k+1}^{t} + u_{i,j,k-1}^{t} - 6 \times u_{i,j,k}^{t} \end{bmatrix} - u_{i,j,k}^{t+1}$$

where time and space are discretized in regular steps. To simplify the problem, we adopt the same space discretization in all axes.

The initial condition is given by a cube with zero Celsius degree. The cube measures *x*, *y* and *z* centimeters, relying on the input problem parameter. Each face of the cube is heated to a temperature of 100 Celsius degree. Thus, sequential code calculates how long all the cube will take to be in 100-Celsius degree, using the previous FDM equation with 1.0 for the material heat velocity propagation.

Input

An input consists of only one test case. The first line contains tow positive values for Δt and Δh (both $\leq$ 1.0). The second line contains three positive values (z, y, x) of a cube ($1 \leq z, y, x \leq 100$).

The input must be read from the standard input.

Output

The output has only one line. It contains the number of time steps to make the entire cube at 100-Celsius degree.

The output must be written to the standard output.

Example

Input example 1	Output example 1
`0.01 0.25` `10 10 20`	`14383`

```
void mdf_heat(double ***   __restrict__ u0,
              double ***   __restrict__ u1,
              const unsigned int npX,
              const unsigned int npY,
              const unsigned int npZ,
              const double deltaH,
              const double deltaT,
              const double inErr,
              const double boundaries){
  register double alpha = deltaT / (deltaH * deltaH);
  register int continued = 1;
  register unsigned int steps = 0;
  while (continued){
    steps++;
    for (unsigned int i = 0; i < npZ; i++){
      for (unsigned int j = 0; j < npY; j++){
        for (unsigned int k = 0; k < npX; k++){
          register double left   = boundaries;
          register double right  = boundaries;
          register double up     = boundaries;
          register double down   = boundaries;
          register double top    = boundaries;
          register double bottom = boundaries;

          if ((k > 0) && (k < (npX - 1))){
            left  = u0[i][j][k-1];
            right = u0[i][j][k+1];
          }else if (k == 0) right = u0[i][j][k+1];
          else left = u0[i][j][k-1];

          if ((j > 0) && (j < (npY - 1))){
            up   = u0[i][j-1][k];
            down = u0[i][j+1][k];
          }else if (j == 0) down = u0[i][j+1][k];
          else up = u0[i][j-1][k];

          if ((i > 0) && (i < (npZ - 1))){
            top  = u0[i-1][j][k];
            bottom = u0[i+1][j][k];
          }else if (i == 0) bottom = u0[i+1][j][k];
          else top = u0[i-1][j][k];

          u1[i][j][k] =  alpha * ( top + bottom + up + down +
left + right  - (6.0f * u0[i][j][k] )) + u0[i][j][k];
        }
      }
    }

    double ***ptr = u0;
    u0 = u1;
    u1 = ptr;

    double err = 0.0f;
    double maxErr = 0.0f;
    for (unsigned int i = 0; i < npZ; i++){
      for (unsigned int j = 0; j < npY; j++){
        for (unsigned int k = 0; k < npX; k++){
          err = fabs(u0[i][j][k] - boundaries);
          if (err > inErr)
            maxErr = err;
          else
            continued = 0;
        }
      }
    }
  }

  fprintf(stdout, "%u\n", steps);
}
```

Problem F

Closest Pair of Points

Given an array of n points in the 2D plane, the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For instance, it can be used to monitor airplanes that come too close together in an air-traffic control system, thus avoiding possible collisions. The following formula is used to determine the distance between two points *p* and *q* in a 2D plane:

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

The given sequential program calculates the distance between the closest pair of points in $O(n(\log\ n)^2)$ time using a Divide and Conquer strategy. Your task is to improve the performance of the program using parallel strategies.

Input

The input contains only one test case. The first line of input contains an integer *N* ($2 \leq N \leq 80{,}000{,}000$), which denotes the number of points. Each of the next *N* lines contains two floats *X* and *Y* ($-10{,}000{,}000 \leq X, Y \leq 10{,}000{,}000$ with exactly 3 digits after the decimal), which denote the coordinates of the i-th point. There is no coincident point in the input data.

The input must be read from the standard input.

Output

The output must contain a single line with a float *D*, denoting the distance between the closest pair of points. *D* must contain exactly 6 digits after the decimal.

The output must be written to the standard output.

Example

Input example 1	Output example 1
5 -5.000 2.000 -15.000 -7.000 0.000 0.000 6.000 3.000 2.000 4.000	4.123106

```
inline double abs2(double a) {
    if (a < 0.0)
        return -a;
    return a;
}

bool compX(const Point& a, const Point& b) {
    if (abs2(a.x-b.x)<EPS)
        return a.y<b.y;
    return a.x<b.x;
}

bool compY(const Point& a, const Point& b) {
    if (abs2(a.y-b.y)<EPS)
        return a.x<b.x;
    return a.y<b.y;
}

inline double sqr(double a) {
    return a * a;
}

inline double distance(Point a, Point b) {
    return sqr(a.x - b.x) + sqr(a.y - b.y);
}

double solve(int l, int r) {
    double minDist = INF;
    double dist;
    int i, j;
    if(r-l+1 <= BRUTEFORCESSIZE){
        for(i=l; i<=r; i++){
            for(j = i+1; j<=r; j++) {
                dist = distance(point[i], point[j]);
                if(dist<minDist){
                    minDist = dist;
                }
            }
        }
        return minDist;
    }

    int m = (l+r)/2;
    double dL = solve(l,m);
    double dR = solve(m,r);
    minDist = (dL < dR ? dL : dR);

    int k = l;
    for(i=m-1; i>=l && abs(point[i].x-point[m].x)<minDist; i--){
        border[k++] = point[i];
    }
    for(i=m+1; i<=r && abs(point[i].x-point[m].x)<minDist; i++){
        border[k++] = point[i];
    }

    if (k-l <= 1) return minDist;

    sort(&border[l], &border[l]+(k-l), compY);

    for(i=l; i<k; i++){
        for(j=i+1; j<k && border[j].y - border[i].y < minDist; j++){
            dist = distance(border[i], border[j]);
            if (dist < minDist){
                minDist = dist;
            }
        }
    }
    return minDist;
}
```