Task 1: **Scheme** Cholesky Decomposition **[8 Points]**

The Cholesky decomposition of a real-valued symmetric positive-definite matrix **A** is a decomposition of the form **A=LL**^{T}, where **L** is a lower triangular matrix with real and positive diagonal entries, and **L**^{T} denotes the transpose of **L**.

The following example shows the Cholesky decomposition of a symmetric real matrix.

$$\begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 6 & 1 & 0 \\ -8 & 5 & 3 \end{pmatrix} \begin{pmatrix} 2 & 6 & -8 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{pmatrix}.$$

$$L_{ij} = \begin{cases} 0 & \text{if } i < j, \\ \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2} & \text{if } i = j, \\ \frac{1}{L_{jj}}(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}) & \text{otherwise.} \end{cases}$$

You can compute the (*i*, *j*) entry if the entries to the left and above are known by the above recursive formula.

In Scheme, write a function *(define (get-ij i j A))* which takes row index *i* and column index *j* of a matrix *A* (represented as a list of lists) of varying size and returns the element at position i,j.
[2 Points]

In Scheme, write a function *(define (create_cholesky A))* which takes a rectangular matrix A of varying size and returns its lower-triangular matrix L of its Cholesky decomposition represented as a list.
[6 Points]

Examples:

Input: (create_cholesky '((1 2)(2 5)))

> ((1 0) (2 1))

Input: (create_cholesky (list (list 4 12 -16) (list 12 37 -43) (list -16 -43 98)))

> ((2 0 0) (6 1 0) (-8 5 3))

Task 2: **Scheme** Certificates for Tree Isomorphism **[7 Points]**

Graph certificates are a tool used to determine graph isomorphism, where two graphs are isomorphic if and only if their certificates are equal. In particular, a simple certificate suitable for trees, i.e., undirected connected graphs that have no cycles, is computed as follows:

1. Label all the vertices of the graph G with the string 01.
2. While there are more than two vertices of G do for each non-leaf x of G:
    a. Let Y be the set of labels of the leaves adjacent to x and the label of x, with the initial 0 and trailing 1 deleted from x;
    b. Replace the label of x with concatenation of the labels in Y sorted in increasing lexicographic order, with 0 prepended and a 1 appended;
    c. Remove all leaves adjacent to x.
3. If there is only one vertex left, report the label of x as certificate.
4. If there are two vertices x and y left, then report the labels of x and y, concatenated in increasing lexicographic order, as the certificate.
5. Otherwise continue with the step 2.

The goal of this task is to implement a function computing this certificate. You will be given a tree-type graph whose vertices are labelled with the string 01. It will be represented as a list of vertices, where each vertex is a list comprising its integer identifier, its so far computed certificate string, and a list of its neighbors represented as integer identifiers (see examples below).

Task 2.1 Compute certificate for a given vertex. [2 pts]
Implement the step 2, i.e., a function *vertex-certificate* that computes the certificate for a given vertex in a given tree as follows:
(vertex-certificate vertex-id tree)
where

- *vertex-id* is an integer identifier of the vertex for which the certificate is to be computed;
- *tree* is a tree-type graph; note, *tree* contains the vertex specified by *vertex-id*;
- and the function returns the new certificate of the node specified by *vertex-id* as a string of *0* a *1* values.

Task 2.2 Compute certificate for the whole tree. [3 pts]
Implement a function *certificate* that computes the certificate for a given tree as follows:
(certificate tree)
where

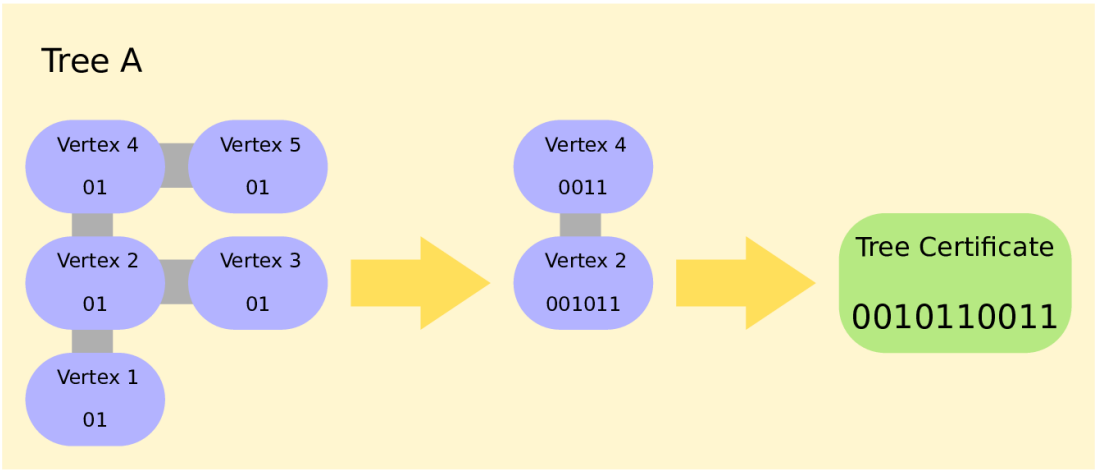
- *tree* is defined as in the Task 2.1;
- and the function returns the respective tree certificate as a string of *0* a *1* values.

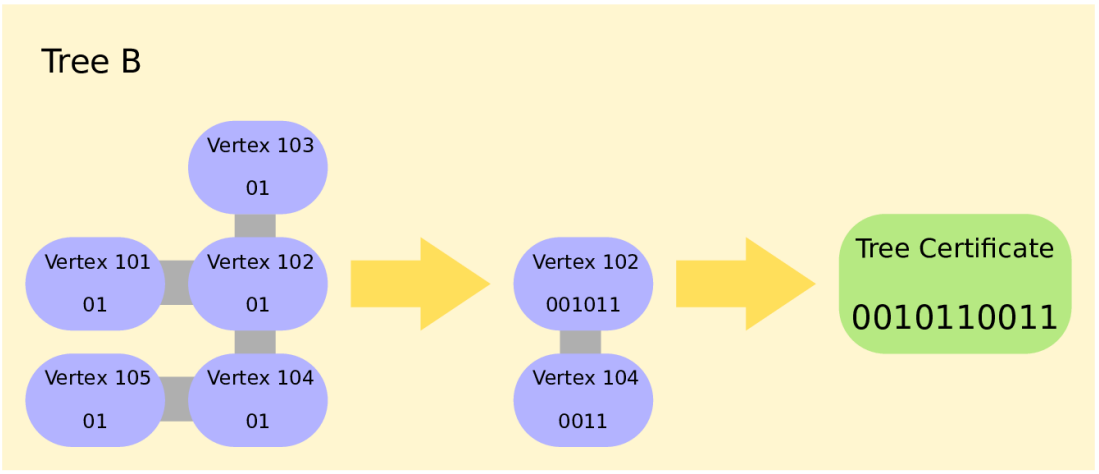
Task 2.3 Determine isomorphism of two trees. [2 pts]
Implement a function *isomorphic?* that determines whether two trees are isomorphic as follows:
(isomorphic? tree1 tree2)
where

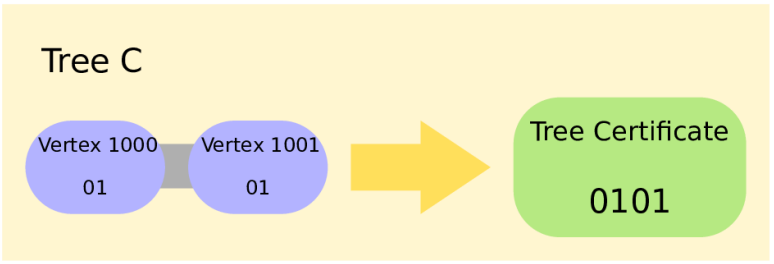
- *tree1* and *tree2* are defined as in the Task 2.1;

- and the function returns *#t* if and only if the two trees are isomorphic, i.e., they have matching certificates.

Examples








```
> (define treeA '( (1 "01" (2)) (2 "01" (1 3 4)) (3 "01" (2)) (4 "01" (2 5)) (5 "01" (4)) ))
> (define treeB '( (103 "01" (102)) (104 "01" (102 105)) (101 "01" (102)) (102 "01" (101 103
104)) (105 "01" (104)) ))
> (define treeC '( (1000 "01" (1001)) (1001 "01" (1000)) ))

> (vertex-certificate 2 treeA)
001011

> (certificate treeA)
```

```
0010110011
> (certificate treeB)
0010110011
> (certificate treeC)
0101

> (isomorphic? treeA treeA)
#t
> (isomorphic? treeA treeB)
#t
> (isomorphic? treeA treeC)
#f
> (isomorphic? treeB treeC)
#f
```

Hints

In Scheme, the *string<?* function can be used for lexicographic string comparison and *string=?* for testing equality. For appending strings in scheme use the *string-append* function. Your implementation will be tested using the r5rs standard, and you should remove the *#lang* before uploading your code.

In order to simplify the implementation, we provide a code template file containing a bunch of useful functions and public test cases. Namely, it contains the filter function, the qsort function implementing quicksort algorithm. Further there are functions trimm, join and add01 manipulating strings. The first one trimms the first and last character from a string, the second joins a list of strings into a single string and the last one takes a string, prepends 0 and appends 1 to it. Finally, there is a function get-node which for a given index *n* and a graph *g* returns the triple representing a node of the index *n*.

Task 3: **Haskell** Certificates for Tree Isomorphism **[7 Points]**

Graph certificates are a tool used to determine graph isomorphism, where two graphs are isomorphic if and only if their certificates are equal. In particular, a simple certificate suitable for trees, i.e., undirected connected graphs that have no cycles, is computed as follows:

1. Label all the vertices of the graph G with the string 01.
2. While there are more than two vertices of G do for each non-leaf x of G:
    a. Let Y be the set of labels of the leaves adjacent to x and the label of x, with the initial 0 and trailing 1 deleted from x;
    b. Replace the label of x with concatenation of the labels in Y sorted in increasing lexicographic order, with 0 prepended and a 1 appended;
    c. Remove all leaves adjacent to x.
3. If there is only one vertex left, report the label of x as certificate.
4. If there are two vertices x and y left, then report the labels of x and y, concatenated in increasing lexicographic order, as the certificate.
5. Otherwise continue with the step 2.

[borrowed from a4m33pal]

Task 3.1 Compute certificate for a given vertex. [2 pts]
Implement the step 2, i.e., a function *vertex_certificate* that computes the certificate for a given vertex in a given tree as follows:
vertex_certificate :: Int -> Graph -> String
and
type Node = (Int, String, [Int])
type Graph = [Node]
where

- the first input is the *Int* identifier of the vertex for which the certificate is to be computed;
- the second input of type *Graph* is defined as a list of nodes, where each type *Node* comprises its *Int* identifier, the so far computed certificate *String*, and a list of its neighbors represented as *Int* identifiers; note, the list contains the vertex specified by the first input;
- and the function returns the new certificate of the specified node a *String* of *0* a *1* values.

Note, your implementation should include the herein specified type definitions.

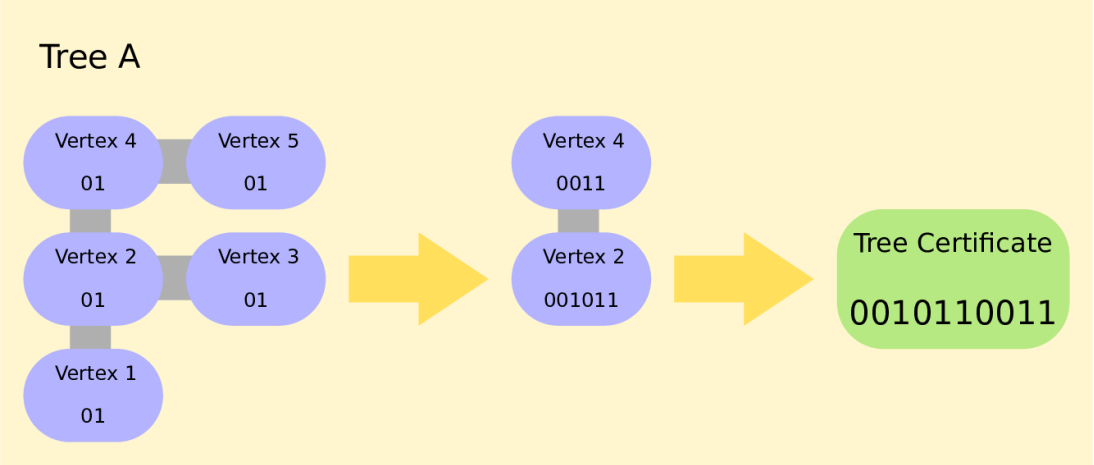
Task 2.2 Compute certificate for the whole tree. [3 pts]
Implement a function *certificate* that computes the certificate for a given tree as follows:
certificate :: Graph -> String
where

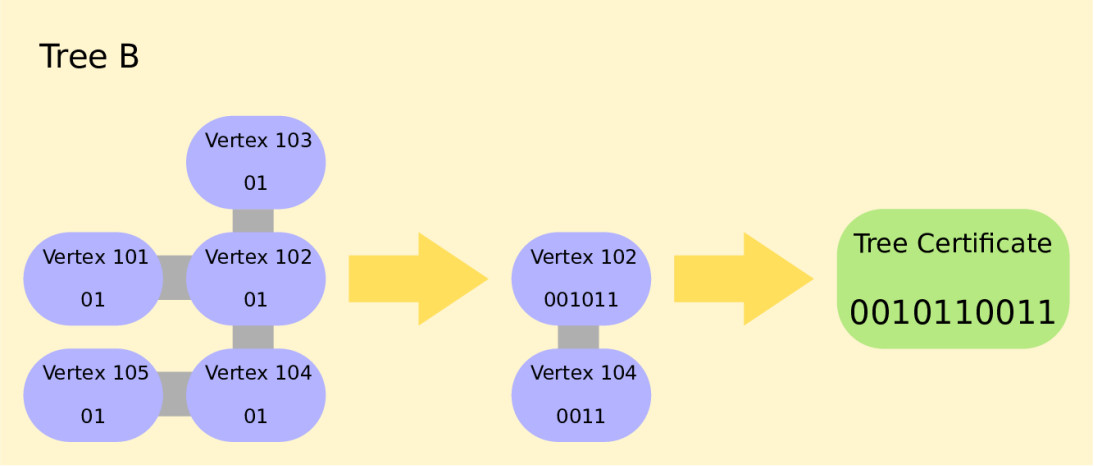
- the first input is defined as in the Task 2.1;
- and the function returns the respective tree certificate as a *String* of *0* a *1* values.

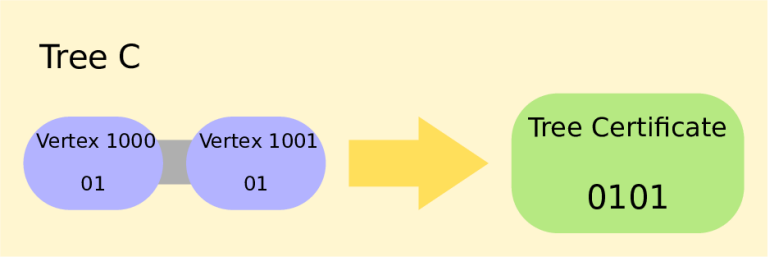
Task 2.3 Determine isomorphism of two trees. [2 pts]
Implement a function *is_isomorphic* that determines whether two trees are isomorphic as follows:
is_isomorphic :: Graph -> Graph -> Bool
where

- the first two inputs are defined as in the Task 2.1;
- and the function returns *True* if and only if the two trees are isomorphic, i.e., they have matching certificates.

Examples







> treeA :: Graph
> treeA = [(1, "01", [2]), (2, "01", [1, 3 ,4]), (3, "01", [2]), (4, "01", [2, 5]), (5, "01", [4])]
> treeB :: Graph
> treeB = [(103, "01", [102]), (104, "01", [102, 105]), (101, "01", [102]), (102, "01", [101, 103, 104]), (105, "01", [104])]
> treeC :: Graph
> treeC = [(1000, "01", [1001]), (1001, "01", [1000])]

> vertex_certificate 2 treeA
“001011”

> certificate treeA
“0010110011”
> certificate treeB
“0010110011”
> certificate treeC
“0101”

> is_isomorphic treeA treeA
True
> is_isomorphic treeA treeB
True
> is_isomorphic treeA treeC
False
> is_isomorphic treeB treeC
False

Hints

In order to sort a list of strings lexicographically in Haskell, import Data.List and call the function sort. We provide a code template file containing the above mentioned import, type declarations, definitions of treeA, treeB, treeC, a function getNode returning a node of a given identifier from a given graph, and functions for manipulating strings trimm, join and add01 (see Task 2).

Task 4 **Haskell IO**: Derivative of a polynomial **[8 Points]**

In mathematics, a **polynomial** is an expression consisting of variables (also called indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables. An example of a **polynomial** of a single indeterminate, x, is $x^2 + 4x$, it's first derivative would be 2x+4.

4.1 Implementation

In **Haskell**, polynomials can be represented as lists of pairs of the form *(coefficient, exponent)*, e.g. $x^2 + 4x$ can be represented as [(4,1), (1,2)]. Implement the function ***derive_polynomial*** *n polynomial,* which takes an integer *n* and list of pairs *polynomial,* and computes the *n-th derivative* of the given polynomial represented as above. The function should return a list of pairs representing the n-th derivative of the given polynomial.

4.2 IO Wrapper

Wrap the ***derive_polynomial*** function in an IO main which first asks the user to input the number of nonzero terms of the following univariate polynomial by requesting "*Enter number of nonzero terms:*". Afterwards, depending on the number, "*Enter term as a pair of the form (coef,exp):*" will require the user to sequentially input the terms. Each of these requests expects the user to input a pair of a coefficient and an exponent of the form *(coefficient,exponent)*. Finally it will ask the user to input which derivative of the polynomial to compute by requesting "*Enter derivative:*". The main function should then print the corresponding derivative of the polynomial represented by the entered list of pairs.

The output of the IO should print the polynomial in the following format:
The polynomial represented by *[(5,1),(9,2),(1,4)]* should be printed as "5*x + 9*x^2 + 1*x^4". Note that terms should be **sorted with respect to the exponents** (lower exponents come first). For the printing you can assume that all the coefficients in test cases are **positive integers**. Thus the output of the IO will always has the format of "a*x^n + b*x^k + ...".
Note the spaces between each of the terms.
In case you implement a type (which is optional) to represent the polynomial internally, make it an instance of Show class by implementing the show function for your polynomial type.

A couple of notes:

> *You can expect only valid input (also only valid number of derivatives requested) as well as positive integer coefficients and integer exponents only.*
> *One option to convert a string str of the form (coef, exp) to a pair of type (Int, Int) is the **read** function as follows: (read str) :: (Int, Int)*
> ***Important note**: Your polynomial should be **sorted by increasing exponents**. So a polynomial read through the IO in the order of [(1,4),(9,2),(5,1)] would be sorted to [(5,1),(9,2),(1,4)].*

*Hint: use the **sortBy** and **compare** functions as follows:*
sortBy (\(_,e1) (_,e2) -> compare e1 e2) [(1,5), (3,2), (5,1)]

Examples

f (x) = x^4 + 9 x^2 + 5x
f' (x) = 5 + 18x + 4x^3
The output ordered in increasing order of exponents:

> derive_polynomial 1 [(1,4),(9,2),(5,1)]
[(5,0),(18,1),(4,3)]

4.2 IO Wrapper

Enter number of non-zero terms:
> 3
Enter term as a pair of the form (coef, exp):
> (1,4)
Enter term as a pair of the form (coef, exp):
> (9,2)
Enter term as a pair of the form (coef, exp):
> (5,1)
Enter derivative:
> 1

*5 + 18*x + 4*x^3*