Computer Science I
Program 1: BigIntegers
Please Check WebCourses for the Data and Program due dates

The Problem

The `unsigned int` type in C requires 4 bytes of memory storage. With 4 bytes we can store integers as large as 2^{32}-1. What if we need bigger integers, for example ones having hundreds of digits? If we want to do arithmetic with such very large numbers we cannot simply use the `unsigned int` data type. One way of dealing with this is to use a different storage structure for integers, such as an array of digits. We can represent an integer as an array of digits, where each digit is stored in a different array index. Since the integers are allowed to be as large as we like, a dynamically-sized array will prevent the possibility of overflows in representation. However we need new functions to operate on these integers. In this assignment, you will develop a function to multiply arbitrarily large integers stored in this manner. To help you, you will be given the functionality to read in a large integer and add two large integers.

Each integer is represented as an array of digits, where the least significant digit is stored in index 0, and the last index stores a non-zero number. (The only exception to this is 0, which is stored in an array of size 1 that has a single element storing 0.) For instance, the value `1234567890` would be stored as:

index	0	1	2	3	4	5	6	7	8	9
array	0	9	8	7	6	5	4	3	2	1

Note: Although this seems counter-intuitive, it makes the code slightly easier, because in all standard mathematical operations, we start with the least significant digit. It also makes sense that the digit at the place 10^i is stored in index i.

Implementation Restrictions

You must use the following struct:

```
struct integer {
     int* digits;
     int size;
}
```

Whenever you store or return a big integer, always make sure not to return it with any leading zeros. Namely, make sure that the value stored in index size-1 is NOT zero. The only exception to this rule is if 0 is being stored. 0 should be stored in an array of size 1.

Here are the prototypes of the functions you will be given:

```
//Preconditions: the first parameter is string that only
//               contains digits and is 10000 digits or
//               fewer. No leading 0s will be included
//               unless the number represented is 0.
//Postconditions: The function will read the digits of the
//             large integer character by character,
//             convert them into integers and return a
//             pointer to the appropriate struct integer.
struct integer* convert_integer(char* stringInt);

//Preconditions: p is a pointer to a big integer.
//Postconditions: The big integer pointed to by p is
//                printed out.
void print(struct integer *p);

//Preconditions: p and q are pointers to struct integers.
//Postconditions: A new struct integer is created that
//                stores the sum of the integers pointed to
//                by p and q and a pointer to it is
//                returned.
struct integer* add(struct integer *p, struct integer *q);
```

Here is the prototype for the function you are to write:

```
//Preconditions: p and q are pointers to struct integers.
//Postconditions: A new struct integer is created that
//                stores the product of the integers
//                pointed to by p and q and a pointer to it
//                is returned.
struct integer* multiply(struct integer *p,
                         struct integer *q);
```

You may write other functions to help you with this task. In fact, you are encouraged to do so. You are free to design the prototypes of these other functions.

Input/Output Specification

Your program will read input from the file, "bigint.txt", which will specify a number of multiplication operations to carry out between pairs of large integers.

The input file format is as follows:

The first line will contain a single positive integer, *n*, representing the number of operations to carry out. The next *n* lines will contain one problem each. Each of these lines will have two non-negative integers separated by white space. The two integers on the line will be the two operands for the problem. You may assume that these two integers are non-negative, are written with no leading zeros (unless the number itself is 0) and that the number of digits in either of the numbers will never exceed 10000. (Note: Thus, the answer of the operation will never exceed 20000 digits.)

You should generate your output to the screen. In particular, you should generate one line of output per each input case, with the following format:

```
Problem #k: X * Y = Z
```

where k is the problem number, starting at 1, X is the first operand given in the problem, Y is the second operand given in the problem, and Z is the product of the two operands.

Sample Input File

```
3
4 5
27 10
1111111111111111111 4
```

Corresponding Output

```
Problem #1: 4 * 5 = 20
Problem #2: 27 * 10 = 270
Problem #3: 1111111111111111111 * 4 = 4444444444444444444
```

Deliverables

Week #1: Turn in one file, *bigint.txt*, with your test data, over WebCourses. Your files must be in the format specified above. You will be graded upon following the input specifications and the thoroughness of your test cases.

Week #2: Turn in a single file, *bigintmult.c*, over WebCourses that solves the specified problem. Please do not make any enhancements to your program. Make sure it solves this specified problem exactly.

Computer Science I: Program #5 – Cop and Robbers
(Adapted from 2011 South East Regional Problem I: Moving Points)

Check WebCourses for the due date and time.

Note: THERE IS NO REDO OPPORTUNITY FOR THIS PROGRAM!!!

You are a cop and have the unfortunate circumstance of being the only cop at the scene of a robbery with multiple robbers. Unfortunately, all of the robbers have equally split the loot and are running in different directions. Lucky for you, due to your superior professional training, you are faster than all of the robbers! It's also helpful that the robbers aren't particularly bright. They haven't bothered to bring a getaway car and once they pick a direction to run, they continue running in that direction no matter what. (So, you know exactly where they'll be at any point in time.)

Of course, there are many other crimes to stop, so you would like to catch all of the robbers as quickly as possible. We assume that you have a instantaneous taser, so that once you catch up to a particular robber, you can taze him immediately and he'll remain in that position until another cop comes along and books him. As soon as this happens, you immediately move onto catching the next robber, without losing any time.

The Problem
Assuming that you start at (0, 0), given your speed, as well as the initial positions, speeds and fleeing directions of all of the robbers, determine the minimum amount of time it will you take to catch all of them.

The Input
There will be several robberies to solve. Each robbery begins with two integers, N and C, separated by a space, denoting the number of robbers ($1 \leq N \leq 7$) and the speed of the cop ($0 < C \leq 100$) in units/sec, respectively.

Each of the next N lines will have four space-separated integers, X, Y, D and S. (X, Y) will representing the initial position of that particular robber at time t = 0 ($-100 \leq X, Y \leq 100$), D represents the direction of movement in degrees (0 degrees is the positive X axis, 90 degrees is the positive Y axis), and S ($0 \leq S < C$) is the speed of that robber in units/sec. It is assumed that all robbers start moving immediately at time t = 0.

All of the input cases will be constructed such that the desired minimum time does not exceed 10000.

The input will end with a line with two 0s.

The Output
For each test case, output a single real number on its own line, representing the least amount of time needed for the cop to catch all of the robbers. Print this number to exactly 2 decimal places, rounded. Any answer within 0.01 of the correct answer will be deemed correct.

Sample Input

```
2 25
19 19 32 10
6 45 133 19
5 10
10 20 45 3
30 10 135 4
100 100 219 5
10 100 301 4
30 30 5 3
0 0
```

Sample Output

```
12.62
12.54
```

Mathematics Background

Given that a cop (or robber) starts at position (x0, y0) at time t0, and that he is moving in the direction D with speed S, then his position at time t0 + T (after T seconds have passed), is:

(x0 + STcos(D), y0 + STsin(D))

If you know where a robber will be at some particular time t0+T and you know where a cop is at time t0, to determine if the cop can catch the robber by time t0+T, calculate the distance, D_{chase} between where the cop is at time t0 and where the robber WILL BE at time t0+T. Then, see if ST, the distance the cop travels in T seconds is greater than or less than D_{chase}. If it is less, it is impossible to catch that robber in that much time. If it is more, than you can catch the robber and might be able to do so faster. When these two values, D_{chase} and ST are very close to one another (say absolute value of the difference less than 0.00000001), then your value for T represents the minimum time it will take to catch that particular robber and that new position represents where the cop will attempt to chase the next robber from.

Implementation Hints

This problem is meant to be an application of the binary search method. The general method of solution is as follows:

1) Try all orderings of catching the robbers. (At most there are 7! of these.)
2) For each ordering, simulate going after the robbers in that fixed order.
3) In simulating, use a binary search over the time to figure out how long it will take you to catch each successive robber. A safe low bound for your search is t=0. A safe high bound for your search is the current distance between you and the robber. Based on the input spec, you are at least 1 unit/sec faster than any robber. So, take this distance and add one to 1, since you are guaranteed to gain at least 1 unit per second on any robber. This will be a safe high bound.
4) Once you get the time to catch one robber, update your position to be where you caught this robber and continue to catch the rest.

Deliverables

Please turn in a single source file, *cop.c*, with your solution to this problem via WebCourses before the due date/time for the assignment. Make sure that your program reads form standard in and outputs to standard out, as previously shown in lab.

Program #5 Grading Criteria

Code Points (40 pts)
Uses standard in – 5 pts
Uses standard out – 5 pts
Reads in all input – 5 pts
Follows output format – 5 pts
Tries to code permutation – 10 pts
Tries to code a binary search – 10 pts

Feel free to award partial credit in each category based on "how well" the program satisfies the criteria presented.

Execution Points (40 pts)
There are 20 cases, 2 points per case. A case is correct if it's within 0.01 of the posted solution. If the program crashes, do not give any credit for future cases even if those may have worked if the program didn't crash.

If you think a simple error caused many, many problems, feel free to spend up to 3 minutes to fix it, retest and if everything is correct, just deduct for that error (5 to 10 points depending on the severity) instead of deducting for each test case missed. But, you are not required to do this. Just running the test cases is adequate.

Style Points (20 pts)
Header comment w/name, program, date – 4 pts
Header comments for each function – 5 pts
Appropriate variable names – 2 pts
Appropriate use of white space – 2 pts
Appropriate indenting – 2 pts
Comments in code – 5 pts

2013 Fall Computer Science I Program #3: Get Out of This Maze!!!
Please consult Webcourse for the due date/time

Queues have many uses. One use of a queue is that it can be used to help find the fastest way out of a maze! The queue is an integral part of a search that is more generally called a Breadth First Search. The idea is as follows. Given a starting point in a maze, travel to all places adjacent to it and mark these as being 1 step away from the start point. Then, for each of these spots that are 1 step away, try travelling to all adjacent spots that are not yet visited. Mark these as 2 spots away. Continue until one of the spots you mark is an exit! By hand, it's quite easy to visually solve this problem. In order to write a program to do it for a 2D array maze, you need a queue that keeps track of each place you have reached that you haven't yet tried to visit new places from. The queue is necessary because to get shortest distances, you must process each possible location in the order in which you arrived to them. In addition, as you travel, you should mark which places you've reached so far, so that you don't try to travel to them again. This second part is critical! Without it, you'll revisit some squares many, many times, resulting in a very slow algorithm.

Let's run through an example by hand, showing what has to happen, visually in code, in both the queue and the 2D array that is storing all shortest distances. We initialize any unvisited square to -1 to indicate that it is in fact, unvisited. When we visit that square, we will update its value to the shortest distance to travel there.

Consider the following maze:

```
~~~~~~~
~XXXXX~
~X-S-X~
~--X-X~
~~~~~~~
```

Let's assume we can only move up, down, left and right from where we currently are. Let S denote the starting square, let X denote an illegal square to move to, let ~ represent the outer boundary to which we are trying to get to, and let – denote a square which is free to visit.

Our initial arrays and queue would look like this:

```
~~~~~~~    -1 -1 -1 -1 -1 -1 -1    Queue: (2,3)
~XXXXX~    -1 -1 -1 -1 -1 -1 -1
~X-S-X~    -1 -1 -1  0 -1 -1 -1
~--X-X~    -1 -1 -1 -1 -1 -1 -1
~~~~~~~    -1 -1 -1 -1 -1 -1 -1
```

At this point, we dequeue the next item and try to go to all possible adjacent squares, which are (2, 2) and (2, 4). We enqueue both of these into the queue, marking that the distance to both of them is 0 + 1 = 1. It does not matter which order we enqueue these two items. (The 0 comes from the distance to (2,3) the +1 is for the move left or right.) After this iteration, we have the following:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (2,2), (2,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1 -1 -1 -1 -1 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

Next, we dequeue (2,2) and look for its unvisited neighbors (note that (2,3) is visited because in the distance array, 0 is stored in that slot, not -1.) The only unvisited neighbor of (2, 2) is (3, 2), so we update the distance to (3, 2) and enqueue it to get:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (2,4), (3,2)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1 -1  2 -1 -1 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

Next, we dequeue (2,4) and look for its unvisited neighbors (note that (2,3) is visited because in the distance array, 0 is stored in that slot, not -1.) The only unvisited neighbor of (2, 4) is (3, 4), so we update the distance to (3, 4) and enqueue it to get:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,2), (3,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1 -1  2 -1  2 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

Now, we dequeue (3,2) and look for its unvisited neighbors The only unvisited neighbors of (3, 2) are (3, 1), and (4, 2). We update the distance to both of these and enqueue to get:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,2), (3,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1  3  2 -1  2 -1 -1
~~~~~~~     -1 -1  3 -1 -1 -1 -1
```

Theoretically, we could be smart enough to recognize that (4, 2) is on the boundary and that the shortest way out is 3. In code though, most people end up waiting to stop the search until (4, 2) is dequeued, so in this trace, we will continue from this point. Here is the current state:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,4), (3,1), (4, 2)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1  3  2 -1  2 -1 -1
~~~~~~~     -1 -1  3 -1 -1 -1 -1
```

Now, we dequeue (3, 4) and get to this state, once we perform all necessary duties:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,1), (4,2), (4,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1  3  2 -1  2 -1 -1
~~~~~~~     -1 -1  3 -1  3 -1 -1
```

Next, we dequeue (3, 1):

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1  Queue:(4,2),(4,4),(3,0),(4,1)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~      4  3  2 -1  2 -1 -1
~~~~~~~     -1  4  3 -1  3 -1 -1
```

Finally, when we get here, we dequeue (4, 2) and realize that we're out of the maze and return our final answer of 3. Note: If you want, you can return this value right when this spot would have been enqueued.

The Problem
Given a maze as described previously, determine whether or not there is a way to escape to the boundary, and if so, what the shortest distance to escape the maze to any of the boundary positions is. At each move, you may move up, down, left or right from your previous spot, so long as the new spot isn't forbidden.

The Input
The first line of the input file will contain a single positive integer, *c* *(c ≤ 100)*, representing the number of input cases. The input cases follow, one per line. The first line of each input case will contain two positive integers, *r(3 ≤ r ≤ 300)*, and *c(3 ≤ c ≤ 300)*, representing the number of rows and columns, respectively, in the maze. The following *r* lines will contain strings of exactly c characters, describing the contents of that particular row using the symbols described above `(~, X, -, S)`. You are guaranteed that the first and last rows and first and last columns will only contain the border character, `~`. You are guaranteed that this character will not appear anywhere else in the grid. Exactly one non border character will be an `S`, so there will always be exactly one starting location. Finally, the rest of the squares will either be `-`, to indicate that that square is valid to travel to, or `X`, to indicate that you may not travel to that square.

The Output
For each case, if it's possible to reach the border of the maze, output the fewest number of steps necessary to reach any border square. If it's not possible, output -1.

Sample Input
```
2
3 3
~~~
~S~
~~~
5 6
~~~~~~
~XXXX~
~XS-X~
~-XX-~
~~~~~~
```

Sample Output
```
1
-1
```

Specification Details
You must implement a queue of ordered pairs or integers to receive full credit on this assignment. A significant portion of the grade will NOT come from execution, but rather the implementation of an appropriate struct to store a queue AND the implementation of the Breadth First Search algorithm described in this problem write-up.

Deliverables
Please turn in a single source file, *maze.c*, with your solution to this problem via Webcourses before the due date/time for the assignment. Make sure that your program reads form standard in and outputs to standard out, as previously shown in lab.

<u>Program #3 Grading Criteria</u>

<u>Code Points (35 pts)</u>
Uses standard in – 5 pts
Uses standard out – 5 pts
Reads in all input – 5 pts
Follows output format – 5 pts
Has a queue struct – 5 pts
Has enqueue – 5 pts
Has dequeue – 5 pts

Feel free to award partial credit in each category based on "how well" the program satisfies the criteria presented.

<u>Execution Points (50 pts)</u>
There are 50 cases, 1 point per case. If the program crashes, do not give any credit for future cases even if those may have worked if the program didn't crash.

If you think a simple error caused many, many problems, feel free to spend up to 3 minutes to fix it, retest and if everything is correct, just deduct for that error (5 to 10 points depending on the severity) instead of deducting for each test case missed. But, you are not required to do this. Just running the test cases is adequate.

<u>Style Points (15 pts)</u>
Header comment w/name, program, date – 4 pts
Appropriate variable names – 2 pts
Appropriate use of white space – 2 pts
Appropriate indenting – 2 pts
Comments in code – 5 pts

2013 Fall Computer Science I Program #1: Organ Transplant Database
Please consult Webcourse for the due date/time

The United States as an Organ Transplant database, to keep track of those who need organ transplants. Depending on a number of factors, when matches are found, organs are donated to those on the list. Typically, one can only receive a donation if the donor's organ is a match for the recipient. In this program, we will simulate a simplified organ transplant data base. To simplify the problem, our data base will simply keep track of the following information for each person in need of an organ:

1) Person's name
2) The organ they need to replace.
3) Their blood type (A+, A-, B+. B-, AB+, AB-, O+, O-)
4) The date (month, day, year) they were added to the list.
5) The time (hours, minutes) in military time they were added to the list.

Your program should use the following structs and constants:

```
#define SIZE 20
#define BLOODTYPESIZE 4

typedef struct {
    int month;
    int day;
    int year;
} dateT;

typedef struct {
    int hour;
    int minute;
} timeT;

typedef struct {
    char name[SIZE];
    char organname[SIZE];
    char bloodtype[BLOODTYPESIZE];
    dateT dateAdded;
    timeT timeAdded;
    int received;
} organT;
```

The name component will store the patient's name, organname will store the name of the organ they are in need of, bloodtype will store one of the 8 aforementioned strings representing bloodtype, the dateAdded and timeAdded will store when the corresponding organ was put on the waitlist while received will store a 0 initially and may store a 1 to indicate that the organ has been transplanted. (Note: In this assignment you will not physically remove the record of any patient, even if they have received their organ. This field will be solely used so that you can skip matching an organ to a patient who has already received one.)

Your program should read in information for a file consisting of the organ wait list, as well as a sequence of organs received for donation. For each organ received, your program should print out the name of the person and the organ they have received. The organ should go to the the person who has been on the waitlist the longest, who is a match for the organ. For the purposes of this assignment, a match occurs when the donor organ is the same AND the bloodtype of the donor is the same as the recipient. Once a match is found for an organ, they should not be matched again.

Input File Format

The first line of the input file will contain a single positive integer, *n* $(n \leq 120000)$, representing the number of organs on the waiting list. The next *n* lines will contain information about one organ each. Each of these lines will contain the person's name, the organ they need replaced, their blood type, the date they were added to the organ database and the time they were added to the organ database. Each of these items will be separated by a space. All names will be comprised of letters and underscores only, all organ names will be comparised of lowercase letters, all bloodtypes will be one of the previously mentioned 8 strings, all dates will be of the format *m/d/y*, where *m, d,* and *y,* represent the numeric month day and year the patient was added to the organ donation list (for this particular organ). Finally, the time will be of the form *hr:min*, where $hr(0 \leq hr \leq 23)$ and $min(0 \leq min \leq 59)$ represent the numeric hour and minutes for the time the patient was added to the organ donation list. You are guaranteed that no two organs were added to the list on the same date and time and that no name or organ name will contain more than 19 characters.

The following line of the input file (line number $n+2$), will contain a single positive integer, *k* $(k \leq 1000)$ representing the number of organs received, during some fixed period in time. The following *k* lines will contain information about the organs received, in the order they were received. Each of these lines will contain two strings separated by a space: the name of the organ and the blood type of the donor. These will both adhere to the specifications previously given.

Output Specification

Output a single line for each organ received. If a matching recipient exists in the database that hasn't yet received an organ, print out the name of the recipient, followed by the organ they received. If no match exists in the database, print out the following on a single line.

```
No match found
```

Sample Input and Output

This is posted separately in the files *organ.in* and *organ.out*.

Specification Details

You must dynamically allocate an array of the appropriate size of type organT to store all of the organs on the waiting list. You must free this memory once it is no longer in use. Though a fancier solution is possible, for this assignment you will be asked to simply run a linear search through the entire organ array to find the best match for each organ. Thus, your program will take a few seconds to run on a maximum sized input file.

Deliverables

Please turn in a single source file, *organ.c*, with your solution to this problem via Webcourses before the due date/time for the assignment. Make sure that your program reads form standard in and outputs to standard out, as previously shown in lab.

Program #1 Grading Criteria

Code Points (35 pts)
Used all structs and constants given as is (okay if others added) – 5 pts
Uses standard in – 5 pts
Uses standard out – 5 pts
Properly allocates memory dynamically – 5 pts
Properly frees the dynamically allocated memory – 5 pts
Reasonable breakdown of functions – (5 pts) 1 pt for each function upto 5 pts, as long as the breakdown is reasonable
No set of 4 or more nested ifs (this is all or nothing) – 5 pts
Feel free to award partial credit in each category based on "how well" the program satisfies the criteria presented.

Execution Points (50 pts)
There are 5 test files, each is worth 10 points. If at least one answer in a test file is incorrect, then it can't get all 10 points. You may simply give a proportional number of points to the number of test cases solved, rounding down. **If you think a simple error caused many, many problems, feel free to spend up to 3 minutes to fix it, retest and if everything is correct, just deduct for that error (5 to 10 points depending on the severity) instead of deducting for each test case missed. But, you are not required to do this. Just running the test cases is adequate.**

Style Points (15 pts)
Header comment w/name, program, date – 4 pts
Appropriate variable names – 2 pts
Appropriate use of white space – 2 pts
Appropriate indenting – 2 pts
Comments in code – 5 pts

2013 Fall Computer Science I Program #3: Organ Transplant Database
Please consult Webcourse for the due date/time

Note: For this assignment, you will be writing a program that implements a different solution to the problem posed in the first program for the course. The only differences in this assignment compared to program #1 will be in the section labeled "**Specification Details**", **one tiny change in organT, and the total number of donated organs received in the input file.** Since most of the sample cases are already known, the grading criteria for this assignment will be posted beforehand. For convenience, the whole description of the problem is reposted below.

The United States as an Organ Transplant database, to keep track of those who need organ transplants. Depending on a number of factors, when matches are found, organs are donated to those on the list. Typically, one can only receive a donation if the donor's organ is a match for the recipient. In this program, we will simulate a simplified organ transplant data base. To simplify the problem, our data base will simply keep track of the following information for each person in need of an organ:

1) Person's name
2) The organ they need to replace.
3) Their blood type (A+, A-, B+. B-, AB+, AB-, O+, O-)
4) The date (month, day, year) they were added to the list.
5) The time (hours, minutes) in military time they were added to the list.

Your program should use the following structs and constants:

```
#define SIZE 20
#define BLOODTYPESIZE 4

typedef struct {
    int month;
    int day;
    int year;
} dateT;

typedef struct {
    int hour;
    int minute;
} timeT;

typedef struct {
    char name[SIZE];
    char organname[SIZE];
    char bloodtype[BLOODTYPESIZE];
    dateT dateAdded;
    timeT timeAdded;
} organT;
```

The name component will store the patient's name, organname will store the name of the organ they are in need of, bloodtype will store one of the 8 aforementioned strings representing bloodtype, the dateAdded and timeAdded will store when the corresponding organ was put on the waitlist while received will store a 0 initially and may store a 1 to indicate that the organ has been transplanted. (Note: In this assignment you will not physically remove the record of any patient, even if they have received their organ. This field will be solely used so that you can skip matching an organ to a patient who has already received one.)

Your program should read in information for a file consisting of the organ wait list, as well as a sequence of organs received for donation. For each organ received, your program should print out the name of the person and the organ they have received. The organ should go to the the person who has been on the waitlist the longest, who is a match for the organ. For the purposes of this assignment, a match occurs when the donor organ is the same AND the bloodtype of the donor is the same as the recipient. Once a match is found for an organ, they should not be matched again.

Input File Format

The first line of the input file will contain a single positive integer, *n* $(n \leq 120000)$, representing the number of organs on the waiting list. The next *n* lines will contain information about one organ each. Each of these lines will contain the person's name, the organ they need replaced, their blood type, the date they were added to the organ database and the time they were added to the organ database. Each of these items will be separated by a space. All names will be comprised of letters and underscores only, all organ names will be comparised of lowercase letters, all bloodtypes will be one of the previously mentioned 8 strings, all dates will be of the format *m/d/y*, where *m, d,* and *y,* represent the numeric month day and year the patient was added to the organ donation list (for this particular organ). Finally, the time will be of the form *hr:min*, where $hr(0 \leq hr \leq 23)$ and $min(0 \leq min \leq 59)$ represent the numeric hour and minutes for the time the patient was added to the organ donation list. You are guaranteed that no two organs were added to the list on the same date and time and that no name or organ name will contain more than 19 characters.

The following line of the input file (line number *n+2)*, will contain a single positive integer, *k* $(k \leq 100000)$ representing the number of organs received, during some fixed period in time. The following *k* lines will contain information about the organs received, in the order they were received. Each of these lines will contain two strings separated by a space: the name of the organ and the blood type of the donor. These will both adhere to the specifications previously given.

Output Specification

Output a single line for each organ received. If a matching recipient exists in the database that hasn't yet received an organ, print out the name of the recipient, followed by the organ they received. If no match exists in the database, print out the following on a single line.

```
No match found
```

Input and Output

I/O for this assignment will be the same five files as assignment #1 as well as one new file with more organs received.

<u>Specification Details</u>

You must store each organ request in a node of a binary search tree. When reading in the list of patients waiting for an organ, you must insert each item into the binary search tree. When processing organs received, you must find the appropriate node in the tree that corresponds to the patient to receive the organ, output his/her name, and delete this node from the tree.

In order to implement this assignment using a binary tree, you must have a method for comparing two organs. **Please use the following:**

For both organs, create a concatenated string, organ name plus blood type. For example, using organ0.in, this concatenated string for John_Lynch is "kidneyA-".

Compare both organs using their concatenated strings.

If this doesn't break the tie, then do so using the dates the two patients have been waiting from (earlier date coming first), and if this is a tie, break it using the time. You are guaranteed that no two patients will have the same organ type, blood type, date and time in the input file.

<u>Deliverables</u>

Please turn in a single source file, *organbintree.c*, with your solution to this problem via Webcourses before the due date/time for the assignment. Make sure that your program reads form standard in and outputs to standard out, as previously shown in lab.

Program #4 Grading Criteria

Code Points (35 pts)
Uses standard in/standard out – 5 pts
Has appropriate binary tree node struct – 5 pts
Has appropriate binary tree functions – 10 pts
Has appropriate compare function(s) – 5 pts
Properly allocates memory dynamically – 5 pts
Properly frees memory – 5 pts

Feel free to award partial credit in each category based on "how well" the program satisfies the criteria presented.

Execution Points (50 pts)
organ 0 through organ 4: 5 pts each, all or nothing since these are posted. Credit should only be given if they run in 3 seconds or less.

organ5: 25 pts, 10 pts for running in time, 15 pts for output – grade proportionally (ie. if 2/3 of the cases are correct, award 10 of these 15 points, judge the number of correct cases by eyeballing the diff in a file compare utility)

If you think a simple error caused many, many problems, feel free to spend up to 3 minutes to fix it, retest and if everything is correct, just deduct for that error (5 to 10 points depending on the severity) instead of deducting for each test case missed. But, you are not required to do this. Just running the test cases is adequate.

Style Points (15 pts)
Header comment w/name, program, date – 4 pts
Appropriate variable names – 2 pts
Appropriate use of white space – 2 pts
Appropriate indenting – 2 pts
Comments in code – 5 pts

2013 Fall Computer Science I Program #2: Possible Passwords
Please consult Webcourse for the due date/time

After learning recursion, you've decided to look for some applications of the new problem solving technique you've learned. It turns out that there's an escalating rivalry between a couple student clubs on campus, "Unkempt Freshmen" and "Frustrated Student Underlings". The SGA at UCF suspects that both clubs are up to some fairly nefarious activities. In order to check up on the clubs, your boss has asked you to write a program that can guess passwords for the email accounts of each club. Luckily, after gathering some data, you know exactly how long each password is and what the possible letters are for each slot. As an example, it's possible you might have narrowed down a particular password to be three letters long where the first letter is from the set {'a', 'b', 'c'}, the second letter is from the set {'x', 'y'} and the third letter is from the set {'d', 'm', 'n', 'r'}. From this data, there are 24 possible passwords. You will have to write a program that can iterate through each possible password, in alphabetical order. Since printing out each of the passwords might create unnecessarily long output, to check to see that your program works, you'll only be asked to output specific ranked possible passwords from the list, instead of the whole list itself.

The Problem

Given the length of a password, a list of possible letters for each letter in the password, and a desired alphabetical rank, determine the possible password of the given rank.

The Input

The first line of the input file will contain a single positive integer, c $(c \leq 100)$, representing the number of input cases. The input cases follow, one per line. The first line of each input case will contain a single positive integer, $m (m \leq 20)$, the length of the password. The following m lines will contain strings of distinct lowercase letters in alphabetical order representing each of the possible letters for each letter in the password. The ith line in this set will store the possible letters for the ith letter in the password. The last line of each test case will contain a single positive integer, r $(r \leq 1048576)$, representing the rank of the possible password to output. (You are also guaranteed that the product of the lengths of these m lines won't exceed 1,000,000,000.)

The Output

For each case, output the correct possible password for the query, in all lowercase letters. It is guaranteed that all queries will be for a valid ranked password.

Sample Input

```
2
3
abc
xy
dmnr
10
2
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
676
```

Sample Output

```
bxm
zz
```

Specification Details

You must use recursion in order to get full credit on the assignment. (There is a very elegant non-recursive solution, but I want you to practice recursion…) The standard recursive solution would be to iterate through all the passwords in the desired order, stopping when you reach the desired rank. This solution can earn you full credit. There's a much faster solution which avoids going through each possible password. If you discover this solution, you may earn some extra credit. None of the TAs or I will give you ANY hint about this efficient solution. I want those who earn the extra credit to truly do so.

Deliverables

Please turn in a single source file, *passwords.c*, with your solution to this problem via Webcourses before the due date/time for the assignment. Make sure that your program reads form standard in and outputs to standard out, as previously shown in lab.

Program #2 Grading Criteria

Code Points (35 pts)
Uses standard in – 5 pts
Uses standard out – 5 pts
Reads in all input – 5 pts
Follows output format – 5 pts
Uses recursion – 5 pts
Uses brute force OR mathematical solution – 5 pts
Properly manages any dynamic memory (give full credit to anyone who has no dynamically allocated memory) – 5 pts

Feel free to award partial credit in each category based on "how well" the program satisfies the criteria presented.

Execution Points (50 pts)
There are 25 cases, give 2 pts per case. If the program crashes, do not give any credit for future cases even if those may have worked if the program didn't crash.

If you think a simple error caused many, many problems, feel free to spend up to 3 minutes to fix it, retest and if everything is correct, just deduct for that error (5 to 10 points depending on the severity) instead of deducting for each test case missed. But, you are not required to do this. Just running the test cases is adequate.

Style Points (15 pts)
Header comment w/name, program, date – 4 pts
Appropriate variable names – 2 pts
Appropriate use of white space – 2 pts
Appropriate indenting – 2 pts
Comments in code – 5 pts

EXTRA CREDIT (20 pts)
If the program uses the fast mathematical method and gets ALL 50 execution points, award an extra 20 points. The fast mathematical method takes on the order of the number of letters in the password and not the order of the rank.

THIS IS ALL OR NOTHING.

Students can ONLY earn this if ALL of the test cases work AND they've used the faster method of solution.