7th Marathon of Parallel Programming WSCAD-SSC/SBAC-PAD-2012

October 17th, 2012.

Rules

For all problems, read carefully the input and output session. For all problems, a sequential implementation is given, and it is against the output of those implementations that the output of your programs will be compared to decide if your implementation is correct. You can modify the program in any way you see fit, except when the problem description states otherwise. You must upload a compressed file with your source code, the *Makefile* and an execution script. The program to execute should have the name of the problem. You can submit as many solutions to a problem as you want. Only the last submission will be considered. The *Makefile* must have the rule *all*, which will be used to compile your source code before submit. The execution script runs your solution the way you design it – it will be inspected not to corrupt the target machine.
All teams have access to the target machine during the marathon. Your execution may have concurrent process from other teams. Only the judges have access to a non-concurrent cluster.
The execution time of your program will be measured running it with *time* program and taking the real CPU time given. Each program will be executed at least twice with the same input and only the smaller time will be taken into account. The sequential program given will be measured the same way. You will earn points in each problem, corresponding to the division of the sequential time by the time of your program (*speedup*). The team with the most points at the end of the marathon will be declared the winner.

This problem set contains 5 problems; pages are numbered from 1 to 15.

Problem A

Bucket Sort

Bucket Sort is another divide and conquer algorithm. It works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm[1]. Figure A.1 shows a simple example.

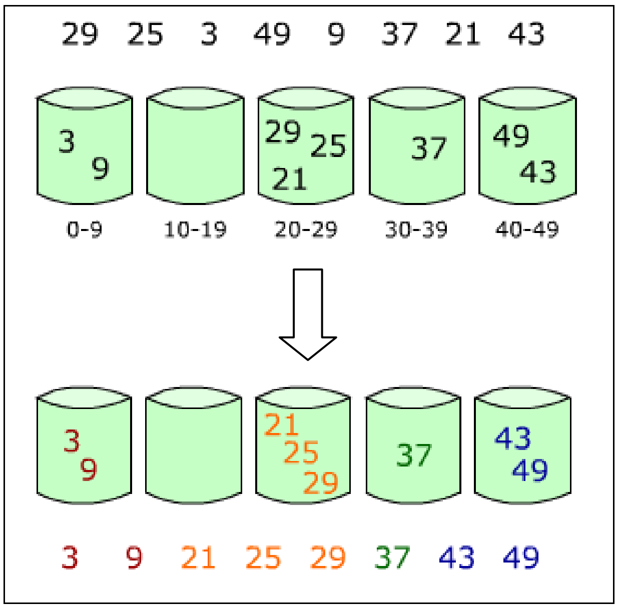

Figure A.1 – Partitioning an array and sorting the buckets.

Write a parallel program that uses a Bucket Sort algorithm to sort keys.

Input

The input file contains only one test case. The first line contains the total number of keys (N) to be sorted ($94 \leq N \leq 10^{10}$). 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 keys are uniformly distributed.

The input must be read from a file named *bucketsort.in*

[1] http://en.wikipedia.org/wiki/Bucket_sort

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 bucketsort.out

Example

Input	Output for the input
`11`	`1234567`
`SINAPAD`	`CTDeWIC`
`SbacPad`	`LADGRID`
`Wscad12`	`MPP2012`
`Sinapad`	`SINAPAD`
`1234567`	`SINApad`
`LADGRID`	`SbacPad`
`WEAC-12`	`Sinapad`
`CTDeWIC`	`WEAC-12`
`sinaPAD`	`Wscad12`
`MPP2012`	`sinaPAD`
`SINApad`	

This example is not a real input file: see input specification above.

```c
#include <stdlib.h>
#include <string.h>
#include "bucketsort.h"

#define N_BUCKETS 94

typedef struct {
        long int *data;
        int length;
        long int total;
} bucket;

void sort(char *a, bucket *bucket) {
        int j, i, length;
        long int key;
        length = bucket->length;
        for (j = 1; j < bucket->total; j++) {
                key = bucket->data[j];
                i = j - 1;
                while (i >= 0
                                && strcmp(a + bucket->data[i] *
length, a + key * length) > 0) {
                        bucket->data[i + 1] = bucket->data[i];
                        i--;
                }
                bucket->data[i + 1] = key;
        }
}

long int* bucket_sort(char *a, int length, long int size) {

        long int i;
        bucket buckets[N_BUCKETS], *b;
        long int *returns;
```

```c
        // allocate memory
        returns = malloc(sizeof(long int) * size);
        for (i = 0; i < N_BUCKETS; i++) {
                buckets[i].data = returns + i * size / N_BUCKETS;
                buckets[i].length = length;
                buckets[i].total = 0;
        }

        // copy the keys to "buckets"
        for (i = 0; i < size; i++) {
                b = &buckets[*(a + i * length) - 0x21];
                b->data[b->total++] = i;
        }

        // sort each "bucket"
        for (i = 0; i < N_BUCKETS; i++)
                sort(a, &buckets[i]);

        return returns;
}
```

Problem B

Mutually Friendly Numbers

Two numbers are *mutually friendly* if the ratio of the sum of all divisors of the number and the number itself is equal to the corresponding ratio of the other number. This ratio is known as the abundancy of a number. For example, 30 and 140 are friendly, since the abundancy of these two numbers is equal. Figure B.1 show this example.

$$\frac{1+2+3+5+6+10+15+30}{30}=\frac{72}{30}=\frac{12}{5}$$

$$\frac{1+2+4+5+7+10+14+20+28+35+70+140}{140}=\frac{336}{140}=\frac{12}{5}$$

Figure B.1 – 30 and 140 are friendly.

This problem consists in finding all pairs of natural numbers that are mutually friendly within the range of positive integers provided to the program at the start of the execution.

Write a parallel program to compute mutually friendly numbers.

Input

The input contains several test cases. Each line contains two integers ($1 \leq S, E < 2^{20}$) that correspond to the range where the mutually friendly numbers will be searched. The test case ends when $S=0$ and $E=0$.

The input must be read from the standard input.

Output

The output contains a message for each mutually friendly numbers found, for each test case.

The output must be written to the standard output.

Example

Input	Output for the input

`30 140`
`100 1000`
`0 0`
	`Number 30 to 140`
	`30 and 140 are FRIENDLY`
	`Number 100 to 1000`
	`102 and 476 are FRIENDLY`
	`114 and 532 are FRIENDLY`
	`120 and 672 are FRIENDLY`
	`135 and 819 are FRIENDLY`
	`138 and 644 are FRIENDLY`
	`150 and 700 are FRIENDLY`
	`174 and 812 are FRIENDLY`
	`186 and 868 are FRIENDLY`
	`240 and 600 are FRIENDLY`
	`864 and 936 are FRIENDLY`

```
int gcd(int u, int v) {
        if (v == 0)
                return u;
        return gcd(v, u % v);
}

void friendly_numbers(long int start, long int end) {
        long int last = end - start + 1;

        long int *the_num;
        the_num = (long int*) malloc(sizeof(long int) * last);
        long int *num;
        num = (long int*) malloc(sizeof(long int) * last);
        long int *den;
        den = (long int*) malloc(sizeof(long int) * last);

        long int i, j, factor, ii, sum, done, n;

        for (i = start; i <= end; i++) {
                ii = i - start;
                sum = 1 + i;
                the_num[ii] = i;
                done = i;
                factor = 2;
                while (factor < done) {
                        if ((i % factor) == 0) {
                                sum += (factor + (i / factor));
                                if ((done = i / factor) == factor)
                                        sum -= factor;
                        }
                        factor++;
                }
                num[ii] = sum;
                den[ii] = i;
                n = gcd(num[ii], den[ii]);
                num[ii] /= n;
                den[ii] /= n;
        } // end for

        for (i = 0; i < last; i++) {
                for (j = i + 1; j < last; j++) {
                        if ((num[i] == num[j]) && (den[i] ==
den[j]))
                                printf("%ld and %ld are FRIENDLY\n",
the_num[i], the_num[j]);
                }
        }

        free(the_num);
        free(num);
        free(den);
}

int main(int argc, char **argv) {
        long int start;
        long int end;

        while (1) {
                scanf("%ld %ld", &start, &end);
                if (start == 0 && end == 0)
                        break;
                printf("Number %ld to %ld\n", start, end);
                friendly_numbers(start, end);
        }

        return EXIT_SUCCESS;
}
```

Problem C

Haar Wavelets

Compressing digital images is the key to store and transmit movies nowadays. This happens with JPG2000 images and MPEG videos (MPEG4 and H.264). Haar Wavelet transform is part of this compression[2].

Each image consists of a fairly large number of little squares called pixels (picture elements). The matrix corresponding to a digital image assigns a whole number to each pixel. For example, in the case of a 256x256 pixel gray scale image, the image is stored as a 256x256 matrix, with each element of the matrix being a whole number ranging from 0 (for black) to 225 (for white).

Before understanding the transform in the matrix, let's see it in a vector. Consider it with *n* elements. The Haar wavelet transform calculates the approximation coefficients (*a*) and details coefficients (*d*) as follows:

$$a = \frac{p_i + p_{i+1}}{2} \text{ e } d = \frac{p_i - p_{i+1}}{2}\text{, where } p_i\ (i < n) \text{ is a value from the vector.}$$

After the transformation, a new vector is formed as follows:

$$t_h = [\ a_1\ \ a_2\ \ \ldots\ \ a_{n/2}\ \ d_1\ \ d_2\ \ \ldots\ \ d_{n/2}\ ]$$

Figure C.1 show an example.

$t_0 = [\ 420\ \ 680\ \ 448\ \ 709\ \ 1420\ \ 1260\ \ 1600\ \ 1600\ ]$

$a_1 = (420+680) \div 2,\ d_1 = (420-680) \div 2$

$a_2 = (448+709) \div 2,\ d_2 = (448-709) \div 2$

$a_3 = (1420+1260) \div 2,\ d_3 = (1420-1260) \div 2$

$a_4 = (1600+1600) \div 2,\ d_4 = (1600-1600) \div 2,\ \therefore$

$t_1 = [\ 550\ \ 578\ \ 1340\ \ 1600\ \ -130\ \ -130\ \ 80\ \ 0\ ]$

$a_1 = (550+578) \div 2,\ d_1 = (550-578) \div 2$

$a_2 = (1340+1600) \div 2,\ d_2 = (1340-1600) \div 2$

$t_2 = [564\ \ 1470\ \ -14\ \ -130\ \ -130\ \ -130\ \ 80\ \ 0\ ]$

$a_1 = (564+1470) \div 2,\ d_1 = (564-1470) \div 2$

$t_3 = [1017\ \ -453\ \ -14\ \ -130\ \ -130\ \ -130\ \ 80\ \ 0\ ]$

Figure C.1 – Haar wavelet transform in an 8-vector.

[2] http://aix1.uottawa.ca/~jkhoury/haar.htm

In general, if the data string has length equal to 2^k, then the transformation process will consist of *k* steps. In the above case, there will be 3 steps since $8=2^3$.

Back to an image, considering a $2^k x 2^k$ matrix, a new matrix can be obtained by alternating the row-transformation and column-transformation, as show in Figure C.2.

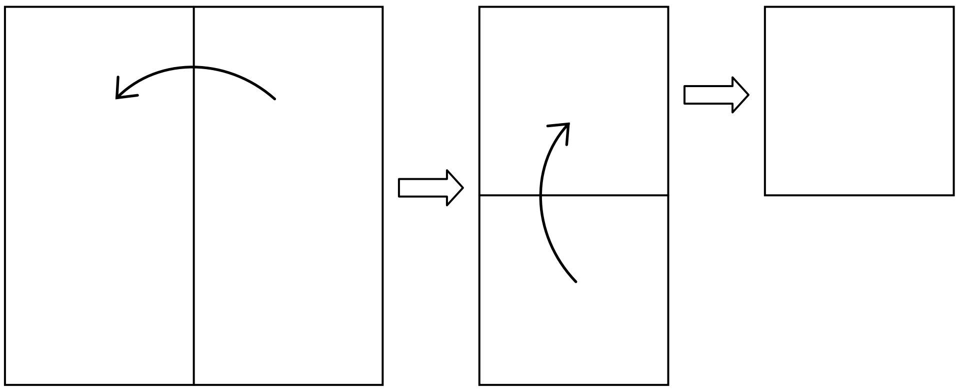

Figura C.2 – Row-transformation and column-transformation.

Write a parallel program that implements the Haar wavelet transform.

Input

The input contains only one test case. The input file is in binary format: the first 32 bits word represents the width (and height) of an image ($2^3 \leq S < 2^{16}$); all the images come next, until the end of file. Each pixel is represented by a 32 bits word.

The input must be read from a file named <u>*image.in*</u>

Output

The output file must be in binary format, keeping the same structure from input file. This output file must have the processed image.

The output must be written to a file named <u>*image.out*</u>

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

#define pixel(x,y) pixels[((y)*size)+(x)]

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

    FILE *in;
    FILE *out;

    in = fopen("image.in", "rb");
    if (in == NULL) {
        perror("image.in");
        exit(EXIT_FAILURE);
    }
    out = fopen("image.out", "wb");
    if (out == NULL) {
        perror("image.out");
        exit(EXIT_FAILURE);
    }

    long long int s, size, mid;
    int x, y;
    long long int a, d;
    double SQRT_2;
    fread(&size, sizeof(size), 1, in);
    fwrite(&size, sizeof(size), 1, out);
    int *pixels = (int *) malloc(size * size *
sizeof(int));
    if (!fread(pixels, size * size * sizeof(int), 1, in)) {
        perror("read error");
        exit(EXIT_FAILURE);
    }

    // haar
    SQRT_2 = sqrt(2);
    for (s = size; s > 1; s /= 2) {
        mid = s / 2;
        // row-transformation
        for (y = 0; y < mid; y++) {
            for (x = 0; x < mid; x++) {
                a = pixel(x,y);
                a = (a+pixel(mid+x,y))/SQRT_2;
                d = pixel(x,y);
                d = (d-pixel(mid+x,y))/SQRT_2;
                pixel(x,y) = a;
                pixel(mid+x,y) = d;
            }
        }
        // column-transformation
        for (y = 0; y < mid; y++) {
            for (x = 0; x < mid; x++) {
                a = pixel(x,y);
                a = (a+pixel(x,mid+y))/SQRT_2;
                d = pixel(x,y);
                d = (d-pixel(x,mid+y))/SQRT_2;
                pixel(x,y) = a;
                pixel(x,mid+y) = d;
            }
        }
    fwrite(pixels, size * size * sizeof(int), 1, out);

    free(pixels);
    fclose(out);
    fclose(in);

    return EXIT_SUCCESS;
}
```

Problem D

Unbounded Knapsack Problem

The unbounded knapsack problem is a classical resource allocation problem with many real-world applications, such as capital investment, microchip synthesis, and cryptography, etc. The problem is as follows.

Suppose *n* items x_1 to x_n where x_i has weight w_i and value v_i. Also, suppose a knapsack with maximum weight capacity *M*. All the previous variables are non-negative integers. The problem is to determine values for all x_i, such that:

$\sum_{i=1}^{n} x_i v_i$ is the maximal, and

$\sum_{i=1}^{n} x_i w_i \leq M$ is true.

The unbounded knapsack problem is NP-hard. The problem is said "unbounded" because there is no limiting value for x_i.

Write a parallel program that solves the knapsack problem.

Input

The input contains only one test case. The first line contains two integers, separated by a space: the number of item *n* and the knapsack's capacity *M*. The remaining *n* lines, one per item x_i, also contain two space-separated integers, the value ($1 \leq v_i < 1024$) and the weight ($1 \leq w_i < 1024$) of item x_i.

The input must be read from the standard input.

Output

The output contains only one line printing the integer that is the maximal value achieved for the knapsack problem.

The output must be written to the standard output.

Example

Input	Output for the input
`3 50`	`300`
`60 10`	
`100 20`	
`120 30`	

```c
typedef struct _item_t {
        int value;   // v_i
        int weight;  // w_i
        float density; // v_i/w_i
} item_t;

int greater_f(const int x, const int y);
int compare_f(const void *x, const void *y);

int knapsack_f(const int n, const int M, const item_t * const
itens) {

        int v = 0, w = 0;
        int r = 0;

        if (n < 1)
                return 0;

        while (M - w >= 0) {
                r = greater_f(r, v + knapsack_f(n - 1, M - w,
&(itens[1])));
                v += itens[0].value;
                w += itens[0].weight;
        }

        return r;

}

int main(int argc, const char *argv[]) {

        int n;
        int M;
        item_t *itens;
        int i;

        scanf("%d %d", &n, &M);
        itens = (item_t*) calloc(n, sizeof(item_t));

        for (i = 0; i < n; ++i) {
                scanf("%d %d", &(itens[i].value),
&(itens[i].weight));
                itens[i].density = (float) (itens[i].value) /
itens[i].weight;
        }

        qsort(itens, (size_t) n, sizeof(item_t), compare_f);

        printf("%d\n", knapsack_f(n, M, itens));

        free(itens);

        return 0;

}

int greater_f(const int x, const int y) {
        return (x > y) ? x : y;
}

int compare_f(const void *x, const void *y) {
        return ((item_t*) x)->density - ((item_t*) y)->density;
}
```

Problem E

We’re Back: 3SAT

Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable.

A *literal* is either a variable or the negation of a variable. A *clause* is a disjunction of literals.

3SAT is a special case of *k*-satisfiability, when each clause contains exactly $k = 3$ literals. For example:

$$E = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4)$$

In this formula, E has two clauses (denoted by parentheses), four variables (x_1, x_2, x_3, x_4), and $k=3$ (three literals per clause).

Write a parallel program that determines if there is an assignment of Boolean values that will satisfy the given 3SAT expression.

Input

The input contains only one test case. The first line contains two integers: the number of clauses (N) and the number of literals (L), separated by a single space ($1 \leq N \leq 255$, $1 \leq L < 2^{64}$). The next N lines contain three integers separated by a space ($1 \leq x_l \leq N$). These integers represent a literal: either a variable or the negation of a variable.

The input must be read from the standard input.

Output

If there is an assignment that satisfies the entire input expression, the output contains an integer that represents the solution followed by its binary representation: each bit represents the variable’s value.

Otherwise, if there is no solution, the output contains only one message ‘Solution not found.’.

The output must be written to the standard output.

Example 1

Input	Output for the input

`4 3`
`3 3 3`
`2 1 -1`
`-3 -2 -3`
`2 1 2`
	`Solution found [5]: 1 0 1`

Example 2

Input	Output for the input

`2 1`
`1 1 1`
`-1 -1 -1`
	`Solution not found.`

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

long solveClauses(short **clauses, int nClauses, int nVar) {

        long *iVar = (long *) malloc(nVar * sizeof(long));
        int i;
        for (i = 0; i < nVar; i++)
                iVar[i] = exp2(i);

        unsigned long maxNumber = exp2(nVar);
        long number;
        short var;
        int c;

        for (number = 0; number < maxNumber; number++) {
                for (c = 0; c < nClauses; c++) {

                        var = clauses[0][c];
                        if (var > 0 && (number & iVar[var - 1]) > 0)
                                continue; // clause is true
                        else if (var < 0 && (number & iVar[-var - 1]) == 0)
                                continue; // clause is true

                        var = clauses[1][c];
                        if (var > 0 && (number & iVar[var - 1]) > 0)
                                continue; // clause is true
                        else if (var < 0 && (number & iVar[-var - 1]) == 0)
                                continue; // clause is true

                        var = clauses[2][c];
                        if (var > 0 && (number & iVar[var - 1]) > 0)
                                continue; // clause is true
                        else if (var < 0 && (number & iVar[-var - 1]) == 0)
                                continue; // clause is true

                        break; // clause is false
                }

                if (c == nClauses)
                        return number;
```

```c
        }
        return -1;
}

short **readClauses(int nClauses, int nVar) {

        short **clauses = (short **) malloc(3 * sizeof(short *));
        clauses[0] = (short *) malloc(nClauses * sizeof(short));
        clauses[1] = (short *) malloc(nClauses * sizeof(short));
        clauses[2] = (short *) malloc(nClauses * sizeof(short));

        int i;
        for (i = 0; i < nClauses; i++) {
                scanf("%hd %hd %hd", &clauses[0][i], &clauses[1][i],
&clauses[2][i]);
        }

        return clauses;
}

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

        int nClauses;
        int nVar;
        scanf("%d %d", &nClauses, &nVar);

        short **clauses = readClauses(nClauses, nVar);

        long solution = solveClauses(clauses, nClauses, nVar);

        int i;
        if (solution >= 0) {
                printf("Solution found [%ld]: ", solution);
                for (i = 0; i < nVar; i++)
                        printf("%d ", (int) ((solution & (long) exp2(i)) /
exp2(i)));
                printf("\n");
        } else
                printf("Solution not found.\n");

        return EXIT_SUCCESS;
}
```