METU DEPARTMENT OF COMPUTER ENGINEERING

Homework Collection

Göktürk Üçoluk

DEPARTMENT OF COMPUTER ENGINEERING

C HOMEWORK COLLECTION

by

Dr. Göktürk Üçoluk

METU

2004

Contents

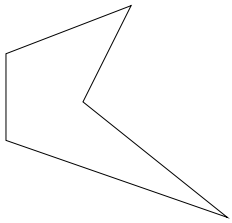
1 CONVEX HULL

'95 HOMEWORK 1

Problem

Consider a set of points in a plane which are represented 2-tuples of real numbers $[x, y]$. We're interested in the boundaries of the point set. Often, when we have a large number of points to process, people looking at a diagram of a set of points plotted in the plane, have little trouble distinguishing those on the "inside" of the point set from those lying on the edge.

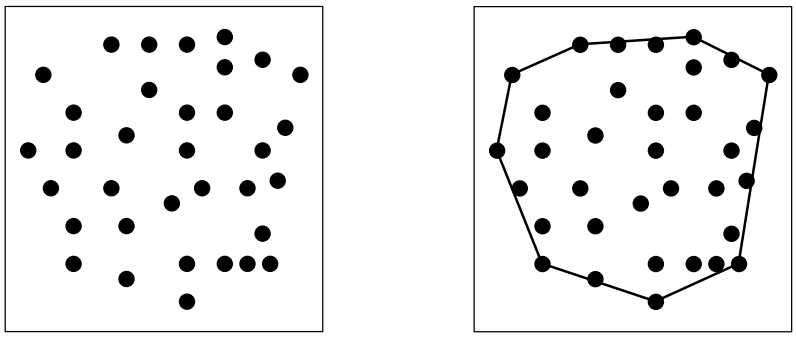
The mathematical way to describe the natural boundary of a point set depends on a geometric property called *convexity*. This is a simple concept that you have encountered before: a *convex polygon* has the property that any line connecting any two points inside the polygon must itself lie entirely inside the polygon. For example the below given polygon is nonconvex;



on the other hand, any triangle or rectangle is convex.

Now, the mathematical name for the natural boundary of a point set is the *convex hull*. The convex hull of a set of points in the plane is defined to be the smallest convex polygon containing them all. Equivalently, the convex hull is the shortest path surrounding the points. An obvious property of the convex hull that is easy to prove is that the vertices of the convex polygon defining the hull are points from the original point set. Given N points, some of them form a convex polygon within which all the others are contained. The problem is to find those points.

Below a sample set of points and their convex hull is shown.



There are 9 points on the hull of the set. In general, the convex hull can contain as few as three points (if the three points form a large¿triangle containing all the others) or as many as all the points (if they fall on a convex polygon, then the points comprise their own convex hull). The number of points on the convex hull of a "random" point set falls somewhere in between these extremes.

A fundamental property of the convex hull is that any line outside the hull, when moved in any direction towards the hull, hits it at one of its vertex points. (This is an alternate way to define the hull: it is the subset of points from the point set that could be hit by a line moving in at some angle from infinity.) In particular, it's easy to find a few points guaranteed to be on the hull by applying this rule for horizontal and vertical lines: the points with the smallest and largest x and y coordinates are all-on the convex hull. This is a fact that you can make use of in determining your starting point.

The input to an algorithm for finding the convex hull is of course an array of points; the output is a polygon, also represented as an array of points with the property that tracing through the points in the order in which they appear in the array traces the outline of the polygon. On reflection, this might appear to require an extra ordering condition on the computation of the convex hull (why not just return the points on the hull in any order?), but output in the ordered form is obviously more useful, and it has been shown that the unordered computation is no easier to do. It is convenient to do the computation in place: the array used for the original point set can also be used to hold the output. The algorithms simply rearrange the points in the original array so that the convex hull appears in the first M positions, in order.

It is helpful to view finding the convex hull of a set of points as a kind of "two-dimensional sort" since frequent parallels to sorting algorithms arise in the study of algorithms for finding the convex hull.

As with all geometric algorithms, we have to pay some attention to degenerate cases that are likely to occur in the input. For example, what is the convex hull of a set of points all of which fall on the same line segment? Depending upon the application, this could be all the points or just the two extreme points, or perhaps any set including the two extreme points would do. Though this seems an extreme example, it would not be unusual for more than two points to fall on one of the line segments defining the hull of a set of points. For your algorithm, we won't insist that points falling on a hull edge be included, since this generally involves more work. On the other hand, we won't insist that they be omitted either, since this condition could be tested afterwards if desired.

Package-Wrapping Method

The most natural convex hull algorithm, which parallels how a human would draw the convex hull of a set of points, is a systematic way to "wrap up" the set of points. Starting with some point guaranteed to be on the convex hull (say the one with the smallest y coordinate), take a horizontal ray in the positive direction and "sweep" it upward until hitting another point; this point must be on the hull. Then anchor at that point and continue "sleeping" until hitting another point, etc., until the "package" is fully "wrapped" (the beginning point is included again).

Of course, we don't actually need to sweep through all possible angles (Why?). [Hint: Implementing a function `theta(`x_1`,`y_1`,`x_2`,`y_2`)` that returns the angle between the horizontal and the line from point (x_1, y_1) to point (x_2, y_2) will be of great help!]

I/O Specifications:

Your program will read the (x, y) pair of coordinates from the *standard input* and print the convex-hull specifying pairs in the counter-clockwise traverse order to the *standard output*. You are given that there are at most 2000 points. Each pair of coordinates are represented as two floating point numbers in a single line separated by at least one blank (No information (like an integer at the start of the input) of how many actual lines are present is provided from the input!).

2 NONSENSE STACK MACHINE

'95 HOMEWORK 2

Background Information

One of the most useful concepts in computer science is that of the *stack*. A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack.

Unlike as for the array, the definition of the stack provides for the insertion and deletion of items, so that a stack is a dynamic, constantly changing object. The question therefore arises, how does a stack change? The definition specifies that a single end of the stack is designated as the stack top. New items may be put on top of the stack (in which case the top of the stack moves upward to correspond to the new highest element), or items which are at the top of the stack may be removed (in which case the top of the stack moves downward to correspond to the new highest element). To answer the question, which way is up? we must decide which end of the stack is designated as its top–that is, at which end items are added or deleted.

The two changes which can be made to a stack are given special names. When an item is added to a stack, it is *pushed onto the stack*, and when an item is removed, it is *popped from the stack*.

Given a stack an item i, performing the operation `push(`i`)` adds the item i to the top of the stack. Similarly, the operation `pop()` removes the top element and returns it as a function value. Thus the assignment operation

```
i = pop();
```

removes the element at the top of the stack and assigns its value to `i`.

There is no theoretical upper limit on the number of items that may be kept in a stack (though a practical limit certainly will exist), since the definition doesn't specify how many items are allowed in the collection. Pushing another item onto a stack merely produces a larger collection of items. However, if a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the *empty stack*. Although the push operation is applicable to any state of the stack, the pop operation cannot be applied to the empty stack because such a stack has no elements to pop. Therefore, before applying the pop operator to a stack, we must ensure that the stack is not empty. The operation `empty()` determines whether or not a stack is empty. If the stack is empty, `empty()` shall return a value meaning TRUE; otherwise it shall return a value meaning FALSE.

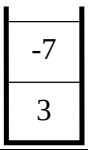
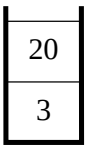
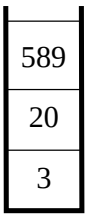
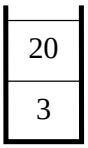
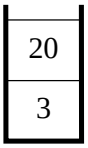
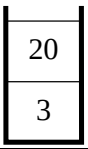
Another operation that can be performed on a stack is to determine what the top item on a stack is without removing it. This operation is written as `stacktop()` and returns the top element of the stack. Like the operation pop, `stacktop` is not defined for an empty stack. The result of an illegal attempt to pop or access an item from an empty stack is called underflow. Underflow can be avoided by ensuring that `empty()` is false before attempting the operation `pop()` or `stacktop()`.

In an imperative language *(like C)* a stack is usually implemented as a one dimensional array of the type of its intended elements. Although the array structure enables random access to any element, the programmer restricts him/herself to use the functions

- `push(`$\triangle$`)`
- `pop()`
- `stacktop()`
- `empty()`

only. So, his/her first job is to implement these four functions that perform the stack operation. These four functions will be accessing the one dimensional array, and modify/query it. It is a trivial fact that at each `push()` and `pop()` operation the contents of the whole array is **not** moved (practically). Instead of this, in a global variable, the index of the last-put (topmost) element in the array is kept and at every `push()` and `pop()` operation this variable is modified in order to represent the last situation.

On the next page examples of some possible actions on a stack are tabulated *(assuming C as the implementation language)*

INSTRUCTION	FINAL STATE OF THE STACK	RESULT OF THE ACTION
`push(3);`		The value `3` is placed on the stack.
`push(-7);`		Similarly another value of `-7` is placed on the stack.
`i = pop();`		The last put value, namely `-7` is removed from the stack, and this value is assigned to the variable `i`.
`push(20);`		The value `20` is placed on the stack.
`push(589);`		The value `589` is placed on the stack.
`pop();`		The last pushed value, `589`, is removed from the stack. since the returned value is not used it is lost.
`j = stacktop();`		The topmost value, namely `20` is returned and hence assigned to `j`, unlike `pop()`, this value is not removed from the stack.
`j = empty();`		`j` receives a value of `0` indicating that the stack is not empty.
`pop();pop();`		Two top values of the stack are disposed. Since they are not saved, the values are lost.
`pop();`	?	ERROR. The stack was empty!

Problem

In this homework you are going to implement a stack and using it, following some rules given below, you will be pushing and popping elements from this stack. The whole task is to output the single number that is at the top of the stack after that procedure is carried out.

1. You will be reading natural numbers from the standard input (keyboard), one by one till an End-of-File (control-D) is inputted.

2. Any number such arrived is *pushed* onto the stack.

3. The top of the stack is examined:

   - If it is not a prime number nothing is done, with step (1) the process continues.
   - If it is a square number *(i.e. is a square of another integer)* then it is replaced by its square root. The process continues from step (3). *(See below for restrictions)*
   - If it is a prime number then a (modulo 3) operation on the number is performed:
     - If the outcome is 1 then the prime number is discarded from stack *(popped)*, then the following two numbers that are on the stack are *popped* and multiplied, the result is pushed onto the stack. Process continues from step (3).
     - If the outcome is 2 then the prime number is discarded from stack *(popped)*, then the following two numbers that are on the stack are *popped* and added, the result is pushed onto the stack. Process continues from step (3).
     - If the outcome is 0 then the same procedure as above is carried out, provided that the two numbers are subtracted *(The smaller one is subtracted from the bigger one, so the result is still a natural number).*

   (See restrictions below)

4. When there is no more input (EOF case), print the top of the stack.

Restrictions

- You are not allowed to use mathematical library functions, like those in `math.h`. Also you are not allowed to implement and use 'Taylor Expansion' (series expansion) of functions. i.e. taking square root shall be done by implementing the high-school-method.

- All numbers coming from the input will be ≥ 0. You don't have to perform an additional check on it.

- A *pop* with an empty stack is an error. Your program shall check this always. If at any stage the procedure requires this illegal operation it shall abort with exactly the following output:

```
error
```

- The input/output specifications are extremely tight. The input is: natural numbers entered one-per-line, and the input is terminated by an *Control-D*. The output is: A single natural number or the above defined error message. *No blank lines, no additional information strings, NOTHING!*

- Use `long int` type for the variables that will hold natural numbers (and also for the type of the stack elements!).

  For your information: To read/print a `long int` type value the format string used in `printf/scanf` shall contain a corresponding `%ld` .

Example

Consider the input of:

```
2
102
150
27
81
625
49
Control-D
```

The expected output is a single line

```
30
```

For your convenience we include the picture of the stack after each internal change (spying at it at step (3)) during the calculation process. **(You are not going to output this information)**

⇓ *Stack bottom*
2
2 102
2 102 150
2 102 150 27
2 102 150 27 81
2 102 150 27 9
2 102 150 27 3
2 102 150 27 0
2 102 123
2 102 123 625
2 102 123 25
2 102 123 5
2 102 123 2
2 125
2 15
2 15 49
2 15 7
2 15 1
30

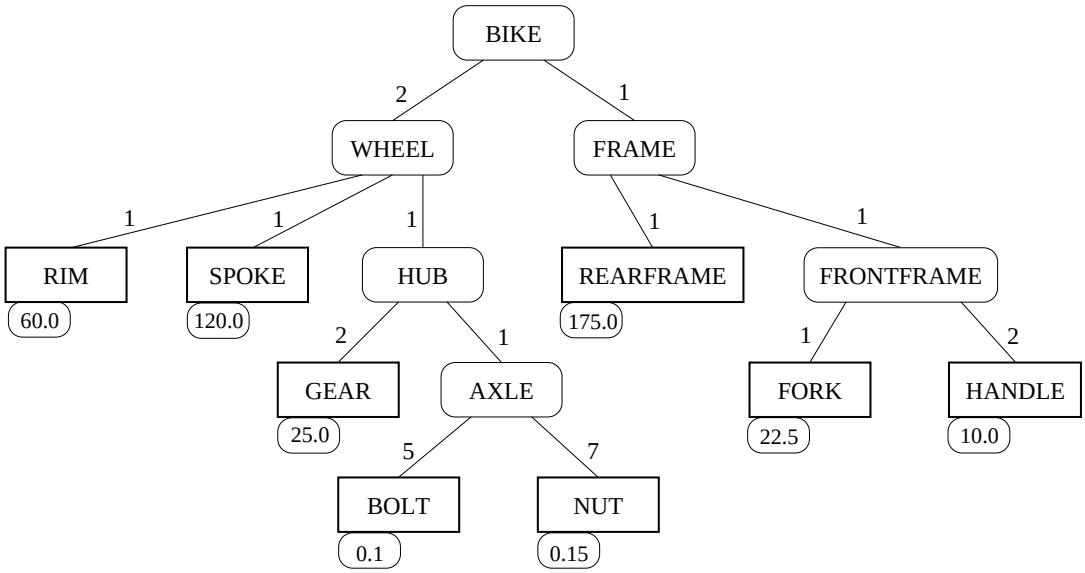
3 PARTS INVENTORY

'95 HOMEWORK 3

Problem

In this homework the purpose is to construct a parts inventory.

Suppose we work in a bicycle factory, where it is necessary to keep an inventory of bicycle parts. If we want to build a bicycle, we need to know what is the total cost of the parts. Each part of the bicycle may have sub-parts, for example each wheel has some spokes, a rim and a hub. Furthermore, the hub can consist of an axle and some gears. Let us consider a tree-structured database that will keep the information of which parts are required to build a bicycle (or any other composite object). There are two kinds of parts that we use to build our bicycle (or generally any such composite object). These are assemblies and basic parts. Each assembly consists of a quantity of basic parts and (may be) a quantity of other assemblies. Since it is possible to calculate the price of any composite part, only the unit price for the basic parts are provided. Below you see such a tree: In this case, for example, a gear has a unit price of 25.0 and you will need 2 such



gears to make a hub.

Your program will input the structure of the composition and the prices of the basic parts through a dialog with the user, and then output the price of the constructed object (in the example the total cost to built a bike!). Here is the exact dialog between the user and your program:

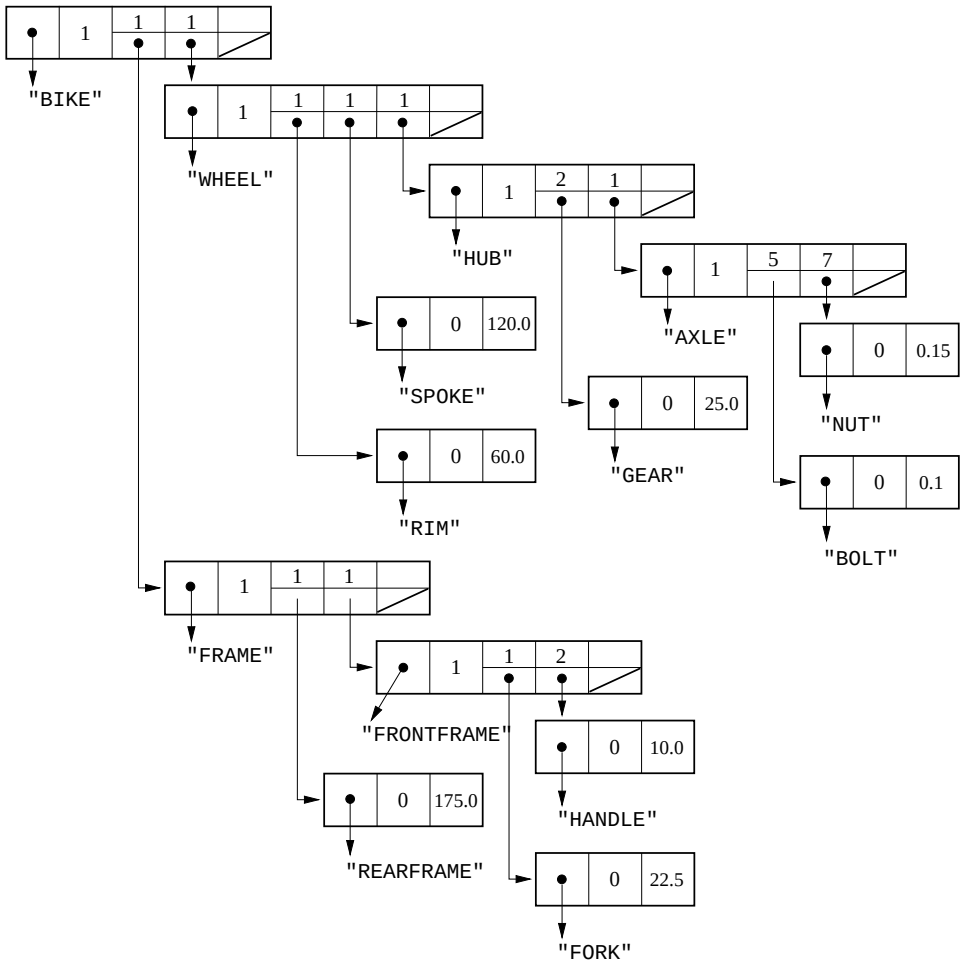
```
Computer: what will you define?
    User: bike
Computer: what is bike?
    User: 2*wheel+1*frame
Computer: what is wheel?
    User: 1*rim+1*spoke+1*hub
Computer: what is rim?
    User: 60.
Computer: what is spoke?
    User: 120.
Computer: what is hub?
    User: 2*gear+1*axle
Computer: what is gear?
    User: 25.
Computer: what is axle?
    User: 5*bolt+7*nut
Computer: what is bolt?
    User: 0.1
Computer: what is nut?
    User: 0.15
Computer: what is frame?
    User: 1*rearframe+1*frontframe
Computer: what is rearframe?
    User: 175.
Computer: what is frontframe?
    User: 1*fork+2*handle
Computer: what is fork?
    User: 22.5
Computer: what is handle?
    User: 10.
Computer: total cost for bike is 780.60
```

The data structure that your program shall construct on-the-fly is given on the following page. As you see, no constant sized memory structures are used at all. The only assumption that you are allowed to make is that no numerical overflow will occur neither at the input nor in the computations and no description input of a composite object will extend over 80 characters (and will be given in a single line).

Hints and Instructions

- Your program shall ignore spaces, provided that they do not occur in numbers, and the name strings. That means it is valid to input one of the lines above as : *(below (␣) is representing a space)*

```
␣␣␣1*␣␣␣␣␣␣rim␣␣␣␣+1␣␣␣␣␣␣␣*␣␣␣spoke␣+1*␣hub␣␣␣␣
```



- You shall make extensive use of the functions defined in `string.h`. Especially: the `str*()` functions and the `*to*()` functions.
- Get each line of user input with `gets()` function, and then process it.
- DO NOT even think of storing the input lines first and then process them all together. This is not possible. The program must intelligently ask questions at each step.
- Use RECURSION.
- For the node representation: Either use unions or define two different structures and perform your own *casting*. If you are not very confident prefer unions.
- Test your program for several cases, check the results manually.
- Since your programs are graded using some automated i/o DO NOT beatify your messages, and the form of the output. Do it exactly as it is shown in the example.

4 SHORTEST PATH

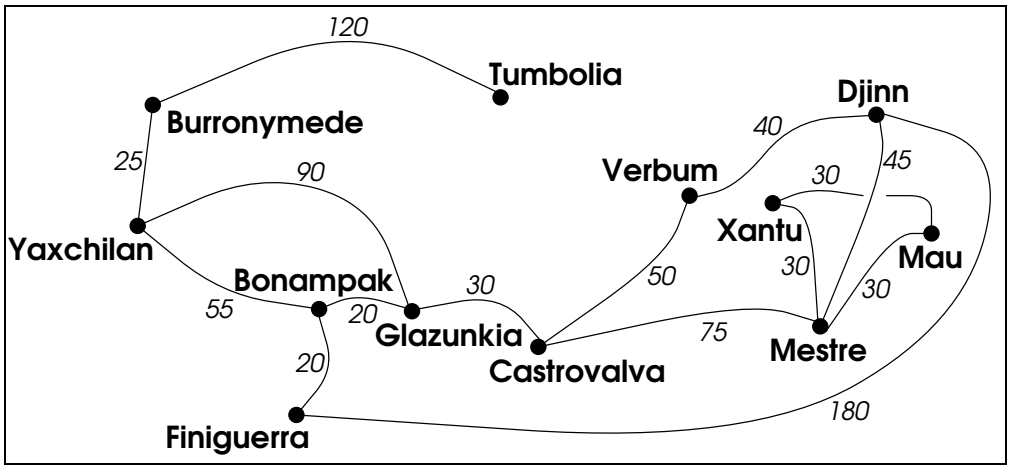
'95 HOMEWORK 4

Problem

In this homework the purpose is to find the shortest path in a given road system between two cities.

Cities are connected to other cities through roads of various lengths. Roads can be used in either direction. As you would expect, a frequently occurring case is that between two cities there might not be a road. If that is the case the way to travel form one city to the other will be via other cities. The input is guaranteed to be such that for any two cities there exists at least one path that connects them. Also it could be the case that there exists more than one such paths. It could also happen that between two cities there is a road (but not more than one) and other paths (via other cities) that connect them.

Below you see an example of such a road map.



With this map to hand the answer of the question:

> *What is the shortest path between **Finiguerra** and **Djinn** and how long is it?*

is:

> **Finiguerra–Bonampak–Glazunkia–Verbum–Djinn**
> *Total path length: 160*

The solutions of this type of problems are well known in computer science and are classified as "Shortest Path Algorithms".

Specifications

- Your program shall read the map information from a file which name will be provided as the first comment line argument to the program. No existence check is required. This map information which will be provided in the below described format.

- The file constitutes of lines each of which corresponds to a road between two cities. There is no predefined order of the lines, in the file. A line is made of 3 entries which are separated by at least one blank. The first and the second entry are the names of the cities which are connected by a road. The third (last) entry in the line is an natural number corresponding to the length of that road. A name of a city is a single word which is made of letters *(Hence* `New Mexico` *is not a valid city name in this homework)*. The naming is case sensitive, upper cases are distinct from lower cases.

  Here is a file which would be one of the descriptions of the above given map.

```
Burronymede Tumbolia 120
Yaxchilan Burronymede 25
Yaxchilan Glazunkia 90
Yaxchilan Bonampak 55
Bonampak Finiguerra  20
Bonampak   Glazunkia 20
Finiguerra Djinn 180
Castrovalva Mestre 75
Castrovalva Verbub 50
Mestre Xantu 30
Mestre Mau 30
Mau     Xantu 30
Mestre Djinn 45
Djinn Verbum 40
Glazunkia Castrovalva 30
```

  The file (as shown) will not include any information of how many lines it includes.

- The program will input the name of the two cities which are the end points of the searched shortest path through a dialogue:

  Computer: `Enter the cities:`
  User: `Finiguerra  Djinn`

  Again the names will be input by seperating them by at least one blank.

- If there exists a shortest path, the output will consist of two lines:

  First line: Length of the shortest path.

Second line: The name of the cities passed on the shortest path, seperated with a single blank. This sequence shall start with the first name which the user input, and end with the second name the user input. If there exist more than one shortest paths then any one of them shall be output.

For the above given example the expected output is:

```
160
Finiguerra Bonampak Glazunkia Verbum Djinn
```

If it was found that there exist no path then the output shall consist of a single line which is exactly the following *(starting from the first character)*:

```
***NO PATH***
```

- You can make the assumption that the file and the user input is error-free, so there is no need to perform error checks.
- **You are forced to use dynamic memory allocation for any array in your program**. So, the use of static or auto array declarations is forbidden.

Evaluation

- A program submitted after the due date/time will be graded **zero**.
- A program that uses compile-time defined arrays will be graded **zero**.
- A working program (in the terms defined above) will receive 40 points, otherwise it will be graded **zero**. Here the term "working" means to run at least the same speed of a program which will be made execute-accessible to you. You will also be provided with some 'large-sized' test data. *(Check tin for further announcements)*
- Your program will receive an additive point P depending on its execution time, calculated as follows:

$$P = 60 \times \frac{Time_{Longest} - Time_{Your\ \ program}}{Time_{Longest} - Time_{Shortest}}$$

Here $Time_{Shortest}$ is the execution time of the best performing submission. $Time_{Longest}$ is the execution time of the worst but 'working' (in the above defined terms) submission.

- All submissions will be compiled and run under strictly equal conditions.

5 Sky Liner

'96 HOMEWORK 1

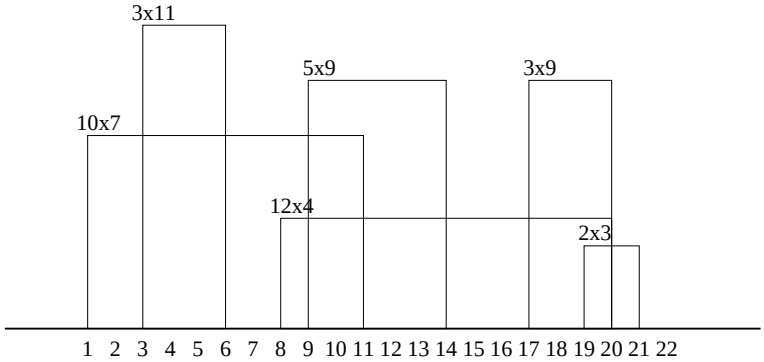
Problem

This problem is known as the *Sky Liner Problem*. It can be summarized as follows: Assume you are looking at a city from a very long distance, the sun is behind the city (from your point of view). You will not be able to recognize individual buildings. The whole city will be observed as a silluette. The problem is:

> Assume you are given the dimensions of the shadow of each building which is a rectangular. Furthermore each of these rectangulars are placed on a common line (the ground) and extent in the same direction (upwards). In addition to the dimensions you are also given the positions of each rectangular. Your duty, if you accept, is to find the vertices of the broken line that surrounds the silluette.

Example

Consider the below given example



As given in the picture above, there are 6 rectangulars. which could be listed as

WIDTH	HEIGHT	POSITION ON GROUND
3	11	3
10	7	1
5	9	9
12	4	8
3	9	17
2	3	19

'Position on Ground' is the position of the lower-left corner of the rectangular. We will represent the broken line that surrounds the silluete by a ordered set of vertices where the leftmost vertice on the broken line is stated at the leftmost position of the set, then

the one which is next on the broken line is following it, etc. A vertex inforamtion will be represented by a 2-tuple:

$$(Position\_on\_Ground, Height)$$

So for this example the solution would be

$$[(1,0),(1,7),(3,7),(3,11),(6,11),(6,7),(9,7),(9,9),(14,9),$$
$$(14,4),(17,4),(17,9),(20,9),(20,3),(21,3),(21,0)]$$

Specifications

- Your program will read the *Width, Height, Position_on_Ground* information from the *standart input*. You are given that there are at most 2000 rectangles. Each input line will contain 3 integers corresponding to the information of one of the rectangles. *(There will be no other characters on the line except these three numbers which are seperated from each other by blanks only)*. No information (like an integer at the start of the input) of how many actual lines are present in the input will be given! The input will terminate with the END-OF-FILE character (*ctrl-D* on the UNIX system).

  A possible input for the example above is:

```
3   9 17
5   9  9
12  4  8
3  11  3
10  7  1
2   3 19
```

  cntr-D

- There is no order is imposed on the input lines. This means that a rectangle's information can be given at any moment of the input process.

- The output is made to the *standart output*. The output will contain the vertex information of the solution in the order specified in the previous subsection. You don't have to output each vertex information on a new line. It is allowed to put them all in a single line separated by at least one blank. Though, not doing so, (i.e. outputting each vertex information on a new line), is also ok. A vertex infomation on the output is two integers separeted by at least one blank. The first integer is the *Position_on_Ground* and the second is the *Height* of the vertex.

  The expected output for the above given example is:

```
1 0 1 7 3 7 3 11 6 11 6 7 9 7 9 9 14 9 14 4 17 4 17 9 20 9 20 3 21 3 21 0
```

6 PROPOSITIONAL LOGIC FORMULAS

Problem

This time we will be dealing with so called **Well-Formed Propositional Logic Formulas**. From now on we will abbreviate it by "WFPLF". A WFPLF is defined as follows:

- Lowercase letters like a,b,c,... are among the constituents of WFPLFs. These letters, individually, are counted as WFPLF too.
- Except the individual lowercase letters, a WFPLF can only contain the characters '(', ')', '-', '&', '|', '>', '=' provided they are used properly.
- If □ and △ are two WFPLF then so are

```
-□       (negation)
(□&△)    (conjunction)
(□|△)    (disjunction)
(□>△)    (implication)
(□=△)    (equivalence)
```

An *instance* of a WFPLF is defined to be a state where all letters (variables) used in that formula have a value. The letter can admit the values T and F, semanticly standing for *Truthness* and *Falsity*. The values associated with the compound WFPLF formulas that involve the operators '-', '&', '|', '>', '=' are defined by the so called *Truth tables* (the table where all instances are covered).

□	△	-□	(□&△)	(□|△)	(□>△)	(□=△)
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

A program that you will write will take any WFPLF from the standard input and produce the truth table.

Specifications

- The length of the formulas are not restricted. That means you shall not make any restrictive assumption about a maximal length. Of course there will be *natural* restrictions imposed by the compiler and hardware *(like pointer size, integer size, memory size, etc.)*. But you shall not impose additional restrictions.
- Though the alphabet is limited to the English lowercase characters it is possible to have formulas in which a letter occurs at different positions.
- All blanks and end-of-lines are to be neglected.
- Parenthesis are not optional nor neutral. That means closing a WFPLF in an parenthesis is forbidden; negation does not have a parenthesis of its argument (doing so is wrong).
- Since, at the moment you receive this worksheet the `struct` and `union` declarations are not introduced, don't build tree structures and do not use struct/union types in your program.
- You will keep the formula as a string, and for each instance reevaluate it. Do not modify the string from instance to instance.
- Make use of *recursion* in your evaluator.
- Your first line of your output shall contain the used characters in the lexicographical order (i.e. form `a` to `z`), separated by exactly one blank. And the upper case letter `R` as the last character in the line (which stands for 'result'). Do not use any other character than `R`.

  The following lines will contain the truth values represented by the letters `T` and `F` (Do NOT use anything else like, `t`,`f`, `True`, `TRUE`, `1`, `0`, etc.)

  You are expected to start with the all variables `T` case as the first line of the truth table and then continue by changing the right-most fastest and the left-most slowest.
- You can assume that all tests and runs are error-free and complies with the above given syntax. Therefore do not perform any error check on the input.
- **Hints:** You may think of using the functions `realloc()`, `getchar()`, `islower()`, `isspace()`.

Example

```
-(-(a&  k) > (  -((a|-k)
          | c   )|           d))
```

is a possible input. The expected output is:

```
a c d k R
T T T T F
T T T F F
T T F T F
T T F F T
T F T T F
T F T F F
T F F T F
T F F F T
F T T T F
F T T F F
F T F T T
F T F F T
F F T T F
F F T F F
F F F T F
F F F F T
```

7 Drawing N-ary Trees

'96 HOMEWORK 3

Introduction

A *tree* is a nonempty collection of *vertices* and *edges* that satisfies certain, requirements. A vertex is a simple object (also referred to as a *node*) that can have a name and can carry other associated information; an edge is a connection between two vertices. A path in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree. One node in the tree is designated as the *root* — the defining property of a tree is that there is exactly one path between the root and each of the other nodes in the tree. If there is more than one path between the root and some node, or if there is no path between the root and some node, then what we have is a *graph*, not a tree.

Though the definition implies no "direction" on the edges, we normally think of the edges as all pointing away from the root or towards the root depending upon the application. We usually draw trees with the root at the top (even though this seems unnatural at first), and we speak of node y as being *below* node x (and x as *above* y) if x is on the path from y to the root (that is, if y is below x as drawn on the page and is connected to x by a path that does not pass through the root). Each node (except the root) has exactly one node above it which is called its *parent*; the nodes directly below a node are called its *children*. We sometimes carry the analogy to family trees further and refer to the "grandparent" or the "sibling" of a node. Nodes with no children are sometimes called *leaves*, or *terminal* nodes. To correspond to the latter usage, nodes with at least one child are sometimes called *nonterminal nodes*. Terminal nodes are often different from nonterminal nodes: for example, they may have no name or associated information. Especially in such situations, we refer to nonterminal nodes as *internal* nodes and terminal nodes as *external* nodes. Any node is the root of a *subtree* consisting of it and the nodes below it.

Sometimes the way in which the children of each node are ordered is significant, sometimes it is not. An *ordered* tree is one in which the order of the children at every node is specified. Of course, the children are placed in some order when we draw a tree, and clearly there are many different ways to draw trees that are not ordered. As we will see below, this becomes significant when we consider representing trees in a computer, since there is much less flexibility in how to represent ordered trees. It is normally obvious from the application which type of tree is called for.

The nodes in a tree divide themselves into *levels*: the level of a node is the number of nodes on the path from it to the root (not including itself). The *height* of a tree is the maximum level among all nodes in the tree (or the maximum distance to the root from any node). The *path length* of a tree is the sum of the levels of all the nodes in the tree (or the sum of the lengths of the paths from each node to the root). If internal nodes are distinguished from external nodes, we speak of *internal path length* and *external path* length.

Trees are intimately connected with recursion. In fact, perhaps the simplest way to

define trees is recursively, as follows:

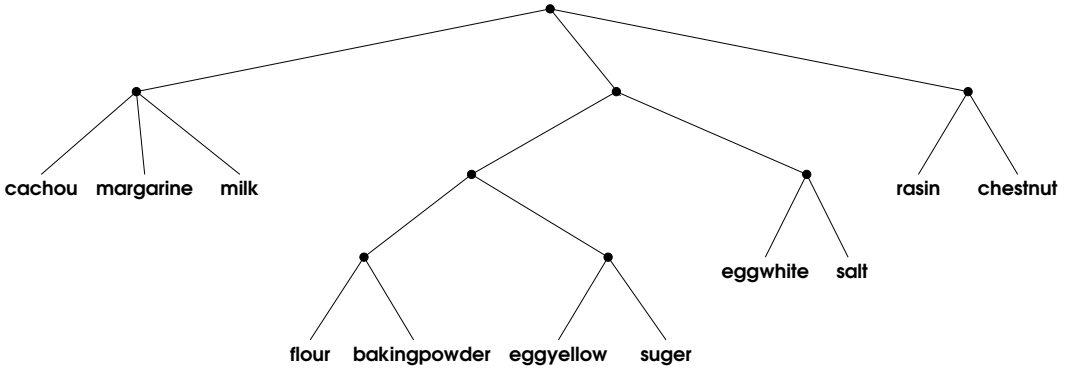
> "a tree is either a single node or a root node connected to a set of trees".

Problem

We will be dealing with ordered trees in which the information is stored in the *external* nodes. For our problem the following specification holds:

- The root is not an external node.
- At any internal node the count of branching is in the range $[2..30]$.
- Each external node holds an information which is an alphanumeric string with length in the range $[2..20]$

The input representation of a tree is a one dimensional expression (a string). In this expression the alphanumeric strings are represented as themself; children of the same parent are grouped into a pair of parenthesis. Here follows an example, consider a subject tree:



the input representation that corresponds to the tree above is

> ((cachou margarine milk) (((flour bakingpowder) (eggyellow sugar)) (eggwhite salt)) (rasin chestnut))

The program that you will write shall draw a tree given in the above representation. The expected appearance is the same of the drawing on the previous page.

Restrictions and How-to-Do's

- The input will be consumed by `getchar()` calls. No storing of the input string as a whole (for a later processing purpose) is allowed.
- The input might be scattered over several lines and may contain white-spaces. All consecutive white-spaces are equivalent to a single blank.

- You are expected to construct a data structure by means of dynamic memory allocation and pointer usage. You shall think of a data structure where it should be possible to store either an external node or an internal one (you may think of making use of `union`). The edges of the tree shall be represented by pointer links.

- A parent can have at most 30 children, not more. But you are not allowed to use this information for allocating memory which you may not use. In other words you are not allowed to make allocation for 30 children where only 4 are present and leave 26 children position unused.

- External nodes are represented by alphanumeric strings (eg. bakingpowder). The maximal length for them is 20. The same rule of the previous item applies for this case as well. You shall not allocate more space for them then actually needed.

- All the structure member field accesses shall be performed by making use of macros (in other words you are not allowed to explicitly use the `'.'` (dot) operator in any function).

- You shall construct the structure on-the-fly, that means while you are reading the input you shall be constructing the internal data structure.

- In the drawing, equal level nodes shall be drawn at the same depth. The vertical distance between two consecutive levels is 30 units.

- Horizontally consecutive external nodes (which are represented by the alphanumerics) shall be equally spaced. (imagine that all the leaves fall from the tree exactly vertical onto a horizontal line and you have the alphanumeric aligned on the same line, now the spaces between two consecutive alphanumeric strings shall be the same). So, for example, the horizontal distance between the milk and flour shall be the same as of the distance between rasin and chestnut (by distance we mean the *distance from end to start*).

- Any parent node shall be horizontally centered between its leftmost child and rightmost child. If such a child is an external node then the reference point of that child used for the centering is the mid of the alphanumeric string of that external node.

- You shall mark the internal node by a filled circle of a radius of 2 units. A practical way to do this is to draw two co-centric circles one of radius 1 the other of radius 2.

- You are expected to use *recursion* in parsing and the drawing. Do not try to use (or even think of) other methods.

- After the construction of the internal data structure you are allowed to run through it at most 2 times.

- You can assume that the input is error free. That means all parenthesis opened are closed; characters in the input stream are either white-spaces, alphanumerics or opening or closing parenthesis (nothing else).

- You are provided with some facilities for drawing in an X-windows environment, these are:

  Some header files: Your top lines of your `hw3.c` file shall contain

  ```
  #include <xview/xview.h>
  #include <xview/canvas.h>
  #include <xview/cms.h>
  #include <xview/xv_xrect.h>
  #include "/shared/courses/ceng-241/hw3.h";
  ```

  An X-window: You will have a graphical x-window poping up when you run your program. **you don't have to do any additional work to create the window**. The window size is fixed to be 1200 units in horizontal, 800 units in vertical direction. The *origin* is at the top-left corner of the window. From now on we will refer to the distance of a point in the window, measured in the horizontal left-to-right direction and starting from the origin as the *x-coordinate*; and similarly, in the vertical top-to-bottom direction (starting form the origin) as the *y-coordinate*.

  Functions for drawing: Through the included `hw3.h` file you have (automatically) three functions available:

  `DrawLine(int x1,int y1,int x2,int y2)` As the name suggests, this draws a straight line from the coordinates $(\texttt{x1}, \texttt{y1})$ to $(\texttt{x2}, \texttt{y2})$

  `DrawCircle(int x,int y,int r)` Draws a circle which has its center at $(\texttt{x}, \texttt{y})$ and is of a radius `r` units.

  `DrawString(int x,int y,char *text)` At $(\texttt{x}, \texttt{y})$ places the string which is pointed by **`t`**ext. The start of the string is at the given coordinate. Each character position is about 6 units wide.

  A `main` function: The actual `main()` function is defined in the header file `hw3.h` . At the start of this `main()` function some initializations are performed (you don't have to bother what these are). Then a function which is named as `student_main()` is called. Therefore in this homework

  - **DO NOT** define a function with name `main()`.
  - **DEFINE** a function with name `student_main()` to do whatever you would normally do in your `main()` function definition.

  A `make` file: You are provided with a make file

  `/shared/courses/ceng-241/Makefile`

  You shall copy this file to your work directory (do not rename it). Then a simple command as

```
make
```

will perform the compilation and linking with the correct libraries of your `hw3.c` file. The executable will have the name `hw3` and will be placed in your working directory. (*Note that the make file does not take an argument like `hw3.c`*).

- Your program, as usual, shall read form standard input.
- The root shall be positioned *allways* at $(600, 30)$ (see below for bonus).

Bonus

Some trees, very naturally, may extend beyond the left or right border. Provided that you comply with the specifications above, nothing is wrong with this. Your X environment is so that a clipping occurs and those portions of the tree will not be seen. You may think of adjusting this if the tree is not that much grown to the other direction by displacing the root. If you solve this problem and (only for those problems) find a better position for the root you get a %20 bonus. This bonus is absolute, that means, if you got a 100 of the compulsory part and also received a 20 points, the hw3 contribution to the overall course grade will be calculated using 120 and will naturally be higher then the contribution of a 100.

8 Minimal Spanning Tree

'96 HOMEWORK 4

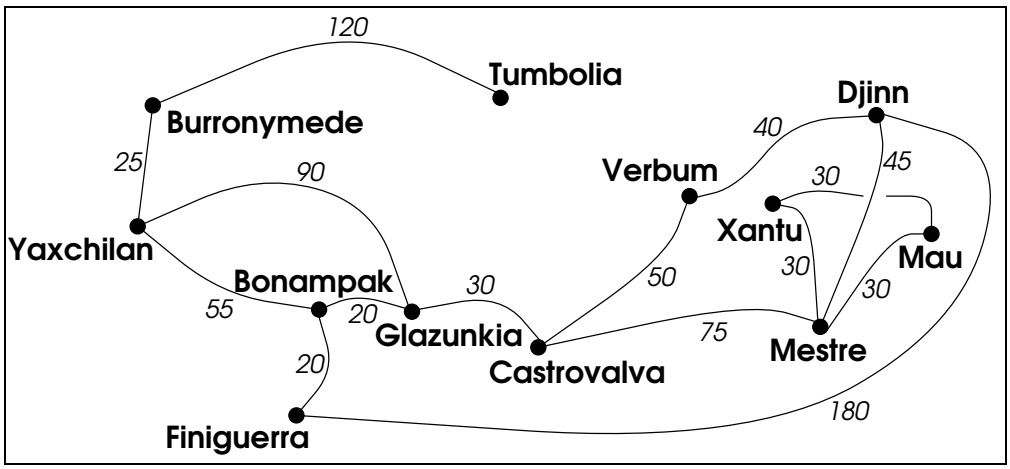
Problem

Cities are connected to other cities through roads of various lengths. Roads are used in either direction. As you would expect, a frequently occurring case is that between two cities there might not be a road. If that is the case the way to travel form one city to the other will be via other cities. The input is guaranteed to be such that for any two cities there exists at least one path that connects them. Also it could be the case that there exists more than one such paths. It could also happen that between two cities there is a road (but not more than one) and other paths (via other cities) that connect them.

The state is in a heavy economical crises. So the government decides to cut down the road maintenance expenses. This means that some unmaintained roads may deteriorate and cease function. But some roads will be kept maintained such that the connectivity remains. That means any city shall be reachable from any other city. Of course some clever goverment official immediately discovers that there are more then one alternatives in choosing the roads that will be kept maintained and the best choice is the one that minimises the expense. The maintenence expense for a road is linearly proportional to its length. It is unnecessary to mention that the increase in some of the distances to reach a city from another one is not to the slightest concern of the government (it is the inviduals who will suffer and not the government).

Your duty (if you accept of course) is to find the roads which will kept maintained.

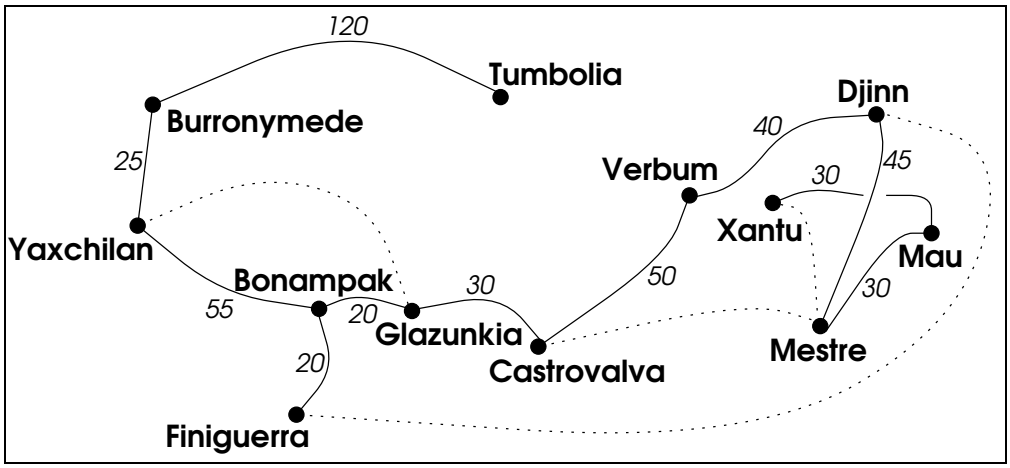
Below you see an example of a road map (look at it: it will destroy itself in 5 secs).



With this map to hand the answer of the question:

> *Which roads shall be kept maintained so that the maintenance cost is minimum and still all cities are connected?*

Here is a solution:



Total maintained road length: 465

Specifications

- Your program shall read the map information from a file which name will be provided as the first comment line argument to the program. No existence check is required. This map information which will be provided in the below described format.

- The file constitutes of lines each of which corresponds to a road between two cities. There is no predefined order of the lines, in the file. A line is made of 4 entries which are separated by at least one blank. The first is the national road number (you will use it in your output) this number can be represented as a `long int`. The second and the third entry are the names of the cities which are connected by a road. The fourth (last) entry in the line is a natural number corresponding to the length of that road (can be represented by an `int`). A name of a city is a single word which is made of (at most 30) letters *(Hence* `New Mexico` *is not a valid city name in this homework)*. The naming is case sensitive, upper cases are distinct from lower cases.

Here is a file which would be one of the descriptions of the above given map.

```
101 Burronymede Tumbolia 120
201 Yaxchilan Burronymede 25
143 Yaxchilan Glazunkia 90
546 Yaxchilan Bonampak 55
12 Bonampak Finiguerra  20
65 Bonampak   Glazunkia 20
34 Finiguerra Djinn 180
33 Castrovalva Mestre 75
4232 Castrovalva Verbub 50
67 Mestre Xantu 30
102 Mestre Mau 30
1343 Mau     Xantu 30
15 Mestre Djinn 45
79 Djinn Verbum 40
90 Glazunkia Castrovalva 30
```

The file (as shown) will not include any information of how many lines it includes.

- The output will start with a single line that will contain the total length of roads that have to be maintained. Each of the following lines will contain a single national road number of a road that shall be maintained (one number per line). This list shall be sorted in ascending order.

  For the above given solution example the expected output is:

  ```
  465
  12
  15
  65
  79
  90
  101
  102
  201
  546
  1343
  4232
  ```

  If there is more then one alternative for the minimal expense then you are free to choose any scheme.

- You can make the assumption that the file is error-free, so there is no need to perform error checks.

- **You are forced to use dynamic memory allocation for any array or structure in your program**. So, the use of static or auto array declarations is forbidden.

Evaluation

- A program submitted after the due date/time will be graded **zero**.
- A program that uses compile-time defined arrays will be graded **zero**.
- A working program (in the terms defined above) will receive 40 points, otherwise it will be graded **zero**. Here the term "working" means to run at least the same speed of a program which will be made execute-accessible to you. You will also be provided with some 'large-sized' test data. *(Check tin for further announcements)*
- Your program will receive an additive point P depending on its execution time, calculated as follows:

$$P = 60 \times \frac{Time_{Longest} - Time_{Your\ \ program}}{Time_{Longest} - Time_{Shortest}}$$

  Here $Time_{Shortest}$ is the execution time of the best performing submission. $Time_{Longest}$ is the execution time of the worst but 'working' (in the above defined terms) submission.
- All submissions will be compiled and run under strictly equal conditions.

9 TURKISH HYPHENATION

'97 HOMEWORK 1

Problem

To start the series of Ceng. 241 homeworks in a gentle manner it has been decided to have a simple one for the first.

The problem is to hyphenate any Turkish word. So here are some examples of what is expected:

Input	*Output*
ak	ak
kal	kal
kalas	ka-las
fikriye	fik-ri-ye
saat	sa-at
kontrbas	kontr-bas
turkce	turk-ce
belirtim	be-lir-tim
belirtmek	be-lirt-mek
belirttirmek	be-lirt-tir-mek
avusturalyalilastiramadiklarimizdanmiymis	a-vus-tu-ral-ya-li-las-ti-ra-ma-dik-la-ri-miz-dan-miy-mis

You don't have to worry about the Turkish special characters, their presence will not change the hyphenation algorithm, since the whole algorithm is based on the order of vowel/consonant properties of the constituent letters of the Turkish word.

Note: The hyphenation algorithm is extremely simple and the coding of it does not exceed 10 lines of C code.

Specifications

Your program will read words each placed on a single line. A Turkish word will not exceed 50 characters. The input is white-space (blanks,tabs) free. After the last word there shall be **no** expectation of a special character/marker. The program shall terminate when the EOF character *(control-D)* is entered. After each line of input the program shall output on a new line the hyphenated word (by placing `'-'` characters where it is appropriate) and continue with the next line of input. Do not use **any** gizmos in your program. That means **do not print** any header like:

```
Welcome to my hw1 program
```

or trailers to the input like:

```
Enter word:
```

nor print trailers for the output:

```
Hyphenation result:
```

or some similar stuff. Your program will be tested automatically (by another program). If you print a single line/space/character more then expected that program may malfunction.

10 SEXPR TOKENIZER

'95 HOMEWORK 1

Introduction

As you are familiar from your Ceng 111 course, Scheme (more generally LISP) was operating in a domain that we call *sexpr*s. An sexpr is one of the followings:

- Number
- Atom
- String
- List[1].

Here we give the formal definition of the syntax for each of these sexpr in the (BNF) notation.

Number

There are two kinds of numbers: integers and floating points. Though the limitation did not exist in Scheme, for sake of simplicity we restrict the sizes so that the integers can 'internally' be represented by `long int` (in the C language) and the floating points by `double`. The BNF definition of 'number' is as follows:

$$
\begin{aligned}
\langle \text{number} \rangle &\longrightarrow \langle \text{sign} \rangle \langle \text{unsigned} \rangle \\
\langle \text{unsigned} \rangle &\longrightarrow \langle \text{unsigned float} \rangle \mid \langle \text{unsigned integer} \rangle \\
\langle \text{unsigned float} \rangle &\longrightarrow \langle \text{unsigned integer} \rangle \texttt{.} \langle \text{exponent} \rangle \\
&\quad \mid \langle \text{unsigned integer} \rangle \texttt{.} \langle \text{unsigned integer} \rangle \langle \text{exponent} \rangle \\
&\quad \texttt{.} \langle \text{unsigned integer} \rangle \langle \text{exponent} \rangle \\
\langle \text{exponent} \rangle &\longrightarrow \langle \text{letter-E} \rangle \langle \text{sign} \rangle \langle \text{unsigned integer} \rangle \mid \varepsilon \\
\langle \text{unsigned integer} \rangle &\longrightarrow \langle \text{digit} \rangle \langle \text{more digits} \rangle \\
\langle \text{more digits} \rangle &\longrightarrow \langle \text{digit} \rangle \langle \text{more digits} \rangle \mid \varepsilon \\
\langle \text{letter-E} \rangle &\longrightarrow \texttt{E} \mid \texttt{e} \\
\langle \text{sign} \rangle &\longrightarrow \texttt{+} \mid \texttt{-} \mid \varepsilon \\
\langle \text{digit} \rangle &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

These are numbers: `27` `4` `-4` `+4` `3.14` `-3.14` `+4` `0.25` `10.25` `00.50` `.25` `25.` `25.0` `-0.25` `-.25` `-.0` `.0` `0.` `-0.` `+0` `+0.` `-12.45e+3` `+012.450e-023` `-.05E01` `-.05E0` `-0.E-0`

These are not numbers: `.` `-.` `x.y` `e-3` `1-3` `1.-3.` `1.25e0.5` `(12)` `"1245"` `"-3.14"`

[1] Actually, speaking in strict LISP terms, this item is a more general one, namely the *Dotted pair*. But, for simplicity, we refrain from this, and replace it by a linear structure, the *list*

Atom

$$\begin{aligned}
\langle \text{atom} \rangle &\longrightarrow \langle \text{non digit character} \rangle \langle \text{more characters} \rangle \\
\langle \text{more characters} \rangle &\longrightarrow \langle \text{atomic character} \rangle \langle \text{more characters} \rangle \mid \varepsilon \\
\langle \text{atomic character} \rangle &\longrightarrow \langle \text{any Ascii printable except } \texttt{") ( ] [. ␣} \rangle \\
\langle \text{non digit character} \rangle &\longrightarrow \langle \text{any atomic character that is not a digit} \rangle
\end{aligned}$$

These are atoms: `x` `x1` `xyz123qrt` `e-2` `-` `-x` `+-` `/12` `AxY` `x2/7` `x+4*y/5`

These are not atoms: `12` `x.` `.` `-.` `a[3]` `(a)` `x.y` `mid␣mid` `"x1"` `"a[3]"` `"2"`

No atom will have more than 50 characters.

String

$$\begin{aligned}
\langle \text{string} \rangle &\longrightarrow \texttt{"} \langle \text{string characters} \rangle \texttt{"} \\
\langle \text{string characters} \rangle &\longrightarrow \langle \text{a string character} \rangle \langle \text{string characters} \rangle \mid \varepsilon \\
\langle \text{a string character} \rangle &\longrightarrow \langle \text{any Ascii printable except } \texttt{"} \rangle
\end{aligned}$$

As you might have realized this is an over simplified definition of *string* (compared to the string convention of the C language) since it does not provide an escape sequence. A string will have at most 300 characters.

These are strings: `"x1"` `"a[3]"` `" "` `"3;;',./&_*)) NY[*]N"` `"-3.14"` `""` `"x.y"`

List

Though in this homework you will **not** recognize *lists*, for completeness we state how they are defined. The following definition will be useful in the next homework.

$$\begin{aligned}
\langle \text{list} \rangle &\longrightarrow \texttt{(} \langle \text{white spaces} \rangle \langle \text{sexpr sequence} \rangle \texttt{)} \\
\langle \text{sexpr sequence} \rangle &\longrightarrow \langle \text{sexpr} \rangle \langle \text{white spaces} \rangle \langle \text{sexpr sequence} \rangle \mid \varepsilon \\
\langle \text{white spaces} \rangle &\longrightarrow \langle \text{white space} \rangle \langle \text{white spaces} \rangle \mid \varepsilon \\
\langle \text{white space} \rangle &\longrightarrow \texttt{␣} \mid \langle \text{end of line} \rangle
\end{aligned}$$

More than one adjacent white spaces are equivalent to a single white space. A white space is only meaningful if its ommitance results in an ambiguity.

The closure: sexpr

Here we give the definition of the term *sexpr*.

$$\langle \text{sexpr} \rangle \longrightarrow \langle \text{number} \rangle \mid \langle \text{atom} \rangle \mid \langle \text{string} \rangle \mid \langle \text{list} \rangle$$

Here is a 'huge' example of an sexpr:

```
(   (family (father (Ahmet OZOKLAV)  ) (mother (Sukriye BATMAZ   ) )
            (children 2 (( munevver 11.5) ( mubeccel 3.5)))
 (  address "Ortaklar Cad. Birlik apt. no: 10/6 Murvetiye/Kertalan")
            (income B-4 1.2e8)
  ))
```

Problem

In this homework you will be writing a *tokenizer* for sexpr's. A tokenizer is a program that takes an element of a language (in our case an sexpr) and returns it basic building blocks in left to right order. For our case the building blocks are:

- integer numbers
- floating point numbers
- atoms
- strings
- left parenthesis
- right parenthesis

For the above given sexpr example the tokenizer would recognize the input as

TOKEN	KIND
(	left parenthesis
(	left parenthesis
family	atom
(	left parenthesis
father	atom
(	left parenthesis
Ahmet	atom
OZOKLAV	atom
)	right parenthesis
)	right parenthesis
(	left parenthesis
mother	atom
(	left parenthesis
Sukriye	atom
BATMAZ	atom
)	right parenthesis
)	right parenthesis
(	left parenthesis
children	atom
2	integer number
(	left parenthesis
(	left parenthesis
munevver	atom
11.5	floating point number
)	right parenthesis
(	left parenthesis
mubeccel	atom
3.5	floating point number
)	right parenthesis
)	right parenthesis
)	right parenthesis
(	left parenthesis
address	atom
"Ortaklar Cad. Birlik apt. no: 10/6 Murvetiye/Kertalan"	string
)	right parenthesis
(	left parenthesis
income	atom
B-4	atom
1.2e8	floating point number
)	right parenthesis
)	right parenthesis
)	right parenthesis

Specification

Your program will take sexpr(s) from the input and until End-of-file is detected will output for each token a triple of information:

[Token] [Token kind] [Process result]

The *Process* is defined as follows:

If token is an integer number: Add 1 (one) to its value.

If token is a floating point number: Halve its value.

If token is an atom: Convert all lower cases to upper cases; all upper cases to lower cases; leave all others unchanged.

If token is a string: Count the spaces in the string.

If the token is a parenthesis: Don't do anything.

So the first 10 lines of the output of your program for the above given example as input would be:

```
[(]  [leftp]  [(]
[(]  [leftp]  [(]
[family]  [atom] [FAMILY]
[(]  [leftp]  [(]
[father]  [atom] [FATHER]
[(]  [leftp]  [(]
[Ahmet]  [atom] [aHMET]
[OZOKLAV]  [atom] [ozoklav]
[)]  [rightp]  [)]
[)]  [rightp]  [)]
```

Lines 20-24 of the output will be:

```
[2]  [integer]  [3]
[(]  [leftp]  [(]
[(]  [leftp]  [(]
[munevver]  [atom] [MUNEVVER]
[11.5]  [float]  [5.750000]
```

The only token which is a string will produce an output line:

```
["Ortaklar Cad. Birlik apt. no: 10/6 Murvetiye/Kertalan"]  [string]  [6]
```

- Do not take any special action when you detect End-of-file, just quit.
- Do not attempt to store all the input. The idea is to read in a line from the input, find the tokens, output them, continue with the next line.
- As it is also clear from the BNF description, it is not possible to *break* numbers, atoms, strings over the end of a line. That means a number, atom or string will resign in a single line in whole. This is not so for a list. A list may have its elements scattered over multiple lines.
- You will be using your tokenizer in the next homework. Therefore write it in a modular style (preferably as a function which at every call produces the next token in line and stores it in some global variables along with a returned type information (a small integer for example)).
- The input will **not** be erroneous. So, you do not have to test for errors.
- *Printable Ascii characters* are those which have theirascii codes in the range [32-127] (decimal).
- Throughout definitions, the ␣ character stands for the Ascii 32 character (space). Also note that, unless it is explicitly stated by the definition, the use of spaces are not allowed. (e.g. if you insert a space into an atom it is no more that atom).

11 SEXPR PARSER

'97 HOMEWORK 3

Introduction

You have written in your previous homework a tokenizer for the primitive data types of the sexpr domain. This time you will be writing a parser for the sexpr domain (which will also include the list type. The purpose is to read in a sexpr and convert it into an internal representation.

Problem

The internal representation will make use of the C concept of *union*s. Most generally an sexpr will be represented internally by a pointer to a `struct` of two fields, namely a *tag* field and another field which is of `union` type. The *tag* field shall be holding the *kind* of the sexpr (ie. atom, integer, float, string, list). As it was explained in the lecture, you are totally free in choosing the tag values. Just as an example, in this sheet we will be using `1`:atom, `2`:integer, `3`:float, `4`:string, `5`:list. It is strongly advisable that you define these through `#define` macros. This will make your code much more readable. The other field of the union will be holding one of the 'internal' representation of the possible sexpr types.

The `union` shall be combining the following 'internal representation' alternatives.

atom: A `struct` consisting of 2 fields which will store:

1. a pointer to the name string of the atom,
2. a pointer to any other sexpr (a pointer to the union!) which is initialized to `NULL`.

We will name these fields as the *printname* and the *value* of the atom, respectively.

There is a special treatment of atoms. Namely, atoms are stored uniquely. That means in a run of your program any atom must be checked for being present at that moment. If there was an atom created with the same name already, you must use the reference (the pointer) to that created. We would strongly advise that you device a dynamic array (growable) in which you store the pointers to the atoms. This will enable you to perform the existence search easily over the atom space. one (and not create a new atom).

integer: A `long int`

float: A `double`

string: A pointer to the content of the string (the double-quotes will not be stored).

list: If the list has n elements then an array of $n + 1$ elements each of which holds a pointer to one of the sexpr elements of the list. ($left \rightarrow right$ order shall be

preserved as $smaller\ index\ value \rightarrow greater\ index\ value$). The last element of the array will be stored with `NULL` serving as a *terminator*.

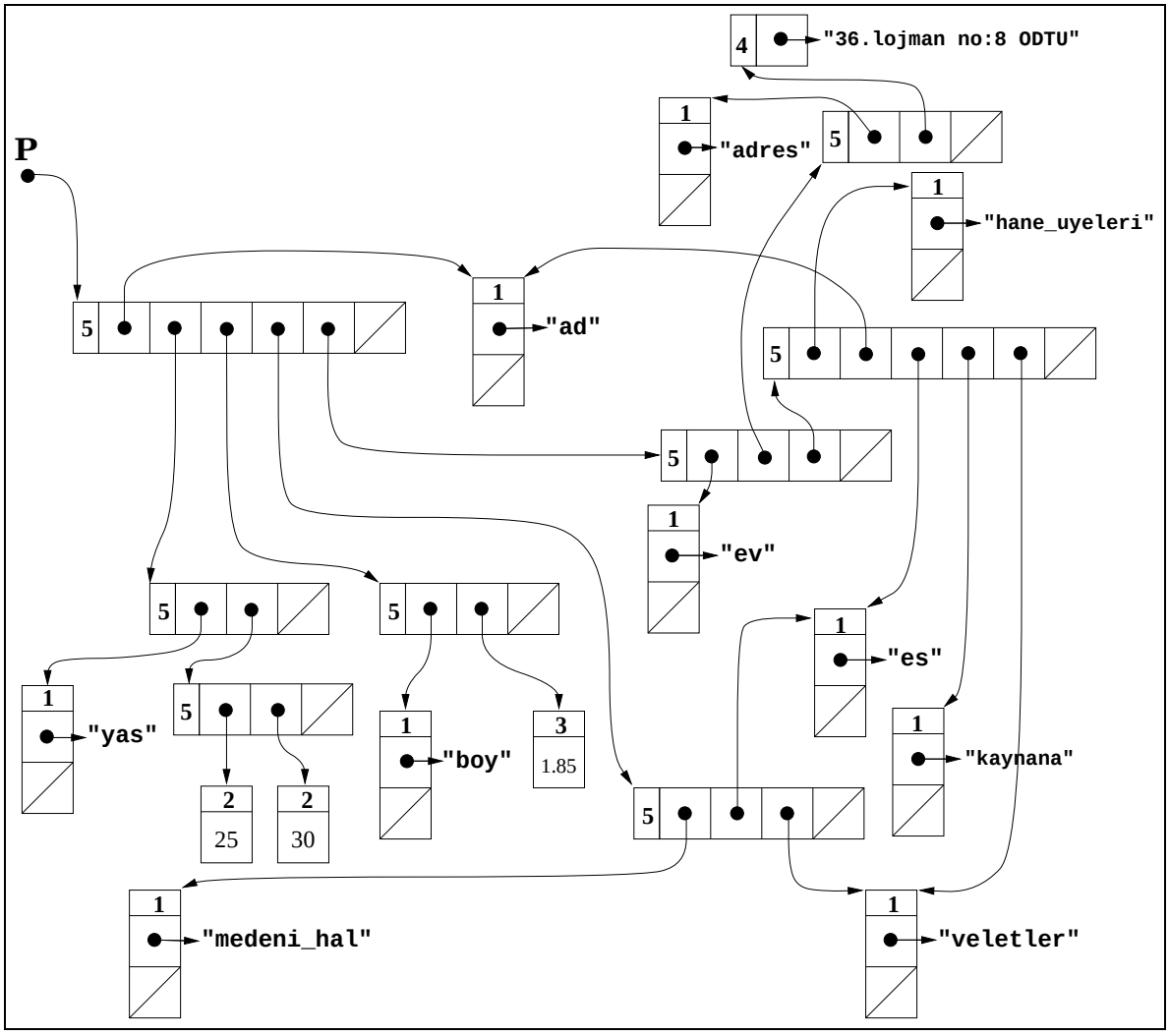
Here is an example:

```
(ad (yas (25 30)) (boy 1.85)
    (medeni_hal es veletler)
    (ev (adres "36.lojman no:8 ODTU")
        (hane_uyeleri ad es kaynana veletler)
))
```

The next page contains a 2-dim schematic of the internal representation for this sexpr input. All what you see in the schematic is in C terms. Pointers are represented with arrows. Though strings are one dimensional arrays stored with the ascii values of the member characters and terminated by zeros to keep the scheme readable this structure is not pictured. C strings are represented by their C language notation. Hence the quotes are not stored and:

The pointer indicated with **P** in the schema is the expected pointer value to be returned.

In the following scheme all placements in the page is random and does not imply any order about the memory addresses.



The program that you are going to develop is going to have *three phases*. In the *I. phase* you will be reading a sexpr and create the internal representation. To do this you shall write a function `read_sexpr()` which reads any sexpr, creates the relevant union and returns a pointer to it. Since lists are containing other sexprs it is obvious that `read_sexpr()` will be making use of itself (that means recursion).

The *II. phase* continues with the some other reading. This time successively, until the hit of EOF, you will be reading and atom which is followed by any sexpr, (and then another atom and another sexpr). Each sexpr read in this way will become the *value* of the preceding atom (the atom that just came before that sexpr). You will store the value into the value field of the relevant atom. It is quite possible that the atom was not existing due to the first phase. It is also possible that some of the atoms may have no value at the end of the second phase.

The *III. phase* is printing. It has a simple rule:

- integers, floats, strings are printed as they are (strings shall have their double quotes).
- if an atom has no value assigned (i.e. still has the `NULL` in its value field) then print its *printname*.

  Otherwise print the sexpr sitting in (pointed by) its *value* field. (note that this can also be another atom with some sexpr value, then that value has to be printed *(I am sure you immediately spot the recursion behind!)*
- lists are going to be printed in coherence with the rules above.

The next subsection* contains for details.

Specification

Your program will read

$sexpr_{grand}$
$atom_1 \quad sexpr_1$
$atom_2 \quad sexpr_2$
$\vdots$
$atom_n \quad sexpr_n$

Of course the scattering of the input over lines is absolutely insignificant. Your tokenizer was and (still is) insensible to End-of-Line.

$sexpr_{grand} \quad atom_1 \quad sexpr_1 \quad atom_2 \quad sexpr_2 \quad \cdots \quad atom_n \quad sexpr_n$

will be an equivalent input.

The output will be:

$sexpr'_{grand}$

Where it is generated following the rule of the *III. phase*, described in in the previous subsection*.

Example:
Assume the following input is given:

```
  (ad (yas (25 30)) (boy 1.85)
      (medeni_hal es veletler)
      (ev (adres "36.lojman no:8 ODTU")
          (hane_uyeleri ad es kaynana veletler)))
 ad vatandas
 veletler ((mubeccel 5) (munevver 8))
 kaynana (kayinvalide-i vatandas)
 vatandas mukremin
 yonetmen demet_akbag
```

The output is expected to be:

```
  (mukremin (yas (25 30)) (boy 1.85)
      (medeni_hal es ((mubeccel 5) (munevver 8)))
      (ev (adres "36.lojman no:8 ODTU")
          (hane_uyeleri mukremin es (kayinvalide-i mukremin)
           ((mubeccel 5) (munevver 8)))
```

The indentation and line breaking is solely for neatness: **You are not expected to do it**.

- The tokenizer will be placed in a file that you will name as exactly as `tokenizer.c`. Your other source code will go into a file that you will name as `hw3.c`. **this time you are not submitting hw3.c only**. You will pack the two files together into a `.tar` file by a command

  ```
  tar -cvf hw3.tar hw3.c tokenizer.c
  ```

  Than you will submit `hw3.tar` by:

  ```
  submit241 hw3.tar
  ```

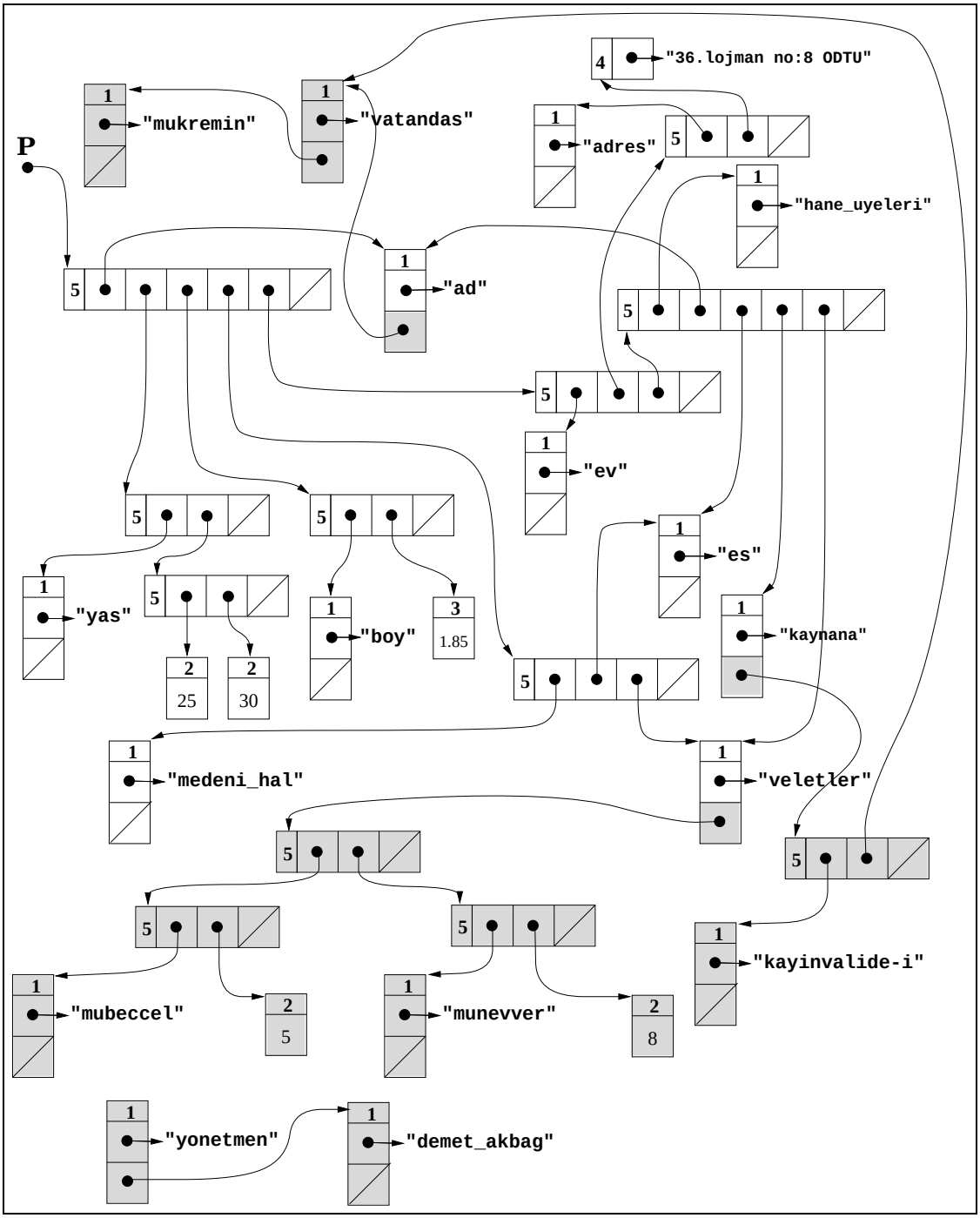
  Resubmission is allowed (till the last moment of the due date), the last will replace the previous, provided you answer the interactive question positively.

- The tokenizer will be called over a single function which shall be named `get_token()` and returns the token kind. The token itself shall be passed through a global `char` array which you will name as `token`. Maximal token length is 100 characters. You are expected, naturally, to introduce `extern` and prototyping lines as necessary.

- Your programs will be evaluated by a test-program. But also be glass-box tested. That means, the evaluation team by no means will modify your code but will look into it. You will also receive/loose points on the programming style.

- You shall make use of `#define` macros in reaching to the subfields of unions and structures.
- No list will contain more then 100 elements. But this knowledge cannot be used to make dynamic memory allocations under the 'worst case' assumption (of that fixed length). But you are allowed to create temporary (auto) variables making use of this knowledge.
- All dynamic memory allocation must be just as much as needed. That means you cannot waste memory.
- It is only the atoms that are stored uniquely. Any other two similar sexprs (e.g. the integers 15 and 15) will have two different internal positions in the memory.
- There will be erroneous input: Expressions which are not sexpr at all; or non-atoms where an atom is expected. In all these error cases you will quit the program by printing on a new line **exactly** the following output:
  `***ERROR***`
- Any unambiguous sexpr printing (where no two-blanks are used consecutively except line starts) is ok, on your side. That means you are allowed to print the whole result in a single line; or break it over lines; or break it overlines and insert more than one blanks at the start of the line (like the output example above). If you decide to scatter over lines the lines have to contain at least 30 characters, provided that it is not the last output line *This just is a restriction to prevent 'ugly' outputs like 'one atom or number per line').*
- Comply **fully** with the input/output specifications. Do not print additional information, prompts, etc.
- Empty lists may exist. Nothing special about them: They read and print as: `( )`
- You are allowed to use any ANSI-C function (the best place for a reference is the book itself).

Here is the scheme of the internal structure after *II. phase* for the example. Gray indicates new/altered.



12 SEXPR UNIFIER

'97 HOMEWORK 4

Introduction

By completing your 3^{rd} homework you are now able to read an *sexpr*, convert it to an internal structure, perform manipulations on it and furthermore print it. In this homework we will make use of this implementation and construct upon it a *unifier*.

The question of *unification* is to take two sexprs which may contain some atoms which we call *variables* and then search for sub sexprs of the sexprs so that when they are substituted for the variables both of the sexprs become identical.

Here is an example, consider the following two sexprs in which `x`, `y`, `z`, `u` are the *variable* atoms.

$sexpr_1$:`(p (f x) y (g y x))`
$sexpr_2$:`(p u (k u) (g z (h w)))`

Now if we investigate carefully both sexprs we discover that if we

substitute `(h w)` for `x`
substitute `(k (f (h w)))` for `z`
substitute `(k (f (h w)))` for `y`
substitute `(f (h w))` for `u`

both $sexpr_1$ and $sexpr_2$ turns into:

```
(p (f (h w)) (k (f (h w))) (g (k (f (h w))) (h w)))
```

For this particular example the result would exactly be the same if `w` would also be a variable atom (ie. it is possible that in the resulting sexpr variable atoms exist too).

We define a *substitution* $\mathcal{S}$ as the a set of pairs s_i/var_i

$$S = \{s_1/var_1,\ s_2/var_2,\ \ldots, s_n/var_n\}$$

with the meaning that, if $\mathcal{S}$ is applied to an sexpr E:

$$\forall i \text{ ssubstitute } s_i \text{ for } var_i \text{ in } E$$

By definition, the substituting expression s_i cannot contain the substituted variable var_i.

If we *apply* the substitution $\mathcal{S}$ to an sexpr E we will dentotationaly write:

$$\mathcal{S}[E]$$

So for the above example we would have written:

$$\mathcal{S} = \{\texttt{(h w)/x},\ \texttt{(k (f (h w)))/z},\ \texttt{(k (f (h w)))/y},\ \texttt{(f (h w))/u}\}$$

Considering the above definition for $sexpr_1$ and $sexpr_2$ we can write

$\mathcal{S}[sexpr_1] = \mathcal{S}[sexpr_2] =$ `(p (f (h w)) (k (f (h w))) (g (k (f (h w))) (h w)))`

The *composition* of two substitutions is another substitution. If $\mathcal{S}$ and $\mathcal{R}$ are two substitutions then the composition $\mathcal{S} \circ \mathcal{R}$ is defined as:

$$(\mathcal{S} \circ \mathcal{R})[E] = \mathcal{S}[R[E]]$$

Assuming

$$\begin{aligned} \mathcal{S} &= \{s_1/v_1,\ s_2/v_2,\ \ldots, s_n/v_n\} \\ \mathcal{R} &= \{r_1/u_1,\ r_2/u_2,\ \ldots, r_m/v_m\} \end{aligned}$$

We obtain $\mathcal{S} \circ \mathcal{R}$ by

$$\mathcal{S} \circ \mathcal{R} = \{S[r_i]/u_i \mid i = 1 \ldots m\} \cup \{s_i/v_i \mid v_i \neq u_j, i = 1 \ldots n, j = 1 \ldots m\}$$

Verbally, the composition of two substitutions $\mathcal{S}$ and $\mathcal{R}$ is denoted by $\mathcal{S} \circ \mathcal{R}$, which is that substitution obtained by applying $\mathcal{S}$ to the substituting sexprs of $\mathcal{R}$ and then adding any s_i/v_i pair of $\mathcal{S}$ having variables v_i not occurring among the variables of $\mathcal{R}$.

For your benefit, we provide you with the algorithmic definition of the function $unify()$ which takes two sexpr to be unified and as a result returns either FAIL or a set which is the substitution required for the unification:

$unify(E_1, E_2) \leftarrow \{$
if either E_1 or E_2 is not a list **then**
{ interchange E_1 and so that E_1 is an atom.
if $E_1 = E_2$ **then return** $(\emptyset)$.
if E_1 is a variable **then**
if E_1 occurs in E_2 **then return** $(FAIL)$
else return $(\{E_2/E_1\})$
if E_2 is a variable **then return** $(\{E_1/E_2\})$
return $(FAIL)$ }
$F_1 \leftarrow$ First element of E_1; $T_1 \leftarrow$ Rest of E_1
$F_2 \leftarrow$ First element of E_2; $T_2 \leftarrow$ Rest of E_2
$\mathcal{Z}_1 \leftarrow unify(F_1, F_2)$
if $\mathcal{Z}_1 = FAIL$ **then return** $(FAIL)$
$G_1 \leftarrow \mathcal{Z}_1[T_1]$
$G_2 \leftarrow \mathcal{Z}_2[T_2]$
$\mathcal{Z}_2 \leftarrow unify(G_1, G_2)$
if $\mathcal{Z}_2 = FAIL$ **then return** $(FAIL)$
return $(\mathcal{Z}_2 \circ \mathcal{Z}_1)$
}

Problem

The statement of the problem is this time very simple. Your program will read 2 sexprs from the standart input and produce a single sexpr as output (do not echo the input on the output). That output sexpr will be either the atom

`FAIL`

or a list of sublists:
((sub_sexpr_1 var_atom_1) (sub_sexpr_2 var_atom_2) (sub_sexpr_n var_atom_n))
with the obvious meaning of being the substitution discovered by the unification.

In the input all atoms that start with a capital letter are variables.

If no substitution is required (ie. two identical sexpr were given as input) the output will be an empty list.

Specification

- all specifications about sexprs syntax and semantics, count of elements, error actions of the previous homework continue to hold.

13 MARKETING POLICY

'97 HOMEWORK 5

Problem

This time the definition of the problem is short and simple. You are responsible of the marketing policy of a gross store.

Each day you will be putting an item from the stock on sale. For simplicity each day there will be only one item on sale.

Furthermore what ever you put on sale you are able to sell that day. The items on stock are numbered starting with 1. Each item contains two informations on its label :

its expiration day d from now on. This is an integer $0 < d < 5000$. After d days you are not allowed to sell the item. It has also an effect on the selling price (see below).

its starting price p. This is another integer $p < 1000$. If you put the item today on sale you will get p money units. If you sell it tomorrow then you get p/d less then the price p. If you sell it the n^{th} day (today is counted as $n = 0$) you get $p - n/d$. When $n = d$ you are not allowed to sell it. (division is floating point division).

You are expected to find out a sequence of item sells so that you get maximal money. You are allowed, of course, to have some items deteriorate in stock (i.e. exceed the expiration date and still have it in stock).

Specification

- Input will be done from a file which will be named as `day-price.txt` . It consists of lines of similar structure. Any i^{th} line of this file is of structure
  $day_i \;\; price_i$
  and is the information about the i^{th} item.

- The output is a sequence of integers, each printed on a distinct line of the output file `sell-order.txt` . the first line will contain a floating point number (you will decide to use `double` or `float`) which is the money you gain with your proposed selling schedule. The following lines will contain the selling schedule. That means the first line after the floating point number is the number of the item which is to be sold *today*. The second line is the item for *tomorrow*, and so on.

- There may be at most 10000 items. But it is possible to have less.

- There is **no restriction** that says the d figure or the p figure cannot repeat.

- Your program will be run on the Sun machines which you have in the lab available. The run time will be **1 minute**. Then it will be terminated. We advise to use the 'time bomb' concept which is introduced on the WEB page. Though we reserve

the right to run your program on faster machines, the execution time will always be restricted to 1 minute. If you are planning to overwrite your file from time to time with *better* solution do this by reopening the file, writing and then closing it.

Evaluation

- A program submitted after the due date/time will be graded **zero**.
- A working program (in the terms defined above) will receive 40 points, otherwise it will be graded **zero**. Here the term "working" means to produce a valid and **correct** output file. (The money calculation mast be correct, there shall be no logical error in the sellings (no repeated sells, no non existing item sell, etc.)
- Your program will receive an additive point P depending on its quality of solution, calculated as follows:

$$P = 60 \times \frac{Money_{Your\ program} - Money_{Lowest}}{Money_{Highest} - Money_{Lowest}}$$

  Here $Money_{Highest}$ is the money earned in the the best performing submission. $Money_{Lowest}$ is the money earned in the worst but 'working' (in the above defined terms) submission.
- All submissions will be compiled and run under strictly equal conditions.

14 POLYGON CLIPPING

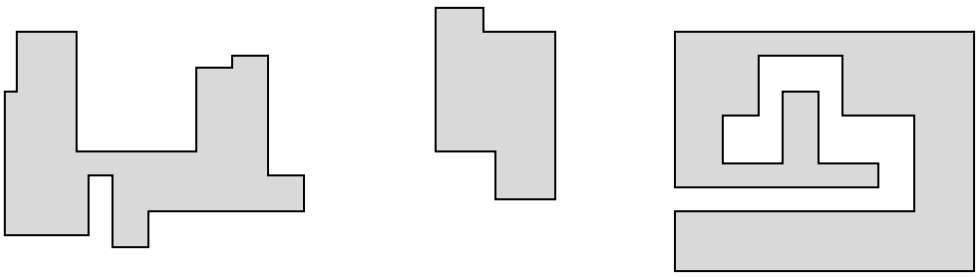
'98 HOMEWORK 1

Problem

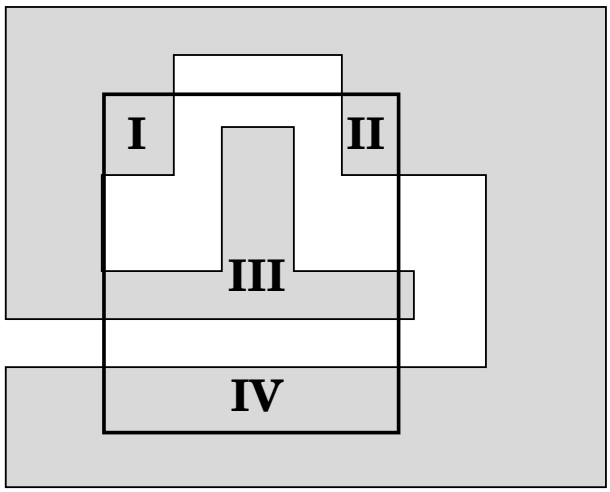
Welcome to the series of Ceng. 140 homeworks. Here is the first one to go!
This is a clipping problem. You will be given a polygon which:

- can be convex or concave (no restriction in this sense),
- has its edges parallel to one of the Cartesian coordinate axes.

Here are a couple of such polygons:



In addition to the given polygon you will be given the size and the position of a rectangular window. The problem is to find how many distinct parts of the polygon falls into the window. For example consider the following polygon and window:



As seen there are exactly four distinct parts of the polygon when viewed through the window (they are marked as **I,II,III,IV** in the figure. So the solution of the problem displayed in the figure is **4**. You are expected to produce this number.

Specifications

- The coordinate system is the Cartesian coordinate system where the apsissa is called X and the ordinate is called Y. Below, we will refer to smaller/greater X as *lower/upper*; smaller/greater Y as *left/right*;
- All coordinates are integers in the range $[0, 20000]$
- Your program will read from standard input.
- The first line of the input consists of four integers namely the coordinates of the window lower-left and upper-right corners. In the following order:
  $X_{lowerleft}$ $Y_{lowerleft}$ $X_{upperright}$ $Y_{upperright}$
- In each of the following line there will be the coordinate of the a corner of the polygon:
  X_{corner} Y_{corner}
  Maximal count of corner is 1000 (but there can be less). The coordinates of neighboring corners will be on successive lines. The corner defined in the first of such lines (the second line of the input) is neighbor to the corner defined in the last line.
- Your output consists of a **single integer**, namely the count of distinct parts (as described in the PROBLEM part, above).
  You shall not print anything more!, for example **DO NOT PRINT** lines as
  `The solution is: 4` or `There are 4 distinct parts.`
  Such programs will receive 0 (zero) grade.

'98 HOMEWORK 2

15 ANOTHER NONSENSE STACK MACHINE

Background Information

See **Background Information** of '95 HOMEWORK 2

Problem

In this homework you are going to implement a stack and using it, following some rules given below, you will be pushing and popping elements from this stack. The whole task is to output the number that is at the top of the stack after that procedure is carried out.

1 You will be reading an integer number in the range [0,1000]. If there is no more input (EOF case: (control-D) is inputted) you print the top of the stack and stop.

2 Any number such arrived is *pushed* onto the stack.

3 The top of the stack is examined:

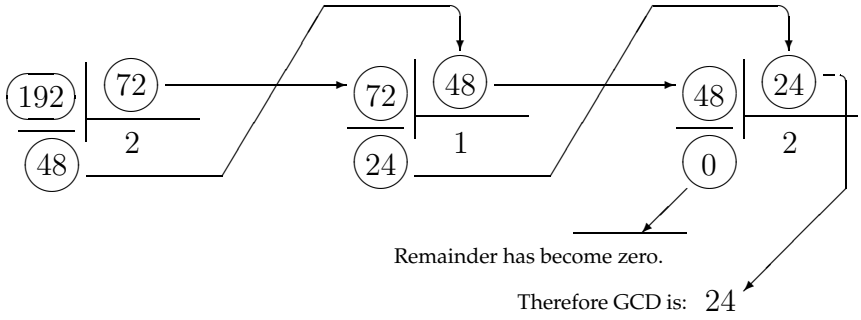
- If it is a square number *(i.e. is a square of another integer)* which is> 1 then it is replaced by its square root. The process continues from step **3**. *(See **Restrictions**)*
- If it is **not** one of the followings
  - `0`
  - `1`
  - A prime number

  nothing is done, with step **1** the process continues.
- Otherwise, a (modulo `5`) operation on the number is performed (lets call the result m). The prime number is discarded from stack *(popped)*, then the following two numbers that are on the stack are *popped* (lets call these numbers p_1 and p_2).
  - If m is `0` the result of $|p_1^2 - p_2^2|$ is pushed onto the stack.
  - If m is `1` the result of $p_1 \times p_2$ is pushed onto the stack.
  - If m is `2` the result of $p_1 + p_2$ is pushed onto the stack.
  - If m is `3` the result of $|p_1 - p_2|$ is pushed onto the stack.
  - If m is `4` the result of $\gcd(p_1, p_2)$ is pushed onto the stack.

  Process continues from step **3**. *(See **Restrictions**)*

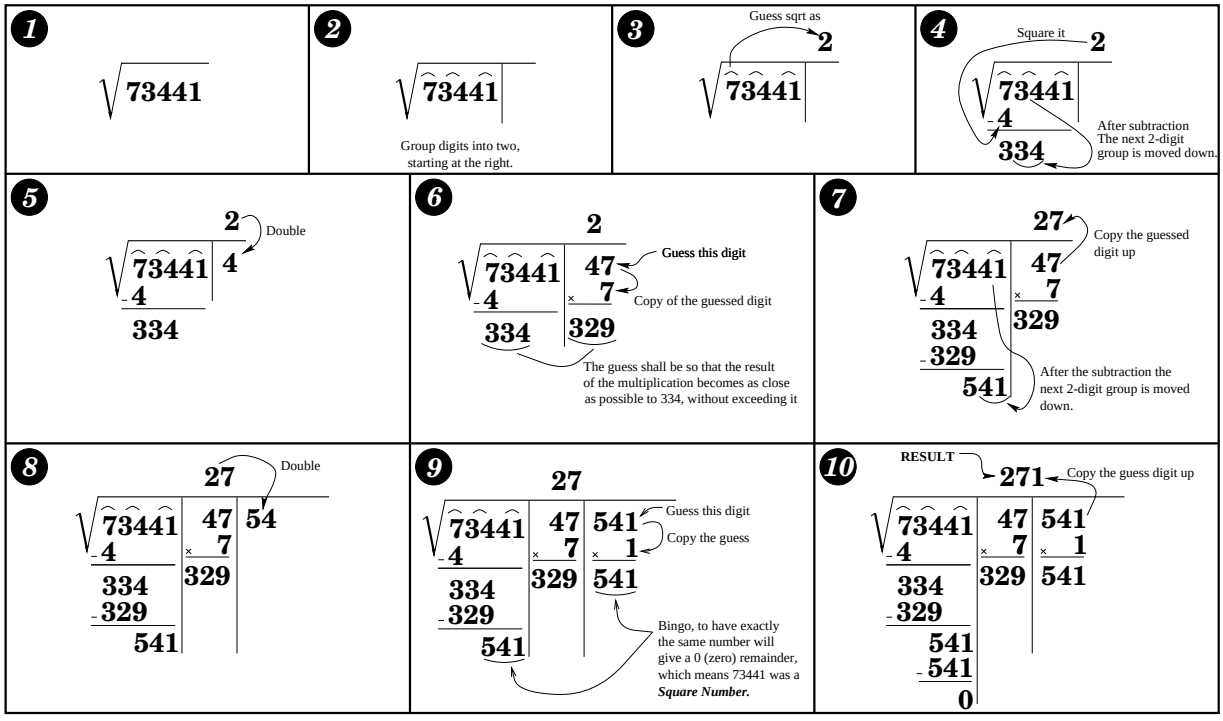
Restrictions

- gcd has to be implemented as a recursive function that implements the Euclid greatest common divisor algorithm. Below you see an example where $\gcd(192, 72)$ is derived. *(It is very difficult to understand why this simple and wonderfull algorithm is not thought at High school!).*
  Note: After you have understood it do not forget to put it in a recursive form!



- You are not allowed to use mathematical library functions, like those in `math.h`. Also you are not allowed to implement and use 'Taylor Expansion' (series expansion) of functions. i.e. taking square root shall be done by implementing the high-school-method.

  Below you see a reminder which goes through the steps of square-rooting 73441.



- No use of `goto` is allowed.
- Implement the stack as an array (that you will name as `stack`) with maximum 5000 elements. This also means that no test case will require more then 5000 positions on stack. Implement `push, pop, empty, stacktop` functions which will operate on this global array. For **any** stack operation only make use these four functions. Never examine/modify the stack array directly (by-passing these functions).
- All numbers coming from the input will be in the range [0,1000]. You don't have to perform an additional check on it.
- The result (as well as the intermediate results) may well be >1000. But it is assured that they will never exceed 10^9.
- A *pop* with an empty stack is an error. Your program shall check this always. If at any stage the procedure requires this illegal operation it shall abort with exactly the following output:

  ```
  error
  ```

- The input/output specifications are extremely tight. The input is: integer numbers entered one-per-line, and the input is terminated by an *Control-D*. The output is: A single integer number or the above defined error message. *No blank lines, no additional information strings, NOTHING!*

16 MANY BODY PROBLEM

'98 HOMEWORK 3

Introduction

This time we will be doing some physics in a graphical environment. Our is to implement a graphical simulation of a gravitational many-body problem.

The invariant property of a *particle* is characterized by a real number which we call its *mass* and denote by the letter m. The dynamical property of a particle is characterized by two vectors, namely its position $\vec{r}$, and its velocity $\vec{v}$. If we pick a Cartesian coordinate system then a vector is represented by three numbers, which we call the *components of a vector* and denote by a three-tuple of real numbers. *(tuple has the meaning of 'ordered set' in computer science)*.

To predict (compute) a particle system's behavior in time, the only thing that we have to know is the masses and the initial values of the dynamic properties of each particle. then the laws of physics predict the future of the dynamics[2]

In the problem that we will handle today, due to Kepler's Law, it is known that if the initial velocities lie in a plane then the velocities in the future will stay in that plane. Therefore we will deal with two-dimensional vectors for position and velocity. So, our vectors will not be three-tuples but two-tuples.

Our system will consists of n particles. We will label our particles with integer subscripts. For example, m_4 will denote the mass of the 4^{th} particle. The values in the following table will be provided as input.

MASS	INITIAL POSITION	INITIAL VELOCITY
m_1	$(x0_1, y0_1)$	$(v0x_1, v0y_1)$
m_2	$(x0_2, y0_2)$	$(v0x_2, v0y_2)$
$\vdots$	$\vdots$	$\vdots$
m_n	$(x0_n, y0_n)$	$(v0x_n, v0y_n)$

On the next page you see such a many-body particle system with 3 particles.

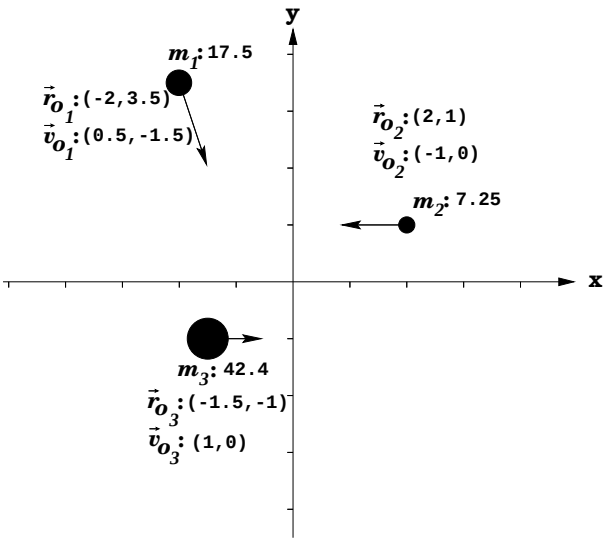
Consider a single particle, as you know, velocity is the change in the position in unit time. Normally we express this as a derivative, considering the unit-time as an infinitely small number:

$$\vec{v} = \frac{d\vec{r}}{dt}$$

If we discretise this, that means approximate the infinitely small by some 'small' number, then we write the above derivative as

$$\vec{v} = \frac{\Delta\vec{r}}{\Delta t}$$

[2]We confine ourself to classical mechanics, which is quite correct at macro scale. If we go down to atomic sizes then this approach will no more predict correct results, you must pick a quantum mechanical approach



The difference is that now Δt is just a 'small' number. So if we call Δt the *unit-time*, then at any time t if a particle is at position $\vec{r}$ then after the *unit-time* the new position will be:

$$\vec{r} + \Delta t \cdot \vec{v} \tag{1}$$

This is always true, even if the velocity is changing. If the particle is under any force then its velocity will change. So at t we will have a velocity value $\vec{v}$ which will be changed after *unit-time* has passed. The change in the velocity is called *acceleration* and is denoted with the vector $\vec{a}$. Newton's second law provides a relation that gives us the *acceleration* at any time the particle will attain, if we know the force acting on the particle at time t as well as the mass of the particle, then:

$$\vec{a} = \frac{\vec{F}}{m}$$

By definition $\vec{a}$ is:

$$\vec{a} = \frac{d\vec{v}}{dt}$$

Again if we discretise the time and consider a *unit-time*, a velocity $\vec{v}$ will change to

$$\vec{v} + \Delta t \cdot \frac{\vec{F}}{m} \tag{2}$$

In order to calculate all the dynamics of a single particle we have to know what forces are acting on it. Force is *superposable*, that means if you know that on a particle there are two forces $\vec{F_1}$ and $\vec{F_2}$ acting, then you can assume that these forces can be substituted by a single force $\vec{F}_{sum}$ which is $\vec{F}_{sum} = \vec{F_1} + \vec{F_2}$. Hence, generally speaking, if a particle is under m number of forces then the resulting force is the vector sum of all of them:

$$\vec{F}_{sum} = \sum_{i=1}^{m} \vec{F}_i$$

Now, what is the force exerted by some other particle p' on the particle p? We will consider in this problem a *gravitational* force. As you now, the gravitational force is inverse quadratic proportional to the distance between the two particles.

$$\vec{F} = \frac{-G \cdot m \cdot m'}{\|\vec{r'} - \vec{r}\|^2} \cdot \hat{u}_{rr'}$$

G is the so called *universal gravitational constant*, which is a real number. Furthermore, $\hat{u}_{rr'}$ is the unit vector in the direction from particle p to p'. It other words it is simply $(\vec{r'} - \vec{r})/\|\vec{r'} - \vec{r}\|$. So, we could have written the force as:

$$\vec{F} = -G \cdot \frac{m \cdot m'}{\|\vec{r'} - \vec{r}\|^3} \cdot (\vec{r'} - \vec{r})$$

If we are going to make use of the *superposition principle* and calculate the resulting force acting on a particle p_i by all other particles, we can write:

$$\vec{F}_{i_{sum}} = \sum_{\substack{j=1 \\ j \neq i}}^{m} -G \cdot \frac{m_i \cdot m_j}{\|\vec{r_j} - \vec{r_i}\|^3} \cdot (\vec{r_j} - \vec{r_i})$$

Taking all constants out and rearranging the order of subtraction :

$$\vec{F}_{i_{sum}} = G \cdot m_i \cdot \sum_{\substack{j=1 \\ j \neq i}}^{m} \cdot \frac{m_j}{\|\vec{r_j} - \vec{r_i}\|^3} \cdot (\vec{r_i} - \vec{r_j}) \tag{3}$$

With the equations (1), (2) and (3) it is possible to calculate new positions and velocities for all particles after a *unit-time* Δt is passed. *(Provided that the initial positions and velocities are given)*

Problem

You will be given the value of the G constant, the *unit-time* Δt, and a set of particles' mass and initial (position and velocity) values. You are provided with some graphics toolbox (some C-functions library) . This toolbox enables you to create a window of a given size, draw lines, rectangles, circles and put text at any place in it. In the proceeding subsections you will find the description of how to use it.

You are expected to draw a simulation of the system's actions with the time changing in *unit-time* steps. The iteration is as follows:

1. Use the initial values for positions and velocities.
2. Draw at the positions the particles (of an area size proportional to their mass)
3. Using the equations (1), (2) and (3) calculate new positions and velocities
4. Continue from item [2].

How-to-Do's

- You are provided with some facilities for drawing in an X-windows environment, these are:

  Some header files: Your top lines of your `hw3.c` file shall contain

  ```
  #include <xview/xview.h>
  #include <xview/canvas.h>
  #include <xview/cms.h>
  #include <xview/xv_xrect.h>
  #include "/shared/courses/ceng140/hw3.h";
  ```

  An X-window: You will have a graphical X-window poping up when you run your program. **you don't have to do any additional work to create the window**. The window size is fixed to be 1200 units in horizontal, 800 units in vertical direction. The *origin* is at the top-left corner of the window. From now on we will refer to the distance of a point in the window, measured in the horizontal left-to-right direction and starting from the origin as the *x-coordinate*; and similarly, in the vertical top-to-bottom direction (starting form the origin) as the *y-coordinate*.

  Functions for drawing: Through the included `hw3.h` file you have (automatically) three functions available:

  `DrawLine(int x1,int y1,int x2,int y2)` As the name suggests, this draws a straight line from the coordinates $(\texttt{x1}, \texttt{y1})$ to $(\texttt{x2}, \texttt{y2})$

  `DrawCircle(int x,int y,int r)` Draws a circle which has its center at $(\texttt{x}, \texttt{y})$ and is of a radius `r` units.

  `DrawString(int x,int y,char *text)` At $(\texttt{x}, \texttt{y})$ places the string which is pointed by **`t`**ext. The start of the string is at the given coordinate. Each character position is about 6 units wide.

  `PenColor(int x)` For `x` use one of the following colors (which are built-in integer constants): `WHITE`, `RED`, `GREEN`, `BLUE`, `ORANGE`, `AQUA`, `PINK`, `BLACK`. The canvas that you draw on is in `WHITE`. You can pick a 'pen' of the provided colors by this function. Until a new 'pen' is picked all drawings is done with that color.

  A `main` function: The actual `main()` function is defined in the header file `hw3.h` . At the start of this `main()` function some initializations are performed (you don't have to bother what these are). Then a function which is named as `student_main()` is called. Therefore in this homework

  - **DO NOT** define a function with name `main()`.
  - **DEFINE** a function with name `student_main()` to do whatever you would normally do in your `main()` function definition.

  A `make` file: You are provided with a make file

  `/shared/courses/140/Makefile`

You shall copy this file to your work directory (do not rename it). Then a simple command as

```
make
```

will perform the compilation and linking with the correct libraries of your `hw3.c` file. The executable will have the name `hw3` and will be placed in your working directory. (*Note that the make file does not take an argument like `hw3.c`*).

- To remove some drawing from the canvas simply pick the white pen, and redraw the drawing.
- To wait a specific amount of time you can use the `usleep()` function. The argument denotes the time (in microseconds) the program will wait before it proceeds.

Specifications

- There will be at least 1 particle and at most 20 particles.
- The input will come from the *standard input*, as usual. Except $Delay$, all values in the input are `float`. That means read them with `"%f"`.
- All coordinates that come from the input are similar to what is in the figure above. So, the input is so that the origin is expected to correspond to the midpoint of the window. It is your resposability to make the necessary conversion (note that X-windows uses a different coordinate axis system for drawing)
- The structure of the input is as follows:

G	$Delay$	Max_x	Max_y	
m_1	$x0_1$	$y0_1$	$v0x_1$	$v0y_1$
m_2	$x0_2$	$y0_2$	$v0x_2$	$v0y_2$
$\vdots$	$\vdots$	$\vdots$		
m_n	$x0_n$	$y0_n$	$v0x_n$	$v0y_n$

It is your responsibility to detect what n is. $Delay$ is an unsigned integer. It is the delay time, that you shall wait between a view of the former system and the following one. You shall implement it as a function call to a library function with the name `usleep`. This function call shall be placed into the iteration after a 'refresh' of the display and before you do the 'new' position calculation. The $Delay$ number is in microseconds and it is quite possible that it is a number which requires 4 bytes. You shall use this number as the argument of the library function `usleep()`. `usleep` has its prototype information in the header file `unistd.h`. On Linux the argument is an `unsigned long` and on Solaris it is an `unsigned int` (as you note on both systems it is a 4 byte number). So, on Linux read it into an `unsigned long` variable using `"%lu"` format, and on Solaris read it into an `unsigned int` variable using `"%u"` format.

Max_x/Max_y are those x/y coordinate values which correspond to the right/top edges of the window. (i.e. if Max_x is `20.5` then a x coordinate value of `20.5` in the input will place the particle exactly on the right edge of the window.)

- The program does not have an exit criteria. The simulation shall go on and on. The X-window kill icon, or *control-C* will be used to kill the program.
- The particle with the smallest mass shall be represented with a circle of radius 2 pixels and largest shall be represented with a circle of radius 20 pixels. All particles shall have areas which are linearly proportional to their masses. It is ok to have a ± 1 pixel approximation.
- In contrary to the previous hw's, in this homework you are allowed to do variations on the output screen (put additional information).

17 POLYNOMIAL ALGEBRA

'98 HOMEWORK 4

Problem

In this homework you will be doing some simple computer algebra. The aim is to write a program which is able to input single variable polynomials, namely in (x), can store them under variable names and is able to evaluate any expression which can involve those variables, their first order derivatives, multiplication, addition and subtraction.

You are expected to store the polynomials internally in link lists; holding only non zero coefficients and the powers of the terms in the elements of the link lists..

Specification

- All more then one whitespaces (including end-of-lines) in the input are redundant and are equal to only one blank.
- The input is in one of the following form
  1. `SET` *variable* `=` *polynomial* `;`
  2. `EVAL` *expression* `;`

  where
  - *variable* is any identifier (C-like) of maximum 10 characters.
  - *polynomial* is a polynomial expression where the polynomial is written in powers of `x`. It is of a flat structure where *monomials* are separated by either a `+` (plus sign) or a `-` (minus sign). (In the input monomials are not necessarily ordered from highest to lowest power, but it is guaranteed that a power appears only once (if it does)).
  - A *monomial* is in one of the following form
    * *a natural number* *(Example:* `453`*)*
    * `x`
    * *a natural number* `x` *(Example:* `453x`*)*
    * `x^` *a natural number* *(Examples:* `x^2`, `x^1`*)*
    * *a natural number* `x^` *a natural number* *(Examples:* `453x^2`, `32x^1`*)*
  - Whitepace is not of any importance and can appear anywhere except in a number. Here are several examples of such polynomials:

```
x ^ 2 - 1
x^2       -1
x ^1001 +673x^   23+7
453x - 13x^102 + 4x^1
1358  x + x^10
```

  - *expression* is one of the following
    * *variable*
    * *variable*`'`
    * (*expression*)
    * *expression*`+`*expression*
    * *expression*`-`*expression*
    * *expression*`*`*expression*

    Multiplication has higher precedence over addition and subtraction. Parenthesis have the usual meaning of grouping.

- The action of `SET` is self-explanatory. It binds the *polynomial* to the *variable* as value.

- The action of `EVAL` is evaluating the expression and obtain a single polynomial. Then print this resulting polynomial. The evaluation scheme is as follows:

EXPRESSION	RESULT
variable	Polynomial bound to *variable*
variable`'`	Derivative polynomial wrt. `x`
expression operation expression	Polynomial *operation*

  The print of a resulting polynomial is expected to be

  - Zero coefficient free: (the term will be dropped).
  - Zero power free: (the `1` and `0` power will not be shown).
  - Unity coefficient free: (A coefficient of `1` will not be shown).
  - In descending power order: (highest power first, lowest power last in line).
  - Not contain any end-of-line in the polynomial.

- All internal representations for coefficients and powers shall be `long int` (which is of size 4 bytes in our department). If any result at any stage of a calculation is going to cause an overflow you are expected to print a single line of output and quit the program:
  `OVERFLOW`

- If a *variable* does not have a bound polynomial then you shall print a single line of output and quit the program:
  `NO VALUE FOR:`*variable*

- Except those above, you can assume that the expression itself is error-free (parenthesis are balanced, etc.)

- The expression will not exceed 80 characters.

- Dynamic memory allocation shall only be used for allocating link list elements. You are not allowed to use an array representation at any stage of the polynomial representation. There is no limit on the size and number of monomials in a polynomial. You are allowed to use arrays only
  - to store [*variable,bounded polynomial*] information;
  - for stack purpose. (you will use stacks in the evaluation phase).
- Deallocate (free) unused memory. (e.g. re-bound variables' former polynomial values (the link list) will no more be used, so deallocate the elements).
- There will be limits set for the execution time. Watch out for announcements at **tin**.

How-to-Do's

- For the reading (parsing) of the *expression* and the evaluation of it there exists a one-pass algorithm which is called "Dijkstra's Algorithm for Infix to Postfix Conversion". You can find it in many Data Structure books.
- Construct the polynomial link list in an ordered form, on-the-fly. While you are reading, search for the position of the monomial to hand and then insert it into the link list.
- Store the *variable, bound polynomial* information into an array of structures. where the structure is of nature:

```
struct value_table
  { char variable[11];
    EP   polynomial; }   /* for EP: look up lecture notes
```

Sample run

Below (•) is denoting a user typed in line and (◦) is denoting a computer generated output. These markers are not part of the input/output text.

- `SET A = x**6 - 18*x**5 + 135*x**4 - 540*x**3 + 1215*x**2 - 1458*x + 729;`
- `SET B = x**3 + 9*x**2 + 27*x + 27;`
- `SET C = 16*x**4 - 288*x**3 + 1944*x**2 - 5832*x + 6561;`
- `SET D = x**2 + 2*x + 1;`
- `SET F = 3;`
- `EVAL A-B;`
- ◦ `x^6 - 18x^5 + 135x^4 - 541x^3 + 1206x^2 - 1485x + 702`
- `EVAL B+C;`
- ◦ `16x^4 - 287x^3 + 1953x^2 - 5805x + 6588;`
- `EVAL A*B;`
- ◦ `x^9 - 9x^8 + 216x^6 - 486x^5 - 1458x^4 + 5832x^3 - 19683x + 19683`
- `EVAL A';`
- ◦ `6x^5 - 90x^4 + 540x^3 - 1620x^2 + 2430x - 1458`
- `EVAL A*D-B;`
- ◦ `x^8 - 16x^7 + 100x^6 - 288x^5 + 270x^4 + 431x^3 - 981x^2 - 27x + 702`
- `EVAL A*(B-D);`
- ◦ `x^9 - 10x^8 + 16x^7 + 116x^6 - 198x^5 - 1728x^4 + 5400x^3 + 972x^2 - 19683x + 18954` *(actual output will not contain any line break)*
- `EVAL C'*(C-B+D*D);`
- ◦ `1088x^7 - 32928x^6 + 436560x^5 - 3258968x^4 + 14685688x^3 - 39730392x^2 + 59554440x - 38112120` *(actual output will not contain any line break)*
- `EVAL D*D-(D-1)*(D+1) = 1;`
- ◦ `1`
- `EVAL B'-F*(X+3)*(X+3) + F = 0;`
- ◦ `3`
- `EVAL A*A*A*A = x**24 - 72*x**23 + 2484*x**22 - 54648*x**21 + 860706*x**20`
- ◦ `OVERFLOW`
- `EVAL A*D-K;`
- ◦ `NO VALUE FOR:K`

18 GENETIC ALGORITHMS-I

'99 HOMEWORK 1

Introduction

Welcome to the series of Ceng. 140 homeworks of this semester. This time you will be experimenting with a problem solving technique of Computer Science (CS) which became popular in the recent years, namely *Genetic Algorithms (GA)*.

As it is not the first time in science history, in this technique CS has stolen (or more politely, 'got inspiration of') the fundamental idea from mother nature.

As you know from your high school biology courses, evolution of life is based on a genetic mechanism. Here the *problem* is to have an organic substance which is able to *survive* it the rough environment that we call *nature*. The *solution* to this problem is a *genetic code* that encodes all the blue prints of a *survival machine* of organic construction the "living specie".

As a solution mechanism the genetic code has some properties:

- Over a long period of time some genetic codes survive, some don't. Those that survive are the *fittest*. So, there is an external *evaluation* mechanism which ranks different solutions (genetic codes) for being the fittest.
- Genetic code interact with other genetic code, in a process that we call *crossover*. Though it is wrapped into some wonderful and most romantic event as 'love', a selfish gene[3] crossover with another selfish gene. So there are two off-spring codes which are a brand of their parent codes.
- Occasionally some part of the code undergoes a random change. This might be due to a γ-ray hitting a molecule, or some erroneous chemical reaction. The key is that this occurs 'rare' and is a very local alternation. We call this *mutation*.
- Each genetic code has a life time. Then it dies. The next generation will live in the next period. The "right to live" for a child is solely based on the result of its evaluation. If it is far from being "fit" then it will be removed from the *genetic pool*.

GA, adopts exactly this scheme. When we decide to solve a world problem by means of GA we do the following:

1. Determine a mapping of the

   $$\text{Problem Solution Space} \longmapsto \text{Space of Binary Strings}$$

   For sake of simplicity we choose a fix length for of the binary strings. In other words, we find a binary string representation for the problem solution.

2. Create a pool of such strings. (usually this is an array of binary strings).

3. Randomly (but with an even randomness) fill out the binary strings with initial values.

[3]The term of *selfish gene* is due to Richard Dawkins, the author of th marvelous book The Selfish Gene published by Oxford Univ. Press. first in 1976

4. Device a function $Eval$

$$Eval \ : \ \text{Space of Binary Strings} \longmapsto \mathcal{R}$$

so that for each binary string s (which encodes a solution), $Eval(s)$ returns a real number that corresponds to the degree of how fit s is (as being a solution to the world problem).

5. Evaluate each member of the pool by means of $Eval()$ and keep the 'evaluation' results.
6. Randomly mate all binary strings in pairs. Among a pair, at a randomly chosen point perform a crossover (do this by cutting both strings into two, exactly at that chosen crossover point, and then switch the binary segments after the crossover point). So obtain two off-springs.
7. Evaluate all off-springs (since a pair of off-springs are obtained by mating a pair of binary strings the number of off-springs are exactly equal to the number of the parent strings).
8. Among the new strings decide who will live and who will die: Replace the least fit n off-springs by the n best fits of the parent (former) strings.
9. Make the new generation the current generation and continue from step (6) if a termination criteria is not met yet.

THE μDICTIONARY OF GA JARGON:

Gene: *A portion of the solution encoding binary string which corresponds exactly to a single feature (parameter) of the solution.*

Chromosome: *The binary string that encodes a solution of the target problem. This is usually an encoding of several parameters which, when determined, form the solution of the problem.*

Allele: *The set of all possible (admissible) values a particular gene can take.*

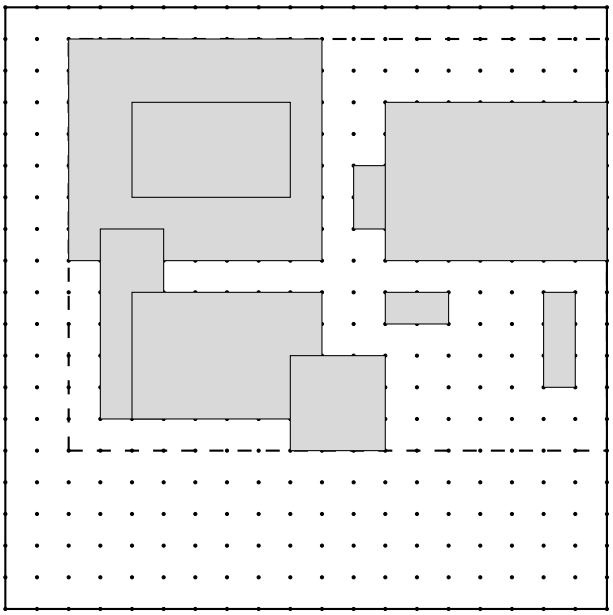
Problem

In a series of three homeworks we will attack a problem which will be explained below and try to solve it by means of a GA technique.

The problem is to place given rectangles on a square grid so that

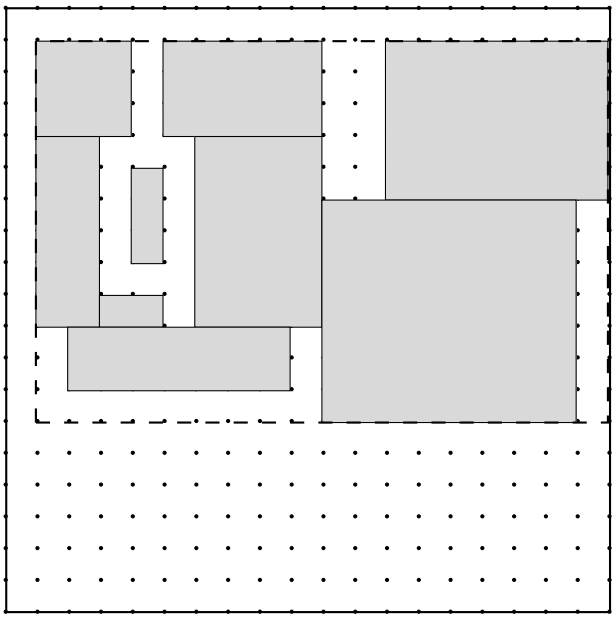
- No rectangles overlap!
- The *bounding box* (a smallest rectangle that includes all the rectangles) has a minimal area.
- Rectangles can be placed are allowed to be placed anywhere provided that one of their sides (any of it) is parallel to the horizontal direction. Furthermore all corners have to remain inside the grid (no clipping).

In the input of the problem you will be given the dimensions of the rectangles. Here is an example of a **random** placement of such given rectangles On a 20×20 grid (this is of course far from being a solution).



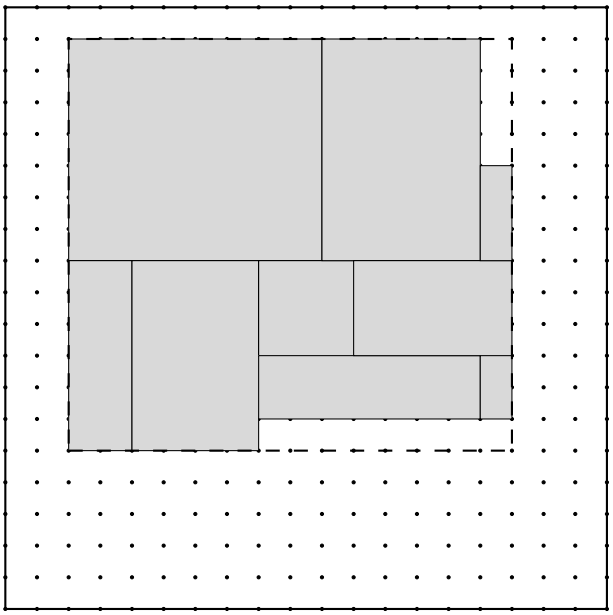
This random placement has a bounding box (indicated in the figure by a dashed line-box) of 13×17 which means an area of 221. But this placement does not qualify to be a solution at all, since there are overlapping regions of some rectangles.

Below is a more organized placement of the same rectangles. This time we have a candidate for a solution.



This placement has a bounding box of 12×17, an area of 204 and no overlapping regions among the rectangles.

Here is a solution which is even a better one: the bounding box is 13×14 which means an area of 182.



PART I

Specifications

- You will be dealing with a grid of 256×256. the origin of the grid is labeled as $(0, 0)$. This means that the upper-right corner of the grid is $(255, 255)$. We will denote the horizontal axis with the letter x and the vertical axis with y. In a 2-tuple representation our convention will be writing (x, y) (as usual).
- Any coordinate (i, j) in this series of homeworks have both i and j positive integers.
- Your program will read the

  Width, Height, x_{left}, y_{lower}, inverted

  information from the *standard input*. You are given that there are at most 1000 rectangles. Each input line will contain 5 byte-integers corresponding to the information of one of the rectangles. *(There will be no other characters on the line except these five numbers which are separated from each other by blanks only)*. No information (like an integer at the start of the input) of how many actual lines are present in the input will be given! The input will terminate with the END-OF-FILE character (*ctrl-D* on the UNIX system).

  A possible input for the example above is:

```
37 19 17  32 0
5   9  9  15 1
12  4  8   0 0
7  11  0   0 0
10  7  1 251 0
2   3 19 101 1
```

  cntr-D

  In this input there are 6 rectangles defined. The rectangle defined in the first line of the input has its lower-left corner placed at $(17, 32)$. The last integer in the line, which will be one from $\{\texttt{1}, \texttt{0}\}$, indicates whether the width is parallel to the horizontal or to the the vertical direction.

  `0`: means width is aligned parallel to x-axis.

  `1`: means width is aligned parallel to y-axis.

  So, the rectangle described in the first line has its upper-right corner placed at $(54, 51)$.

  The rectangle described in the second line is inverted and has its lower-left corner at $(9, 15)$, its upper-right corner at $(18, 20)$

  The fifth line describes a placement which is exceeding the right boundary of the grid. We call such placements as *illegal*.

- There is no order imposed on the input lines.

- The output consists of a single line of four numbers: an integer, followed by two unsigned byte-integers, and then a long integer:

  $Count_{illegals}$ $Boundingbox_{width}$ $Boundingbox_{height}$ $Area_{overlap}$

  Here

  $Count_{illegals}$ is the *count* of rectangles that have not all of its corners in the grid. If no such rectangle exists this number will be 0 (zero).

  $Boundingbox_{width}$ is the *width* of the bounding box that encloses all the rectangles except the *illegals*.

  $Boundingbox_{height}$ is the *height* of the bounding box described in the item above.

  $Area_{overlap}$ is the *total overlapping area* (attention: consider their overlapping area only once when you consider two rectangles). Do not consider illegal rectangles.

19 GENETIC ALGORITHMS-II

'99 HOMEWORK 2

PART II

Problem

Mainly the task of this homework is to visualize the work done in PART I (previous homework. You will be creating a window which will display the placement of some given rectangles on the grid. Also you will evaluate the placement (which will later be used in the GA) and display this evaluation result (a floating point number).

Specifications

You are provided with some facilities for creating and drawing in an X-windows environment, these are explained in the How-to-Do section.

The job can be described by means of the following items:

- The first line of the input consists of three floating point *weights*.

$$Weight_{illegals} \quad Weight_{bounding\_box\_area} \quad Weight_{overlap\_area}$$

- The input following the first line is exactly the same as in Homework 1.
- None overlapping regions of rectangles will be drawn by their edges. Black will be used for edges, white for interior.
- Overlapping regions will be drawn by a grading from Light Pink to Dark Red. Light-Dark interpretation is as follows:

  Exactly 2 rectangles are overlapping Lightest Pink.

  Exactly 3 rectangles are overlapping One degree darker Pink than above.

  Exactly 4 rectangles are overlapping One degree darker than above.

  ⋮

  Exactly 10 rectangles are overlapping One degree lighter than Dark Red.

  11 rectangles or more are overlapping Dark Red.

- An illegal placement will be drawn with Black fill.
- Any color (Light Pink - Dark Red) over black has (more) visibility. (ie. If a non-illegal rectangle overlaps with an illegal one, then the overlapping region will be drawn by a Reddish color (according to the coloring rule explained above)). This applies also for the edges.
- The bounding box is expected to be drawn (as a full drawn black edge) rectangle. (hint: draw it at last).

- In the bottom-right corner an overall evaluation grade for the input will be displayed (in black). This is a floating point number computed as follows:

$$\begin{gathered} Weight_{illegals} \times Count_{illegals} \\ + \\ Weight_{bounding\_box\_area} \times (Boundingbox_{width} \times Boundingbox_{height}) \\ + \\ Weight_{overlap\_area} \times Area_{overlap} \end{gathered}$$

  Here the meanings of

$$Count_{illegals} \quad Boundingbox_{width} \quad Boundingbox_{height} \quad Area_{overlap}$$

  were explained in the previous homework.

HOW-TO-DO's

- You are provided with some facilities for drawing in an X-windows environment, these are:

  Some header files: Your top lines of your `hw2.c` file shall contain

```
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/cms.h>
#include <xview/xv_xrect.h>
#include "/shared/courses/ceng140/hw2.h";
```

  An X-window: You will have a graphical X-window poping up when you run your program. **You don't have to do any additional work to create the window**. The window size is fixed to be 800 units in horizontal, 900 units in vertical direction. The **origin of this drawing environment** is at the top-left corner of the window.
  Please note that this is different than the grid description. The grid has its origin at the lower-left corner. While you are drawing you have do to a conversion to get the picture correct!...think about it.

  The grid will have its top-left corner at the point (20,20) [units] of the drawing environment, and will cover a horizontal span of ($256 \times 3 = 768$ units) and again ($256 \times 3 = 768$ units) in the vertical span. This means a grid space of 1 will be represented by 3 units in the drawing environment.

  The bottom 800×132 stripe is where you will print the evaluation.

  From now on we will refer to the distance of a point in the window (the drawing environment), measured in the horizontal left-to-right direction and starting from the origin as the *x-coordinate*; and similarly, in the vertical top-to-bottom direction (starting form the origin) as the *y-coordinate*.

  Functions for drawing: Through the included `hw2.h` file you have (automatically) three functions available:

  `DrawRectangle(int x1,int y1,int x2,int y2)` As the name suggests, this draws a rectangle where the coordinates of two corners which sit on a diagonal (of the rectangle) is given. It is the current pen color which is used for the edges.

`FillRectangle(int x1,int y1,int x2,int y2)` Similar to above, but the interior is filled with the current pen color.

`DrawString(int x,int y,char *text)` At (x, y) places the string which is pointed by **`t`**ext. The start of the string is at the given coordinate. Each character position is about 6 units wide.

`PenColor(int x)` For `x` you can use one of the following colors

`WHITE` (is a define constant for 0),

`BLACK` (is a define constant for 1),

An integer in the range [`2`,`11`] for a color in the range [Light Pink,Dark Red].

The canvas that you draw on is in `WHITE`. You can pick a 'pen' of the provided colors by this function. Until a new 'pen' is picked all drawings is done with that color.

A `main` function: The actual `main()` function is defined in the header file `hw2.h` . At the start of this `main()` function some initializations are performed (you don't have to bother what these are). Then a function which is named as `student_main()` is called. Therefore in this homework

- **DO NOT** define a function with name `main()`.
- **DEFINE** a function with name `student_main()` to do whatever you would normally do in your `main()` function definition.

A `make` file: You are provided with a make file

```
/shared/courses/140/Makefile
```

You shall copy this file to your work directory (do not rename it). Then a simple command as

```
make
```

will perform the compilation and linking with the correct libraries of your `hw2.c` file. The executable will have the name `hw2` and will be placed in your working directory. (*Note that the make file does not take an argument like* `hw2.c`).

- To remove/overwrite some drawing from the canvas simply pick the white/(some color) pen, and redraw the drawing.
- While you develop your program you may need to wait a specific amount of time, for this purpose you can use the `usleep()` function. The argument denotes the time (in microseconds) the program will wait before it proceeds. **DO NOT LEAVE THIS IN THE CODE THAT YOU TURN IN.**

20 GENETIC ALGORITHMS-III

'99 HOMEWORK 3

– I understand...
– And understanding is happiness.

Rama Revealed
A.C. Clarke

PART III

Problem

In this third homework in the series you will be finding placement solutions for a set of rectangles. The aim is to find a two dimensional distribution of the rectangles such that

- No rectangle overlaps,
- The bounding box is minimal.

In your first homework you have devised an evaluation function and in the second one you have coded a display facility to visualize possible solutions.

This homework is the one in which a GA engine will be coded. The first homework introduced how to realize a GA engine. In the following section a step-by-step construction procedure will be described.

Specifications and How-to-Do's

- The *italic* writings below are similar explanations given in HW1. They are followed by technical explanations.
- Below the term *string* does not refer to the 'string' concept (convention) of the C language. What is meant to be is merely an *sequence (array) of byte values*. So, don't get confused.

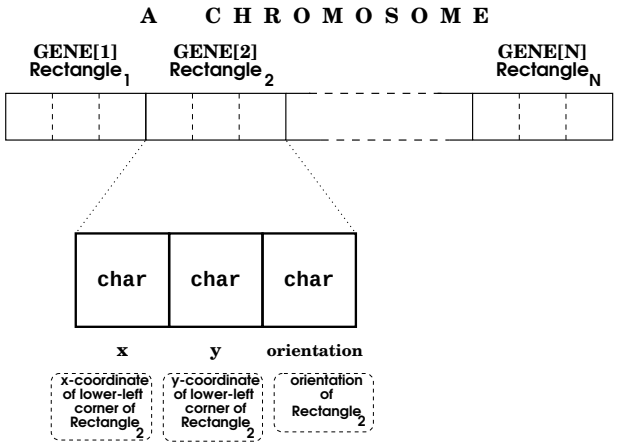
1. *Determine a mapping of the*

   $$\textit{Problem Solution Space} \longmapsto \textit{Space of Binary Strings}$$

   You shall pick an array representation (all allocations shall be done dynamically) in which the i th array element corresponds to the placement of the lower-left corner of the i th rectangle and the orientation information (which is from $\{0, 1\}$ and is explained in HW1).

   ```
   struct gene
         { char x, y, orientation; };
   ```

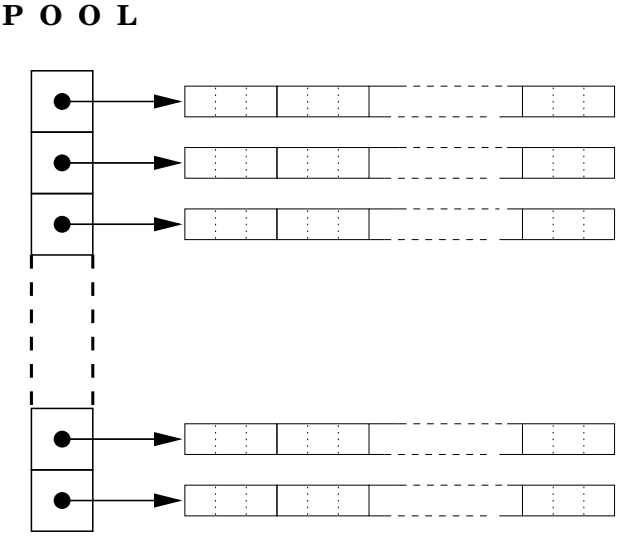
   will serve well as the the array element's structure.



By this you can represent a *chromosome* as a one-dimensional array of `struct gene` and of a size N where N $(N \leq 100)$ will be determined from input (the count of rectangles to be placed).

2. *Create a pool of such chromosomes (usually this is an array of the binary strings)*[4].

   In C this can be easily achieved by a *pointer array* of the pool size.



   The size of the pool shall be 300. You shall use in your program a `#define` constant :

```
#define POOLSIZE 300
```

3. *Randomly (but with an even randomness) fill out the binary strings of $\mathcal{P}$ with initial values.*

   You can make use of the `random()` function, which prototype is defined in `stdlib.h`. To understand the details do : `man random`.

   For your convenience here are two functions that you may use, based on these library functions.

[4] Actually, you will need a second pool to store the child chromosomes at each generation, so it is wise to create **two** such pools right away, we will call these pools $\mathcal{P}$ and $\mathcal{P}'$, respectively.

```
int random1(int n)
{
  return (random()%n);
}

void randomize(void)
{
 srandom((int) (time(NULL)));
}
```

`random1(100)` may return random integer values in the range $[0, 99]$. `randomize()` initialize the *seed* of the random sequence to some arbitrary value (fetched from the real time clock of the system). If you do not `randomize()` the sequence that you will obtain by successive calls to `random1()`, will be the same.

4. *Devise a function* $Eval$

$$Eval \ : \ \textit{Space of Binary Strings} \longmapsto \mathcal{R}$$

so that for each chromosome s *(which encodes a solution),* $Eval(s)$ *returns a real number that corresponds to the degree of how fit* s *is (as being a solution to the world problem).*

This is exactly what you have done as a part of HW2 (based on the work of HW1). The 'weights' which were inputted in HW2 is now something that you will be determining (in order to get good solutions)[5]. Make them into `#define` constants which you define at the top of your program. Write a function `eval_chromosome()` which takes as input a *chromosome* (presumably the pointer to it) and returns the real number that was printed at the lower left corner of the screen of HW2.

5. *Evaluate each member of the pool by means of* $Eval()$ *and keep the 'evaluation' results.*

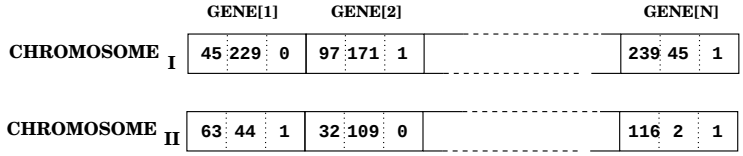
This is quite clear, evaluate each member of the pool by means of this `eval_chromosome()` and keep the 'evaluation' results in an array which is the same size of the pool.

6. *Randomly mate all chromosomes in pairs. Among a pair, at a randomly chosen point perform a crossover (do this by cutting both strings into two, exactly at that chosen crossover point, and then switch the binary segments after the crossover point). So obtain two off-springs.*

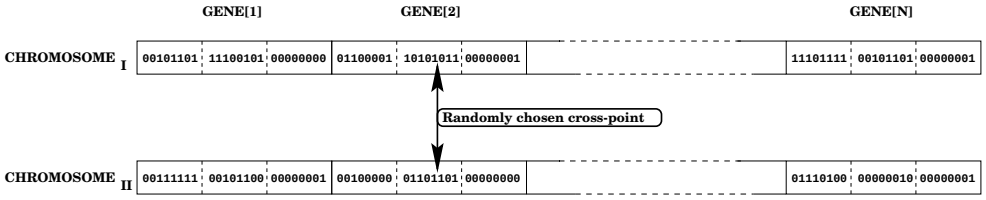
The second pool $\mathcal{P}'$ (the same size and structure of the formerly explained one $\mathcal{P}$) will be filled with the new generated chromosomes. The generation is basically done by crossing existing chromosomes and producing two children per crossed pair. In this generation process you will randomly choose two chromosomes out of the pool, mate them, cross them, hence produce two new chromosomes, and do this until no (unmated) chromosomes remain. To do this, somehow keep an information that they are chosen (to avoid re-choosing an already mated one). You have to device a function `cross_chromosome()` which takes in two (pointers to) chromosomes, and two places (presumably pointers to the places in the second pool) where the result of the crossing will be stored.

[5]This part is extremely important, after you got your program working we want you to experiment with the *genetic engine* and tune it. This is a point where you will put some analytic thought in and try to improve your GA engine's performance. Your grading will be effected by the performance of your engine.

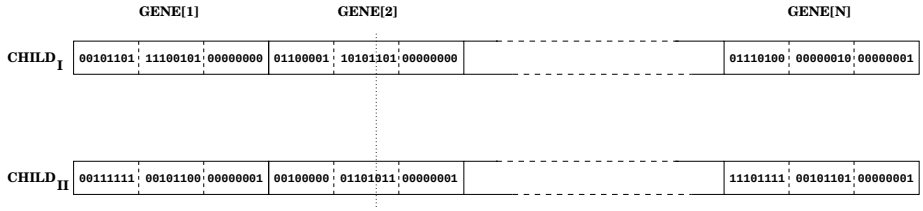
The *crossing* process is somewhat important, so it is worth to spare some words: Assume two mated chromosomes are:



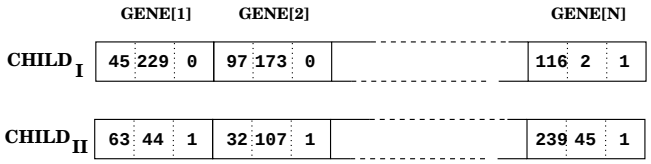
If we take a closer look (at bit level) and also consider a **randomly chosen** crossing point we will see:



As you may observe, it is quite possible that the crossing point will hit some intermediate bits. Your crossing algorithm shall be able to do this type of a random choice and furthermore produce the correct off-springs (children). Here is what is expected for the two example chromosomes crossed at the indicated point:



The numerical values these bits corresponds to, can be visualized as:



At the end of this crossing process, beside the existing chromosomes in the first pool, you have a second pool full of child chromosomes.

7. *Randomly pick some chromosome and mutate.*

   Mutation is extremely simple, it is based on the biological fact that while cross over takes place, randomly errors occur. These errors sometimes help to find better solutions. In your case you will take a number of chromosomes from $\mathcal{P}'$ and change a single bit in there. The probability of having an error is 1 per 2000 bit-crossovers. Making use of the chromosome length you can calculate how many bits have to be mutated:

$$\text{Total no. of bits crossed} = N \times (2 \times 8 + 1) \times \frac{1}{2} \times \texttt{POOLSIZE}$$

So, (Total no. of bits crossed)/2000 bits in the pool $\mathcal{P}'$ have to be mutated. In a `for` loop randomly pick among $\mathcal{P}'$ and *flip* a random bit in the chromosome. Do not keep a record of this, so there is a probability of a mutated chromosome to get once more mutated.

Pay attention not to generate invalid values for the `orientation` fields (do not touch any bit except the least-significant one). This 'sensitivity' does not have to exist in generating invalid placements (so do not worry about the mutations you do on the `x` or `y` fields. [6]

8. *Evaluate all off-springs.*

   Similar to the evaluation process carried out for $\mathcal{P}$, do an evaluation on the chromosomes in $\mathcal{P}'$. You shall keep these results also. It is wise to write a function which takes two arguments in: a pool and a fitness result array (of `double` type elements) and fills out the array (i th element holds the fitness value of the i th chromosome in the pool).

9. *Among the new strings decide who will live and who will die: Replace the least fit n off-springs by the n best fits of the parent (former) strings.*

   This is called *elitism*. Those who were the fittest of the *former* generation gets a right to live for one more generation (cute eh??).

   To do this you have to *sort* the pool according to to the fitness. For your convenience we provide you such a sorting routine:

```
void pool_sort(gene  *pool[],double fitness[], int n)
{
 /* fitness holds result of Eval() with the meaning big number => bad fit */
 /* Sorts in descending fitness order i.e. fitness[0]<fitness[1] .        */
 /* Remember that the (0.0) would be the best fitting: so fitness[0] is the*/
 /* smallest number hence is the best of the pool.                        */
 /* n is the count of elements to be sorted.                              */

 register int i, s; int f;
 gene  *ivaluep, *eltp;
 double elt,ivalue;

 for(i=1; i<n; i++)
        { elt = fitness[i];  eltp = pool[i];
          s = i;
          f = (s-1)/2;
          while(s>0 && fitness[f]<elt)
                { pool[s] = pool[f]; fitness[s] = fitness[f];
                  s = f;
                  f = (s-1)/2; }
          fitness[s] = elt; pool[s] = eltp; }
 for(i=n-1; i>0; i--)
        { ivalue = fitness[i];     ivaluep = pool[i];
          fitness[i] = fitness[0]; pool[i] = pool[0];
          f = 0;
          if (i==1) s = -1;
          else  s = 1;
          if (i>2 && fitness[2]>fitness[1]) s=2;
          while (s>=0 && ivalue<fitness[s])
                 { fitness[f] = fitness[s];    pool[f] = pool[s];
                   f = s;
                   s = 2*f+1;
                   if (s+1 <= i-1 &&  fitness[s] < fitness[s+1])  s++;
```

[6] The reason for this is that the chromosome is punished for an illegal placement by the fitness function anyway, so in whole this bad feature will try to correct itself over the evolution process. This does not exist, however, for the orientation, i.e. the fitness function does not posses any punishment for illegal orientation values.

```
                    if (s > i-1) s = -1; }
               fitness[f] = ivalue;   pool[f] = ivaluep;
          }
}
```

The value of n is something that you shall be determining. Experience shows that this shall be 5-10% of the pool size. Set this value by a `#define` parameter that you shall name as `KEEPCOUNT`, to an appropriate value you decide.

10. *Make the new generation the current generation and continue from step (6) if a termination criteria is not met yet.*

    This is simple, just copy the new generation (pool $\mathcal{P}'$) to the current generation (pool $\mathcal{P}$).[7]

    The termination criteria is reaching a number of iteration (going through this generation-evaluation cycle). This number will be given to you in the input data. It is of `long` type.

11. After every m number of generation you are supposed to call your 'display' routine (making use of HW2) for the best in the pool. Modify your routines to display the *generation count* in the lower-left corner of your screen. The m value will be provided at input. An m value of `0` (zero) means no X-window display of the placement is desired in that run.

12. When the termination criteria is met, output (to stdout) the information stored in the *best* chromosome. The format of the output is explained in the I/O section that follows.

13. The maximal count of rectangles is 100, but this does, by no means, give you the right to make a maximal memory allocation of 100 genes. All bulk memory allocations have to be done dynamically on the principle of *exactly as much as needed*. We will be checking the size of the memory your program is requesting, at run time.

14. You will be given inputs which have solutions.

15. Implementations of other placement methods (other than the described GA) will receive 0 (zero) grade, regardless of their performance (Yes, we will be looking into the code).

16. You shall include necessary header files (like the ones of HW2).

I/O

Input

The input consists of a single line which holds some parameter values of the run followed by maximum of 100 lines, of similar nature, each of which corresponds to a rectangle.

First line:

$Count\_of\_iteration \quad m$

Each of the following lines:

$Rectangle_{width} \quad Rectangle_{height}$

[7] Actually you even don't need to copy it, it is enough to swap the pointers in the pool arrays ($\forall i,\ \mathcal{P}[i] \leftrightarrow \mathcal{P}'[i]$).

Output

First line:

$Boundingbox_{width}$ $Boundingbox_{height}$

Each of the following lines:

x y $orientation$

These lines shall be in the same order of the input, and contain information about the placement of the corresponding rectangle. The values x and y refer to the coordinates of the lower-left corner of the rectangle. $orientation$ is of the same meaning of the last value in any line of the HW1 input.

21 DNA FINGERPRINTING

2000 HOMEWORK 1

Introduction

This time you will be experimenting with the DNA Fingerprinting techniques. You will be writing a program by which you will perform paternity/maternity identification.

What is DNA?

DNA (Deoxyribonucleic acid) is a chemical structure that forms chromosomes. A piece of a chromosome that dictates a particular trait is called a *gene*.

Structurally, DNA is a double helix: two strands of genetic material spiraled around each other. Each strand contains a sequence of *bases* (also called nucleotides). A base is one of four chemicals (adenine, guanine, cytosine and thymine).

The two strands of DNA are connected at each base. Each base will only bond with one other base, as follows: Adenine (**A**) will only bond with thymine (**T**), and guanine (**G**) will only bond with cytosine (**C**). Suppose one strand of DNA looks like this:

A-A-C-T-G-A-T-A-G-G-T-C-T-A-G

The DNA strand bound to it will look like this:

T-T-G-A-C-T-A-T-C-C-A-G-A-T-C

Together, the section of DNA would be represented like this:

T-T-G-A-C-T-A-T-C-C-A-G-A-T-C
A-A-C-T-G-A-T-A-G-G-T-C-T-A-G

DNA strands are read in a particular direction, from the top (called the *5′* or "five prime" end) to the bottom (called the *3′* or "three prime" end). In a double helix, the strands go opposite ways:

5′ **T-T-G-A-C-T-A-T-C-C-A-G-A-T-C** *3′*
3′ **A-A-C-T-G-A-T-A-G-G-T-C-T-A-G** *5′*

What is DNA Fingerprinting?

The chemical structure of everyone's DNA is the same. The only difference between people (or any animal) is the order of the base pairs. There are so many millions of base pairs in each person's DNA that every person has a different sequence. Using these sequences, every person could be identified solely by the sequence of their base pairs. However, because there are so many millions of base pairs, the task would be very time-consuming. Instead, scientists are able to use a shorter method, because of repeating patterns in DNA. These patterns do not, however, give an individual "fingerprint," but they are able to determine whether two DNA samples are from the same person, related people, or non-related people. Scientists use a small number of sequences of DNA that are known to vary among individuals a great deal, and analyze those to get a certain probability of a match.

Every strand of DNA has pieces that contain genetic information which informs an organism's development (*exons*) and pieces that, apparently, supply no relevant genetic information at all (*introns*). Although the introns may seem useless, it has been found that they contain repeated sequences of base pairs. These sequences, called *Variable Number Tandem Repeats* (VNTRs), can contain anywhere from twenty to one hundred base pairs.

Every human being has some VNTRs. To determine if a person has a particular VNTR, a *Southern Blot* is performed, and then the Southern Blot is probed, through a hybridization reaction, with a radioactive version of the VNTR in question. The pattern which results from this process is what is often referred to as a DNA fingerprint.

How is DNA Fingerprinting Done?

The Southern Blot is one way to analyze the genetic patterns which appear in a person's DNA. Performing a Southern Blot involves:

1. Isolating the DNA in question from the rest of the cellular material in the nucleus. This can be done either chemically, by using a detergent to wash the extra material from the DNA, or mechanically, by applying a large amount of pressure in order to "squeeze out" the DNA.

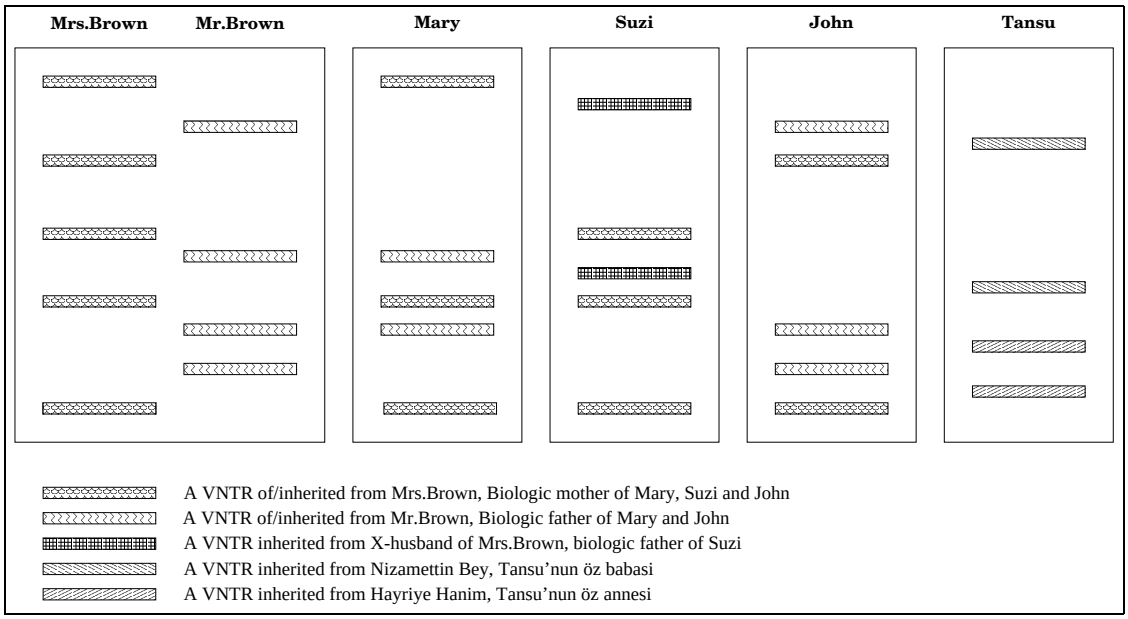
2. Cutting the DNA into several pieces of different sizes. This is done using one or more restriction enzymes.

3. Sorting the DNA pieces by size. The process by which the size separation, "size fractionation," is done is called gel electrophoresis. The DNA is poured into a gel, such as agarose, and an electrical charge is applied to the gel, with the positive charge at the bottom and the negative charge at the top. Because DNA has a slightly negative charge, the pieces of DNA will be attracted towards the bottom of the gel; the smaller pieces, however, will be able to move more quickly and thus further towards the bottom than the larger pieces. The different-sized pieces of DNA will therefore be separated by size, with the smaller pieces towards the bottom and the larger pieces towards the top.

4. Denaturing the DNA, so that all of the DNA is rendered single-stranded. This can be done either by heating or chemically treating the DNA in the gel.

5. Blotting the DNA. The gel with the size-fractionated DNA is applied to a sheet of nitrocellulose paper, and then baked to permanently attach the DNA to the sheet. The Southern Blot is now ready to be analyzed.

A given person's VNTRs come from the genetic information donated by his or her parents; he or she could have VNTRs inherited from his or her mother or father, or a combination, but never a VNTR either of his or her parents do not have. Shown below are the VNTR patterns for Mrs.Brown, Mr.Brown, and their four children: Mary (the Browns' biological daughter), Suzi (Mr.Brown's step-daughter, child of Mrs.Brown and her former husband), John (the Browns' biological son), and Tansu (the Browns' adopted daughter, not biologically related).



Problem

Assume you are given the Southern Blot results for a number of persons. For each person this will be a sequence of distances, the distances of the rendered DNA strands (the parallel lines in the figure above), all measured from the same edge of the paper. In reality the lines have some thickness but we will assume that this is zero. Now, having to hand this sequence of distances for each individual, the problem is to determine the parent-child relations (if any) among these individuals.

Specifications

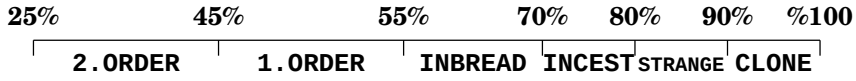
- You will be dealing with at most 100 individuals' data.
- The distance sequence of each individual may contain at most 500 distances.
- A $distance$ is an integer in the range [0,100000] (which suggests that you shall use `long int` representation).
- Your program will read the data from the *standard input*. Each individual's data is given in an input line of the structure:

  $$name \quad n \quad distance_1 \quad distance_2 \quad \ldots \quad distance_n$$

  Where $name$ is a string without any whitespaces. The distances in a line may not be sorted!
  A possible input would be (this is a toy version):

```
Ahmet     7    10067 200 4001 300 3564 78089  6000
Mehmet    8    42 789 456 23005 10083 400 101 1002
Zuleyha   8    2980 458 103 807 67987 2 11 72100
```

- You are expected to print a list of geneticly related pairs. Those who have no relation or a relation of higher order than 2 **will not be printed**. Each pair which is decided to be printed will only be printed once. You will decide among two individuals X and Y (assume X is appearing at the standard input before Y) by counting the matching distance data and dividing this by n_{max} (where n_{max} is the $max(< n \text{ of } X >, < n \text{ of } Y >)$) and, hence obtaining a percentage. Having this percentage to hand you will be using the below given range information to determine the relation among X and Y. If the percentage is less then 25% you shall decide that this pair will not be printed. A percentage at the lower boundary is inclusive (i.e. 45% means `1.ORDER`, as an exception to this 100% is still named `CLONIC`).



The *match* criteria of two distances d and d' is defined as the hold of the following inequality.

$$\frac{|d - d'|}{\max(d, d')} < 0.01$$

- The output will be done to the standard output and consist of lines where each line corresponds to a pair. The structure of a line for a pair of the individuals X and Y is (assuming X appeared at the standard input before Y)

  [$name_X$,$name_Y$] $relation$

  where $relation$ is one of the strings (`2.ORDER`, `1.ORDER`, `INBREAD`, `INCEST`, `STRANGE`, `CLONIC`)

  So a possible three lines of an output would be

```
[Zulfikar,Bahriye] INBREAD
[Mahmut,Mustafa] 1.ORDER
[I-Pedro,II-Pedro] INBREAD
[DOLLY,SALLY] CLONIC
```

  The line order is insignificant. You are free on this.

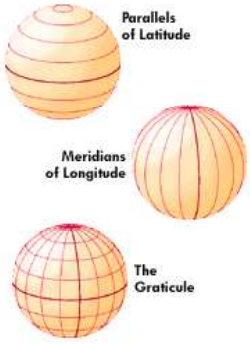
- The program runs and the checking of the results will be performed by a computer program. Therefore it is mandatory to strictly obey the output format specifications. **Do not print additional information, do not attempt to beautify your output**.

22 WHERE TO MEET ON THE GLOBE?

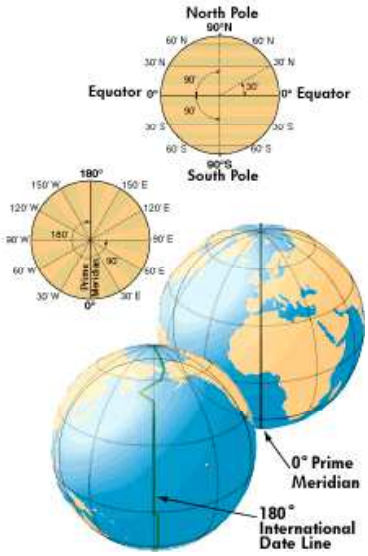
2000 HOMEWORK 2

Introduction

In this homework you will be dealing with position information on our Globe.



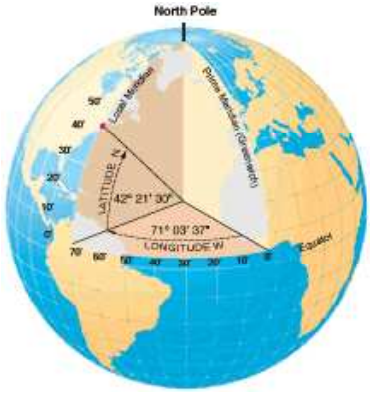
Latitude and longitude form a geographical coordinate system used for locating places on the surface of the earth. They are angular measurements, expressed as degrees of a circle measured from the center of the earth. The earth spins on its axis, which intersects the surface at the north and south poles. The poles are the natural starting place for the graticule, a spherical grid of latitude and longitude lines.



LATITUDE
Halfway between the poles lies the equator. Latitude is the angular measurement of a place expressed in degrees north or south of the equator.
Latitude runs from $0°$ at the equator to $90°N$ or $90°S$ at the poles. Lines of latitude run in an east-west direction. They are called parallels because they are equally spaced.
LONGITUDE Lines of longitude, called meridians, run in a north-south direction from pole to pole. Longitude is the angular measurement of a place east or west of the prime meridian. This meridian is also known as the Greenwich Meridian, because it runs through the original site of the Royal Observatory, which was located at Greenwich, just outside London, England. Longitude runs from $0°$ at the prime meridian to $180°$ east or west, halfway around the globe. The International Date Line follows the $180°$ meridian, making a few jogs to avoid cutting through land areas.

DEGREES, MINUTES, SECONDS

A degree (°) of latitude or longitude can be subdivided into 60 parts called minutes (′). Each minute can be further subdivided into 60 seconds (″). One degree of latitude equals approximately 111 km. Because meridians converge at the poles, the length of a degree of longitude varies, from 111 km at the equator to 0 at the poles (longitude becomes a point at the poles). The diagram at left is an example of a place located to the nearest second. It is written as:
$42°21'30''N \quad 71°03'37''W$
This place is city center, Boston, Massachusetts.



Problem

You will be given a list of places (name and coordinate). Then you will have two tasks to solve.

Task 1: Find that member of the given list (which we will call *the meeting place*) so that, the sum of the flight distances from all the other places to this *meeting place* is minimum.

Task 2: You will be given a number n together with the list. The question now is, which n places shall be removed from the list and which place shall be chosen as the *meeting point* so that the sum of flight distances from the other places is minimum among all such possibilities.

Specifications

- Assume our World is a perfect sphere of circumference 40075.4 km. Use this as the base for all calculations.
- You will be dealing with at most 300 places.
- $1 \leq n \leq 200$.
- For arithmetic calculation use `double` type representation.
- Your program will read the data from the *standard input*.
  The first line contains the n integer, alone. All the following lines are similar and are of the structure:

  $$name \quad xx\texttt{:}xx\texttt{:}xx\mathcal{L}_1 \quad xx\texttt{:}xx\texttt{:}xx\mathcal{L}_2$$

  Where

  $name$ is a string without any whitespaces of maximal length 20.

  $xx\texttt{:}xx\texttt{:}xx\mathcal{L}_1$ is the *latitude* information in which xx stands for degrees, minutes, seconds respectively, and are separated by (`:`) colons. $\mathcal{L}_1$ is one of the letters `N`,`S`.

  $xx\texttt{:}xx\texttt{:}xx\mathcal{L}_2$ is the *longitude* information in which xx stands for degrees, minutes, seconds respectively, and are separated by (`:`) colons. $\mathcal{L}_2$ is one of the letters `E`,`W`.

  The first lines of a possible input would look like (if e.g. $n = 3$):

```
3
ankara 39:55:00N 32:55:00E
dublin 53:20:10N 06:15:15W
```

- Your output will consist of exactly **two** lines.

  First line: Answer to **Task 1** as
  distance in kilometers (two decimal fraction digits) which is the sum of all flights to the *meeting place* followed by the name of the *meeting place* (one space in between).

  Second line: Answer to **Task 2** as
  distance in kilometers (two decimal fraction digits) which is the sum of all flights to the *meeting place* followed by the name of the (new) *meeting place* (one space in between) which is then followed by n number of $name$s which you have decided to remove.

  A possible output would look like (if e.g. $n = 3$):

```
6432.23 konya
4601.00 ankara siirt urfa hakkari
```

- You can find interesting data/info at the following URLs.

```
http://members.aol.com/bowermanb/maps.html
http://www.infoplease.com/ipa/A0001769.html
http://www.indo.com/distance/
```

23 LARGEST COMMON SUBTREE

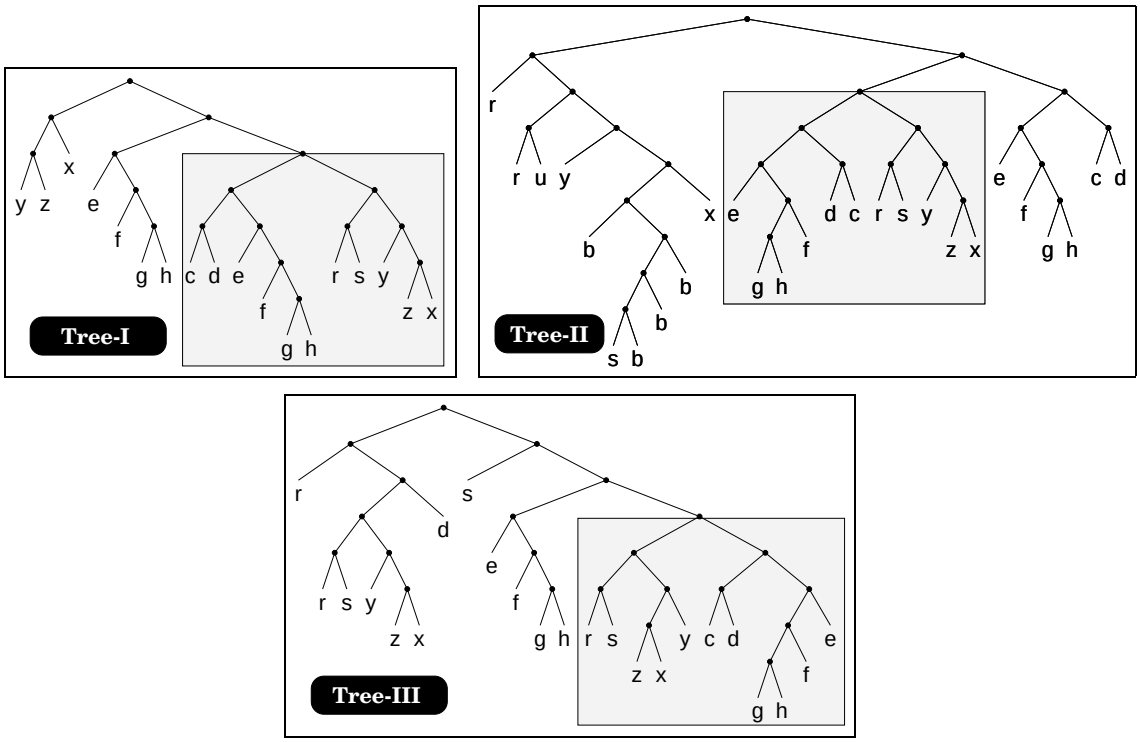
2000 HOMEWORK 3

Introduction

This homeowork is a warm up for pointer usage.

Problem

You will be given a set of binary trees, represented in a linear form (below introduced). The problem is to find the largest sub-tree contained in all of the given trees. For example consider the below given three trees:



The subtree in shaded area is the largest common subtree. The largeness is based on the counts of nodes (terminal+nonterminal). Also note that, in this problem there is no difference in a left or right branching. So, what is drawn as a left or right branch should actually be considered as a sub-branch only (discarding its property of being on the left or right).

Specifications

- A tree is given in the input as a dotted-pair. A dotted-pair, for this problem, is defined in the BNF notation as follows:
  (⟶ has the same meaning as ::=)

$$\langle \text{dotted-pair} \rangle \longrightarrow \texttt{(}\ \langle \text{dotted-pair} \rangle\ \texttt{.}\ \langle \text{dotted-pair} \rangle\ \texttt{)} \mid \langle \text{letter} \rangle$$
$$\langle \text{letter} \rangle \longrightarrow \langle \text{a lower case letter from the english alphabet} \rangle$$

The input that would correspond to **TREE-I**, **TREE-II**, **TREE-III** of above would be:

```
(((y.z).x).((e.(f.(g.h))).(((c.d).(e.(f.(g.h)))).((r.s).(y.(z.x))))))
((r.((r.u).(y.((b.(((s.b).b).b)).x)))).((((e.((g.h).f)).(d.c)).((r.s).(y.(z.x)))).((e.(f.(g.h))).(c.d))))
((r.(((r.s).(y.(z.x))).d)).(s.((e.(f.(g.h))).(((r.s).((z.x).y)).((c.d).(((g.h).f).e))))))
```

Each tree will be provided on a single line of input. There is no restriction on the length of the line. There might be at most 100 trees.

- The output of your program is a single line, which is a dotted-pair, too. It is the dotted-pair that corresponds to the largest subtree. There is no restriction on the left-right ordering of the dotted-pairs you are going to print. A possible output for the above input is:

  ```
  (((c.d).(e.(f.(g.h)))).((r.s).(y.(z.x))))
  ```

  Another one is:

  ```
  (((c.d).(e.(f.(h.g)))).((r.s).(y.(x.z))))
  ```

- If you decide that more than one subtrees are the 'largest' then output any one of them.
- You cannot use arrays to store the trees themselves. An array usage is allowed only to store the root node pointers to the trees.

How to Do's

- represent a node by a structure

  ```
  struct node
       { char letter;
         struct node *left, *right; };
  ```

  You can store, for example, `0` (the zero byte) for the interior (nonterminal) nodes into the `letter` subfield, and the ASCII code for the leaf characters. In addition, you can store also the `left`, `right` branch pointers subfields with `NULL`.

- Get each node memory by a separate `malloc()` call.
- Use `getchar()` to consume from the input. Do not store the whole line of input. You do not need this. The tree data structure can be created while you pass over the input character by character.
- Do use recursion. Both in input-to-data structure construction, the subtree search and the print phase.

24 MENDELIAN GENETICS

2000 HOMEWORK 4

Introduction

This time we will be experimenting with Mendelian Genetics. Any inherited trait such as eye color is referred to as *phenotype*. All phenotypes result from the presence of a specific gene or combination of genes, the *genotype*. In this homework we will assume that a phenotype is related to a single gene value, the *allele*. Each individual carries two (*alleles*) of each gene. But only one of these two, is activated to produce the phenotype. We call that allele the *observed* one and the other the *hidden* one.

Homozygous individuals carry two identical alleles. *Heterozygous* individuals carry two different alleles. In case of dominance, one is called the *dominant* allele and the other the *recessive* allele. A heterozygous individual will have a phenotype of its dominant allele. In this homework it is also possible that there is no dominance defined among two different alleles.

In a sexual reproduction, for each trait, a random selection process takes place. One allele will come from the mother's alleles, the other from the father's alleles. So, an unbiased draw among the mother's alleles for that trait is performed. This yields one of the alleles of the child. Similarly, another draw among the father's alleles will determine the other allele of the child (for that trait). Now if there is a dominance relation defined for these alleles of the child, then the child's trait is determined according to the dominance rule. Otherwise the determination is unbiased random.

Problem

You will be given a set of individuals and an experimentation lab in the form of a pre compiled code (object code). This object code accommodates information about individuals of some population. This information includes

- count of individuals,
- what traits are present,
- what are the possible phenotypes for each trait,
- all dominance relations
- For each individual, what
  - is the gender
  - are the phenotypes

You will be able to retrieve this information by calling some functions. In addition to this you will also be able to perform reproduction by crossing any two individuals, you chose, of opposite sex (provided that you obey the non-incest rule defined below). Each reproduction will cost you some points. This price will be announced later in the news group of this course. For this

moment you shall only concentrate on doing minimal number of reproductions. Each reproduction produces a new individual. Incest of the first and second kind are forbidden. That means (parent,child), (sibling[8],sibling), (uncle/aunt[9],nephew/nice) marriages are disallowed.

Your job is to find out the hidden alleles for those individuals which id-number will be given to you in the input line.

This homework differs from the previous ones. It involves lots of randomness. For some inputs, probably, at a convenient stage, you will make random guesses. To compensate for this, your program will be extensively tested with numerous test data.

You will be provided with a fully functional object code in which the population and the properties of the individuals are fixed. By no means assume that this will be the population that will be used for grading. Also do not waste your valuable time on reverse engineering it.

The path where the object codes (for various Unix systems of the department) are located will be announced in the news group of the course. There will be no DOS or Windows versions!!

Specifications

- Individuals are represented by non-negative, consecutive integers. There will never be a population greater than 100 individuals. The first individual has id-no `0`. All individuals are consecutively numbered. There is no gap in the numbering.
- Traits are represented by non-negative, consecutive integers (starting with zero), too. Though your implementation shall not make a use of it, just for psychological reasons: you can assume that there will be at most 50 traits. And a trait will have at most 40 possible phenotypes.
- Phenotypes are represented by strings which do not contain any whitespace.
- Any population will contain at least one individual of the opposite sex.
- The individuals of the initial population are not paternally/maternally related. So, reproduction among any of them is free. The non-incest rule defined, applies only starting with their children.
- There is no catholic-marriage.
- There is no limitation on the sexual productivity of any individual, also the individuals do not age. Also, they are productive the moment they are born.
- Reproduction is done by calling a function (described below). By calling that function more than once (with the same arguments), you can create siblings. This can be done any time (by this we mean after the first child, any of the parents can have other affairs!, and then, again, produce another child).
- If there is a freedom in any response, the action of the object code is random, that means exactly the run of the same code may produce different results. Your programs will be tested this way.

[8] We define sibling as having at least one parent in common

[9] A sibling of any parent. Note that this definition excludes the wives/husbands of them

- `main()` is defined in the object code. You will be defining `student_main()` exactly the same way you would define `main()`. The `main()` which is defined in the object code will be calling your `student_main()`.
- Functions of the object code are as follows:

  `int get_initial_pool_size(void)`
  returns the count of individuals in the initial pool.

  `int get_trait_count(void)`
  returns the count of traits.

  `phenotype *get_phenotypes(int trait)`
  returns a pointer to the first (`0` th) element of an array which has `phenotype` as elements. This array will have `get_trait_count()` many elements. `phenotype` is a user defined type and holds information about a single phenotype. It is defined as

  ```
  typedef struct phenotype
           {
              char *name;
              int dominance;
           }
          phenotype;
  ```

  the `name` points to a string (something like `"brown"`), namely to the name of a phenotype. `dominance` is a an integer which tells you the dominance weight of this phenotype. If another phenotype for this trait (some other array element) is `"blue"` and has as `dominance=1` where `"brown"` was =`3` then if in an individual `"brown"` and `"blue"` appears as alleles, then `"brown"` will be the *observed* and `"blue"` will be the *hidden* one. If the weights are equal then the *observed* one is determined by an unbiased random draw.

  By going over this array you can obtain all phenotypes related to that `trait`. An invalid value for `trait` will yield a `NULL` pointer return value.

  `individual *get_individual(int n)`
  returns a pointer to a memory location which holds the information of the individual. This pointer and the structure pointed by it is not unique for each individual. So, you shall make a copy of the content which is pointed by that pointer if you will need it later. On illegal `n` value `NULL` is returned. The user defined type `individual` is defined as:

  ```
  typedef struct individual
            {
              int id_number;
              int gender;
              char **phenotype_names;
            }
          individual;
  ```

  `id_number` is the same as `n`.
  `gender` is `1` for male, `2` for female.
  `phenotype_names` holds a pointer to the `0` th element of an array which has `(char`

*) as elements. The n th element of this array holds a pointer that points to a string, namely to the phenotype of this individual for trait n. By going over this array you can obtain all phenotype information related to that individual. Of course there is no way to obtain both alleles for a trait of the individual (to find that out is the whole purpose of this homework, anyway). What you will have in the `individual` structure is always the allele that caused the *observed* property.

`individual *make_individual(int father, int mother)`
will create a new individual by crossing `father` and `mother`. In case of an illegal cross (e.g. a non existing individual as parent, or homosexual reproduction attempt) the return value is the `NULL` pointer. If the cross is legal, then the *observed* properties of him/her will be returned in a structure which is pointed by the return value. The function will observe dominance relations. All random draws that it will perform (if any) are unbiased.

- All data which are pointed by those pointers which are return values of the functions defined, are volatile[10]. That means, a new function call may overwrite the previous information. This includes all strings as well. So, you shall make copies, if you need them for later use.

  Furthermore, use this data read-only. Do not write to that area. This may have unforeseen effects.

- There will be a time limit for execution of 60 sec.

I/O Specification, Compiling and Running

- **Input** is a single line of id-numbers, separated by one blank at least. You have to find the *hidden* allele values for them.

- **Output** is n lines, where n is the count of id-numbers given in input. Each line corresponds to one individual specified in the input. The line order has to be exactly the same order of the input. Each line will have exactly the same number of strings, separated by one blank. The strings are the hidden allele values of that individual in the trait order. So the first string in the line is the hidden allele for the trait=0; the next is the the hidden allele for the trait=1;...and so on.

- For **compiling** do:

  – Copy the object code and the header file from the place specified in the news group to your working directory with the names

  ```
  mendelab.o
  mendelab.h
  ```

  – In order to let the compiler know about the functions that will come from the object code, include a line into your program as:

  ```
  #include "mendelab.h"
  ```

  – Assuming your C source code is located in the file `hw4.c` you do the compilation by:

[10] not in terms of the `volatile` declaration of C

```
gcc -o hw4  hw4.c mendelab.o
```

This will produce an executable with the name `hw4`.

- To **run** the executable, we have provided a run-time option to control randomization: if you run your program with no command line option as

  ```
  hw4
  ```

  then each run has the same randomization sequence. That means two runs of the same executable will produce the same result. But if you run your program with a command line option

  ```
  hw4  -truerandom
  ```

  each run will use a different randomization sequence and so two runs will (almost) never be the same, as far as the randomization involved.

Interactive Programming Aid

We have provided an additional object code which you can use for the development phase of your homework. It incorporates the functionality of a simple interactive environment, which (when actitivated) waits for simple commands from stdin and outputs information about the state of the population you are experimenting with.

The interactive state is activated by calling the function

```
interactive();
```

at any position of your program. You will be entering an interactive mode, which expects some simple input from you. Commands in this environment start at the first column position and are expressed by single letters, followed by one or more arguments (the count of arguments cannot exceed 16):

P Prints information about individuals in the pool. There are two alternatives either you use it as

`P *`

which prints **all** the population, or you enter the individuals that you are interested in, by their id-number, following the letter `P`. Here is an example:

`P 3 5 8`

The response to the input of above is display like:

```
[  3] female<**+**> {    ela}/ela     {benekli}/benekli {  kepce}/sivri   {  bodur}/bodur   {portlek}/portlek {uzunkuyruk}/uzunkuyruk
[  5] male  <**+**> {  siyah}/yesil   {    duz}/duz     { kivrik}/kivrik  {  sirik}/kisa    {portlek}/gocuk   {topkuyruk}/kuyruksuz
[  8] male  <**+**> {    ela}/cakir   {benekli}/benekli {  sivri}/sivri   {   uzun}/kisa    {  cekik}/gocuk   {catalkuyruk}/kuyruksuz
```

Here `<**+**>` means that the father and mother information is unknown. If this field would be `< 9+12>` this would mean that the father of this individual is `9` and the mother is `12`. The following alleles in the line are the list of all trait values of that individual. As an example,

`{  siyah}/yesil`

means that the surface allele (the one which is observable) is `siyah` and the hidden one (the one that you are after!) is `yesil`.

T Prints information about traits. There are two alternatives either you use it as
`T *`
which prints **all** the traits, or you enter the traits that you are interested in, by their numbers, following the letter `T`. Entering nonexisting trait numbers may cause severe errors.

M Is used for doing reproduction. It is used as
`M` id_{father} id_{mother}
like
`M 5 3`
If it is a valid crossing (noninsest, nonhomosexual) then the child is produced. In this case the output is the observable (phenotypes) of the new individual. Otherwise you get a warning which informs that the reproduction has not taken place.

? Is similar to `M` but does not produce a child. It is merely for asking for the validity of a reproduction.

Q Quits the `interactive()` function. All the state changes that are done interactively are persistent.

The interactive input routine is not *bullet-proof* for strange inputs. Do not temper with it.

Your programs will be evaluted with the interactive environment being removed. Also the internal data structure may get modified. So, rely only on the functions described in the specifications.

25 CHARACTER SEQUENCE

'01 HOMEWORK 1

Introduction

Welcome to the series of Ceng. 140 homeworks. You had a brief introduction to the *stack* concept of Computer Science in the lectures. in this homework you are going to make use of it.

Problem

You will be given a sequence of characters of unknown length which can be expressed in the BNF notation as:

$$
\begin{aligned}
\langle \text{sequence} \rangle &::= \langle \text{sequence} \rangle \langle \text{sequence} \rangle \mid \\
&\quad \texttt{[}\ \langle \text{sequence} \rangle\ \texttt{]}\ \langle \text{number} \rangle \mid \\
&\quad \langle \text{letter} \rangle \\
\langle \text{letter} \rangle &::= \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
\langle \text{number} \rangle &::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle \\
\langle \text{digit} \rangle &::= \texttt{0} \mid \texttt{1} \ldots \mid \texttt{9}
\end{aligned}
$$

We define the transformation $\mathcal{F}$ on a sequence as

$$
\begin{aligned}
\mathcal{F}(S_1 S_2) &= \mathcal{F}(S_1)\mathcal{F}(S_2) \\
\mathcal{F}(\texttt{[}S\texttt{]}n) &= \underbrace{\mathcal{F}(S)\mathcal{F}(S)\cdots\mathcal{F}(S)}_{n \text{ times}} \\
\mathcal{F}(c) &= c
\end{aligned}
$$

where S, S_1, S_2 represents any $\langle sequence \rangle$, c represents any $\langle letter \rangle$ and n represents any $\langle number \rangle$.

A *full reduction* of a sequence means to apply the transformation $\mathcal{F}$ until no application is possible.

Specifications

- You will be reading a single sequence from the standard input and output the **full reduction sequence** to the standard output.
- There is no limitation on the length of the sequence. Furthermore there is no way to know about the length of the sequence until the end of the standard input, namely the end-of-file is encountered.
- For any $\texttt{[}S\texttt{]}n$ case: $n \geq 2$.
- You can assume that any full reduction of $\mathcal{F}(\texttt{[}S\texttt{]}n)$ will end in a sequence of 1000 $\langle letter \rangle$s, at most.
- Except the `main()` function you are **not allowed** to define functions.

- You can define a single one dimension array of `char` type of size 1100 at most, if you do so you have to name this array as `bobby`.
- Also, you are allowed to define a stack array of `int` type, of the size 350 at most, if you do so you have to name this array as `stack`.
- No other array definitions are allowed. Furthermore, since it is not covered by the lectures yet, the use of pointers, dynamic memory allocations, string functions are **disallowed**. Even for input and output you are expected to use `getchar()` and `putchar()`.
- The use of `#define` is restricted to the purpose of naming constants only. The use of `#define` with arguments is forbidden (this is also not covered in the lectures yet).
- You shall assume that the input is error free and fully comply with these **specifications**.
- Do not beautify your output by some messages, new lines, spaces, etc. The output is a single $\langle sequence \rangle$ and nothing else. This has to be strictly followed since the evalution of your program will be automated by means of an evaluation program.

Example

Input:

```
ali[[ba]3nab[al]2]2dedi
```

Output:

```
alibababanabalalbababanabalaldedi
```

26 MASTERMIND

'01 HOMEWORK 2

Introduction

Mastermind is a game almost all of us used to play some time in our past. It is a paper and pencil game played between two individuals. The game is symmetric and the idea is that one side holds a (secret) number of a given number of digits and then the other side proposes numbers in each turn to which s/he gets some well-specified response about the count of digits which are exactly in place and the count of digits which exist in the secret number but are misplaced in the proposal.

Problem

Your problem is to write a program which plays as the 'code breaker' of mastermind. In other words your program, by making 'intelligent' proposals, will guess the number which was 'hold' by the user (the code maker).

The rules of mastermind are as follows:

1. A *digit* is an integer in the range $[0, D]$.
2. A *valid number* is defined to be an N digit number $(N \leq D)$ where the high-most (left-most) digit cannot be 0 and any digit appears only once in the number.
3. The code maker chooses a valid number (we will call it the *secret number*).
4. (At each turn) the code breaker proposes a valid number. Turns are counted (Starting with 1).
5. As response to the proposal, the code maker provides two counts:

   First count (C_{exact})**:** The count of digits of the proposed number that match in place of the secret number.

   Second count $(C_{misplaced})$**:** The count of digits of the proposed number which do exist in the secret number but are not in place.

6. A C_{exact} value of N stops the game and the turn-count is recorded; otherwise the game continues from step (3).

Your program will be limited by some maximal turn-count (Max_{turn}) and some maximal process-time (Max_{time}) (in milliseconds). When these values are exceeded the evaluation program will abort your program. These limiting figures will be made available to your run as a part of the input data. A breach of these limits will result in an abortion of that run, and you will get 0 (zero) points for that run.

Specifications

- The first line of the standard input will contain 4 integers:
  $D\ \ N\ \ Max_{turn}\ \ Max_{time}$
  The meaning of these parameters are explained in the preceding section. Furhermore, $D \leq 9$. The last one, (Max_{time}) shall be read as a `long`, all the others are ok to be read as `int` type. The line ends with an end-of-line.

- Your program is expected to output a single integer, namely your proposal for that turn, followed by an end-of-line.

- In response to this, your program will be able to read two integers form the standart input, seperated by one blank:
  $C_{exact}\ \ C_{misplaced}$
  The line ends with an end-of-line.

- When you read in a C_{exact} value that is `N` that means you have found the secret number, congratulations... Do not print anything, simply stop execution (for example by a call to `exit(0);`). (Do not go into a loop with the expectation of a new problem).

- You can decide at any moment to quit, this will not cause any loss or gain of points. If you decide to do so stop execution.

- You are expected to submit a one page written report that explains your solution approach and algorithm. The due date is the same day as of the electronic submission, but the time is 17:00.

Grading

- Your program will be run with an extensive amount of different data. Each run will be performed with a different data. This data will be the same for all students.
- You will receive a grade which is calculated on the base of your cummulative performance with respect to the best performance of the class.
- Assume you found the secret code N_{bingo} times out of N_{total} runs. Your grade (out of 100) will be calculated as:

$$\frac{Performance_{your}}{Performance_{best}} \times 60 + \frac{N_{bingo}}{N_{total}} \times 30 + \langle \text{report \& code quality} \rangle$$

  Here the $Performance$ of an individual will be calculated as

$$\sum_{\text{All runs}} \frac{Max_{turn} + 1 - Turn\_Count}{Max_{turn} + 1}$$

  $Turn\_Count$ is the turn count at which the secret code is found for that run. If a particular run is quited or aborted due to the limitations, the $Performance$ contribution for that run is calculated as 0 (zero).

Hint

```
man clock
```

27 BALANCING CHEMICAL EQUATIONS '01 HOMEWORK 3

Introduction

The syntax of an unbalanced chemical reaction equation can be defined as:

⟨reaction equation⟩ ::= ⟨side⟩ `->` ⟨side⟩
⟨side⟩ ::= ⟨molecule⟩ | ⟨molecule⟩ `+` ⟨side⟩
⟨molecule⟩ ::= ⟨atom⟩ | ⟨atom⟩ ⟨integer⟩ | ⟨complex molecule⟩ | `(` ⟨complex molecule⟩ `)` ⟨integer⟩
⟨complex molecule⟩ ::= ⟨molecule⟩ ⟨molecule⟩ *mid* ⟨molecule⟩ ⟨complex molecule⟩
⟨atom⟩ ::= ⟨an existing atom of the periodic chart⟩

According to this syntactic definition:

CONSTRUCT	COMPLEX MOLECULE	MOLECULE	ATOM
`Fe`	no	yes	yes
`Gq`	no	no	**no**
`Fe2`	no	yes	no
`FeFe`	yes	yes	no
`CO`	yes	yes	yes
`(C)O`	**no**	no	no
`C(O)2`	**no**	no	no
`(CO)`	yes	yes	no
`(COOH)`	yes	yes	no
`(CO2H)`	yes	yes	no
`(CO3H2S)`	yes	yes	no
`((FeS)2CO)3`	yes	yes	no

Please note that this definition in BNF notation is merely syntax related, nothing about semantics, (ie. whether such a molecule can chemically exist or not) is considered.

We relax the syntax by allowing the existence of any number (possibly zero) of blanks in the reaction equation provided that

- no blank exists in any construct that is also a ⟨molecule⟩
- no blank exists in the mid of the "`->`" sign.

Balancing a reaction equation is to find the coefficients of each '`+`' separated molecule on both sides (if there is only one molecule on one side then the coefficient of that molecule) so that the count of atoms on both sides are equal (and this is true for for all the elements present in the reaction equation).

Problem

You will be given a reaction in the above defined syntax in a single line. Here follows such an example input line.

```
H3PO4+(NH4)2MoO4   + HNO3 ->   (NH4)3PO4(MoO3)12   + NH4NO3     +  H2O
```

When this example is balanced the coefficients read as:

1 H3PO4 + **12** (NH4)2MoO4 + **21** HNO3 -> **1** (NH4)3PO4(MoO3)12 + **21** NH4NO3 + **12** H2O

You are expected to produce the balancing coefficients:

```
1 12 21 1 21 12
```

Of course the set of balancing coefficients is not unique, because the multiplication off all the elements of the set by a constant value will produce another set of balancing coefficients (for that reaction equation). Therefore we introduce the following restriction:

- All of the coefficients will be positive integers with a GCD (Greatest Common Divisor) of 1.

Specifications

- You will be reading a single line from the standard input which is the reaction equation (unbalanced).
- Your output is a single line of single space separated positive integers which are the balancing coefficients for the input. Their order is the molecule order of the input line (see example above).
- The total count of molecules of both sides is ≤ 20.
- No molecule exceeds a length of 400 characters.
- Do not beautify your output. Do not print any additional messages, (pre/pro)ceeding lines. Programs that produce non compliant outputs will be automatically graded as zero.
- You are expected to submit a one page written report that explains your solution approach and algorithm. The due date is the same day as of the electronic submission, but the time is 17:00.
- You can assume that the input is error free and complies with the input definition above. Furthermore, it is guaranteed that there exists a unique solution. Also, no coefficient will be greater then a 2 byte `unsigned int`.
- The chemical reactions that your program will be tested with may be fictive (non existing). Also it is not guaranteed that they are correct as far as electron binding and exchange is concerned. The molecules may at zero charge sum.

How to Do's

- Though the lycee education puts it differently, the best way to do this job is to solve n-1 linear equations in n unknowns (the coefficients).[11] Each equation is due to an atom involved in the reaction and represents the fact that the count of that specific atom on both sides shall be equal. The *restriction* introduced at the end of the **Problem** section above, serves to eliminate the last degree of freedom.

[11] Another reason for this is that conventional ad-hoc techniques (like redox) may not work in our cases due to the last item of the **Specification** section.

- This means getting involved in *matrix inversion*. Keep in mind that you have to work with *fractions* and set up the matrix inversion algorithm based on fractions. You shall set up a `struct` that holds the numerator and denominator and construct the four-operation algebra of it (also consider the GCD removal). While doing so, take into account the overflow possibilities. **Do not choose a floating point representation for the matrix operations**.

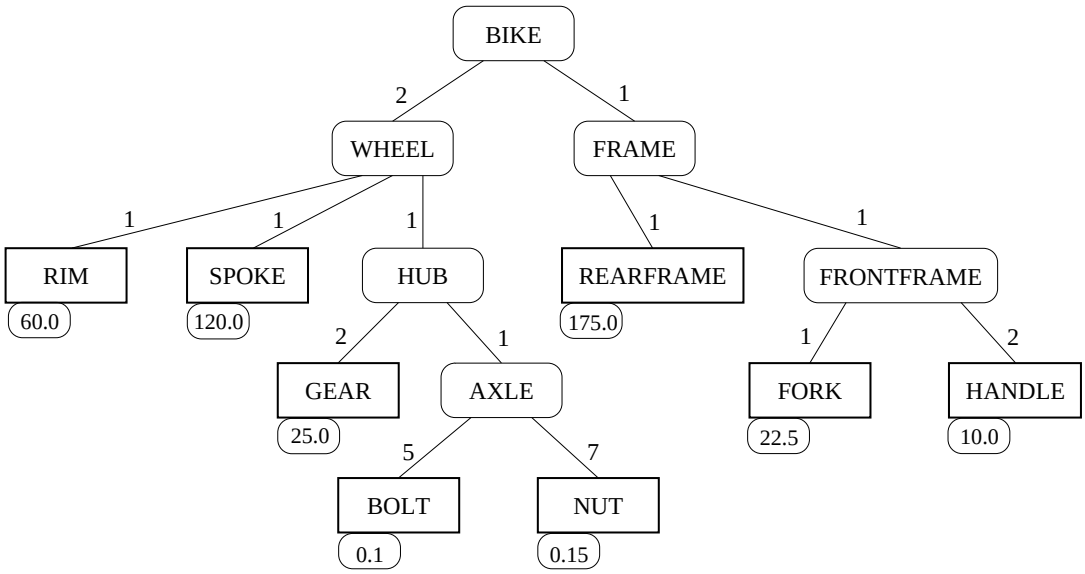
28 PARTS INVENTORY (variant)

'01 HOMEWORK 4

Introduction

In this homework the purpose is to construct a parts inventory.

Suppose we work in a bicycle factory, where it is necessary to keep an inventory of bicycle parts. If we want to build a bicycle, we need to know what is the total cost of the parts. Each part of the bicycle may have sub-parts, for example each wheel has some spokes, a rim and a hub. Furthermore, the hub can consist of an axle and some gears. Let us consider a tree-structured database that will keep the information of which parts are required to build a bicycle (or any other composite object). There are two kinds of parts that we use to build our bicycle (or generally any such composite object). These are assemblies and basic parts. Each assembly consists of a quantity of basic parts and (may be) a quantity of other assemblies. Since it is possible to calculate the price of any composite part, only the unit price for the basic parts are provided. Below you see such a tree:



In this case, for example, a gear has a unit price of 25.0 and you will need 2 such gears to make a hub.

Problem

Your program will read the tree information from the standard input. A possible input that would correspond to the tree example of above would be (see below text for input specifications):

```
BIKE(2*WHEEL(RIM[60.0  ],
             SPOKE[120.],
             HUB(2*GEAR[25.],AXLE(5*BOLT[0.1],    7    * NUT[.15]))),
       FRAME(REARFRAME [175.00],
             1*FRONTFRAME  (FORK[22.5]  ,2  *HANDLE[10.])))
```

(All intendations, spaces and line breakings are optional. The same input could be provided in a single line without any of them inserted)

The expected output is a single floating point:

```
780.6
```

which is the price to build a BIKE.

The input line syntax defined in BNF notation is as follows:

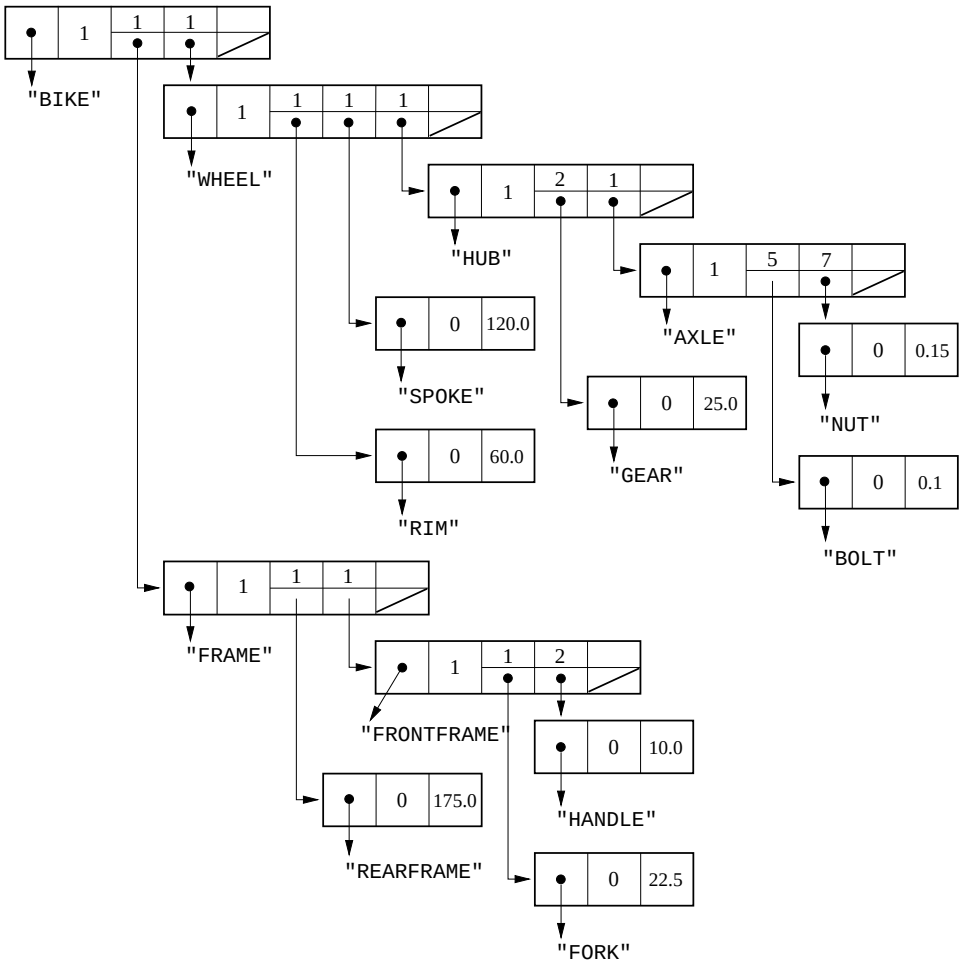
$$
\begin{aligned}
\langle \text{part} \rangle &::= \langle \text{composite part} \rangle \mid \langle \text{basic part} \rangle \\
\langle \text{composite part} \rangle &::= \langle \text{name} \rangle \mid \langle \text{name} \rangle \texttt{(} \langle \text{list of items} \rangle \texttt{)} \\
\langle \text{basic part} \rangle &::= \langle \text{name} \rangle \mid \langle \text{name} \rangle \texttt{[} \langle \text{price} \rangle \texttt{]} \\
\langle \text{item} \rangle &::= \langle \text{part} \rangle \mid \langle \text{quantity} \rangle \texttt{*} \langle \text{part} \rangle \\
\langle \text{list of items} \rangle &::= \langle \text{item} \rangle \mid \langle \text{item} \rangle \texttt{,} \langle \text{list of items} \rangle \\
\langle \text{name} \rangle &::= \langle \text{uppercase letter sequence of maximal length 20} \rangle \\
\langle \text{quantity} \rangle &::= \langle \text{positive integer} \rangle \\
\langle \text{price} \rangle &::= \langle \text{floating point} \rangle
\end{aligned}
$$

We relax the syntax by allowing the existence of any number (possibly zero) of blanks or end-of-lines in the input provided that they do not exists in any *name, quantity* or *price*.

The data structure that your program shall construct on-the-fly is given below. As you see, no constant sized memory structures are used at all. The only assumption that you are allowed to make is that no numerical overflow will occur neither at the input nor in the computations.

Furthermore, things can be more complex then in the example. At any stage, a part (either basic or composite) which is defined at another place of the input can be used (refereed to). Of course if that is the case no description of it will follow the point where it is used. The point of such a usage is definitely not fixed. What we mean is that it is quite possible that some *part* which definition is due to come (but not read in yet) can be refereed.

Let us give an example: Let us assume that in the previous example WHEEL itself is making use two AXLEs and FRONTFRAME is making use of one AXLE (attention: HUB is still making use one AXLE, nothing is changed there) then the input line would read as:




```
BIKE(2*WHEEL(RIM[60.0  ],
             2*AXLE,
             SPOKE[120.],
             HUB(2*GEAR[25.],AXLE(5*BOLT[0.1],    7    * NUT[.15]))),
       FRAME(REARFRAME [175.00],
             1*FRONTFRAME  (FORK[22.5]  ,AXLE, 2  *HANDLE[10.])))
```

Specifications

- The input is a single *part* which will be read from the standard input.
- The output is a single computed *price* of that part which will be printed to the standard output.
- As the example displays very clearly an omitted *quantity* figure means that it is 1.
- You can assume that the input is error free and complies with the input definition above. If a multiple usage of a *part* exists then it is guaranteed that exactly at one and only one point of these usages that part is defined (in terms of its sub parts or price).

Furthermore, it is guaranteed that no part X is making using a part that uses X. In other words there are no cycles in the structure (For your information: if such cycles would exist the resulting structure would not be called as a tree at all).

- No *quantity* figure in the input line will exceed 100.
- No total quantity of any (sub) *part* in any part will exceed the size of a 4-byte `long`.
- You shall use `double` for all floating point data.
- No numerical information is provided about an upper limit for count of items on a *list of items*.
- No numerical information is provided about the depth of the tree. Your program shall not limiting numerical assumptions on these. Of course, due to the nature of the computer infrastructure there are some practical limits.

Hints and Instructions

- You shall make extensive use of the functions defined in `string.h`. Especially: the `str*()` functions and the `*to*()` functions.
- If you do any `realloc()` be careful about consequences. (remember the lecture).
- DO NOT even think of storing the input lines first and then process them all together. This is not possible. The program must have a one-pass-algorithm.
- Use RECURSION.
- For the node representation: Either use unions or define two different structures and perform your own *casting*. If you are not very confident prefer unions.
- Create only ONE node per part. ie. do not create more then one node if a part is multi used. (What are pointers for, any way?). In the second example above (the example in the box), AXLE is multi used. The structure representing AXLE will still be unique. But There will be pointers pointing to it from the structures representing WHEEL, HUB and FRONTFRAME.
- Since your programs are graded using some automated i/o DO NOT beautify the form of the output. Do it exactly as it is shown in the example.

29 BUYING BY INSTALMENT

'02 HOMEWORK 1

Introduction

Welcome to the series of Ceng. 140 homeworks. It is quite common, in these days, to buy some good on the base of monthly instalments. Let us assume the price part that you do not pay in advance, namely the loan is (A). Furthermore let us assume that you have agreed on a payment which will last (n) months. Now what is the mathematical method that determines the amount of montly payment (x). There is one more ingradient in this problem which is the montly interest that you have to pay for the amount that was on loan for each month, we will call this figure (f).

Now let us try to derive the equation which combines all this. For the ease of the calculation we define $\xi = (1 + f)$

The moment you sign the agreement: Your debt is A.

At the end of the first month: Your debt has grown to ξA. You walk into the shop, you pay your montly instalment, which is x, and now the debt, you owe, is reduced to $\xi A - x$. You go back home and enjoy life for one month more.

At the end of the second month: Your debt has grown to $\xi \times (\xi A - x)$. And you pay again the amount of x. So, your dept is now $\xi \times (\xi A - x) - x$ which can be rewritten as $\xi^2 A - x(\xi + 1)$

At the end of the third month: Your debt has grown again by multiplying it by ξ, again you pay x and (after rearrangement) your dept boils down to: $\xi^3 A - x(\xi^2 + \xi + 1)$

... ...

At the end of the n^{th} month: Generalizing, we can easily tell that our debt is:

$$\xi^n A - x(\xi^{n-1} + \xi^{n-2} + \cdots + \xi + 1)$$

which can neatly be written as:

$$\xi^n - x \sum_{i=0}^{n-1} \xi^i$$

Well, but at this point in time my debt should have been reduced to 0 (zero). That was the whole arrangement.

So, we have our equation:

$$\xi^n - x \sum_{i=0}^{n-1} \xi^i = 0$$

The sum in here computes to (see your high school math books)

$$\sum_{i=0}^{n-1} \xi^i = \frac{1 - \xi^n}{1 - \xi}$$

So we have

$$\xi^n - x\frac{1-\xi^n}{1-\xi} = 0$$

Doing the rearrangement we arrive at the following equation:

$$\xi^{n+1} - \left(1 + \frac{x}{A}\right)\xi^n + \frac{x}{A} = 0$$

Specifications

Your problem is to solve numerically this equation, for one unknown, provided that all the others are given. A is not subject to this, it will always be provided. So, you will be writing a program that will solve the equation for the following possibilities:

1. n, x are given, f is the unknown.
2. n, f are given, x is the unknown.
3. x, f are givem, n is the unknown.

Remember that $\xi = (1 + f)$.

Specifications

- The first line contains two numbers, one floating point in the range $[100, 1000000]$, the value of A; and then an integer that corresponds to the question type: one of (1,2,3).
- All values are realistic so no negative values of f, n and x is possible. Furthermore if f is asked for (type 1) the outcome is in the range $(0, 1]$.
- The second line is holding
  - for type 1 question: n and x (seperated by at least one blank).
  - for type 2 question: n and f (seperated by at least one blank).
  - for type 3 question: x and f (seperated by at least one blank).

  n is an integer all others are floating points. Any given x value is in the range $[100, 1000000]$ but an x that you calculate can be greater than this.
- Your output is a single line of a pure number (no remarks please) which is the unknown.
- Do not beautify your output by some messages, new lines, spaces, etc. The output is a single $\langle sequence \rangle$ and nothing else. This has to be strictly followed since the evalution of your program will be automated by means of an evaluation program.
- You shall assume that the input is error free and fully comply with these **specifications**.

'02 HOMEWORK 2

30 PAYMENT PLANING BY SIMULATED ANNEALING

Introduction

A large group of Computer Science (CS) problems are problems in which a solution to a well defined problem is searched for. We call such tasks *search* problems. Finding a solution to a search problem means to determining the value of a set of parameters. The kind of parameters may vary from problem to problem. So parameters may be real numbers, some integers, some discrete identification of some values (like colors, gender, etc). It is also quote possible that the parameters are heterogenous, that means they are not of the same type.

There are various search techniques to tackle these problems.

The first is to consult mathematics for a *analytic* solution. This, if exists is obtainable by solving some formula. Sometimes the solution does not exist in *closed form*, so approximation techniques is used. The area of *Numerical Analysis* is the battle field of this approach.

The second that can come into ones minds is *exhaustive search*, the search we call in which we try systematically all possibilities. Normally, a search problem is too hard to find a solution by trying out all possibilities (all possibilities are named as the *search space* in CS).

The third is a guided tour in the search space. Here at every step you have some candidate solutions, and by a prescription (we call it the *search algorithm*), looking at the candidate(s) you decide either

- to stop, and be satisfied with what you have got,
- or, continue with some new candidates which you pick from the search space, according to your search algorithm, by making use of the candidates you have in your hand.

This last category is full of detailed techniques. Your first homework was an example of this where you had a single candidate in your hand, and at each step you could have picked a better one by a *binary search* algorithm. This technique was applicable because

- there was a single solution,
- the search space was made up of values of a monotonic increasing function (for the given domain).

This time, life will not be so easy. The problem will be defined under the title **Problem**, but before considering it, it is worth to have some introduction to two new search technique (that fall into the third category of above). The following section[12] explains *Hillclimbing* and *Simulated Annealing* running through an example. To make your life easy about understanding it here is the explanation of some terminology:

[12]This section is in great extend pp. 26-29 of the book *Genetic Algorithms+Data Structures=Evolution Programs* by Zbigniew Michalewicz
To the student: The book has no other relevance to the problem that you received in this homework, so don't spend your energy by trying to locate a copy of the book with the intention of finding some 'hints'.

Objective function: It is the formulation of the problem to be solved as a function that quantitatively describes the quality of a solution candidate.

Optimization task: To find the best solution.

Optimum The best solution. Often we speak of a global optimum and some local optimum. The global optimum is the best solution all over the search space. A local optimum is the best solution in the neighborhood of that solution. To define a neighborhood you need to have a measure in the space. So, you can talk about the distance of two members of the solution space. This measure is imposed by you, having a feeling of the 'likeliness' of two candidate solutions. It is a measure where small changes (perturbations) to solution candidates yield in small changes in measure.

Hill climbing: A primitive search technique in which you have a solution candidate at each moment, and to improve it, you search the neighborhood of the candidate for a better solution (a solution that makes you more happy when you plug it in the objective function than the candidate to hand). You proceed to the best of these in the 'neighborhood', dump your old candidate and take that one to be the current candidate.

Simulated annealing: Read the following item about temperature. Still unsatisfied? Proceed to the next section.

Temperature: Simulated annealing as a computational process is patterned after the physical process of *annealing* in which physical substances such as metals are melted (i.e., raised to high energy levels) and then gradually cooled until some solid state is reached. The goal of this process is to produce a minimal-energy final state. Thus this process is one of valley descending in which the objective function is the energy level. Physical substances usually move from higher energy configurations to lower ones, so the valley descending occurs naturally. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function

$$probability = e^{-\frac{\Delta E}{kT}}$$

where ΔE is the positive change in the energy level, T is the temperature, and k is Boltzmann's constant. Thus, in the physical valley descending that occurs during annealing, the probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Thus such moves are more likely during the beginning of the process when the temperature is high, and they become less likely at the end as the temperature becomes lower. One way to characterize this process is that downhill moves are allowed anytime. Large upward moves may occur early on, but as the process progresses, only relatively small upward moves are allowed until finally the process converges to a local minimum configuration.

The rate at which the system is cooled is called the *annealing schedule*. Physical annealing processes are very sensitive to the annealing schedule. If cooling occurs too rapidly, stable regions of high energy will form. In other words, a local but not global minimum is reached. If, however, a slower schedule is used, a uniform crystalline structure, which corresponds to a global minimum, is more likely to develop. But, if the schedule is too slow, time is wasted. At high temperatures, where essentially random motion is allowed, nothing useful happens. At low temperatures a lot of time may be wasted after the final

structure has already been formed. The optimal annealing schedule for each particular annealing problem must usually be discovered empirically.

These properties of physical annealing can be used to define an analogous process of simulated annealing, which can be used (although not always effectively) whenever simple hill climbing can be used. In this analogous process, ΔE is generalized so that it represents not specifically the change in energy but more generally, the change in the value of the objective function, whatever it is. The analogy for kT is slightly less straightforward. In the physical process, temperature is a well-defined notion, measured in standard units. The variable k describes the correspondence between the units of temperature and the units of energy. Since, in the analogous process, the units for both E and T are artificial, it makes sense to incorporate k into T selecting values for T that produce desirable behavior on the part of the algorithm. Thus we use the revised probability formula

$$probability = e^{-\frac{\Delta E}{T}}$$

Where ΔE is nothing else but $f(\mathbf{v}_c) - f(\mathbf{v}_n)$, namely the difference of the current candidate objective function value and the new candidate's.

Thermal equilibrium: Low T value, where the system no more excited to hop around.

Hillclimbing, Simulated Annealing

The search space is a set of binary strings $\mathbf{v}$ of the length 30. The objective function f to be maximized is given as

$$f(\mathbf{v}) = |11 \cdot one(\mathbf{v}) - 150|,$$

where the function $one(\mathbf{v})$ returns the number of `1`s in the string $\mathbf{v}$. For example, the following three strings

$$\begin{aligned} \mathbf{v}_1 &= (\texttt{11011010111010111111011011011}), \\ \mathbf{v}_2 &= (\texttt{11100010010011011001010100011}), \\ \mathbf{v}_3 &= (\texttt{00001000001100100000010001000}), \end{aligned}$$

would evaluate to

$$\begin{aligned} f(\mathbf{v}_1) &= |11 \cdot 22 - 150| = 92, \\ f(\mathbf{v}_2) &= |11 \cdot 15 - 150| = 15, \\ f(\mathbf{v}_3) &= |11 \cdot 6 - 150| = 84, \end{aligned}$$

($one(\mathbf{v}_1) = 22$, $one(\mathbf{v}_2) = 15$, and $one(\mathbf{v}_3) = 6$).

The function f is linear and does not provide any challenge as an optimization task. We use it only to illustrate the ideas behind these two algorithms. However, the interesting characteristic of the function f is that it has one global maximum for

$$\mathbf{v}_g = (\texttt{11111111111111111111111111111}),$$

$f(\mathbf{v}_g) = |11 \cdot 30 - 150| = 180$, and one local maximum for

$$\mathbf{v}_l = (\texttt{00000000000000000000000000000}),$$

$f(\mathbf{v}_l) = |11 \cdot 0 - 150| = 150$.

There are a few versions of hillclimbing algorithms. They differ in the way a new string is selected for comparison with the current string. One version of a simple (iterated) hillclimbing algorithm (MAX iterations) is given below (steepest ascent hillclimbing). Initially, all 30 neighbors are considered, and the one $\mathbf{v}_n$ which returns the largest value $f(\mathbf{v}_n)$ is selected to compete with the current string $\mathbf{v}_c$. If $f(\mathbf{v}_c) < f(\mathbf{v}_n)$, then the new string becomes the current string. Otherwise, no local improvement is possible: the algorithm has reached (local or global) optimum $local = TRUE$. In a such case, the next iteration $t \leftarrow t + 1$ of the algorithm is executed with a new current string selected at random.

It is interesting to note that the success or failure of the single iteration of the above hillclimber algorithm (i.e., return of the global or local optimum) is determined by the starting string (randomly selected) . It is clear that if the starting string has thirteen `1`s or less, the algorithm will always terminate in the local optimum (failure). The reason is that a string with thirteen `1`s returns a value 7 of the objective function; and any single-step improvement towards the global optimum, i.e., increase the number of is to fourteen, decreases the value of the objective function to 4. On the other hand, any decrease of the number of `1`s would increase the value of the function: a string with twelve `1`s yields a value of 18, a string with eleven `1`s yields a value of 29, etc. This would push the search in the "wrong" direction, towards the local maximum.

For problems with many local optima, the chances of hitting the optimum (in a single iteration) are slim.

```
iterated_hillclimber() ← {
                          t ← 0
                          repeat
                                local ← FALSE
                                select a current string v_c at random
                                evaluate v_c
                                repeat
                                      select 30 new strings in the neighborhood of v_c
                                        by flipping single bits of v_c
                                      select the string v_n from the set of new strings
                                        with the largest value of objective function f
                                      if f(v_c) < f(v_n)
                                        then v_c ← v_n
                                         else local ← TRUE
                                until local
                                t ← t + 1
                          until t = MAX
                         }
```

The structure of the simulated annealing procedure is given below.

```
simulated_annealing() ← {
                          t ← 0
                          initialize temperature T
                          select a current string v_c at random
                          evaluate v_c
                          repeat
                                repeat
                                      select a new string v_n at random
                                        in the neighborhood of v_c
                                        by flipping a single bit of v_c
                                      if f(v_c) < f(v_n)
                                        then v_c ← v_n
                                         else if random[0,1) < exp{(f(v_n) − f(v_c))/T}
                                                 then v_c ← v_n
                                until (termination condition)
                                T ← g(T, t)
                                t ← t + 1
                          until (stop-criterion)
                         }
```

The function $random[0,1)$ returns a random number from the range $random[0,1)$. The (termination-condition) checks whether 'thermal equilibrium' is reached, i.e., whether the probability distribution of the selected new strings approaches the Boltzmann distribution. However, in some implementations this repeat loop is executed just k times (k is an additional parameter of the method).

The temperature T is lowered in steps ($g(T,t) < T$ for all t). The algorithm terminates for some small value of T: the (stop-criterion) checks whether the system is 'frozen', i.e., virtually

no changes are accepted anymore:

As mentioned earlier, the simulated annealing algorithm can escape optima. Let us consider a string

$$\mathbf{v}_4 = (\texttt{11100000010011011100101010000}),$$

with twelve 1s, which evaluates to $f(\mathbf{v}_4) = |11 \cdot 12 - 150| = 18$. For $\mathbf{v}_4$ as the starting string, the hillclimbing algorithm (as discussed earlier) would approach the local maximum

$$\mathbf{v}_l = (\texttt{00000000000000000000000000000}),$$

since any string with thirteen 1s (i.e., a step 'towards' the global optimum) evaluates to 7 (less than 18). On the other hand, the simulated annealing algorithm would accept a string with thirteen 1s as a new current string with probability

$$p = exp\{(f(\mathbf{v}_n) - f(\mathbf{v}_c))/T\} = exp\{(7 - 18)/T\},$$

which for some temperature, say $T = 20$, gives

$$p = e^{-\frac{11}{20}} = 0.57695$$

i.e., the chances for acceptance are better than 50% .

Problem

Remember the first homework. Life was easy in those days (which were not so far away). This time our hero(ine) is a person that has his/her living based on a salary. Also has a burning desire to do shopping. Being fascinated of a new installment mechanism that s/he reads in some shop display, s/he reads the announcement, again and again:

- The loan can be paid back in n months, where n is in the range of [3-36].
- It is up to you to make up your pay-back plan. The only restriction is that for any month except the first, your installment has to be in the range of $\pm 50\%$ of your previous month's installment. You are free to choose any value for the first month's installment.
- Each month the debt you owe will be increased by a factor of ξ (yes the good old ξ).
- Come on! you are well again, you are well!

S/he understands the first three but the fourth makes no sense (though looks somewhat familiar).

Being happy like a child, s/he goes home and sits down for calculation. The country s/he lives in is extremely stable. All changes of the forthcoming 3 years period is predictable, and done so by financial experts. So, the monthly bank interests are known for the next 3 years. Due to some 'Banker's Protection Law' deposit accounts may last only one month.

Now, our hero(ine) tries to figure out a pay-plan so that when all what is left over from paying his/her installment is put on a deposit account (for one month, according to the law) and gains some interest with the interest rate know for that month, at the end of n months the *saving* (in the bank) is maximized. Cute, eh?

Your job is to write a program that will take as input n, the monthly bank interests for all the n months, the amount of the loan A, the constant interest rate for the loan f[13], and output n numbers, namely the installments of each month.

As you are very eager to get into the professional life, we will simulate it for you, by:

- If your plan is not *working*, in other words if is not balanced, or proper, ie. at any stage (month) you attempt to pay more than you have in bank, or if you brake the $\pm 50\%$ rule, we will fry you. You get zero.
- If you have a 'working' plan you receive 40 points, then what adds up on this (maximum 60 points) is proportional to your performance, compared to your classmates. According to the following formula:

$$P = 60 \times \frac{Saving_{Your\ program} - Saving_{Worst\ program}}{Saving_{Best\ program} - Saving_{Worst\ program}}$$

  Where $Worst\ program$ is the program of one of your classmates that produces a working plan but the saving is the least among all working plans of the class. Similarly, $Best\ program$ is the one that has the highest saving in the class.

Example

Let us assume the input was as follows:

```
1000000.0 6 500000.0 0.08
0.02 0.09 0.08 0.07 0.05 0.01
```

Furthermore, assuming that the following is a solution you came up with: *(By no means the following is the best plan, it is just an example for a 'working plan')*

```
200000.  150000.  190000.0 260000.0 240000.0 287126.083584
```

You can assume that you pay a month's installment just the last moment of that month (through internet banking) [we will call the moment just before the payment t_1 and the moment after the payment t_2]. And then just a few seconds later it is the first day of the next month an you bank account is automatically receiving your salary for the starting month [this moment in time we mark as t_3]. Furthermore, you can assume that your purchase was exactly done at a t_3 timemark. Now the calculation is done as the following:

Months after purchase: 1

Bank is holding@t_1: $500000 \times 1.02 = 510000$

Debt@t_1: $1000000 \times 1.08 = 1080000$

You pay: 200000

Bank holding@t_2: $510000 - 200000 = 310000$

Bank holding@t_3: $310000 + 500000 = 810000$

Debt remaining@t_3: $1080000 - 200000 = 880000$

[13]This f has nothing to do with the f() of the previous section.

Months after purchase: 2

Bank is holding@t_1: $810000 \times 1.09 = 882900$

Debt@t_1: $880000 \times 1.08 = 950400$

You pay: 150000

Bank holding@t_2: $882900 - 150000 = 732900$

Bank holding@t_3: $732900 + 500000 = 1232900$

Debt remaining@t_3: $950400 - 150000 = 800400$

Months after purchase: 3

Bank is holding@t_1: $1232900 \times 1.08 = 1331532$

Debt@t_1: $800400 \times 1.08 = 864432$

You pay: 190000

Bank holding@t_2: $1331532 - 190000 = 1141532$

Bank holding@t_3: $1141532 + 500000 = 1641532$

Debt remaining@t_3: $864432 - 190000 = 674432$

Months after purchase: 4

Bank is holding@t_1: $1641532 \times 1.07 = 1756439.24$

Debt@t_1: $674432 \times 1.08 = 728386.56$

You pay: 260000

Bank holding@t_2: $1756439.24 - 260000 = 1496439.24$

Bank holding@t_3: $1496439.24 + 500000 = 1996439.24$

Debt remaining@t_3: $728386.56 - 260000 = 468386.56$

Months after purchase: 5

Bank is holding@t_1: $1996439.24 \times 1.05 = 2096261.202$

Debt@t_1: $468386.56 \times 1.08 = 505857.4848$

You pay: 240000

Bank holding@t_2: $2096261.202 - 240000 = 1856261.202$

Bank holding@t_3: $1856261.202 + 500000 = 2356261.202$

Debt remaining@t_3: $505857.4848 - 240000 = 265857.4848$

Months after purchase: 6

Bank is holding@t_1: $2356261.202 \times 1.01 = 2379823.81402$

Debt@t_1: $265857.4848 \times 1.08 = 287126.083584$

You pay: 287126.083584

Bank holding@t_2: $2379823.81402 - 287126.083584 = 2092697.730436$

Debt remaining@t_3: 0

If you provide this plan you are claiming that the last **Bank holding@t_2** figure is maximal (as far as you could have found) with this plan. It is this figure that will be used for grading.

Specifications

- Your input consists of 2 lines, and your output consists of a single line described below.
- The **first input line** contains four numbers

  1st number a floating point in the range $[100, 1000000]$, the value of A;

  2nd number an integer in the range $[3 - 36]$, the value of n;

  3rd number a floating point in the range $[100, 1000000]$, the salary of our hero(ine);

  4th number a floating point in the range $[0.001, 2.0]$, the value of f, which the monthly (fixed) interest rate your loan is subject to. Remember that $\xi = 1 + f$.

  The **second input line** is holding n number of floating points, namely the bank interest rates f_i for the $first$, $second$, $\ldots n^{th}$ month. f_i is in the range [0.001,2.0]
- The output line shall contain n floating points in the range $[0, 50000000]$, which are the installments you decided to pay for the $first$, $second$, $\ldots n^{th}$ month.
- You are guaranteed that **at least** one working plan exists.
- Precision announcements for the first homework are valid for this one also.
- A time limit will be given that restricts the execution time. This limit will be announced in due time, on the 'tin' group.
- You are expected to solve the problem by Simulated Annealing. Fine trimmings on the algorithm is allowed. Your main effort shall go into determining the cooling process (how T is going to change over iterations), and how you define the neighborhood of a candidate solution.

- You can make use of the `random()` function, which prototype is defined in `stdlib.h`. To understand the details do : `man random`.

  For your convenience here are two functions that you may use, based on these library functions.

```
int random1(int n)
{
 return (random()%n);
}

void randomize(void)
{
 srandom((int) (time(NULL)));
}
```

  `random1(100)` may return random integer values in the range $[0, 99]$. `randomize()` initialize the *seed* of the random sequence to some arbitrary value (fetched from the real time clock of the system). If you do not `randomize()` the sequence that you will obtain by successive calls to `random1()`, will be the same.

- Do not beautify your output by some messages, new lines, spaces, etc. The output is specified above and is nothing else. This as to be strictly followed since the evaluation of your program will be automated by means of an evaluation program.

- You shall assume that the input is error free and fully comply with these **specifications**.

31 LOGIC CIRCUIT SIMULATION

'02 HOMEWORK 3

Introduction

This is a replica of the last homework of Ceng 111, this year. So, we expect that you feel some comfort.

The solution can be done in two approaches. We are forcing, though, one of it (the easer approach). You will find a rough analysis in the How-to-do section.

Problem

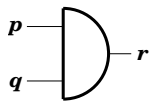
In this homework you will be writing a digital circuit simulator for the i/o (input/output) level. the digital circuit elements that you will be dealing with are forming a small subset of the real world. Furthermore, restrictions are imposed to ease your job.

The circuits that you will be dealing with can contain four types of components:

- AND gate
- OR gate
- NOT gate
- τ-flip-flop

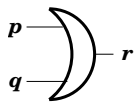
Their schematic representations as well as the functions that describe their actions are expressed as truth tables below:

AND



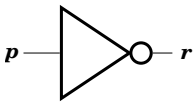
p	q	r
0	0	0
0	1	0
1	0	0
1	1	1

OR



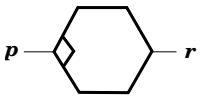
p	q	r
0	0	0
0	1	1
1	0	1
1	1	1

NOT



p	r
0	1
1	0

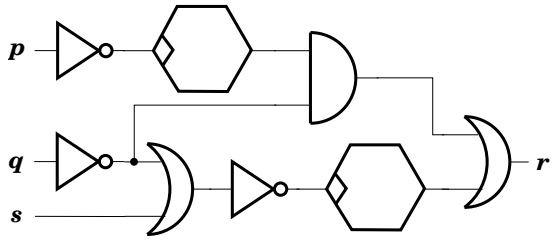
T-FLIP-FLOP



p	$former\ r$	r
0	0	0
0	1	1
1	0	1
1	1	0

The τ-flip-flop has a hidden input which is invoked at the start of each experiment. So, that all τ-flip-flops start a *run* with their $former\ output$ state set to 0.

Below you see a circuit example:



We name a set of input values of a circuit as an *input vector* for the circuit. The input vector has as many elements as the circuit has inputs. A *run* of the simulator is defined as a sequence of input vectors being presented to the inputs.

The purpose of the simulation is to determine what the output of the circuit (in the example r) is after a run.

Note that this output value is not only a function of the last element of the sequence (the last presented input vector). Because the behavior of a τ-flip-flop is depending on its former state of its output. So the circuit exhibits a memory effect. Hence, even if the same input vector is presented to the input for the second time, it is quite possible that the output may not be the same.

Consider the circuit above. Assume the circuit is presented with two input vectors, which are of the form (p, q, s) and are (0,1,0) and (0,1,1), in sequence. We call this a single *run*.

The p value of the first input, a 0, will be inverted by the not gate to a 1 and fed into the τ-flip-flop. Since at the start of a run all τ-flip-flops are set to have a $former\ output$ state of 0, according to the third row of the truth table, the output of that τ-flip-flop will be 1. This 1 is then fed into the and gate, and so on.

The p value of the second input is again a 0. Again it will be inverted by the not gate that is connected to the p input. This 1 will enter the τ-flip-flop as it was in the previous case. But this time the $former\ output$ state is not 0 but is 1. So according to the fourth line of the truth table the output of that τ-flip-flop will be 0 this time. This time this 0 is fed into the and gate which is different then the previous case.

Specifications

- The circuit is guaranteed to be of combinatorial type. That means if you consider a one directional flow of information from the inputs to the output of each circuit element, no loop is formed in this information flow in the whole circuit.
- You are expected to write a program that takes input from standard input and produces either `0` or `1` as the result on the standard output (as explained above). The input lines are as follows:

  First line: The input variables.
  For the example above: `p q s`

  Second line: A sexpr like list expression that describes the circuit. [14]
  For the example circuit above this would be:

  ```
  (or(and(tflipflop (not p)) zort)(tflipflop (not(or (zort (not q)) s))))
  ```

  There is no limitation on the input line length.

  Remaining lines: Each line holds an input vector where the first of these lines is the input vector that is first to be applied. There is no limitation on the count of such lines. Practically it will terminate by the detection of the EOF.
  A possible example (with two input vectors) for the circuit above:

  ```
  0 1 0
  0 1 1
  ```

[14] Space in an sexpr is only required between two atoms, furthermore, the use of more then one adjacent spaces is equal to one space.

Note that there are no parenthesis around the vectors.

Here the `zort` word is used to label an output of some circuit element. The form of usage and definition is as follows:
A usage of a label (`zort` in the example) in an input position of a circuit element means that there is a wire connection from this point to the point of a circuit element that appears as the second element of a list which has as the first element the same label.
For the example circuit above: The input of the and gate is such a connection. It is labeled with the label `zort`. This label appears also as a part of the sexpr as `(zort (not q))`. This all together, means that there is a wire connection from the circuit output of `(not q)` to the second input of the and gate.

It is possible that the same label is used at several input positions (meaning that all those inputs are connected by wire and share the same signal data). But, it is not possible that such an label refers to more than one outputs. In other words such an label will show up exactly once as the first member of a list. Note that these labels are not given prior to the sexpr line. So, it is a part of your job to find them out.

- On any path of the signal flow that connects any input to the output you are guaranteed to have at most one τ-flip-flop.
- *The output of the program with the above mentioned example inputs will be `1`.*

How-to-Do

- A simple approach is to store only the topology (structure) of the circuit. That means leaving off the simulation values of each element of the circuit (do not create positions to store these values in) and calculate the behavior of the circuit by a function, taking the topology and the input vectors as input. This technique will work provided that we pay attention to (handle) the 'memory effect' of the flip-flop.

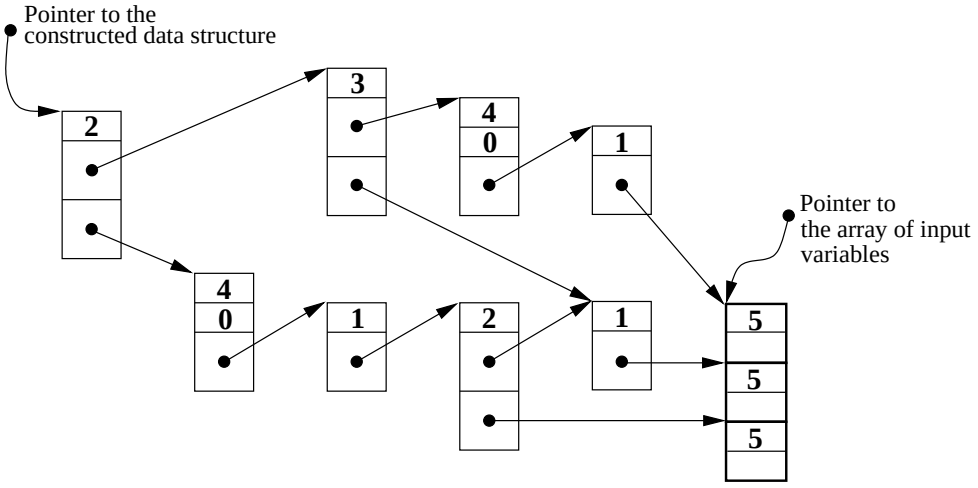
  Though *union*s are usable for the implementation, we advise you not to do so and define the following structures. Then you use type casting to achieve the union-like functionality. So you benefit by having different structure elements of different sizes and not a single entity (the union) of the maximal size of all possible elements.

```
typedef struct {char kind; void *fed_by;} not;
typedef struct {char kind; void *fed_by_1, *fed_by_2;} binary_operator;
typedef struct {char kind; void *fed_by; char former_state} tflipflop;
typedef struct {char kind; char value;} input;

#define NOT       1
#define OR        2
#define AND       3
#define TFLIPFLOP 4
#define INPUT     5

#define KIND(p)                  ( ((not *) p) -> kind )
#define NOT_FED_BY(p)            ( ((not *) p) -> fed_by )
```

```
#define BINOP_FED_BY_1(p)         ( ((binary_operator *) p) -> fed_by_1 )
#define BINOP_FED_BY_2(p)         ( ((binary_operator *) p) -> fed_by_2 )
#define TFLIPFLOP_FED_BY(p)       ( ((tflipflop *) p) -> fed_by )
#define TFLIPFLOP_FORMER_STATE(p) ( ((tflipflop *) p) -> former_state )
#define INPUT(p)                  ( ((input *) p) -> value )
```



Above you see the data structure that corresponds to the example, just after it is constructed. As you might have understood already, the structure `binary_operator` is used to store both the and and or elements' information. The difference is in the value stored of the `kind` subfield (a value of 2 means and, 3 means or)[15].

- Write a parsing function that does a one pass over the sexpr and constructs the data structure. The only problem here is to deal with the *labels*.

  1. For input variables create a dynamically allocated array of `input` type (keep the pointer to this array). So, you have access to the variables both through the data structure you have built, as well as through an array. So, all variables become accessible by simply accessing the members of the array (by indexing).
  2. Create and use (consult) a table for the ⟨label,pointer to element⟩ information.
  3. For the 'used before defined' cases insert a pointer to special structure which can be defined by

     ```
     #define LABEL 6
     #define LABEL_NAME(p)   ( ((label *) p) -> label )
     typedef struct {char kind; char *label;} label;
     ```

  4. Then, later when all of the sexpr input is processed, do a search (presumably recursive) on the data structure you have built, and replace all 'pointers to labels' by the appropriate element (by finding the (pointer to the) actual element by a search for the label in the table).

     In the last form of the data structure no (`label *`) shall exist any more.

[15] In your programming you better use the `NOT`, `OR`,...,`INPUT` macros instead of using `1`, `2`,...`5`, directly.

- After the data structure is built (like the one that is displayed above), you are ready to process the input vectors. First you store the values in the input variables by accessing them as array members and than you evaluate the data structure. You shall do this by calling an evaluation function. This function will of course be recursively defined. Starting with the pointer that corresponds to the output of the data structure ((r) in the example), calculate the action of the element the pointer is pointing to. The function will *only* do a calculation and change nothing on the structure. But after the calculation for a vector is done, the memory fields of all the τ-flip-flop must be updated. For this case, you must device a mechanism that will change the `former_state` field. Think about it, you will find out that there are several solutions to this problem[16].

[16] If you need you can make minor modifications to the `struct` definitions.

32 MAP EXTRACTION

'02 HOMEWORK 4 [17]

Introduction

One of the problems in robotics is map extraction. A robot simply has two interactions with the bounded environment it is in: *perception* and *action*. It perceives information about a small portion of the environment via its sensors, which could possibly be a camera, a touch sensor, thermal sensor etc. Processing its sensory information, it picks one of the possible actions it can perform, and executes it. For a real robot, this action may be turning to a direction by a certain angle, dashing forward/backward by activating its step motors etc. While it moves in the environment, it processes the environmental information it has been collecting, in order to construct an almost-accurate map of the environment. Since the power resource of the robot is finite, an important problem in map extraction is how to minimize the number of actions while accurately constructing the map from sensory information history.

Problem Description

- In this homework, you will be dealing with a simplified version of map extraction problem. We will assume that the bounded environment, in which the robot will be navigating, is an $n \times m$ grid world.
- In this world, each cell is either *empty* or *blocked*.

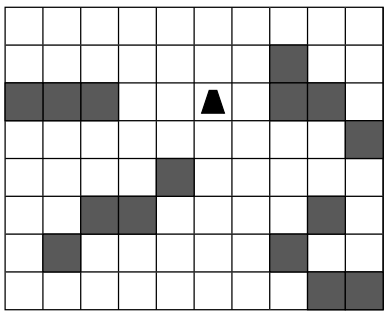


Figure 1: A sample grid world with a robot in one of the empty cells. Shaded cells are blocked.

- A robot is initially placed into a randomly chosen empty cell in this synthetic domain.
- The robot performs actions in order to navigate among the grid cells. Each action takes 1 discrete time step in our world. The robot has the ability to move into 4 directions in every discrete time step: *west*, *east*, *north*, *south*. Initially, time is 0.
- Executing one of these actions, the robot moves one cell towards the chosen direction, provided that the destination cell is an empty cell. If the destination cell is not an empty cell, the action has no effect (i.e. the robot stays at its location).

- Unfortunately, our robot is blind. It can not see anything around; however, it can understand whether the action it performed had been successful or not.
- Our goal is to design a robot that navigates in the grid world in order to build up a map. In other words, the mission of the robot is to come up with a solution which states
  - the dimensions of the rectangular grid world
  - the coordinates of the block cells

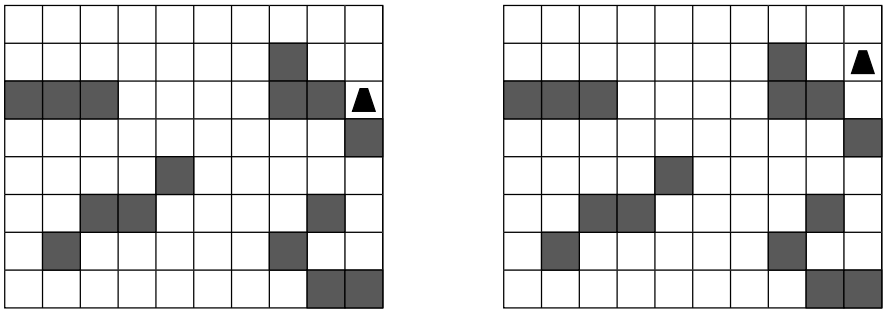


Figure 2: Suppose the robot is positioned as in the left figure. If the robot performs one of the actions *east*, *west* or *south*, its location will not change. If it performs the action *north*, the new location of the robot is shown in the right figure.

Specifications

- The problem stated in the previous section is very well suited to *client-server architecture*.
  - Using UNIX terminology, a *server* process/program (server, in short) is a process/program, which continually waits for information/task (service, in short) requests from other processes/programs, after it starts execution. When such a request comes, it does some computation (related to the domain it is responsible to serve for) inside, prepares a response for that request, and sends this response back to the calling process/program.
  - The process/program that requests service from the server is called a *client* process/program (client, in short).
- For our problem, a server that simulates the grid world environment is available. When starts execution, the server creates a grid world, places a robot randomly to an empty cell, and waits for a client to request service.
- When a client connects to the server, the client has the opportunity to move the robot in the world, and get responses from the server.
  - The client may send to server one of the strings `"e"`, `"w"`, `"n"`, `"s"` representing the actions *east*, *west*, *north*, *south* respectively.

  - Upon receiving this action request from the client, the server tries to move the robot to the requested direction. It increments the time counter by 1.
  - If the action is successful, the server sends back the string `"+"` representing *success* or the string `"-"` representing *failure*.
  - If the server receives an illegal action request, action is assumed to fail, thus it reports back *failure*.
  - Server does not perform any change in robot location unless the client asks for an action to move the robot.

- Server can give service to only one client at a time. You can download the server program through the web address
  `www.ceng.metu.edu.tr/~ceng140/hw4/`
  You will also find help on usage.

- Communication among server and client will be realized through network *sockets*. Information about socket programming is available through web address
  `www.ceng.metu.edu.tr/~ceng140/hw4/`

- As you might expect, you will write the client program that communicates with the server through network, directs the robot actions, and build up the environment map using the responses that come back from the server. Your program will accept 2 command line arguments: an IP address and a port number. For example, if your client program has the name `hw4`; assuming that the server program has started execution on the machine that has the internet address `144.122.71.23` and has opened the socket port number `30000`, you will execute your client by
  `hw4 144.122.71.23 30000`

- Remember that the client you will implement will not have any prior information about the grid world it will try to learn.

- Output of your program will be a sequence of numbers in the following format:

  - The first line will consist of two positive integers separated by one blank, which are the vertical dimension and horizontal dimension of the grid world respectively.
  - The lines following the first line will be the integer coordinates of the blocks. Assuming north-west corner cell of the world is the origin, cell with coordinate (0, 0), each line will consist of two integers separated by one blank: first integer will be the vertical ordinate, and the second integer will be the horizontal ordinate. For example, for the grid world shown in first figure, after a long navigation in the environment, your program should come up with a solution by printing out the following:

```
8 10
1 7
2 0
2 1
2 2
2 7
2 8
3 9
```

```
4 4
5 2
5 3
5 8
6 1
6 7
7 8
7 9
```

  - The output order of coordinates is not important.

- You can assume that, there will not be any unreachable regions in the environment. This may happen if a number of empty cells are surrounded by blocks, for example.
- The limitations in dimensions of the grid world and the number of obstacles are the same with server's limitations. See the server web page for details.
- Your program will be graded according to the following parameters:
  - Code quality; i.e. organization, modularity and indentation,
  - Solution quality; i.e. number of time steps to reach the solution and computational speed,
  - Originality of your solution method
- You should also submit a one page report on your solution strategy (or algorithm), *before* the homework submission deadline. In this report, explain very briefly and clearly what is the method you used to solve the problem.

  ATTENTION: In this homework, it is very important that you *design* your OWN solution method. Do not share ANY information about your solution approach with your friends!

33 STEGANOGRAPHY

'03 HOMEWORK 1

Introduction

Steganography has its origin in the Greek words "stegonos" (invisible, hiding) and "graphos" (graphics, writing style, recording method). It is a technique for transmitting information without disclosing it to any other people.

This idea is in these days adopted for digital information sources. *Digital watermark* is the new name of this old idea. It is now quite common to embed (hide) digital information (either text or image) in some other digital information (usually image or sound). The encoding and decoding is done by a pair of programs (or two subunits of a single program). The *encoder* takes in two inputs ((T) the information that is going to be hidden and another digital information (S) which will hold it) and produces an 'almost' similar digital information (S') which is in hold of T. The key point is that the produced S' is very similar to the original S. The *decoder* takes in S' and produces T out of it.

It is common that S is a digital image. Digital images are stored in a variety of ways. One way is to consider the image as a two dimensional matrix. Such images are called *raster images*. In a raster image each matrix element is holding a visual information (color/gray level/black or white info) which is called a *pixel value*. If an image of $Height \times Width$ is represented by a $N \times M$ matrix, then the matrix element[18] at (i,j) is in hold of the visual information of a rectangle which has its top-left corner placed $\frac{i \times Height}{N}$ down, and $\frac{j \times Width}{M}$ to the left of the top-left corner of the image. The small rectangle which is generated from a single matrix element is a *pixel* of the image. A pixel has only a single visual information.

The *visual information* is a color encoding for a color image; a gray level value (degree of grayness) for a b/w image where gray tones are allowed; or a binary (0/1) information for a b/w (no gray tones allowed) image.

There exist various *color encodings*. One is RGB another is CMYK (others exist as well). Almost all of the encodings are based on the same idea of simulating a color by a linear combination of some fixed colors (frequencies). This can be done because the eye can be tricked as far as colors are concerned. An eye perceive such a combination of colors as some single color (some frequency on the spectrum).

RGB refers to the primary colors of light, Red, Green and Blue, that are used in monitors, television screens, digital cameras and scanners. CMYK refers to the primary colors of pigment: Cyan, Magenta, Yellow, and Black. These are the inks used on the press in "4-color process printing", commonly referred to as "full color printing".

Let us return to the subject of the formats to store raster images. You are probably aware of a variety of such formats. Files, in hold of digital images, bear file extensions that indicate the format the digital image is stored in. Among them you may recall JPG, BMP, TIF, GIF, PPM and may be some others. Some of these also incorporate compression mechanisms designed for images.

In this homework you will be dealing with the plain PPM (ascii) format. PPM is the abbreviation of "portable pixmap". Plain PPM has a very simple structure lay out:

[18] assuming the indexing start a 0

- It starts with the ascii codes `'P'` and `'3'` (this "P3" is called the magic number: a kind of signature that tells this is a "Plain PPM" format)
- Whitespace (any number (≥ 1) of blanks, TABs, CRs, LFs).
- A width M, formatted as ASCII characters in decimal.
- Whitespace.
- A height N, again in ASCII decimal.
- Whitespace.
- The maximum color-component value, again in ASCII decimal.
- Whitespace.
- $Width \times Height$ many pixels, each three ASCII decimal values between 0 and the specified maximum color-component value, starting at the top-left corner of the image, proceeding in normal English reading order. The three values for each pixel represent red, green, and blue, respectively; a value of 0 means that color is off, and the maximum value means that color is maxxed out.
- Characters from a '#' to the next end-of-line are ignored (comments).
- No line should be longer than 70 characters.

Here follows an example

```
P3
#An example image data in ppm format enjoy it.
4 4
15
 0  0  0    0  0  0    0  0  0   15  0 15
 0  0  0    0 15  7    0  0  0    0  0  0
 0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

Problem

In this homework you will be writing a simple digital watermark encoder/decoder. The T information, in our case, is plain ascii code taking values from the set ['A'-'Z','a'-'z',' ','.'] and S is a image in PPM (portable pixmap) format.

You are expected to write two programs, namely `encoder.c` and `decoder.c`. `encoder.c` will be reading a plain PPM image data from its standard input. The text T which is going to be hidden in the image will be given on the second line of the input, as a comment (i.e. prefixed with a # and ended with the EOL character '\n'. The output of this program is PPM encoding of S'. **This PPM output will not have any comment line present (a line starting with a #).**

`decoder.c` is expected to read a plain PPM image data, created by the `encoder.c` program and produce on the standard output the hidden text T.

Various methods, ranging from high quality to low quality, can be used for the 'embedding'. The quality is steaming from the degree of difference of the images S and S'. A minimal difference is preferred and corresponds to a high quality encoding. We quantitatively define this quality as:

$$Quality = \frac{NM}{\sqrt{\sum_{\substack{\text{all pixels} \\ C \in \{red,green,blue\}}} \left(S_{ij_C} - S'_{ij_C}\right)^2}}$$

Your grading will be linearly depending on the quality of your encoding.

Specifications

- You will be dealing with plain PPM image inputs in the context explained in the previous section.
- Image Height and Width values will be in the range [50,500]. These values of an input image cannot be altered in the corresponding output image.
- The input image maximum color component value will be ≤ 254.
- The following will hold for the text T:
  - The text will be provided on the second line of input starting with #.
  - Text may contain upper/lower case letters (from the ascii table), blanks and a dot.

  - Text is consisting of some words of English separated by single blanks and ended with a '.'. The # is not a part of the text but exactly the first character following it, is.
  - The text length (including the dot –which is also a part of the text) is in the range [2,50].
  - The text, though made of words of English, may not have a meaning and may not be grammatically correct.

- The output image will **not** contain any comment line (a line starting with #).
- There will be no direct or indirect information passing, by no means, from the encoder program to the decoder program. This includes files, environment variables, connection to external data sources. Any program doing so will receive zero.
- Your program has to complete in less then 1 sec.
- What you will submit is a tar file. Tar is a program of Unix, used to pack files togather. You have to pack your files by the command:

```
tar -cf hw1.tar encode.c decode.c
```

Grading

- Your program will be run with an extensive amount of different test data, ranging from small to big images. Each run will performed with a different data. This data will be the same for all students.
- For each student, for each run the $Quality$ value will be computed. If your decoder produces the encoded text correctly then you receive, for that run, a grade contribution of

$$\frac{1}{R} \times \left(\frac{Quality_{your}}{Quality_{best}} \times 40 + 60 \right)$$

  Where R is the count of tests. $Quality_{best}$ is the best $Quality$ value observed for that test data among all students.
- If your total grade is less then 30 then your program will be eye-inspected by the TA and judged to receive a grade in the range [Your grade, 30]. This will be final end decisive. No objection will be considered for this judgment.

34 SYMBOLIC DERIVATIVE

'03 HOMEWORK 2

Introduction

In this homework we will be dealing with the differentiation of single variable mathematical expressions.

To ease the pain of parsing a restricted and simplified version of mathematical expression will be considered. Below you will find the BNF description of the syntax of the mathematical expressions we will be dealing with.

$$
\begin{array}{rcl}
\langle \text{expression} \rangle & ::= & \langle \text{expression} \rangle \, \langle \text{operator} \rangle \, \langle \text{expression} \rangle \mid \\
& & \langle \text{expression} \rangle \, \texttt{\^{}} \, \langle \text{natural number} \rangle \mid \\
& & \texttt{(} \, \langle \text{expression} \rangle \, \texttt{)} \mid \\
& & \langle \text{atomic} \rangle \\
\langle \text{atomic} \rangle & ::= & \langle \text{natural number} \rangle \mid \texttt{X} \mid \texttt{sin} \mid \texttt{cos} \mid \texttt{tan} \mid \texttt{sh} \mid \texttt{ch} \mid \texttt{ln} \\
\langle \text{operator} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\langle \text{natural number} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{100}
\end{array}
$$

In addition to this an expression can contain any number of whitespace provided that does not split an $\langle atomic \rangle$. The semantics is as follows

- The single variable X is denoted by the letter `X`.
- `sin`, `cos`, `sh`, `ch`, `tan`, `ln` stand for $\sin(X)$, $\cos(X)$, $\sinh(X)$, $\cosh(X)$, $\tan(X)$ and $\ln(X)$ respectively.
- $\langle expression \rangle$`^`$n$ means $\langle expression \rangle^n$
- Precedence (greater value means performed first) and associativity of the operators are

Operator	Precedence	Associativity
`+ -`	1	left
`* /`	2	left
`^`	3	right

Problem

Your program will read a single expression and will produce the derivative expression (with respect to X) of this expression.

The resulting expression doesn't have to be in the *simplest* form. Furthermore it is allowed to have parenthesis which actually could be omitted. Though, having no more then 20% more then necessary parenthesis will be awarded with an additional 20 points. So, you have the chance to receive 120 points from this homework.

Here is an example,

Input:

```
(X^2-1)*tan - ln^2   /X
```

Output:

```
(X^2-1)*(tan^2+1)+2*X*tan+(2*ln*X-ln^2)/X^2
```

This is quite an elaborated output. Yours does not have to be that advanced. (But if you manage to do so you get bonus!)

```
(((((X^2)-1)*((tan^2)+1))+((2*X)*tan))+((((2*ln)*X)-(ln^2))/(X^2)))
```

is another alternative with some redundant parenthesis present.

Specifications

- You will be reading a single expression from the standard input which is the expression you will take the derivative of. The input may be scattered over lines, contain blanks (not corrupting any atomic) and will be terminated by EOF.
- Your output is the derivative expression on a single line containing no blanks.
- The count of operations in the input is ≤ 20.
- The following derivative table is provided for your convenience:

Expression	*Expression′*
N (*natural number*)	`0`
`X`	`1`
`sin`	`cos`
`sh`	`ch`
`ch`	`sh`
`ln`	`1/X`
$\square_1$ `+` $\square_2$	$\square_1'$ `+` $\square_2'$
$\square_1$ `*` $\square_2$	$\square_1'$ `*` $\square_2$ `+` $\square_1$ `*` $\square_2'$
$\square_1$ `/` $\square_2$	($\square_1'$ `*` $\square_2$ `-` $\square_1$ `*` $\square_2'$)/$\square_2^2$
$\square$ `^`N	N`*`$\square$`^(`N`-1)*`$\square'$

- Any exponent given in the input will be ≥ 2.
- Simplification is not required in this homework. This is so even for operations only involving numbers with one exception: the exponent. Any exponent in your output must be a reduced to a single number.
- No `cos` will be appear in the input. *(The reason for this is to keep the problem simple: we do not have unary minus!)*
- If you do some simplification, do not produce negative numbers, or introduce unary minus signs, this may mislead the evaluation process.
- The correctness of your output expressions will be tested by evaluating the expression at several numerical values of `X`. Therefore, the order and form of your expression is unimportant, since arithmetic obeys the Church-Russer property.

- You are guaranteed that the input is well formed and does not contain any error, complies with these specifications. Therefore, you do not have to bother with any error check.
- Please note that the variable `X` is in uppercase. Do not produce lowercase `x`'s accidentally.

How to Do's

- Implement Dijkstra's algorithm (Phase 1).
- Implement evaluation algorithm (Phase 2), but push on this stack not the values but the reconstructed infix expressions. The best way to do this is pushing on the stack pointers pointing to dynamically created strings that are the infix representations of the sub expressions. When you pop two such subexpression strings from the stack with the intention to join them under an operation and push it back on the stack, do not forget to free the space allocated for the operants, after having done the 'joining'.
- **Golden hint:** It is very vise to run a parallel (shadow) stack during Phase 2, where the derivatives (also in infix form) are kept (generated) in parallel to the infix reconstruction process.

35 MODIFIED QUADTREES

'03 HOMEWORK 3

Regulations

Due date: 13 June 2003, Friday *(Not subject to postpone)*

Submission: Electronically. You will be submitting your program source code through a file which you will name as `hw3.c` by issuing the following single line at the Unix prompt.

```
submit140 hw3.c
```

Resubmission is allowed (till the last moment of the due date), the last will replace the previous, provided you answer the interactive question positively.

For the written part see text.

Team: There is **no** teaming up. The homework has to be done/turned in individually.

Cheating: All parts involved (source(s) and receiver(s)) get zero. **Yes, we definitely mean it!**

I/O Specification: You are expected to write programs that %100 comply with the I/O specifications expressed in this worksheet. Do not beautify you I/O.

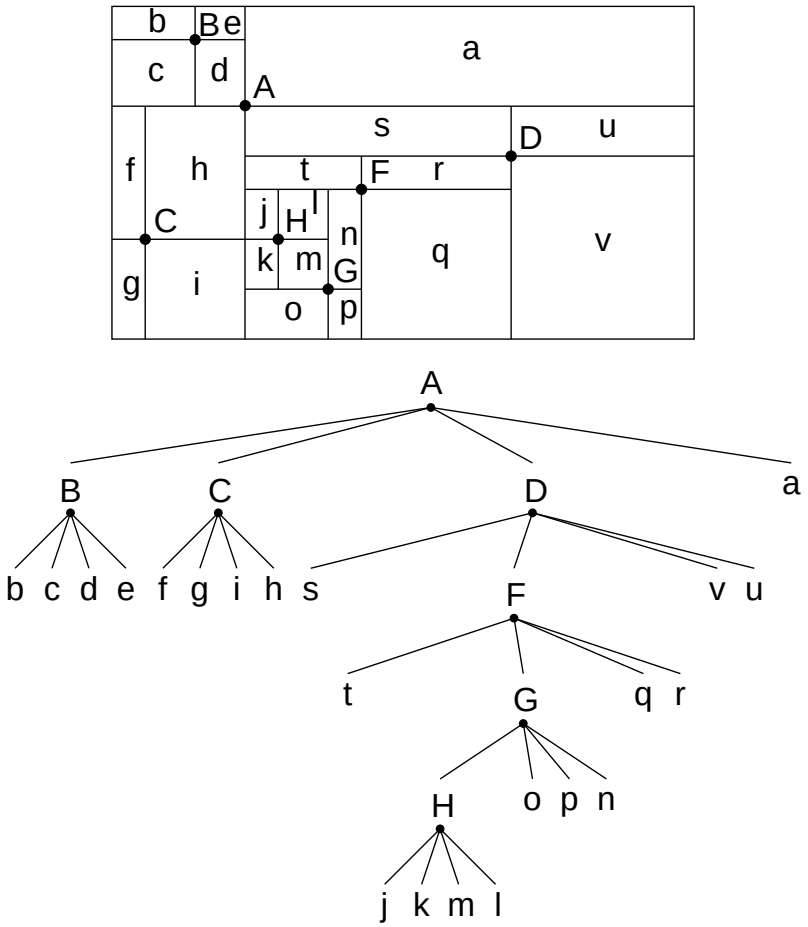
Introduction

This homework is about a modified version of quadtrees (an internet search will not harm but will not do any good neither). For your information only, a quadtree is used in image compression where a rectangular image is subdivided into four equal sized quadrants this process (of dividing a quadrant into further quadrants) is continued until all the pixels in a quadrant is of the same color. This (recursive) division is then represented by a tree where each node is either a terminal node of a single color or a intermediate node that has exactly four children.

Our modified quadtrees differ from this idea by the followings.

- A rectangle is divided into neighboring rectangles, but these do not have to be of equal size.
- Each rectangle (whether subdivided further or not) is labeled by a unique alphabetical string of maximal length 10.
- We are not interested in the colors but the dimensions of all rectangles (that will be the output)

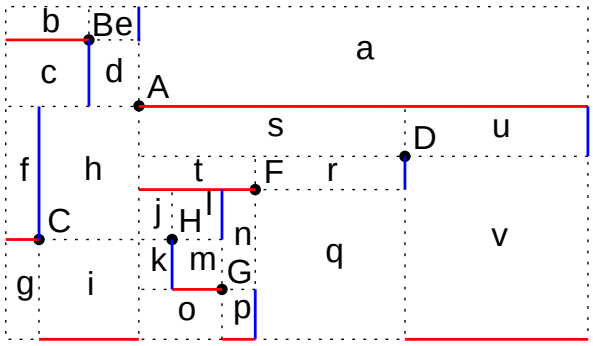
Below you can see such a possible subdivision of a rectangular area, and the quadtree corresponding to it.
(For simplicity we use single character labels in the example, furthermore subdivided rectangles are labeled with uppercase letters these are not part of the specification)



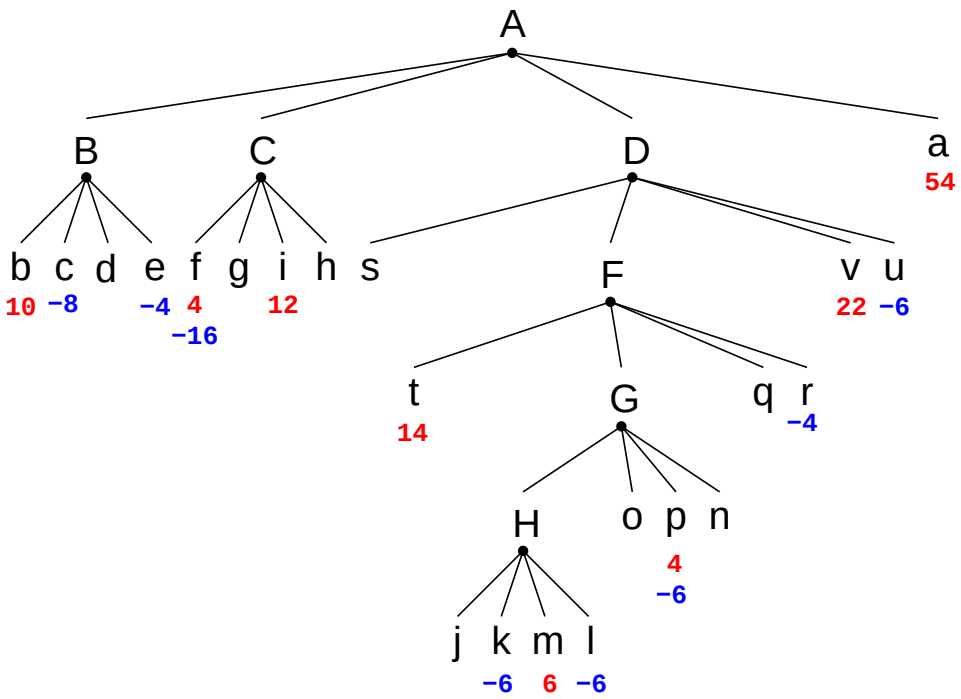
Problem

You will be given a quadtree information in a linear format. The information also includes some width and/or height of the terminals (i.e. rectangles which are no further subdivided). The provided information is assured to be sufficient for the calculation of any dimension of all rectangles.

Assume the following widths/heights are given for the above example.



This will lead to a tree structure of
(unsigned number is a width, negative sign indicates height)



This would be given in an input as:

```
(A (B [b 10] [c -8] d [e -4]) (C [f 4 -16] g [i 12] h) (D s (F [t 14] (G
(H j [k -6] [m 6] [l -6]) o [p 4 -6] n) q [r -4]) [v 22] [u -6]) [a 54])
```

Based on this you are expected to output all the width and height informations. Here are a few lines from the output.

```
⋮
A 70 40
B 16 12
⋮
F 32 22
⋮
a 54 12
```

```
b 10 4
c 10 8
⋮
o 10 6
p 4 6
⋮
```

(The order of the lines is absolutely not important.)

Specifications and How to Do's

- *(For explanation on additional whitespaces see next two items)*

$$
\begin{aligned}
\langle \text{tree} \rangle &::= \langle \text{terminal} \rangle \mid \langle \text{nonterminal} \rangle \\
\langle \text{nonterminal} \rangle &::= \texttt{(}\ \langle \text{label} \rangle\ \langle \text{upper left quandrant} \rangle\ \langle \text{lower left quadrant} \rangle \\
&\quad\ \langle \text{lower right quadrant} \rangle\ \langle \text{upper right quadrant} \rangle\ \texttt{)} \\
\langle \text{terminal} \rangle &::= \langle \text{label} \rangle \mid \texttt{[}\ \langle \text{label} \rangle\ \texttt{␣}\ \langle \text{pinteger} \rangle\ \texttt{]} \mid \texttt{[}\ \langle \text{label} \rangle\ \texttt{␣-}\ \langle \text{pinteger} \rangle\ \texttt{]} \mid \\
&\quad \texttt{[}\ \langle \text{label} \rangle\ \texttt{␣}\ \langle \text{pinteger} \rangle\ \texttt{␣-}\ \langle \text{pinteger} \rangle\ \texttt{]} \\
\langle \text{upper left quandrant} \rangle &::= \langle \text{tree} \rangle \\
\langle \text{lower left quadrant} \rangle &::= \langle \text{tree} \rangle \\
\langle \text{lower right quadrant} \rangle &::= \langle \text{tree} \rangle \\
\langle \text{upper right quadrant} \rangle &::= \langle \text{tree} \rangle \\
\langle \text{pinteger} \rangle &::= \texttt{1} \mid \texttt{2} \mid \ldots \mid \langle \text{upper limit explained in text} \rangle
\end{aligned}
$$

- Two consecutive labels are separated by one whitespace at least.
- The input may contain any number of whitespaces at any place provided that no label or (positive or negative) number is split.
- The input/output is from/to standard input/output.
- All width and heights (calculated or given) are positive integers that can be represented as a 32 bit (`unsigned long int`).
- Labels are case sensitive (as it is in the example).
- The inputs will have at least one nonterminal.
- There is no limitation on the size of the tree. All memory allocations (except a few variables) have to be done dynamically.
- The output is a list of

$$\langle \text{label} \rangle\ \texttt{␣}\ \langle \text{width} \rangle\ \texttt{␣}\ \langle \text{height} \rangle$$

  where both $\langle \text{width} \rangle$ and $\langle \text{height} \rangle$ are $\langle \text{pinteger} \rangle$. All $\langle \text{label} \rangle$ that were present in the input must exist once and only once in the output. Each $\langle \text{label} \rangle$ has to be on a seperate line of the output.
- Your program is expected to have a time complexity of $\mathcal{O}(N \log N)$. Your program will be tested with various sized trees under a time limit that is twice the sufficient amount for a program to run for the largest data, with the given complexity (written by the evaluating team).
- You cannot keep the input. You must consume it by a one pass algorithm that constructs the tree on-the-fly. This algorithm necessarily has to be of a recursive nature (think about it).
- Unless you have a very profound C programming knowledge do not think of alternative data structures other then trees. Though, for efficiency reasons minor modifications to tree structures can be thought of.

36 FOURIER TRANSFORM

'04 HOMEWORK 1

Introduction

The Fourier transform that we will be dealing in this home work has many applications, in fact any field of physical science in which signals can be expressed as a sum of sinusoidal signals with different amplitudes will make use of Fourier series and Fourier transforms.

In our forthcoming homeworks we will have a basic application of the image reconstruction technique behind the (CT) Computerized Tomography or MRI (Magnetic Resonance Imaging) (also known as NMR). The details will be explained in the next homework in the series, but this time (in this homework) we will concentrate on Fourier Transform and the computer realization *Discrete Fourier Transform* (DFT), a mathematical transformation which is in the very core of this technique.

The continuous form of the Fourier transform is:

$$F(\alpha) = \int_{-\infty}^{+\infty} f(x)\, e^{-i\alpha x}\, dx$$

We *discretize* the continuous function $f(x)$ with some constant interval (Δx). so $x \to n\Delta x$. If the sampling is taken at N (equally spaced) points, then $\alpha x \to \frac{2\pi nk}{N}$ and the integral is transformed into a summation over the whole space of samples:

$$F(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n)\, e^{-i\frac{2\pi nk}{N}} \qquad k = 0, 1, \ldots, N-1$$

here of course we redefined F and f over indexed values. (ie. $f(n) \equiv f(n\Delta x)$).

Problem

The problem is to write a program that computes and outputs the $F(k)$ values when the values $f(n), n = 0, 1, \ldots, N-1$ are given as input. Note that $F(k)$ values are normally complex. For your computation you will be making use of the identity:

$$e^{i\xi} = \cos\xi + i\sin\xi$$

Specifications

- The N value will not exceed 1000.
- You will be using `double` precision.
- Input is from standard input and output is to standard output.
- The input consists of a sequence of $f(n)$ values. They will be separated by at least one white space. It is quite possible that the count of white spaces vary. You will not be knowing the actual value of N prior to the input of $f(n)$ values. The end will be detected by an end-of-file at the standard input.

- The output will **only** consist of N lines. The k'th line (we start counting lines at 0) will be containing exactly two `double` values separated by a **single** blank: If $F(k)$ is calculated as the complex number $A + iB$ then the kth line will consist of A followed by B separated by a single blank. (absolutely do not use any other separator).
- The actual test data will not be provided.
- **You are strongly advised to solve the problem on the computer prior to your lab session and also practice to have full control over the editor, compiler, etc.**

37 CONTROLLING ELEVATORS

'04 HOMEWORK 3

Introduction

This time you will be dealing with the control of N number of elevators. As we all have lots of experience on the subject, elevator control can be done always 'more intelligently' (or at least we get such a feeling).

The problem is that the demand on the human side is stochastic, though it may display some characteristic patterns as well. For example, in the morning time people will mostly emerge from their homes and get to the elevator aiming to get to floor 1, where in the evening time it is just the reverse. In the mid of the day the usage intensity will drop, but such a pattern of the demand type will be less observable.

Furthermore, there are some hidden rules: like an elevator going down, shall not change course (due to a pushed button), and go in the reverse direction (even for one level) and pickup the person who pushed the button.

Problem

We will be dealing with a simplified N-elevators problem. In our case the only stochastic behavior will be in 'pressing buttons'. The getting in and getting out of the elevator will be a constant time (which is certainly not so in real life). Furthermore, our elevators will have an infinite load capacity.

Your time is also discretized. That means you will call a function (you have to do so) to get the events of the next time instance (after unit time has passed). Your response is with a set of actions for the elevator motors. If you do not specify an action the previous state of the motor is continued. (Yes, if you do something wrong you may crash the elevator). There are some parameters that you will read from the standard input, prior to running the simulation. These parameters will be about the value of N (how many elevators you have), how many requests you will receive in total, the time (measured in unit time) it takes to move from one level to the other and the (fixed) time spend when you stop to serve a request.

Interactively (interacting by calling at each time step a function) you will learn about the status of new requests and in reaction to this you (your program of course) will decide about the changes of the actions of the elevator motors. Per elevator you will be able to do this by issuing a command that is one of `up`, `down`, `stop`.

Your aim is to minimize the sum of squared times that is spend by individuals that performed a request and get their wish served. In other words, we sum the square of the elapsed times, assuming each individual has a stopwatch that s/he starts when s/he presses a request button and stops when s/he gets out at his/her destination floor. As a programmer our aim is to minimize this sum.

Specification

Your program will start by reading from the standard input the following values (they will be separated by at least one blank):

N**:** A value in the range [1-5] denoting the count of elevators.

R**:** Total count of (external) requests that will be made [1-10000].

L**:** A value in the range [1-200] denoting the count of floors.

t**:** An integer in the range [1-10] denoting the time (measured in unit time) required to move the elevator from one level to the next (either up or down).

d**:** An integer in the range [1-100] denoting the time (measured in unit time) the door will remain open when somebody gets in or out of the elevator.

T**:** Maximal time interval (measured in unit time) in which all servings has to be completed after the last external request is made. This is an integer in the range [1-10000].

There are some rules about the simulation:

- There are no run-away requesters. All requests keep their positions until they are served.
- There are no I-have-changed-my-mind cases among the individuals that entered the elevator car (and hence pressed a button). Everybody will leave the car at the level s/he pressed the stop-at-level-request button for.
- There is no information about the 'count of individuals that made the elevator-call-request'. There can be more than one individuals be waiting at a certain level. A single request for the elevator (we will call this *external request*), may later (when the elevator car arrives at the request point) chain more than one stop-at-level-request button pressing.

  And there is nothing wrong with this. Absolutely each individual will press a stop-at-level-requests for him/herself (we will call these *internal requests*). This is so even if two individuals will have the same destination (the button will be pressed twice).
- It is also possible that while the door is open, an individual arrives, enters the car and immediately presses the button. So, during the d time (including its end point in time) it is possible to receive internal requests. If the the elevator is empty, the door stays open indefinitely. When an individual enters such a car (an empty one), if the the time passed (door-is-open time) is greater or equal to d then the door closes, otherwise the elevator waits (having the door still open) to complete d and then closes it.
- An elevator cannot move when its door is open. Trying to do so is not an error but this command will be ignored.
- The time d is independent of how many individuals gets in or out.
- While the car is moving in a direction (and occasionally stopping on the way), it cannot change this direction until it has served all the stop-at-level-requests in a direction made by the individuals in the car. So if the car was empty and two individuals that have stop-at-level-request that are in opposite different directions entered the car, you can move in

either direction. But if the car is not empty and you have a new comer that has a request which requires to change direction, you cannot. If you decided to move in one direction, you have to serve all individual having the destination in that direction and just then you are allowed to change direction.

If your program orders a move which is a violation of this rule this will be considered an error.

- There is only one external request button on each floor level. If the button is pressed once, until the elevator serves that floor, the effect of consecutive pressings of the same button have no effect. They are not recorded nor any information of such additional pressings is conveyed to the program.
- An individual entering the car immediately presses the stop-at-level-request button (in the car) even if the button was already pressed. All the (new) pressed button information is available in the first following time frame (function call). This includes those buttons that were pressed once and now pressed again (otherwise we could not calculate the sum of squares of times).
- As far as your actions are concerned: If you continue to run the motor of an elevator beyond the boundaries (lower then the lowest floor or higher then the highest floor) this will be considered as an error.
- `In case of an error:` You loose that test case completely. The testing will stop and you will receive from that set of test data a zero. The testing process will continue with the remaining test sets.
- Initially all elevators are resting at floor 1 (the lowest floor).
- The first call to `status()` is marked as $time = 0$. At the next call to `status()` the time passed is 1 unit time. *(What `status()` is is explained below)*
- It is possible that the first `status()` call returns some internal requests.
- It is possible that you decide to stop more than one elevator simultaneously arriving at the same level. In this case all of the people will get into one of the elevator. This elevator will be chosen randomly by the `status()` function implementation. So, you do not have any control over such an event.
- It is possible that while the door is open for a while d people continue to enter the elevator car (without making any external request). But it is for sure that any person entering the elevator car immediately presses his internal request button.
- If the door closes at a level (since a time of d has passed), it is not possible to reopen the door at that level without moving the car. Also note that, the closing of the door does not chain a move up/or down automatically. It is your responsibility to issue exactly at the moment the door closes the up/or down move command. If you fail to give this command the door will close and the elevator car will stay still (until you issue the command). If you issue the command earlier than the door closes, this will not be considered as an error, but will simply be ignored.

The final program that you turn in has to use an external `extern` function that will be named as `status()` and has the following prototype:

```
struct request *status(void);
```

Here `struct request` is defined as:

```
struct request
   { char request_type;
     char elevator;
     unsigned char floor; };
```

Each time `status()` is called this function will:

- Free any heap allocation done in the previous call to `status()`.
- Advance the internal clock by one unit time.
- Fill out a fresh allocated heap array with the struct requests occurred in the last unit time.
- A terminator is stored at the end of this information, namely a `struct request` that has the `request_type` subfield set to 0 (zero).
- Return the pointer to the first element in this new allocated and filled out region.

The meaning of the information stored into a `struct request` is as follows:

`request_type:`

`0:` Array terminator.

`1:` External request button is pressed at the floor level which is indicated in the subfield `floor`.

`2:` A person who has just entered the elevator car has pressed an internal request button indicating his/her destination floor. (This floor information is available in the `floor` subfield.)

`elevator:` An integer value in the range $[1 - N]$ indicating which elevator is in concern. This subfield information is only relevant if the request is an internal one (i.e. `request_type` has value 2).

`floor:` An integer value in the range $[1 - L]$ indicating which floor level is in concern.

As said before, for the same floor level, until that level is served by an elevator, it is not possible to receive two external requests (i.e. `request_type = 1`). But in contrary to this, it is quite possible to receive, (even in the same `status()` call, two identical `struct request`'s indicating the same internal request (that means distinct individuals in that car have the same destination).

Note that the heap region you receive by a `status()` call will be automatically taken away in the next `status()` call. So, if you want to keep some of this information, you have to secure it to a place of your own.

At grading time, your program will be linked with the evaluation program which will provide

- the `main()` function
- the `status()` function

These functions will not be available during the development time, so, for test purposes, you have to write them on your own. **Do not forget to remove those definitions prior you turn in you solution.**

Your program has to provide a function that you will name as `action()`. This function will have the following prototype:

```
void action(char *elevator);
```

At grading time the `main()` written by the evaluator will loop through

- calling `action()` and then
- inspecting the correctness of the action you have stored in the place that is pointed by the argument of action and
- if the action is correct bookkeeping of state and score etc.

You get the power when the `action()` function is called. Presumably your first action will be a call to `status()`. This will advance the time by one time unit, (during this period your (previous) actions are also carried out) and get the new events. The time marks of all these events are this new time. For example, all the events return by the third call of `status()` are assumed to have occurred at time=2. (In the remaining stay of you in `action()`), time is frozen at that point in time. We assume that you will do your own computation then, make some decisions about the motor actions, submit these decisions by storing them into the place which is pointed by the argument (`char *elevator`) of `action()`, and return from `action()`.
This content that you will store will be as follows:

`elevator[0]:` Leave empty. Will not be inspected.

`elevator[1]:` Action for the motor of elevator$_1$.

`elevator[2]:` Action for the motor of elevator$_2$.

⋮

`elevator[`N**`]:`** Action for the motor of elevator$_N$.

Actions are as follows:

`-1:` Continue the previous action.

`0:` Stop the motor.

`1:` Motor moves the elevator car down.

`2:` Motor moves the elevator car up.

As it was stated earlier, you can reverse the move direction of a car if it arrives and/or leaves empty.

For all the individuals that enter at a certain level the waiting time that has passed will be the same value. This value is the time elapsed from the moment the (external) request at that level was made to the moment the car stops and the door opens.

An individuals total time (that will be squared and summed up) is this waiting time + the time spend on the elevator car until the door opens at his/hers destination floor.

After R th external request is received, you have to complete all servings in T unit time. To *complete* means: the very last individual(s) quit(s) at his/her/their destination floor. The evaluation program will keep track of the individuals waiting to be served. When all get served, the evaluation program will detect this and stop calling `action()`, and will grade your performance. You are welcome to finish earlier (not spend all T unit times). If you cannot manage to finish in T unit time (after the last request was made) this will be considered as an error.

After completion, the final state of the elevator motors are unimportant (you don't have to stop them).

Each `action()` call will exactly call `status()` once and only once.

Normally, each `action()` has to return promptly. If you are wasting an eye-observable time: you are doing something wrong. The evaluation program, will consider your program to perform erroneously in this case.

Grading

- Your program will be run with an extensive amount of different test data, ranging from small to big. Each run will be performed with a different data. This data will be the same for all students.
- For each student, for each run the $Quality$ value will be computed. This quantity is defined as:

$$Quality = \frac{S}{\sqrt{\sum_{i=1}^{S} \langle \text{individual's total time} \rangle_i}}$$

  Where S is the count of carried individuals.

  If your program runs correctly you receive, for that run, a grade contribution of

$$\frac{1}{Z} \times \left(\frac{Quality_{your}}{Quality_{best}} \times 40 + 60 \right)$$

  Where Z is the count of tests. $Quality_{best}$ is the best $Quality$ value observed for that test data among all students.
- If your total grade is less then 30 then your program will be eye-inspected by the TA and judged to receive a grade in the range [Your grade, 30]. This will be final end decisive. No objection will be considered for this judgment.

'04 HOMEWORK 4

38 MATH EXPRESSION PARSING & PRINTING

Introduction

In this homework we will be dealing with multivariate infix expressions.

Below you will find the BNF description of the syntax of the expressions we will be dealing with.

$$
\begin{array}{rcl}
\langle \text{expression} \rangle & ::= & \langle \text{expression} \rangle \, \langle \text{operator} \rangle \, \langle \text{expression} \rangle \mid \\
 & & \texttt{(} \, \langle \text{expression} \rangle \, \texttt{)} \mid \\
 & & \langle \text{atomic} \rangle \\
\langle \text{atomic} \rangle & ::= & \langle \text{natural number} \rangle \mid \langle \text{variable} \rangle \\
\langle \text{operator} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\^{}} \\
\langle \text{natural number} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{1000000} \\
\langle \text{variable} \rangle & ::= & \langle \text{letter} \rangle \mid \langle \text{letter} \rangle \, \langle \text{variable} \rangle \\
\langle \text{letter} \rangle & ::= & \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z}
\end{array}
$$

In addition to this an expression can contain any number of whitespace provided that no $\langle atomic \rangle$ is split (this could be expressed also in BNF notation but that would severely injure readability). Precedences (greater value means performed first), associativity and arity of the operators are

Operator	Precedence	Associativity	arity
+ -	1	left	+:nary, -: binary
* /	2	left	*:nary, /: binary
^	3	right	binary

Problem

You are supposed to write two functions that you will name `infix_internal` and `internal_infix`.

`infix_internal` will take a string of a well-formed infix expression of the syntax above and will construct the tree structured internal representation of it and return the pointer to this structure.

`internal_infix` will take a pointer of such an internal structure and form the infix expression as a string. The most important property of this print is that the infix form shall contain only necessary parenthesis.

In the next section strict definitions of the internal structure is given. Certainly when your function `internal_infix` is evaluated it will not necessarily be your `infix_internal` function that will provide the argument (and vice versa). So it is not sufficient to have a lock-key matching between `infix_internal` and `internal_infix` but also a very tight confirmation with the standard of this worksheet.

Specifications

You have to use the following type/structure/macro/prototype definitions:

```
typedef
   struct expression
       { char kind;
         union
            { char *variable;
              long number;
              struct
                  { char operator;
                    struct expression *operand[2]; }  operation;
            } data;
        }
   expression, *EP;

#define VARIABLE  1
#define NUMBER    2
#define OPERATION 3

EP infix_internal(char *expr);
char *internal_infix(EP tree);
```

The content of the union is recorded (of course by you) into the `kind` subfield using the `#define`s above.

The `variable` subfield of the union is a pointer to a string that holds the ascii codes of the letter sequence that make up the $\langle variable \rangle$. Don't forget to make proper allocations for these strings (or use `strdup()`). It is up to you to use the same pointer value for the same $\langle variable \rangle$s. This will not make you gain nor loose any points.

Similarly the `number` subfield will hold an expression that is a $\langle natural\ \ number \rangle$.

If the expression is an operation, the `operator` subfield will hold the operator (so it will be holding one of `'+', '-', '*', '/', ^`). The `operand` array will be holding:

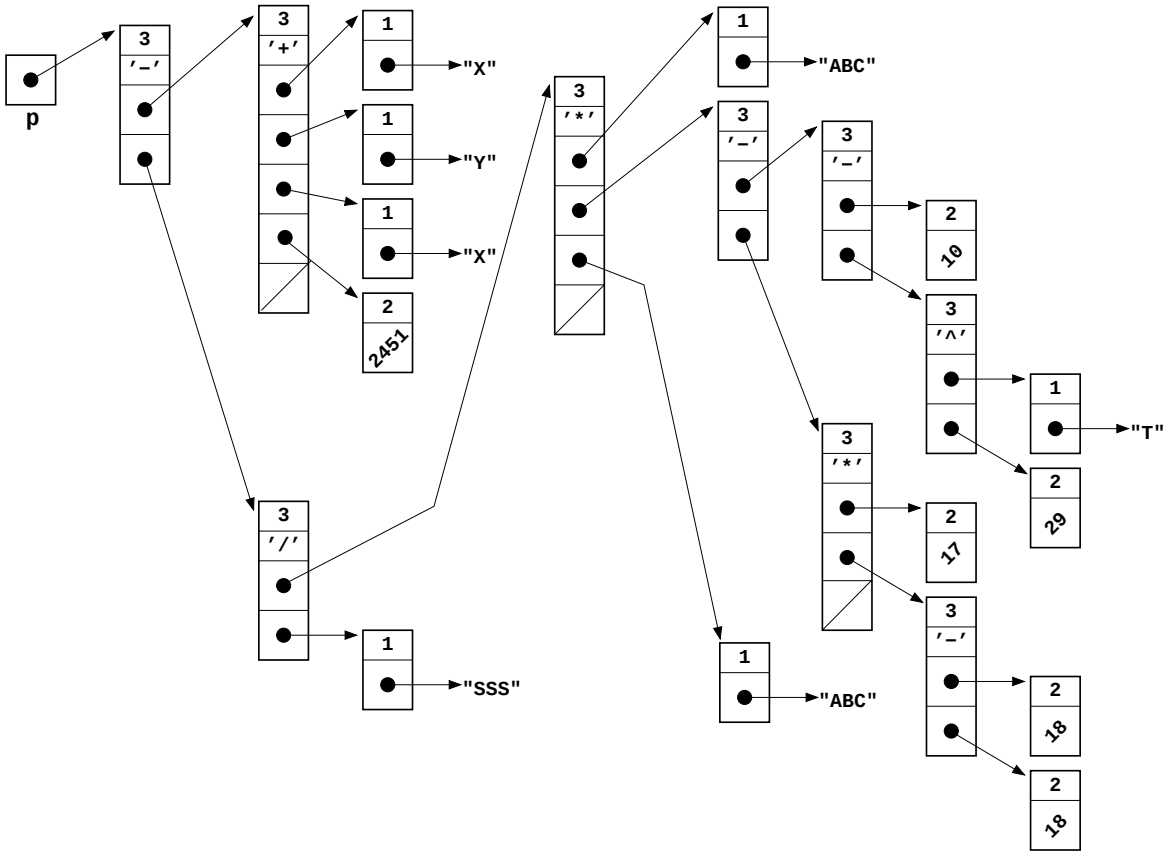
if the operator is binary the pointers to the corresponding internal structure of the left hand side of the operation (in `operand[0]`) and the right hand side of the operation (in `operand[1]`).

if the operator is nary then you implement the trick explained in the lecture: Having allocated the necessary amount of memory for the array, you store the left-most operand of the nary operator into `operand[0]`, the next into `operand[1]`,... up to `operand[n]` in which you store the right-most operand. You have to store a terminator `NULL` into `operand[n+1]`. So, don't forget to make the allocation as big enough to have $n + 1$ proper positions.

Here is an pictorial example. Assume you are calling `infix_internal` as follows:

```
p = infix_internal(
       "(X + Y)+ (X+2451) -ABC*(10-T^29     -(17*(18-18)))*ABC/SSS"
                  )
```

The internal structure `p` is pointing to shall look like:



And the `internal_infix(p)` call shall construct and return the string:

```
"X+Y+X+2451-ABC*(10-T^29-17*(18-18))*ABC/SSS"
```

Note that no spaces are allowed in this constructed string.

You are strongly encouraged to start with Dijkstra's algorithm.

You are also strongly encouraged to write and use macros for each subfield given an `EP`. Your submitted code will be inspected for this.

You are expected to submit a C code under the `hw4.c` name which

- Does not contain a `main` function definition.
- Defines two functions: `infix_internal`, `internal_infix` confirming with the given prototype.
- Is able to separately compile into an object file.

Of course your code can include any count of other functions, global variables, types, pragmas etc. As you can guess, our test program will contain the definition of `main` and will be linked with the object code that is obtained by compiling your program. **Attention: You are expected to submit a C code and not the compilation result (namely the object code), compilation will be done by us at evaluation time).**