ECE 51220 Programming Assignment 1

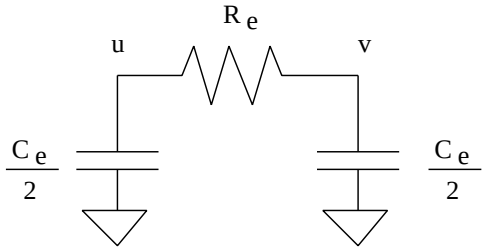
You are required to implement a program to insert inverters into an RC (resistive-capacitive) tree, which is represented by a strictly binary tree. The objective is to insert as few inverters as possible, under the constraints that (i) the maximum Elmore delay at each stage is under a specified time constraint, and (ii) all leaf nodes of the binary tree after inverter insertion are non-inverting.

Elmore Delay

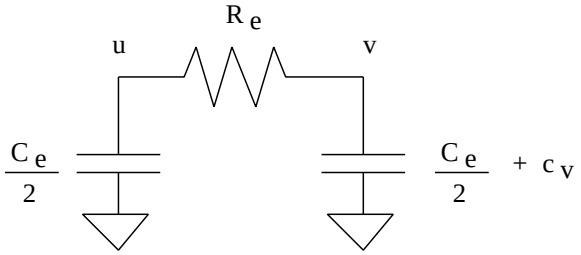
Every tree edge in the strictly binary tree is a wire with some resistance and some capacitance. Consider a tree edge $e = (u, v)$ of length l_e connecting a parent node u and to a child node v. The resistance and capacitance of the wire are given by

$$R_e = r \times l_e, \quad C_e = c \times l_e,$$

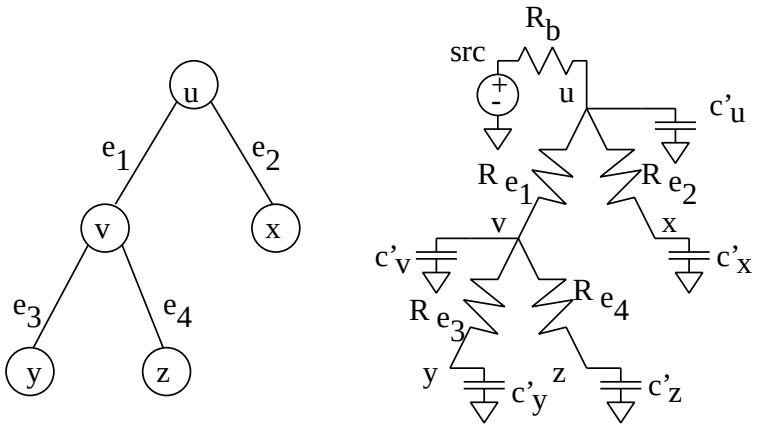
where r and c are per-unit-length resistance and per-unit-length capacitance, respectively. (If it helps, you may assume that a unit-length is 1μm or 1nm.) A first-order model of the wire is given below:



If the child node v happens to be a leaf node, it has an additional capacitance value associated with it, called the sink capacitance, denoted by c_v. The corresponding circuit is given below:



The tree is driven at its root by a driver (or inverter) with a output resistance R_b and output capacitance C_o (the product R_bC_o is the intrinsic delay T_b of the inverter). Therefore, the corresponding RC-tree of a binary tree is as follows:



(a) A binary tree (b) The corresponding RC tree

In the preceding figure, c'_i corresponds to the total capacitance at node i. Therefore,

$$\begin{aligned} c'_u &= \frac{C_{e_1}}{2} + \frac{C_{e_2}}{2} + C_o, \\ c'_v &= \frac{C_{e_1}}{2} + \frac{C_{e_3}}{2} + \frac{C_{e_4}}{2}, \\ c'_x &= \frac{C_{e_2}}{2} + c_x, \\ c'_y &= \frac{C_{e_3}}{2} + c_y, \\ c'_z &= \frac{C_{e_4}}{2} + c_z. \end{aligned}$$

The output capacitance of the inverter (driver) is lumped together with other wire capacitances at node u. Moreover, the driver output resistance R_b is between a source node, denoted as *src* and the root of the strictly binary tree u.

Given an RC tree T, the Elmore delay model [1] gives an good approximation of the delay from the source node *src* to node j. Let $\text{Path}(src, j)$ denote the path in the tree from node *src* to node j. Moreover, let $R_{\text{Path}(src,i)\cap\text{Path}(src,j)}$ denote the total resistance along the path common to $\text{Path}(src, i)$ and $\text{Path}(src, j)$. Furthermore, let T_v denote the subtree whose root node is v. The Elmore delay from *src* to j is given by the following two equivalent expressions:

$$\begin{aligned} t_j &= \sum_{i \in T} c'_i \times R_{\text{Path}(src,i)\cap\text{Path}(src,j)} \qquad (1) \\ &= \sum_{e=(u,v)\in\text{Path}(src,j)} R_{(u,v)} \sum_{k \in T_v} c'_k. \qquad (2) \end{aligned}$$

Using the first expression, the Elmore delays of nodes v and z, for example, are:

$$\begin{aligned} t_v &= c'_u R_b + c'_v(R_b + R_{e_1}) + c'_x(R_b) + c'_y(R_b + R_{e_1}) + c'_z(R_b + R_{e_1}), \\ t_z &= c'_u R_b + c'_v(R_b + R_{e_1}) + c'_x(R_b) + c'_y(R_b + R_{e_1}) + c'_z(R_b + R_{e_1} + R_{e_4}). \end{aligned}$$

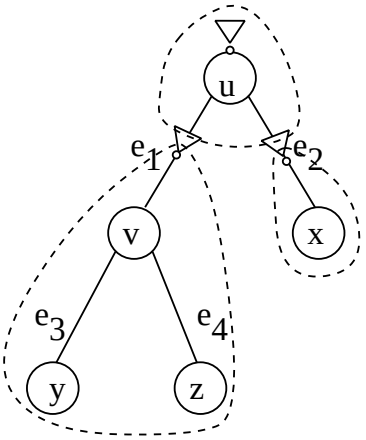
Using the second expression, the Elmore delays of nodes v and z can be written as:

$$\begin{aligned} t_v &= R_b(c'_u + c'_v + c'_x + c'_y + c'_z) + R_{e_1}(c'_v + c'_y + c'_z), \\ t_z &= R_b(c'_u + c'_v + c'_x + c'_y + c'_z) + R_{e_1}(c'_v + c'_y + c'_z) + R_{e_4}(c'_z). \end{aligned}$$

For an input tree, the driver (inverter) is not explicitly spelled out and all leaf nodes are inverting.

Stage Delay

The following figure shows a tree with two inverters inserted. The implicit inverter at the root node of the tree is now shown. Each inverter isolates the downstream (its output) from the upstream (its input). Each leaf node of the tree is now non-inverting, meeting one of the two constraints of the inverter insertion problem.



The inverters divide the tree into stages, with each stage driven by an inverter. The figure shows three stages all together. The top-most stage includes the driver (implicit inverter) at the source, part of the edges e_1 and e_2, and the input capacitances (C_b) of the two inserted inverters. Here, the input capacitances of the two inserted inverters serve as the sink capacitances of the RC tree of the top-most stage. To determine the stage delay, you have to calculate the Elmore delays from the source to the input capacitances of the two inserted inverters and pick the largest of the two. You have to account for the output resistance and output capacitance of the driver (implicit inverter) in your Elmore delay calculation.

The bottom left stage includes an inserted inverter, the remaining part of e_1 and its downstream. To determine the stage delay, you have to calculate the Elmore delays from the corresponding inverter to the two leaf nodes y and z and pick the largest of the two. You have to account for the output resistance and output capacitance of the inserted inverter in your Elmore delay calculation.

The bottom right stage includes the other inserted inverter, the remaining part of e_2 and its downstream. The stage delay is the Elmore delay from the corresponding inverter to the leaf node x. Again, you have to account for the output resistance and output capacitance of the inserted inverter in your Elmore delay calculation.

For your solution to be valid, all leaf nodes (of the original tree) must be non-inverting. For this example, each leaf node can be reached from the root node with two stages. It is not necessary that

all leaf nodes can be reached from the root node with the same number of stages. Each must be reached with an even number of stages. Moreover, all stage delays must not be greater than some specified time limit.

Deliverables

In this assignment, you are required to develop your own include file and source files, which can be compiled with the following command:

```
gcc -std=c99 -pedantic -Wvla -Wall -Wshadow -O3 *.c -o pa1 -lm
```

It is recommended that while you are developing your program, you use the "`-g`" flag instead of the "`-O3`" flag for compilation so that you can use a debugger if necessary. All declarations and definition of data types and the functions associated with the data types should reside in the include file or source files. The main function should reside in one (and only one) of the source files.

If you provide a Makefile in your submission, your Makefile must create `pa1` with the command:

```
make pa1
```

If your assignment is written in C++, you must provide a Makefile because the `gcc` command used in the grading script will not be able to generate an executable `pa1`.

If we invoke the executable `pa1` as follows:

```
./pa1 time_constraint in_name1 in_name2 in_name3 out_name1 out_name2 out_name3
out_name4
```

the executable takes a time constraint (`argv[1] time_constraint`), reads the parameters of an inverter from a file (`argv[2] in_name1`), reads the parameters of wires from a file (`argv[3] in_name2`), and constructs an RC tree from a file (`argv[4] in_name3`). The executable then outputs the topology of the RC tree in pre-order fashion to a text file (`argv[5] out_name1`), calculates the Elmore delays of all leaf nodes in the RC tree and outputs them to a binary file (`argv[6] out_name2`), inserts inverters such that all stages meet the given time constraint and all leaf nodes are non-inverting, and outputs the resulting tree to a text file (`argv[7] out_name3`) and a binary file (`argv[8] out_name4`). When the time constraint is too stringent for your algorithm to produce a solution, it is fine to leave the third and fourth output files empty. When the arguments to the executable allow you to produce four meaningful output files, the executable returns `EXIT_SUCCESS`; the executable should return `EXIT_FAILURE` otherwise.

`argv[1]` stores the constraint delay limit. You can obtain the stage delay limit as a double using the `atof` function.

Format of first input file

`argv[2]` stores the name of the file that contains the parameters of the inverter to be inserted (and also the driver at the source of the given RC tree). The file contains a single line printed with the format `"%.10le %.10le %.10le\n"`, where the first `double` is the input capacitance (in F), the second `double` is the output capacitance (in F), and the third `double` is the output resistance (in Ω). The file `inv.param` included for this assignment is as follows:

```
3.4500000000e-14 5.8000000000e-14 1.1300000000e+02
```

Format of second input file

`argv[3]` stores the name of the file that contains the wire parameters. This file should contain a single line printed with the format `"%.10le %.10le\n"` where the first `double` is the per-unit-length wire resistance (in Ω/unit length) and the second `double` is the per-unit-length wire capacitance (in F/unit length). The file `wire.param` included for this assignment is as follows:

```
1.0000000000e-04 2.0000000000e-19
```

Format of third input file

`argv[4]` stores the name of the file that contains the topological information of the RC tree. The file is divided into lines, and each line corresponds to a node in the strictly binary tree. The order in which the nodes are printed is based on a post-order traversal of the binary tree. If it is a leaf node (which is a sink), it has been printed with the format `"%d(%.10le)\n"`, where the `int` is the label of the sink, and the `double` is the sink node capacitance (F). If it is a non-leaf node, it has been printed with the format `"(%.10le %.10le)\n"`, where the first `double` is the wire length to the left child and the second `double` is the wire length to the right child.

The file `5.txt` contains a 5-sink example. Note that sink 5 has a sink capacitance of 0fF, which is different from the other sinks. It also has an zero-length edge to the source of the RC tree. (This is actually a fake sink that I created to make the tree strictly binary.)

```
2(3.5000000000e-14)
4(3.5000000000e-14)
(1.5300000000e+06 8.7000000000e+05)
1(3.5000000000e-14)
3(3.5000000000e-14)
(1.2300000000e+06 1.3700000000e+06)
(1.2700000000e+06 1.2700000000e+06)
5(0.0000000000e+00)
(5.0000000000e+06 0.0000000000e+00)
```

Format of first output file (text)

The first output file (`argv[5]`) is a pre-order printing of the given RC tree. Each line corresponds to a node in the binary tree. If it is a leaf node (which is a sink), you print the node with the format `"%d(%.10le)\n"`, where the `int` is the label of the sink, and the `double` is the sink node capacitance (F). If it is a non-leaf node, you print with the format `"(%.10le %.10le)\n"`, where the first `double` is the wire length to the left child and the second `double` is the wire length to the right child.

For the 5-sink example, the corresponding output file `5.pre` is as follows:

```
(5.0000000000e+06 0.0000000000e+00)
(1.2700000000e+06 1.2700000000e+06)
(1.5300000000e+06 8.7000000000e+05)
```

```
2(3.5000000000e-14)
4(3.5000000000e-14)
(1.2300000000e+06 1.3700000000e+06)
1(3.5000000000e-14)
3(3.5000000000e-14)
5(0.0000000000e+00)
```

Format of second output file (binary)

The second output file (`argv[6]`) should contain the labels and Elmore delays of the leaf nodes of the given RC tree in binary. You should be using the function `fwrite` to write the Elmore delays. For each node, you write an `int` for its label followed by a `double` for its Elmore delay. The order in which you write the labels and Elmore delays is based on a pre-order traversal of the RC tree. If there are n leaf nodes, the size of the output file should be $n \times$ (`sizeof(int)` + `sizeof(double)`). The corresponding output file for the 5-sink example is `5.elmore`.

Format of third output file (text)

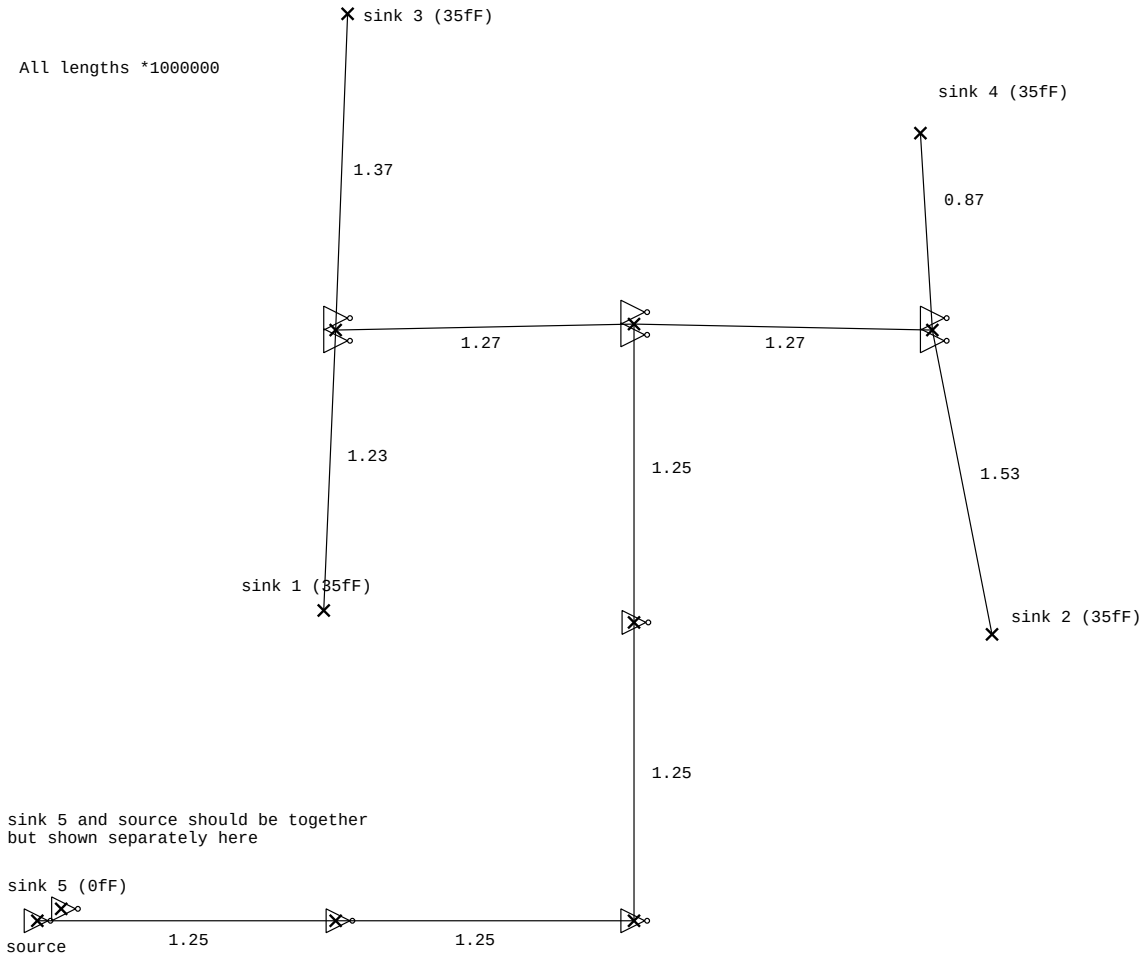
The third output file (`argv[7]`) should contain the topology of the RC tree with inverters inserted. The topology should be printed in post-order fashion, similar to the third input file. However, there are some differences in the format so that we can capture the presence of inverters. Moreover, it may be necessary to break a long wire into multiple segments and use an inverter to drive each of them. The following figure shows a solution for the 5-sink example.

In this solution, we allow multiple inverters to be connected in parallel to create an inverter of larger size. Suppose we connect $k > 1$ inverters in parallel, the input capacitance of this inverter is k times as large as that of the single inverter, the output capacitance k times as large, and the output resistance k times as small.

In this example, there are six inverters (of different sizes) along each source-to-sink paths for sinks 1 through 4. Sink 5 has a source-to-sink path that contains two inverters. In this circuit, the source and sink 5 should coincide. For illustration purpose, they are shown separately in the figure. The inverter at the source drives two branches, one branch leads to sinks 1 through 4, and the other branch is an inverter driving sink 5 directly. In your solution, it is not necessary that all source-to-sink paths have the same number of inverters. The only requirement is that each path has an even number of inverters (or stages).

The topology is stored in the text file `5.ttopo` as follows:

```
2(3.5000000000e-14)
4(3.5000000000e-14)
(1.5300000000e+06 8.7000000000e+05 2)
1(3.5000000000e-14)
3(3.5000000000e-14)
(1.2300000000e+06 1.3700000000e+06 2)
(1.2700000000e+06 1.2700000000e+06 2)
(1.2500000000e+06 -1.0000000000e+00 1)
(1.2500000000e+06 -1.0000000000e+00 1)
(1.2500000000e+06 -1.0000000000e+00 1)
5(0.0000000000e+00)
```




```
(0.0000000000e+00 -1.0000000000e+00 1)
(1.2500000000e+06 0.0000000000e+00 1)
```

Each sink node is printed with a similar format as the input `"%d(%.10le)\n"`, which corresponds to the sink label (`int`) and the sink capacitance (`double`).

Each non-leaf node is printed with the format `"(%.10le %.10le %d)\n"`, which corresponds to the wire length to the left child, the wire length to the right child (both `double`), and the number of parallel inverters at this location (`int`). **One major change is the inclusion of the number of parallel inverters as the last field. The other major change is that we allow an internal node to have only one (left) child.** This happens when the wire length to its "right" child is of value -1.0, indicating that the right child is non-existent. For example, we split the long wire at the root node into 4 segments, creating three additional nodes. Each of these additional nodes has only one left child.

Another example is the inverter driving sink 5. Here, the following lines

```
5(0.0000000000e+00)
(0.0000000000e+00 -1.0000000000e+00 1)
```

indicate that sink 5 is directly driven by an inverter (and sink 5 appears as a left branch of the inverter with zero wire length while the right branch of the inverter is non-existent.)

At the root node, the field indicating the number of parallel inverters should always be 1. Unlike the input file, we explicitly indicate the presence of an inverter as a driver in the output file.

You can insert a repeater instead of an inverter. Assume that you want an repeater made up of a single inverter driving 4 parallel inverters. Suppose we want to insert such a repeater at the non-leaf node

```
(1.2700000000e+06 1.2700000000e+06 2)
```

We would replace the line with the following two lines:

```
(1.2700000000e+06 1.2700000000e+06 4)
(0.0000000000e+00 -1.0000000000e+00 1)
```

Of course, such a topology is not a valid solution overall for this example because some sink nodes will now be inverting.

Your algorithm may want to insert an inverter to drive only one of the two branches. For example, instead of inserting two parallel inverters to drive both branches leading to sink 2 and sink 4, we want to insert two parallel inverters to drive only the branch leading to sink 2. We would modify the following lines

```
2(3.5000000000e-14)
4(3.5000000000e-14)
(1.5300000000e+06 8.7000000000e+05 2)
```

to

```
2(3.5000000000e-14)
(1.5300000000e+06 -1.0000000000e+00 2)
4(3.5000000000e-14)
(0.0000000000e+00 8.7000000000e+05 0)
```

Again, this is not a valid solution because not all sinks are non-inverting.

Format of fourth output file (binary)

We can also store the representation of a topology (in post-order traversal) in a binary file format (`argv[8]`) (using the `fwrite` function). For a leaf node, it is stored as an `int` (sink label) and a `double` (sink capacitance).

For a non-leaf node, it is stored as an `int` (a value of -1), a `double` (wire length to left child), a `double` (wire length to right child), and an `int` (number of parallel inverters). If it is a non-leaf node with only a branch, the wire length to right child should be of value -1.0.

The topology of the 5-sink example is stored in binary file `5.btopo`.

Electronic Submission

The project requires the submission (electronically) of the C-code (source and include files) through Brightspace. You should create and submit a zip file called `pa1.zip`, which contains the source and include files. For example, if you have only the necessary source and include files in the current directory, you can create the zip file as follows:

```
zip pa1.zip *.[ch]
```

A zip file created in this fashion does not contain a folder. You can add a Makefile to the zip file:

```
zip pa1.zip Makefile
```

If your `pa1.zip` contains a `Makefile`, we will use your `Makefile` to create `pa1` using the command:

```
make pa1
```

You should always copy the created zip file to a different directory, unzip the file in that directory, compile the program in that directory, and run some test cases in that directory. Submit the zip file only after you are certain that you have the correct zip file.

Your zip file should not contain a folder or directory (that contains the source files, include files, and Makefile). The zip file should contain only the source files, include files, and Makefile.

Grading

The assignment will be graded based on the evaluation of your program using some examples. In the following, we assume that each example (or each test case) is graded based on 100 points.

The first output accounts for 10 points, the second output accounts for 20 points.

The remaining 70 points will be awarded based on the quality and validity of your solution in the fourth output if there exist feasible solutions for a given time constraint. For a time constraint that is too stringent for a feasible solution, a correct return status of your program will earn you credit. Note that even when you cannot produce the fourth output because there are no feasible solutions, you still have to produce the first and second output files.

By quality, we refer to the number of inverters you have inserted into a given tree. The fewer there are, the higher the quality.

For a solution in the fourth output to be considered valid, all leaf nodes must be non-inverting and all stages must meet the time constraint. Moreover, when all inserted inverters are removed from your solution, the topology remains the same as the original RC tree, the label and the capacitance of every leaf node remain intact, and the wire length of every branch remains intact. When you divide a branch into shorter segments for the insertion of inverters, your program may introduce some rounding error in the wire length of that branch. Some reasonable amount of rounding error is acceptable. In other words, we do not expect the sum of the segment lengths to be exactly the same as the original length of the divided branch.

Out of the 70 points, 40 points will be awarded for a solution that is valid and 30 points will be awarded based on the quality of a valid solution. **Please note that we do not expect your algorithm to be optimal. In other words, we do not expect your algorithm to be able to find an optimal solution for every test case. Typically, a successful implementation of a reasonable greedy approach will allow you to earn full credit for a test case.**

The third output file is meant to help you with debugging. It will not be used for the evaluation of your solution.

It is important that your program can accept any legitimate filenames as input or output files. Even if you cannot produce all output files correctly, you should still write the `main` function such that it produces as many correct output files as possible. Any output files that you cannot produce, you should leave them as empty file or not create them at all. Essentially, you are graded based on your ability to construct a tree, compute Elmore delays, and perform inverter insertion.

For each input example, we expect your program to complete its execution within a run-time limit that is determined based on the problem size. The run-time limits for the evaluation of your program will range from 6 seconds to 6 minutes, depending on the problem size.

It is important that all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. We will use `valgrind` to check for memory issues. While you are most familiar with memory leaks, `valgrind` can be useful in helping you find the cause of a segmentation fault and identify allocated memory locations that have not been initialized properly. The tool is useful only when you pay attention to all messages that `valgrind` reports. One useful programming habit is to keep your code `valgrind`-clean at any stage of programming. In other words, do not leave any memory issues unresolved at any stage of programming. Any memory issues reported by `valgrind` will result in a 50-point penalty.

What you are given

In the zip file `pa1_examples.zip`, it contains a folder `examples`, within which there are 2 sample input files (`3.txt`, `5.txt`) and two sets of sample output files (`3.pre`, `3.elmore`, `3.ttopo`, `3.btopo`, `5.pre`, `5.elmore`, `5.ttopo`, `5.btopo`). You are also given `inv.param` and `wire.param`, and `fake_inv.param` and `fake_wire.param`.

The purpose of `fake_inv.param`, `fake_wire.param`, and `3.txt` are used to help you to develop the Elmore computation function because they use integer values for parameters and wire lengths. The file `3.txt` corresponds to the 3-sink example used in this document, with leaf nodes x, y, and z being labeled numerically as 1, 2, and 3, respectively. The file `3.pre` stores the pre-order printing of this example, and `3.elmore` stores the Elmore delays of sink nodes of this example. The files `3.ttopo` and `3.btopo` are the text and binary files storing the topology of the solution with inverters inserted.

The Elmore delays of `5.txt` are computed using `inv.param` and `wire.param`.

Note that the sample solutions in the `*.[bt]topo` files are not optimal. You should not try to produce similar solutions. They are provided so that you understand the output formats (text and binary).

In addition, we also provide in `pa1_examples.zip` three more sample input files without any associated output files: `p1.txt`, `s1423.txt`, and `s5378.txt`. These are three of the input files that will be used for the evaluation of your submission.

Additional information

If there are any updates to the instructions, we will post them on Brightspace. Pay attention to announcements on Brightspace regarding these changes.

References

[1] W. C. Elmore. The transient response of damped linear networks with particular regard to wide-band amplifiers. *Journal of Applied Physics*, 19(1):55–63, January 1948.