

**Essential Algorithms and Data Structures  
for Computational Design in Grasshopper  
First Edition**

**Rajaa Issa**

Robert McNeel & Associates



Essential Algorithms and Data Structures for Computational Design, First edition, by Robert McNeel & Associates, 2020 is licensed under a [Creative Commons Attribution-Share Alike 3.0 United States License](#).

# Table of Contents

<b>Preface</b>	4
<b>Chapter One: Algorithms and Data</b>	5
1_1: Algorithmic design	5
1_2: Algorithms parts	5
1_3: Designing algorithms: the 4-step process	7
1_4: Data	12
1_5: Data sources	12
1_6: Data types	13
1_7: Processing data	15
1_7_1: Numeric operations	15
1_7_2: Logical operations	17
1_7_3: Data analysis	18
1_7_4: Sorting	18
1_7_5: Selection	19
1_7_6: Mapping	19
1_8: Pitfalls of algorithmic design	23
1_8_1: Invalid or wrong type input	23
1_8_2: Incorrect input	24
1_8_3: Incorrect order of operation	24
1_8_4: Mismatched data structures	25
1_8_5: Long processing time	25
1_8_6: Poor organization	26
1_9: Algorithms tutorials	26
1_9_1: Unioned circles tutorial	26
1_9_2: Sphere with bounds tutorial	28
1_9_3: Data operations tutorial	29
1_9_4: Pitfalls tutorial	30
<b>Chapter Two: Introduction to Data Structures</b>	32
2_1: Overview	32
2_2: Generating lists	33
2_3: List operations	35
2_4: List matching	39
2_5: Data structures tutorials	44
2_5_1: Variable thickness pipe tutorial	44
2_5_2: Custom matching tutorial	46
2_5_3: Simple truss tutorial	47
2_5_4: Pearl necklace tutorial	49
<b>Chapter Three: Advanced Data Structures</b>	52
3_1: The Grasshopper data structure	52

3_1_1 Introduction	52
3_1_2 Processing data trees	52
3_1_3 Data tree notation	54
3_2: Generating trees	56
3_3: Tree matching	59
3_4: Traversing trees	62
3_5: Basic tree operations	64
3_5_1: Viewing the tree structure	64
3_5_2: List operations on trees	64
3_5_3: Grafting from lists to a trees	66
3_5_4: Flattening from trees to lists	67
3_5_5: Combining data streams	68
3_5_6: Flipping the data structure	68
3_5_7: Simplifying the data structure	70
3_6: Advanced tree operations	73
3_6_1: Relative items	73
3_6_2: Split trees	78
3_6_3: Path mapper	83
3_7: Advanced data structures tutorials	89
3_7_1: Sloped roof tutorial	89
3_7_2: Diagonal triangles tutorial	92
3_7_3: Zigzag tutorial	93
3_7_4: Weaving tutorial	93

# Preface

The **Essential Algorithms and Data Structures for Computational Design** introduces effective methodologies to develop complex 3D modeling algorithms using Grasshopper. It also covers extensively the data structure adopted by Grasshopper and its core organization and management tools.

The material is directed towards designers who are interested in parametric design and have little or no background in programming. All concepts are explained visually using Grasshopper® (GH), the generative modeling environment for Rhinoceros® (Rhino). This book is not intended as a beginners guide to Grasshopper in terms of user interface or tools. Basic knowledge of the interface and workflow is assumed. For more resources and getting started guides, go to the *learn* section in [www.rhino3d.com](http://www.rhino3d.com).

The content is divided into three chapters. Chapter 1 discusses algorithms and data. It introduces a methodology to help create and manage parametric solutions. It also introduces basic data concepts such as data types, sources and common ways to process them. Chapter 2 reviews basic data structures in Grasshopper. That includes single items and lists. Chapter 3 includes an in-depth review of the tree data structure in Grasshopper and practical applications in design problems. All Grasshopper examples and tutorials are written with Rhinoceros version 6 and are included in the download.

*Rajaa Issa*

Robert McNeel & Associates

# Chapter One: Algorithms and Data

Algorithms and data are the two essential parts of any parametric design solution, but writing algorithms is not trivial and requires a skill that does not come easy to intuitive designers. The algorithmic design process is highly logical and requires explicit statement of the design intention and the steps to achieve them. This chapter introduces a methodology to help creative designers develop new algorithmic solutions. All algorithms involve manipulating data and hence Algorithms and Data are tightly connected. We will introduce the basic concepts of data types and processes.

## 1\_1: Algorithmic design

We can define algorithmic design as a design method where the **output** is achieved through **well-defined steps**. In that sense, many human activities are algorithmic. Take, for example, baking a cake. You get the **cake** (output) by using a **recipe** (well-defined steps). Any change in the **ingredients** (input) or the baking process results in a different cake. We will analyze the parts of typical algorithms, and identify a strategy to build algorithmic solutions from scratch.

Regardless of its complexity, all algorithmic solutions have 3 building blocks: **input**, **key process**, and **output**. Note that the key process may require additional input and processes.

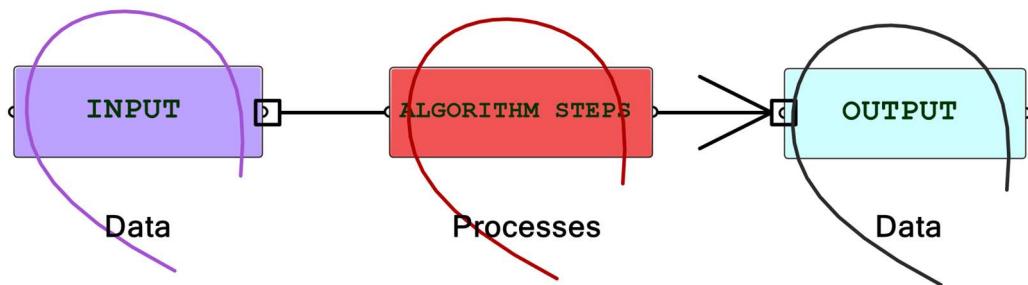


Figure (1): The building blocks of algorithmic solutions

Throughout this text, we will organize and label the solutions to identify the three blocks clearly. We will also use consistent color coding to visually distinguish between the parts. This will help us become more comfortable with reading algorithms and quickly identify input, key processing steps, and properly collect and display output. Visual cues are important to develop fluency in algorithmic thinking.

In general, reading existing algorithmic solutions is relatively easy, but building new ones from scratch is much harder and require a new set of skills. While it is useful to know how to read and modify existing solutions, it is essential to develop algorithmic design skills to build new solutions from scratch.

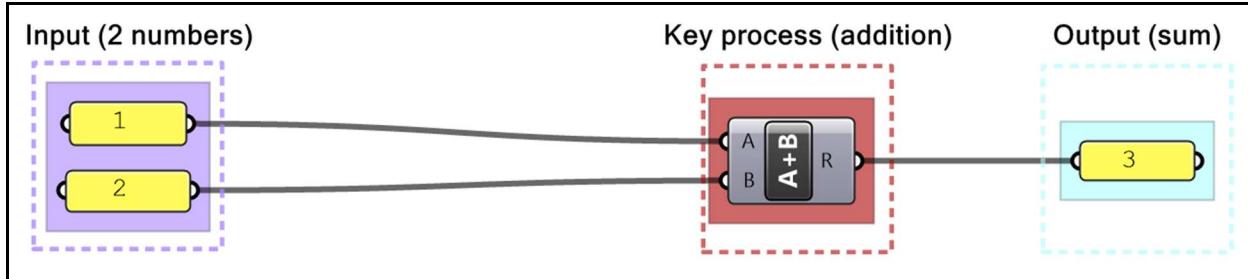
## 1\_2: Algorithms parts

In Grasshopper, a solution flows from left to right. At the far left are input values and parameters, and the far right has the output. In between are one or more key processes, and sometimes additional input and output. Let's take a simple example to help identify the three parts of any algorithm (input, key process, output). The simple addition algorithm includes two numbers (input), the sum (output)

and one key process that takes the numbers and gives the result. We will use purple for the input, maroon for the key processes and light blue for the output. We will also group and label the different parts and adhere to organizing the Grasshopper solutions from left to right.

### Example 1-2-1:

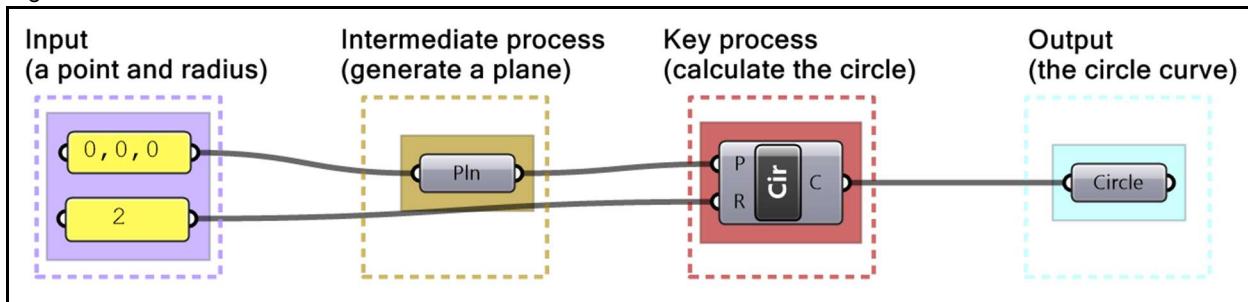
Algorithm to add 2 numbers



Algorithms may involve intermediate processes. For example, suppose we need to create a circle (output) using a center and a radius (input). Notice that the input is not sufficient because we do not know the plane on which the circle should be created. In this case, we need to generate additional information, namely the plane of the circle. We will call this an intermediate process and use brown color to label it.

### Example 1-2-2:

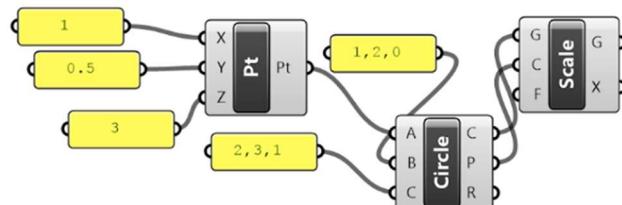
Algorithm to create a circle on the XY-Plane from a center and a radius



Some solutions are not written with styles and hence are hard to read and build on. It is very important that you take the time to organize and label your solutions to make it easier to understand, debug and use by others.

### Tutorial 1-2-3: Read existing algorithm

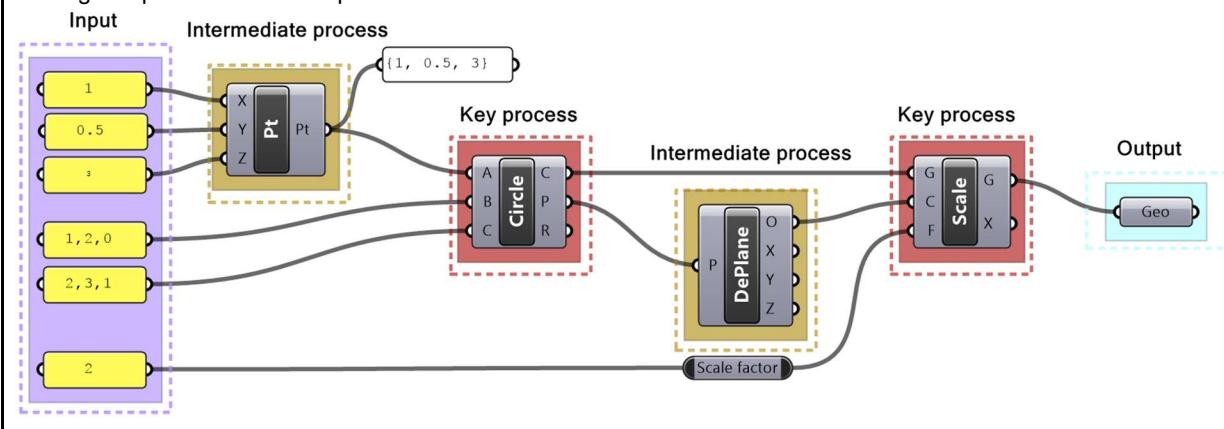
Given the following definition, write a description of what the algorithm does, identify input, the main process(s) and output, then label and color-code all the parts. Re-write the solution to make it more readable.



**Solution**

In order to figure out what the algorithm is meant to do, we need to group the input on the left side, and collect the output on the right side, then organize the processes in the order of execution. We then step through the solution from left to right to deduce what it does. We can examine and preview the output in each step.

The example of the tutorial is meant to create a circle that is twice as large as another circle that goes through three given points. One of the points is constructed out of its 3 coordinates.



## 1\_3: Designing algorithms: the 4-step process

Before we generalize a method to design algorithms, let's examine an algorithm we commonly use in real life such as baking a cake. If you already have a recipe for a cake, you simply get the recommended ingredients, mix them, pour in a pan, put in preheated oven for a certain amount of time, then serve. If the recipe is well documented, then it is relatively straightforward to use. As you become more proficient in baking cakes, you may start to modify the recipe. Perhaps add new ingredients (chocolate or nuts) or use different tools (cupcake container).

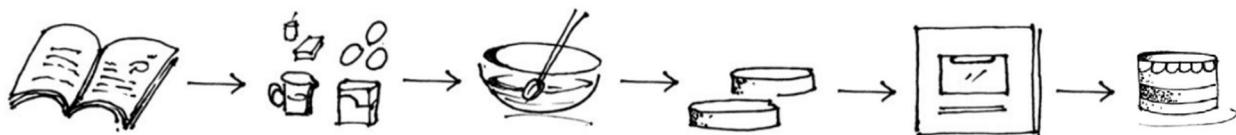


Figure (2): Steps to follow existing recipe

When designers write algorithms, they typically try to search for existing solutions and modify to fit their purposes. While this is a good entry point, using existing solutions can be frustrating and time-consuming. Also, existing solutions have their own flavor and that may influence design decisions and limit creativity. If designers have unique problems, and they often do, they have no choice but to create new solutions from scratch; albeit a much harder endeavor.

Back to our example, the task of baking a cake is much harder if you don't have a recipe to follow and have not baked one before. You will have to guess the ingredients and the process. You will likely end up with bad results in the first few attempts, until you figure it out! In general, when you create a new recipe, you have to follow the process in reverse. You start with an image of the desired cake, you then guess the ingredients, tools and steps. Your thinking goes along the following lines:

- The cake needs to be baked, so I need an **oven** and **time**,
- What goes in the oven is a **cake batter** held by a **container**,

- The batter is a mix of **ingredients**

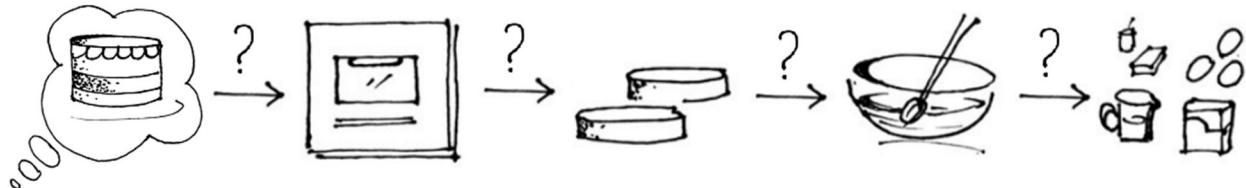


Figure (3): Steps to invent a new recipe from scratch

We can use a similar methodology to design parametric algorithm from scratch. Keep in mind that creating new algorithms is a “skill” and it requires patience, practice and time to develop.

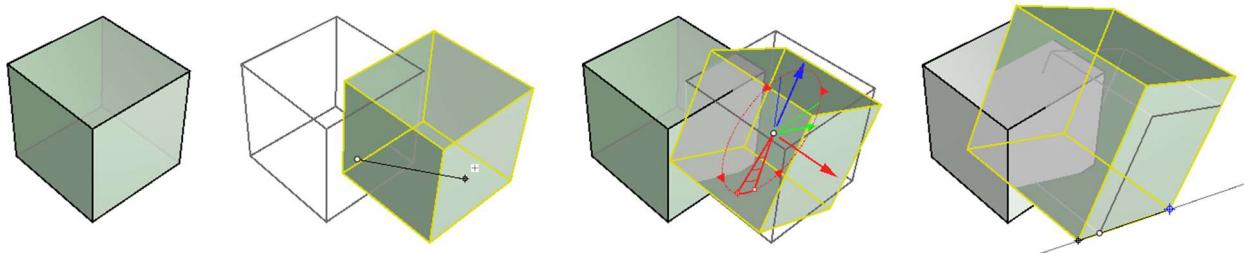
### Algorithmic thinking in 3D modeling vs parametric design

3D modeling involves certain level of algorithmic thinking, but it has many implicit steps and data. For example designing a mass model using a 3D modeler may involve the following steps:

- 1- Think about the output (e.g. a mass out of few intersecting boxes)
- 2- Identify a command or series of commands to achieve the output ( e.g. run **Box** command few times, **Move**, **Scale** or **Rotate** one or more boxes, then **BooleanUnion** the geometry).

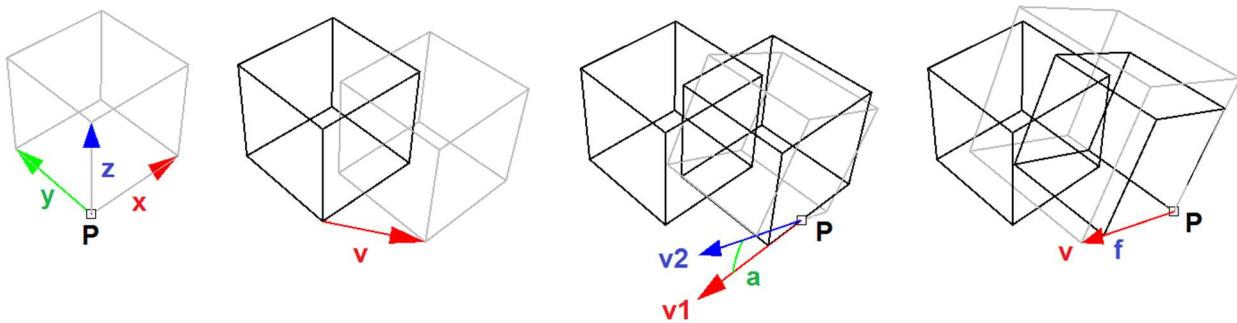
At that point, you are done!

Data such as the base point for your initial box, width, height, scale factor, move direction, rotation angle, etc. are requested by the commands, and the designer does not need to prepare ahead of time. Also, the final output (the boolean mass) becomes directly available and visible as an object in your document.



Figure(4): Interactive 3D modeling to create and manipulate geometry uses visual widgets and guides

Algorithmic solutions are not interactive and require explicit articulation of data and processes. In the box example, you need to define the box orientation and dimensions. When copy, you need a vector and when rotate you need to define the plane and angle or rotation.



Figure(5): Algorithmic solutions involve explicit definition of geometry, vectors and transformations

## Designing algorithms

Designing algorithms requires knowledge in geometry, mathematics and programming. Knowledge in geometry and mathematics is covered in the ***Essential Mathematics for Computational Design***<sup>1</sup>. As for programming skills, it takes time and practice to build the ability to formulate design intentions into logical steps to process and manage geometric data. To help get started, it is useful to think of any algorithm as a 4-step process as in the following:

1- Clearly identify the desired outcome	Output
2- Identify key steps to reach the outcome	Key processes
3- Examine initial data and parameters	Input
4- Define intermediate steps to generate missing data	Intermediate input + processes

Thinking in terms of these 4 steps is key to developing the skill of algorithmic design. We will start with simple examples to illustrate the methodology, and gradually apply on more complex examples.

### Example 1-3-1: Add two numbers

Use the 4-Step process to write an algorithm to add two numbers

1st Number      →      Addition Calculation      →      Sum  
2nd Number



<sup>1</sup> Issa, Essential Mathematics for Computational Design, 4th edition, 2019. Free download of the PDF and examples: <https://www.rhino3d.com/download/rhino/6/essentialmathematics>

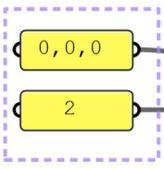
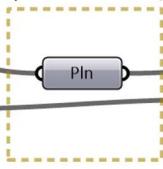
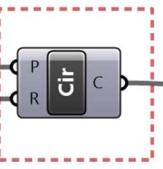
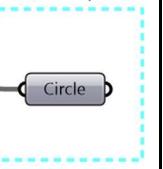
<b>Step 2: Key process:</b> Addition. Use the <b>Addition</b> component that takes 2 numbers and gives the sum.	<p>Key process (Addition)</p> <p>A + B → R</p> <p>Output (Sum)</p>
<b>Step 3: Input:</b> 2 numbers. Use a <b>Panel</b> to hold the values of input numbers.	<p>Input (2 numbers)</p> <p>1 → A</p> <p>2 → B</p> <p>Key process (Addition)</p> <p>A + B → R</p> <p>Output (Sum)</p> <p>3 → R</p>

### Example 1-3-2: Create a circle

Use the 4-Step process to create a circle from a given center and radius

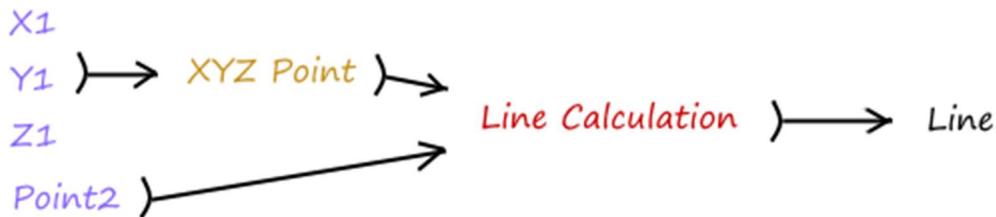


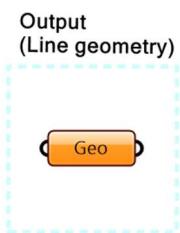
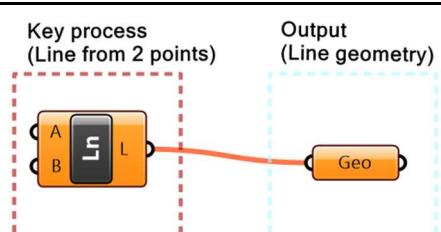
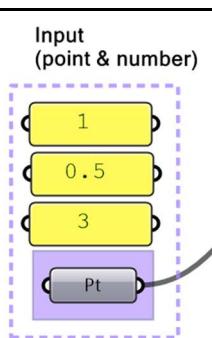
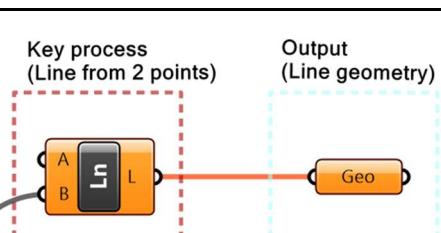
<b>Step 1: Output:</b> Circle. Use the <b>Circle</b> parameter to collect the output.	<p>Output (Circle)</p> <p>Circle</p>
<b>Step 2: Key process:</b> Identify a key process that generates a circle from a radius. Use the <b>Circle</b> component in Grasshopper.	<p>Key process (Generate Circle)</p> <p>P → Cir → C</p> <p>Output (Circle)</p> <p>Circle</p>
<b>Step 3: Input:</b> Use the given input (center and radius). Feed the radius to the <b>Circle</b> component.	<p>Input (point &amp; number)</p> <p>0,0,0 → P</p> <p>2 → R</p> <p>Key process (Generate Circle)</p> <p>P → Cir → C</p> <p>Output (Circle)</p> <p>Circle</p>

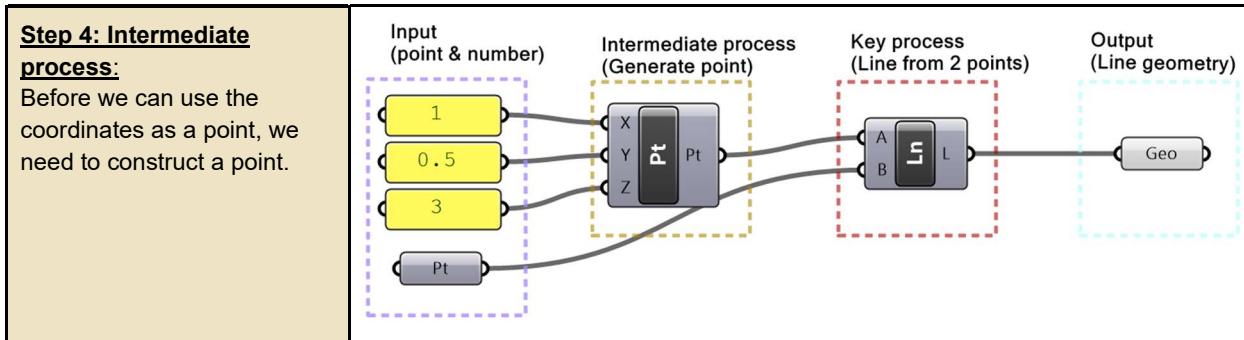
<b>Step 4: Intermediate process:</b>	Input (point & number)	Intermediate process (Generate Plane)	Key process (Generate Circle)	Output (Circle)
The circle needs the center, and also the plane on which the circle is located. Assume the circle is on a plane parallel to the XY-Plane and use the circle center as the origin of the plane.				

### Example 1-3-3: Create a line

Use the 4-Step process to create an algorithm to generate a line from 2 points. One point is referenced from Rhino, and the other is created using three coordinates ( $x=1$ ,  $y=0.5$  and  $z=3$ ).



<b>Step 1: Output:</b> The line geometry. Use the <b>Geometry</b> parameter to collect the output.		
<b>Step 2: Key process:</b> Identify a key process that generates a line from 2 points. Use the <b>Line</b> component in Grasshopper.		
<b>Step 3: Input:</b> Use the given input (referenced point and 3 coordinates). Feed one point to one of the ends of the line.		



In more complex algorithms, we will need to analyze the problems, investigate possible solutions and break them down to pieces whenever possible to make it more manageable and readable. We will continue to use the 4-step process and other techniques to solve more complex algorithms throughout the book.

## 1\_4: Data

Data is information stored in a computer and processed by a program. Data can be collected from different sources, it has many types and is stored in well defined structures so that it can be used efficiently. While there are commonalities when it comes to data across all scripting languages, there are also some differences. This book explores data and data structures specific to Grasshopper.

## 1\_5: Data sources

In Grasshopper, there are three main ways to supply data to processes (or what is called components): internal, referenced and external.

<b>Data sources: in Grasshopper</b>	
<b>1- Internally set data</b> Data can be set inside any instance of a parameter. Once set, it remains constant, unless manually changed or overridden by external input. This is a good way when you do not generally need to change the data after it is set (constant). Data is stored inside the GH file.	
<b>2- Referenced data</b> Data can be referenced from Rhino or some external document. For example, you can reference a point created in a Rhino document. When you move the point in Rhino, its reference in Grasshopper updates as well. Grasshopper files are saved separately from Rhino files, and hence if the GH file has referenced data, the Rhino file needs to be saved and	

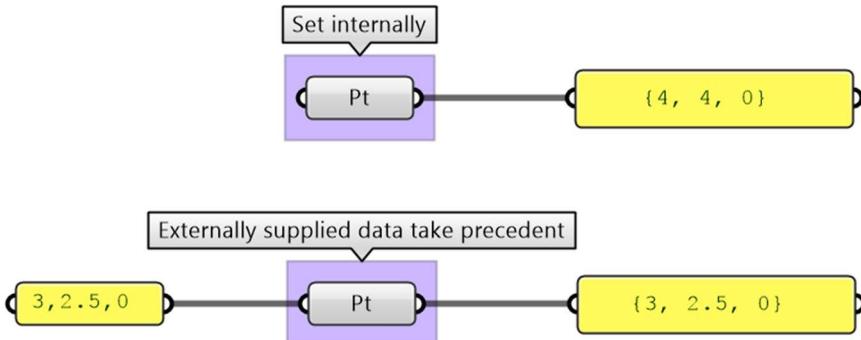
passed along with the GH file to avoid any loss of data.

Right mouse click the component, then "Set one point"



### 3- Externally supplied data

Data can be supplied from previous processes. This method is best suited for dynamic data or data controlled parametrically. Externally supplied data to a parameter takes precedent over the internal or referenced values (when both exist).



## 1\_6: Data types

All programming languages identify the kind of data used in terms of the values that can be assigned to and the operations and processes it can participate in. There are common data types such as **Integer**, **Number**, **Text**, **Bool** (true or false), and others. Grasshopper lists those under the **Params > Primitives** tab.

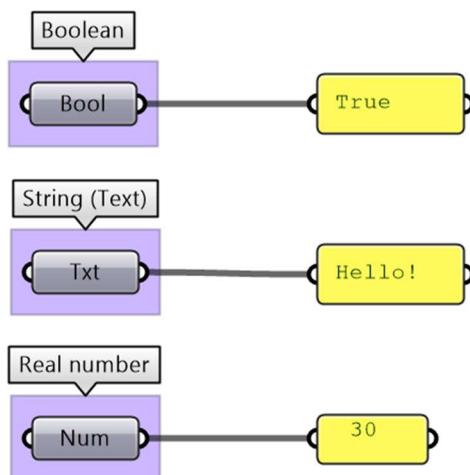


Figure (6): Examples of primitive data types common to all programming languages

Grasshopper supports geometry types that are useful in the context of 3D modeling such as **Point** (3 numbers for coordinates), **Line** (2 points), **NURBS Curve**, **NURBS Surface**, **Brep**, and others. All geometry types are included under the **Param> Geometry** tab in GH.

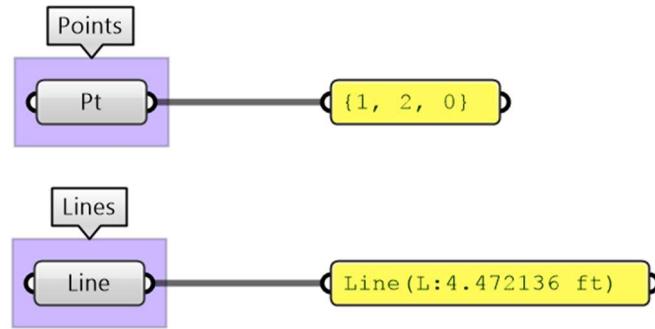


Figure (7): Examples of geometry data types

There are other mathematics types that designers do not usually use in 3D modeling, but are very common in parametric design such as **Domains**, **Vectors**, **Planes**, and **Transformation Matrices**. GH provides a rich set of tools to help create, analyze and use these types. To fully understand the mathematical as well as geometry types such as NURBS curves and surfaces, you can refer to the **Essential Mathematics for Computational Design** book by the author.

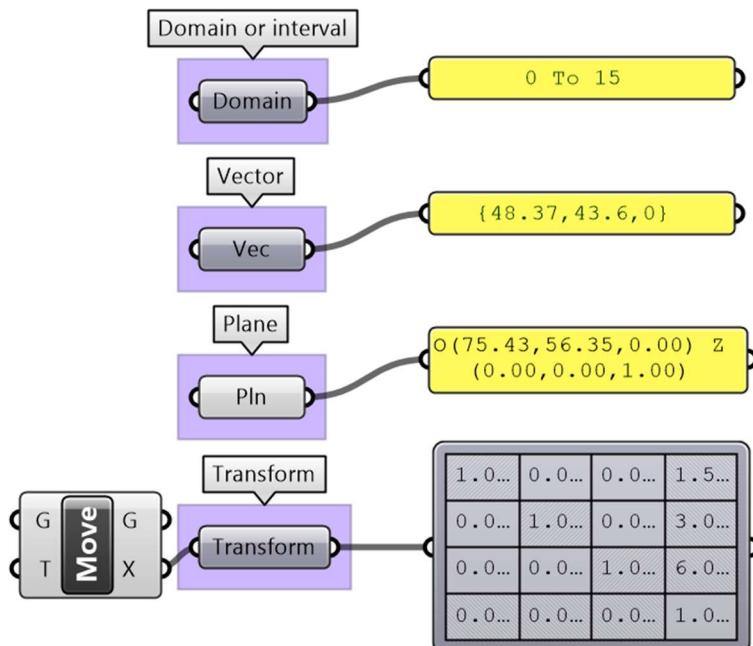


Figure (8): Examples of data types common in computer graphics

The parameters in GH can be used to convert data from one type to another (cast). For example if you need to turn a text into a number, you can feed your text into a **Number** parameter. If the text cannot be converted, you'll get an error.

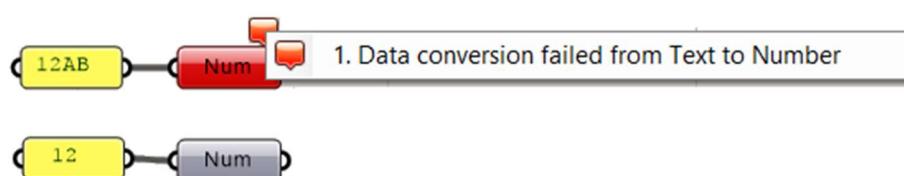


Figure (9): Data conversion (casting) inside parameters in Grasshopper

Grasshopper components internally convert input to suitable types when possible. For example, if you feed a “text” to **Addition** component, GH tries to read the text as a number. If a component can process more than one type, it uses the input type without conversion. For example, equality in an expression can compare text as well as numbers. In such case, make sure you use the intended type to avoid confusion.

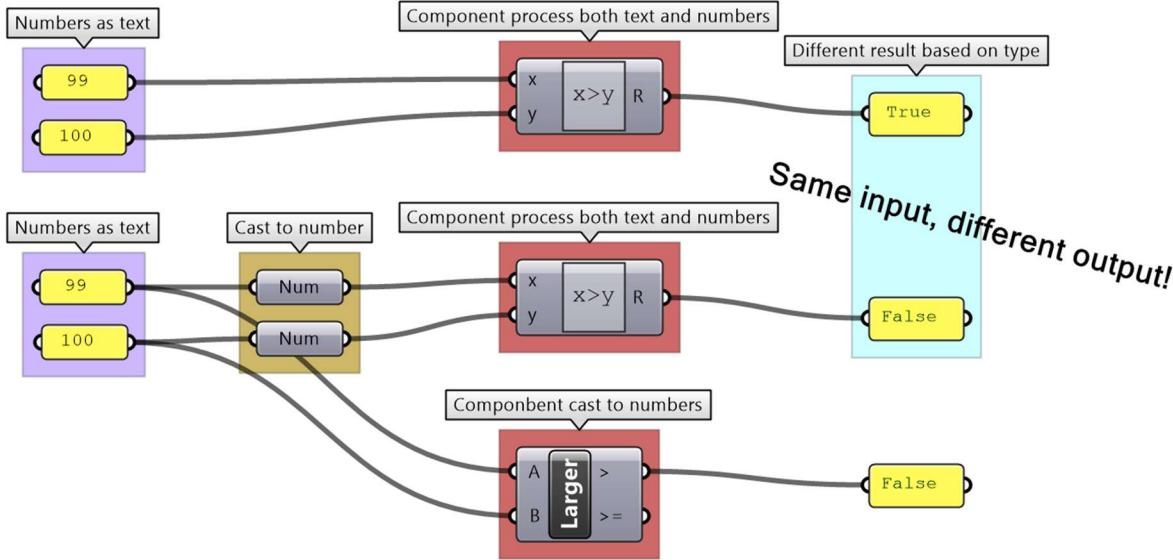


Figure (10): Some operations can be performed on multiple types. Cast to the intended type especially if the component is capable of processing multiple types (such as **Expression** in GH)

It is worth noting that sometimes GH components simply ignore invalid input (null or wrong type). In such cases, you are likely to end up with an unexpected result and hard to find the bug. It is very important to verify the output from each component before using it.

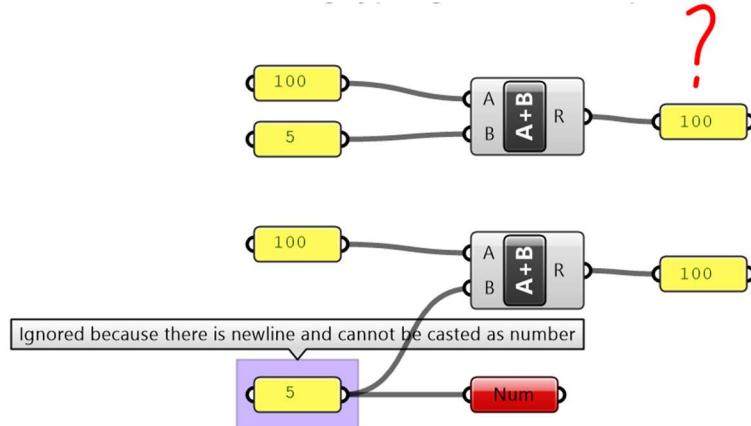


Figure (11): Invalid input is ignored and a default value is used. For example a number inside a **Panel** component can be interpreted as a text and hence become invalid input to an **Addition** component

## 1\_7: Processing data

Algorithmic designs use many data operations and processes. In the context of this book, we will focus on five categories: numeric and logical operations, analysis, sorting and selection.

### 1\_7\_1: Numeric operations

Numeric operations include operations such as arithmetic, trigonometry, polynomials and complex numbers. GH has a rich set of numeric operations, and they are mostly found under the **Math** tab.

There are two main ways to perform operations in GH. First by using designated components for specific operations such as **Addition**, **Subtraction** and **Multiplication**.

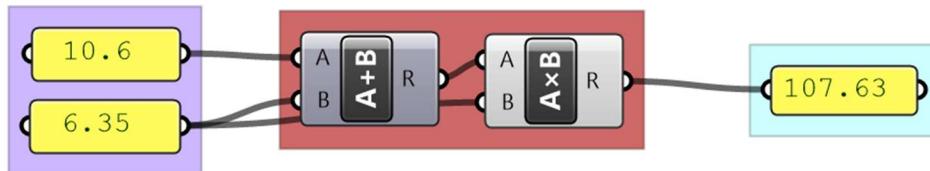


Figure (12): Examples of numeric operations components in GH

Second, use an **Expression** component where you can combine multiple operations and perform a rich set of math and trigonometry operations, all in one expression.

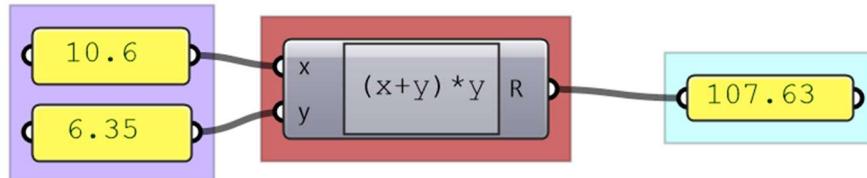


Figure (13): **Expression** component in GH can be used to perform multiple operations

The **Expression** component is more robust and readable when you have multiple operations.

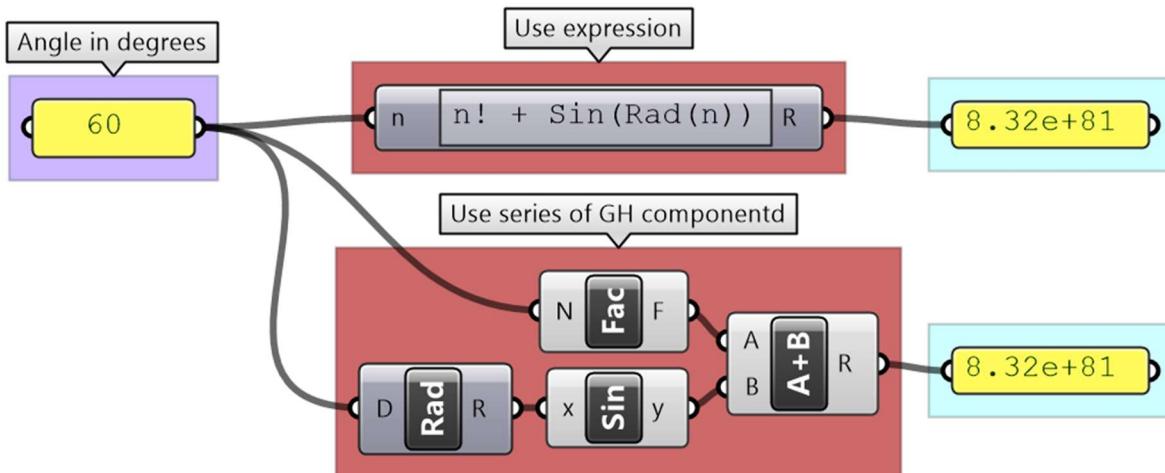


Figure (14): When a chain of operations are involved, using the **Expression component** is easier to maintain

Input to Expressions can be treated as text depending on the context.

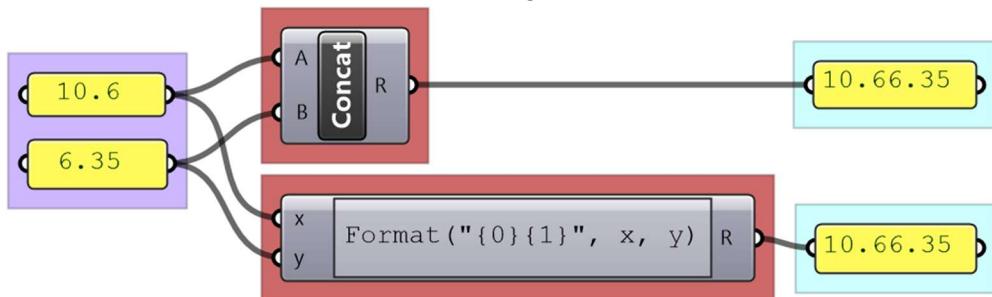


Figure (15): **Expression** component can process and format text

It is worth mentioning that most numeric input to components allow writing an expression to modify the input inline. For example, the Range component has N (number of steps) input. If you right mouse

click on “N”, you can set an expression. You always use “x” to represent the supplied input regardless of the name.

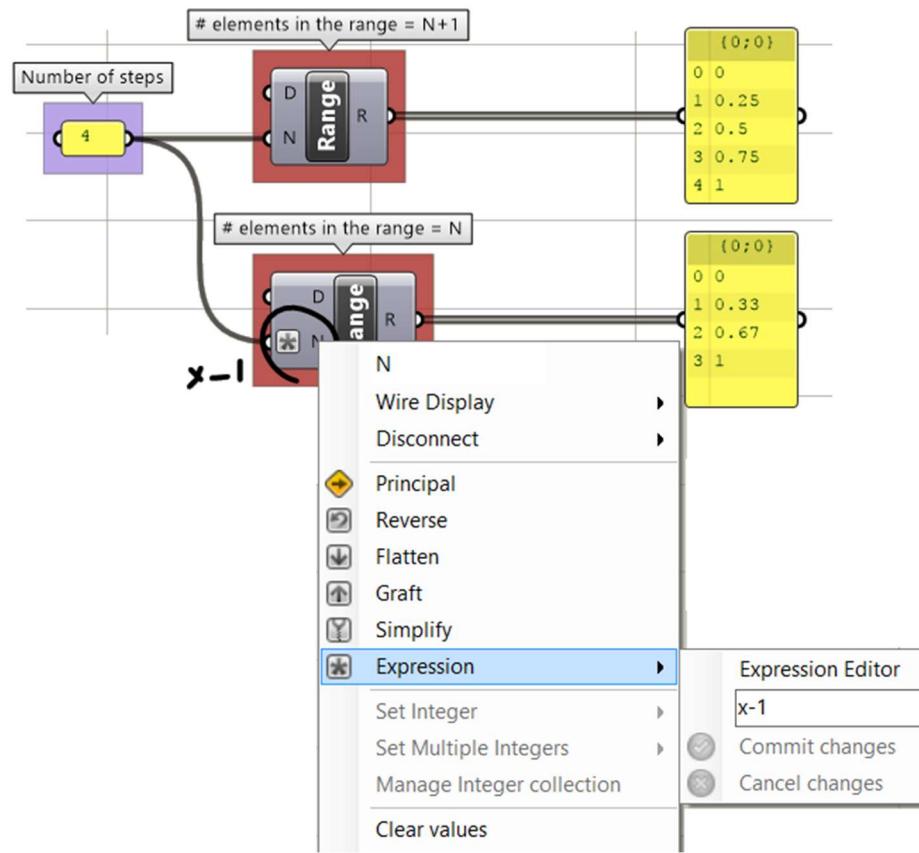


Figure (16): Expression can be set inside the input parameter. Variable “x” refers to the supplied input value

### 1\_7\_2: Logical operations

Main logical operations in GH include equalities, sets and logical operations.

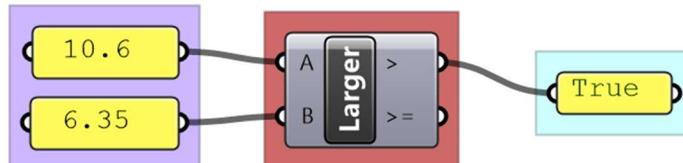


Figure (17): GH has multiple components to perform Logical operations

Logical operations are used to create conditional flow of data. For example, if you like to draw a sphere only when the radius is between two values, then you need to create a logic that blocks the radius when it is not within your limits.

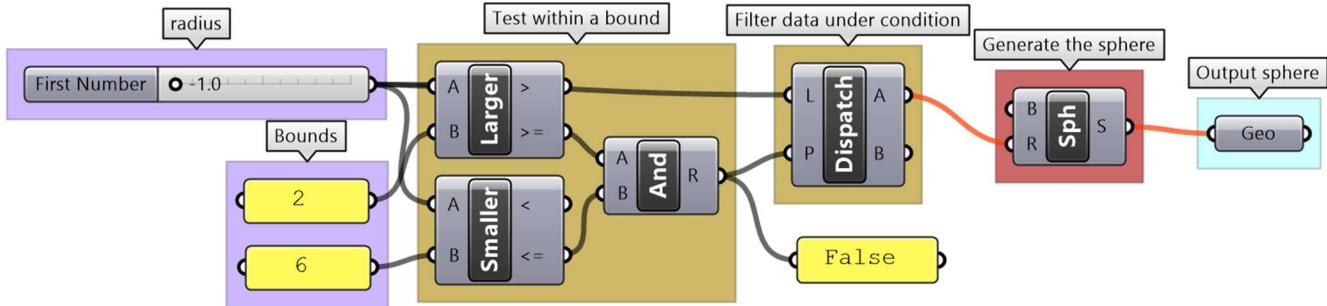


Figure (18): Data flow control using logical operations

### 1\_7\_3: Data analysis

There are many tools in GH to examine and preview data. **Panel** is used to show the full details of the data and its structure, while the **Parameter Viewer** shows the data structure only. Other analysis components include **Quick Graph** that plots data in a graph, and **Bounds** to find the limits in a given set of numbers (the min and max values in the set).

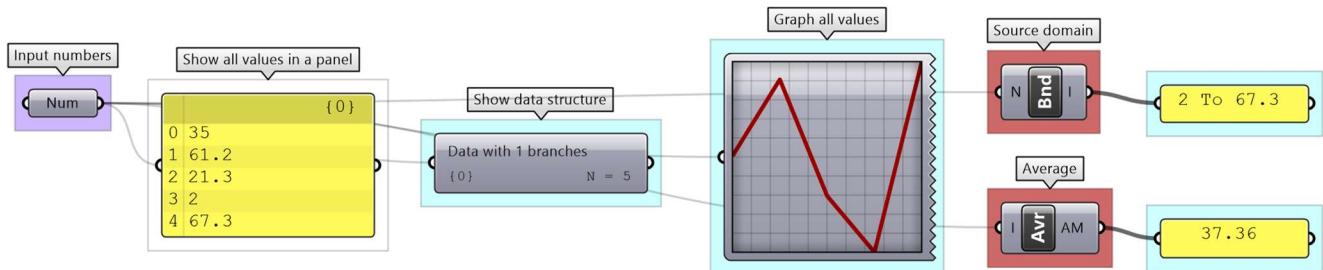


Figure (19): Some of the ways to analyze data in Grasshopper

### 1\_7\_4: Sorting

GH has designated components to sort numeric and geometry data. The **Sort List** component can sort a list of numeric keys. It can sort a list of numbers in ascending order or reverse the order. You can also use the **Sort List** component to sort geometry by some numeric keys, for example sort curves by length. GH has components designated to sort geometry sets such as **Sort Points** to sort points by their coordinates.

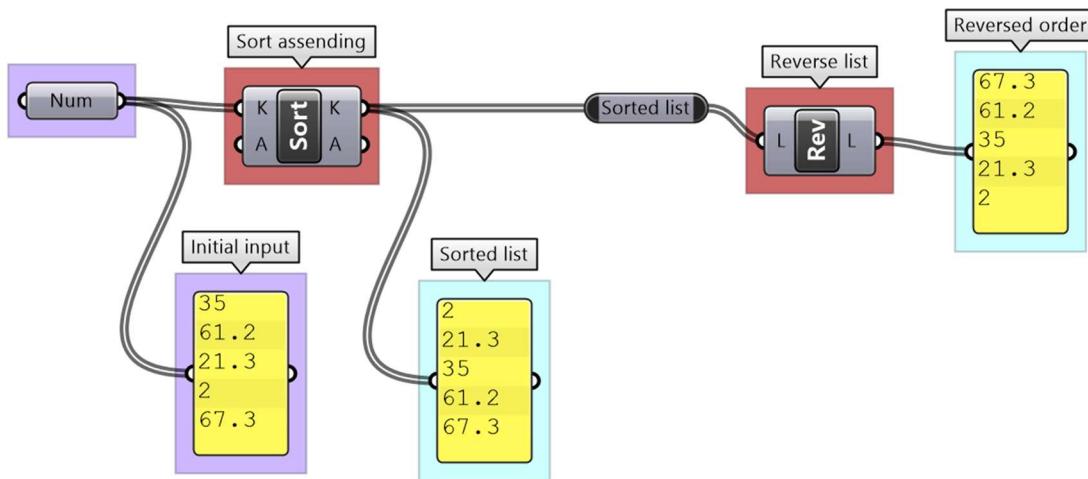


Figure (20): Sorting numbers in Grasshopper

## 1\_7\_5: Selection

3D modeling allows picking specific or a group of objects interactively, but this is not possible in algorithmic design. Data is selected in GH based on the location within the data structure, or by a selection pattern. For example **List Item** component allows selecting elements based on their indices.

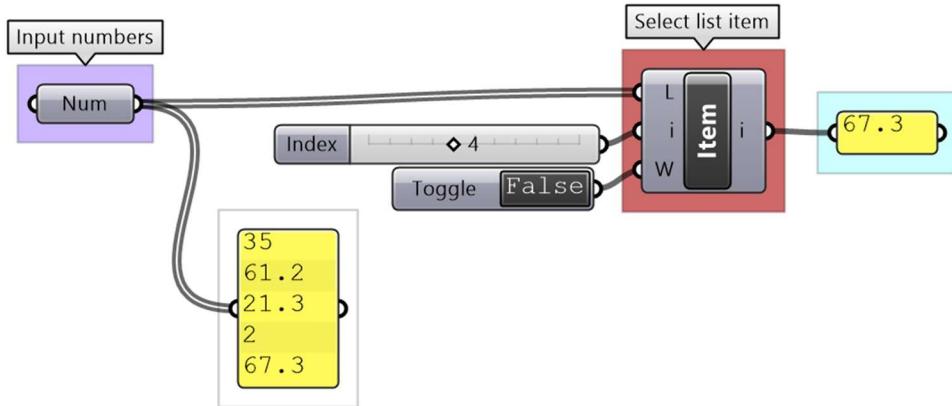


Figure (21): Select items from a list in Grasshopper

The **Cull Pattern** component allows using some repeated pattern to select a subset of the data.

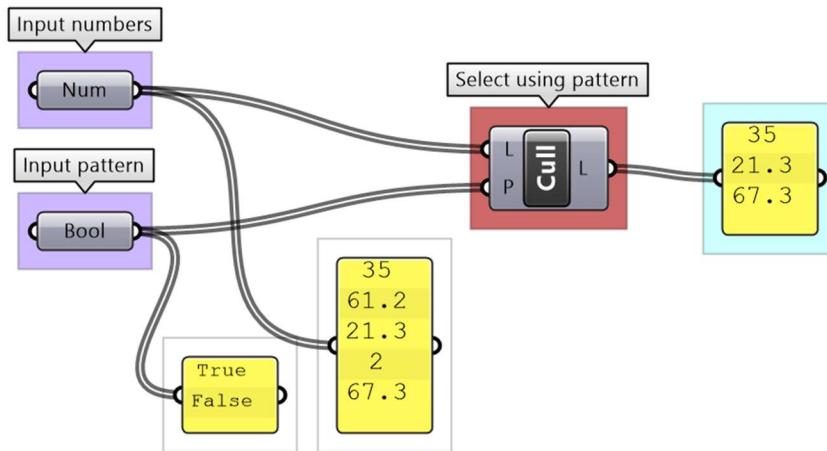


Figure (22): An example to select every other item in a list

As you can see from the examples, selecting specific items or using cull components yield a subset of the data, and the rest is thrown away. Many times you only need to isolate a subset to operate on, then recombine back with the original set. This is possible in GH, but involves more advanced operations. We will get into the details of these operations when we talk about advanced data structures in chapter 3.

## 1\_7\_6: Mapping

That refers to the linear mapping of a range of numbers where each number in a set is mapped to exactly one value in the new set. GH has a component to perform linear mapping called **ReMap**. You can use it to scale a set of numbers from its original range to a new one. This is useful to scale your range to a domain that suits your algorithm's needs and limitations.

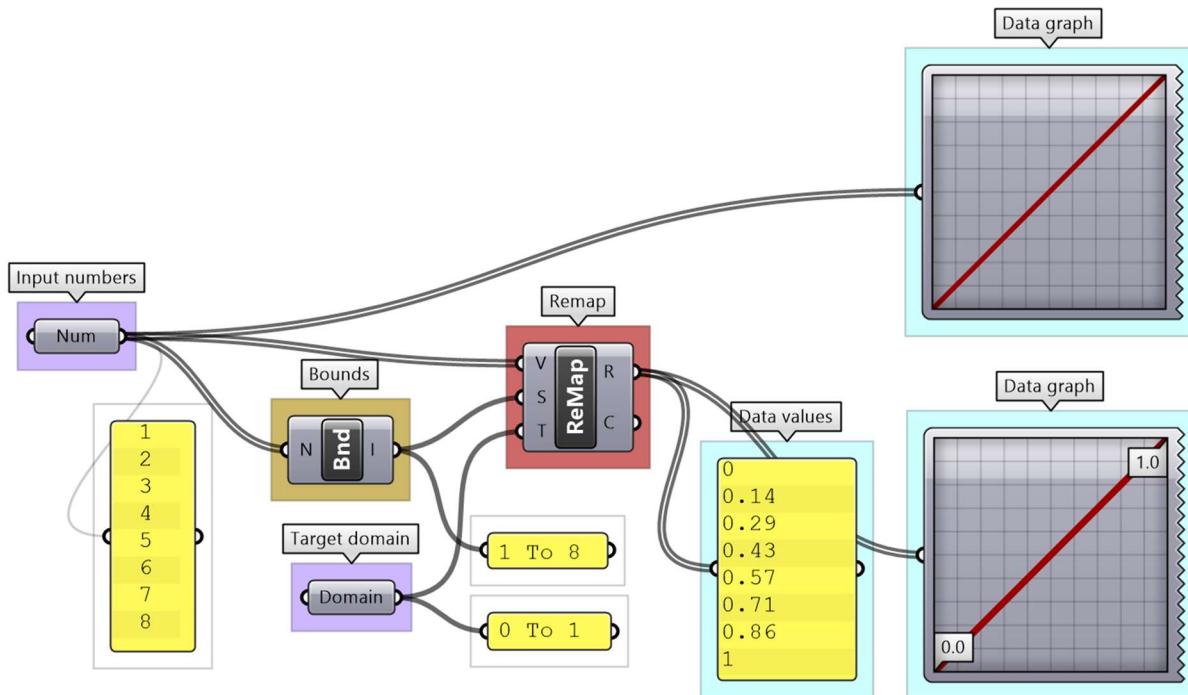


Figure (23): An example of linear remapping of numbers in Grasshopper

Converting data involves mapping. For example, you may need to convert an angle unit from degrees to radians ( GH components accept angles in radians only).

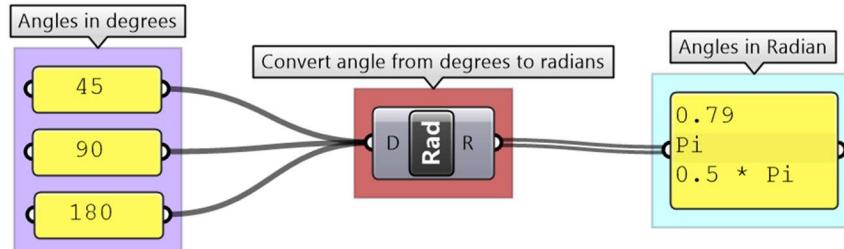


Figure (24): Convert angles from degrees to radians

As you know, parametric curves have “domains” (the range of parameters that evaluate to points on the curve). For example, if the domain of a given curve is between 12.5 to 51.3, evaluating the curve at 12.5 gives the point at the start of the curve. Many times you need to evaluate multiple curves using consistent parameters. Reparameterizing the domain of curves to some unified range helps solve this problem. One common domain to use is “0 To 1”. At the input of each curve in any GH component, there is the option to **Reparameterize** which resets the domain of the curve to be “0 to 1”.

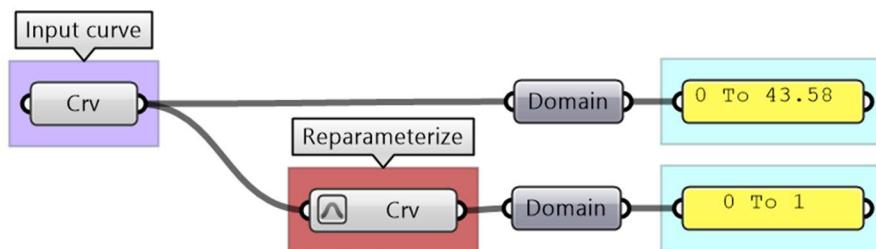
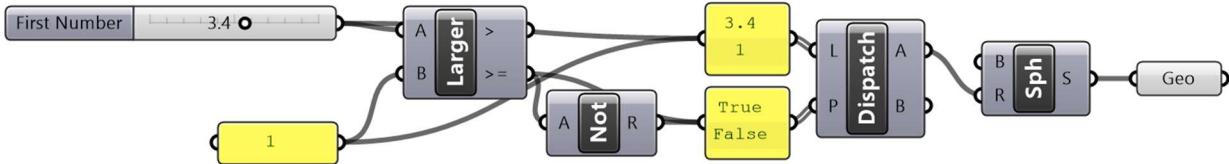


Figure (25): Normalize the domain of curves (set to 0-1). Use Reparameterize input flag in Grasshopper

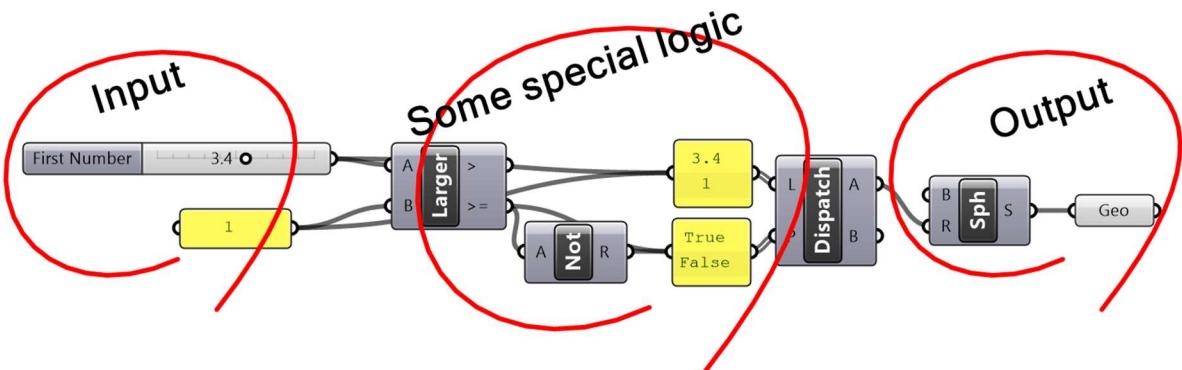
### 1-7-1: Flow control tutorial

What is the purpose of the following algorithm? Note and color code to describe the purpose of each part.



### Analyze the algorithm

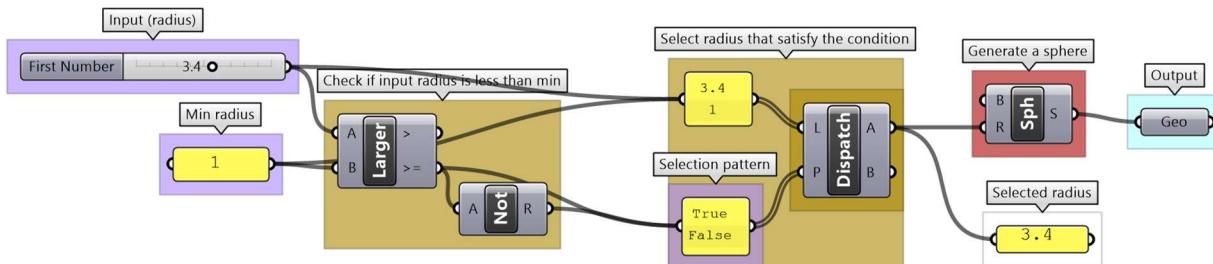
The algorithm has an output that is a sphere, a radius input and some conditional logic to process the radius.



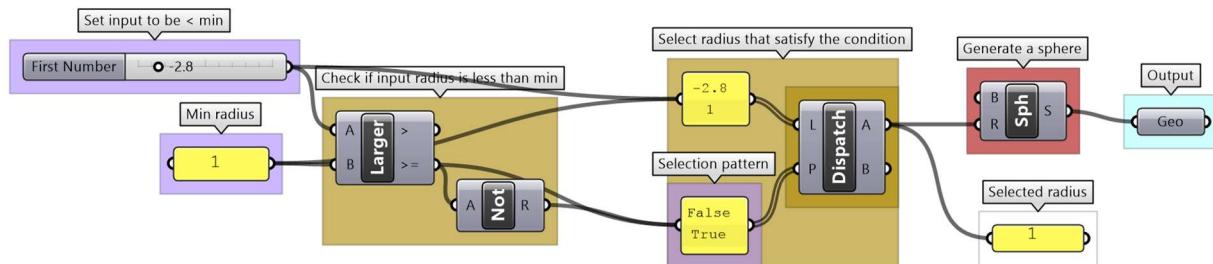
### Note and color-code the solution

From testing the output and following the steps of the solution it becomes apparent that the intention is to make sure that the radius of the sphere cannot be less than 1 unit.

Test with radius  $> 1$



Test with radius  $< 1$



## 1-7-2: Data processing tutorial

Given a list of point coordinates, do the following:

1- Analyze the list to understand the data.

3- Write an algorithm to use the input to construct a list of type *Point* with coordinates mapped to a domain between 3 and 9.

Note that the input list is organized so that the first 3 numbers refer to the x,y,z of the first point, the second 3 numbers belong to the second point and so on.

## Algorithm analysis

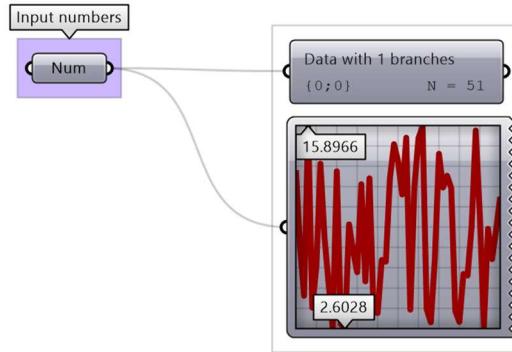
There is a list of 51 numbers (3 coordinates for each point implies the list includes 17 points)

Using a **QuickGraph**, we can see that the range of values are between 2.60 and 15.89. We can also see that the values are distributed randomly.

One other input is the target domain:

Target domain

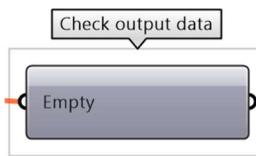
3 to 9



## Use the 4-step process to solve the algorithms

### Output

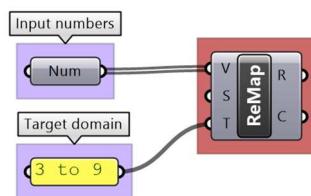
List of points



### Key Process #1 Remap Coordinates:

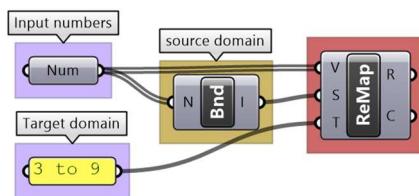
Map the coordinates list from its current domain to a new domain 3 to 9

Use **ReMap** component



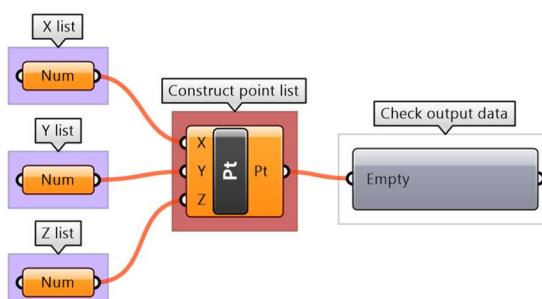
### Intermediate processes #1

The input domain is missing and can be extracted using **Bounds** component



### Key Process #2 Construct Points:

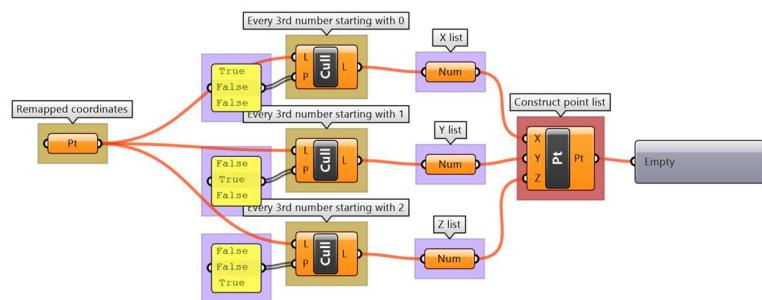
Construct points from coordinates  
Use **Construct Point (Pt)** component



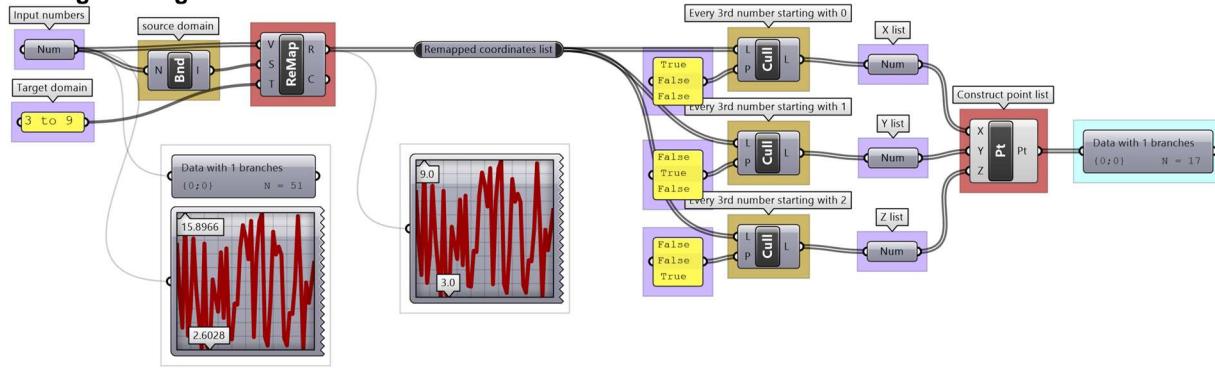
### Intermediate processes #2

Extract all X coordinates as one list, Y in another and Z in the third. Use **Cull Pattern** component with appropriate pattern to extract each coordinate as a separate list.

The input to **Cull** is the remapped points from process #1



### Putting it all together



## 1\_8: Pitfalls of algorithmic design

Writing elegant algorithms that are efficient and easy to read and debug is hard. We explained in this chapter how to write algorithms with style using color-coding and labeling. We also articulated a 4-step process to help develop algorithms. Following these guides help minimize bugs and improve the readability of the scripts. We will list a few of the common issues that lead to incorrect or unintended result.

### 1\_8\_1: Invalid or wrong type input

If the input is of the wrong type or is invalid, GH changes the color of components to red or orange to indicate an error warning, with feedback about what the issue might be. This is helpful, but sometimes faulty input goes unnoticed if the components assign a default value, or calculate an alternative value to replace the input, that is not what was intended. It is a good practice to always double check the input (hook to a panel or parameter viewer and label the input). To avoid using wrong types, it is advisable to convert to the intended type to ensure accuracy.

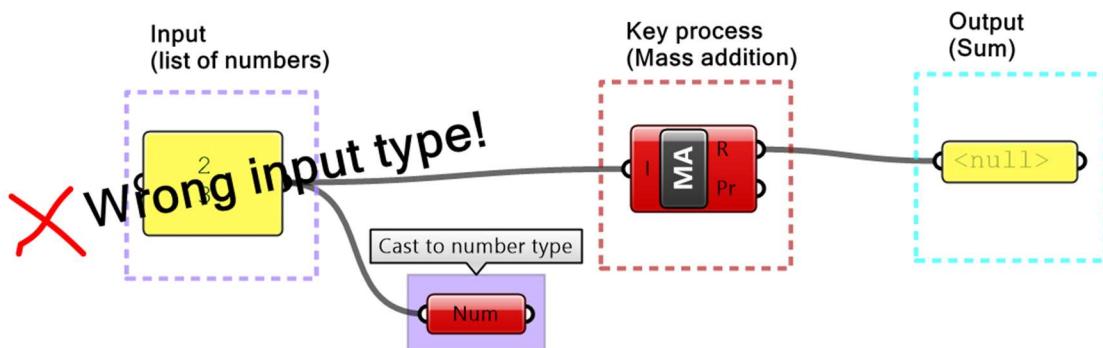


Figure (26): Error resulting from wrong input type

### 1\_8\_2: Incorrect input

Input is prone to unintended change via intermediate processes or when multiple users have writing access to the script. It is very useful to preview and verify all key input and output. The **Panel** component is very versatile and can help check all types of values. Also you can set up guarding logic against out of range values or to trap undesired values.

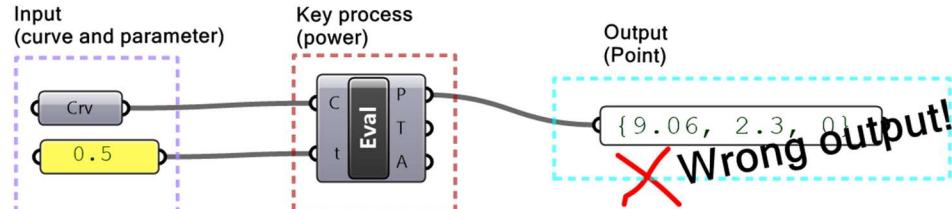


Figure (27): Error resulting from incorrect input. Cannot assume curve domain is 0-1 and use 0.5 to evaluate the midpoint.

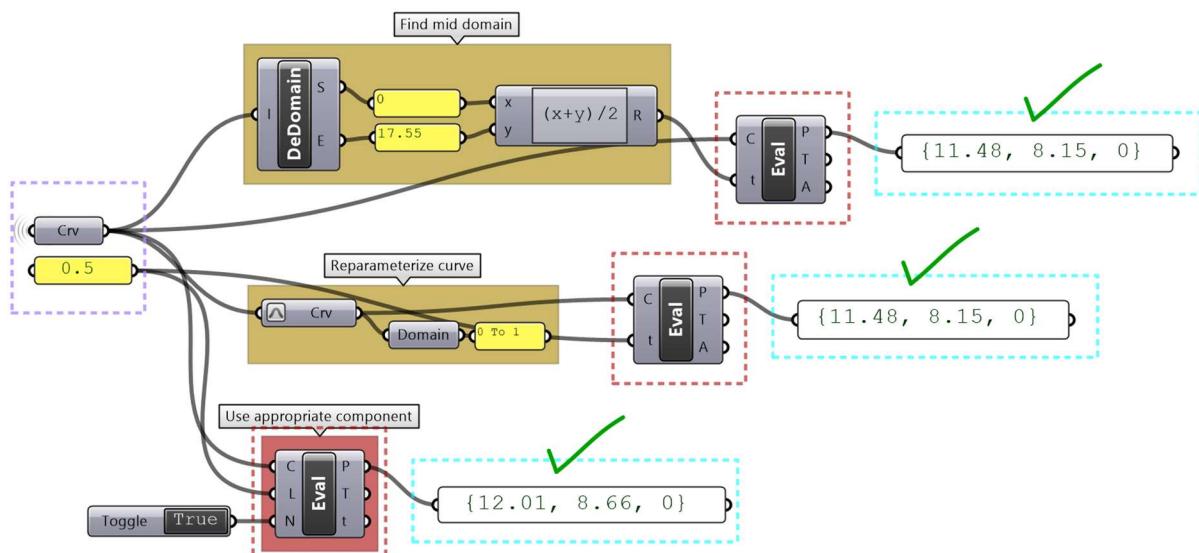


Figure (28): Example of a robust solution to evaluate the midpoint of a curve

### 1\_8\_3: Incorrect order of operation

You should try to organize your solutions horizontally or vertically to clearly see the sequence of operations. You should also check the output from each step to make sure it is as expected before continuing on your code. There are also some techniques that help consolidate the script, for example use **Expression** when multiple numeric and math operations are involved. The following highlights some unfavorable organization.

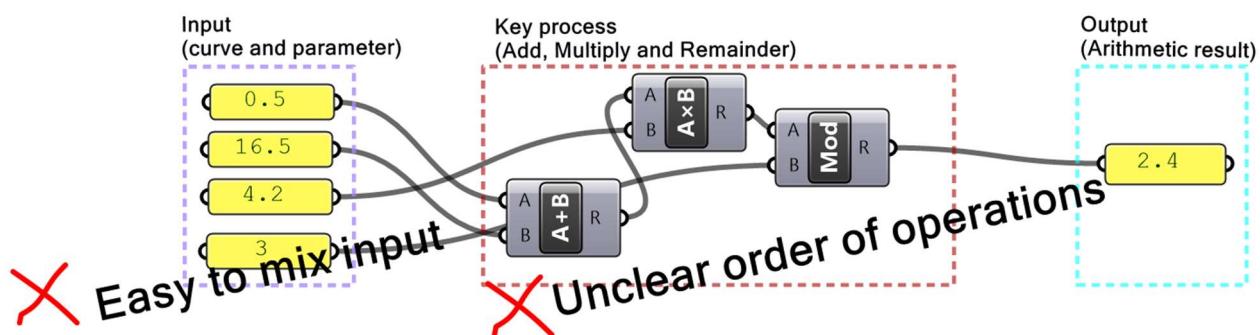
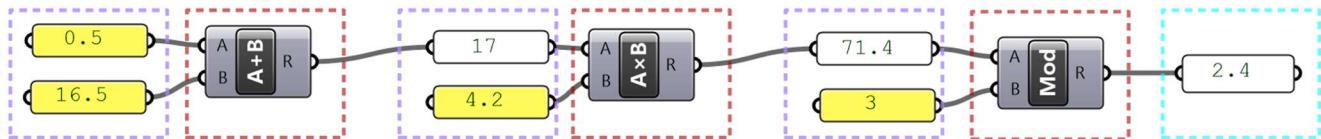


Figure (29): Easy to confuse input to operations with poor organization

The following shows how to rewrite the same code to make it less error prone.

### View and align input



### Consolidate/simplify processes when possible

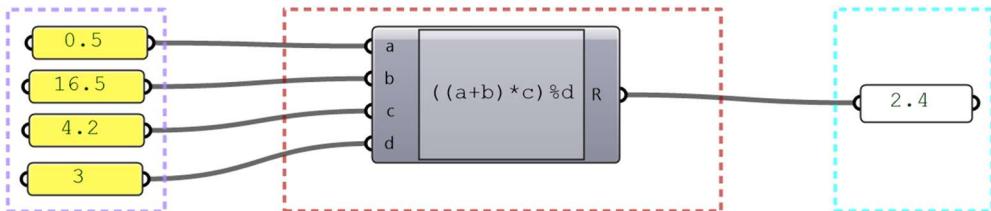


Figure (30): Best practices to align input with processes, or use **Expressions**

### 1\_8\_4: Mismatched data structures

Mismatched data structures as input to the same process or component is particularly tricky to guard against in GH, and has the potential to spiral the solution out of memory. It is essential to test the data structure of all input (except trivial ones) before feeding into any component. It is also important to examine desired matching under different scenarios (data matching will be explained at length later).

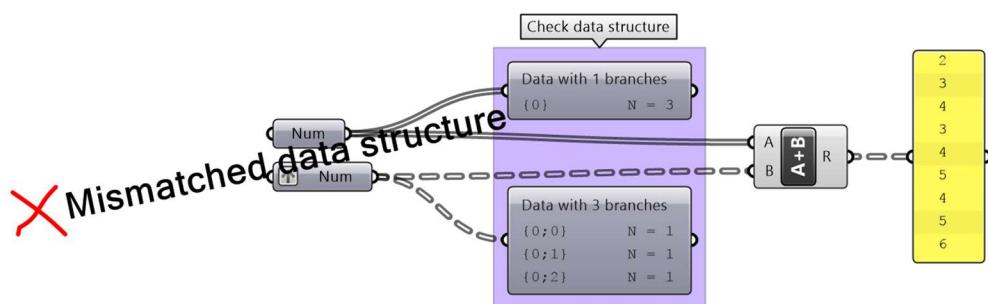
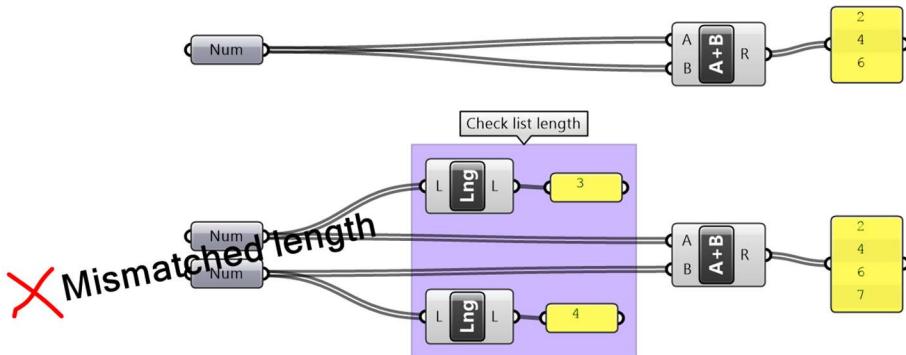


Figure (31): Mismatched data structures of input can cause errors in the output

### 1\_8\_5: Long processing time

Some algorithms are time consuming, and you simply have to wait for it to process, but there are ways to minimize the wait when it is unnecessary. For example, at the early cycles of development, you

should try to use a smaller set of data to test your solution with before committing the time to process the full set of data. It is also a good practice to break the solution into stages when possible, so you can isolate and disable the time consuming parts. Also, it is often possible to rewrite your solution to be more optimized and consume less time. Use the GH **Profiler** to test processing time. When a solution takes far too long to process or crashes, you should do the following: before you reopen the solution, disable it, and disconnect the input that caused the crash.

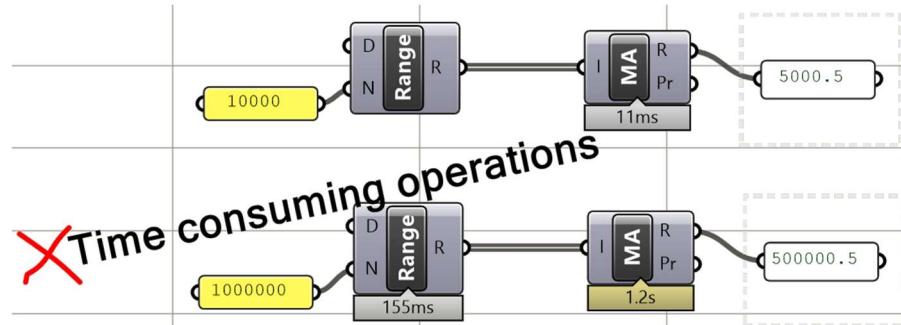


Figure (32): Grasshopper Profiler widget help observe processing time

### **1\_8\_6: Poor organization**

Poorly organized definitions are not easy to debug, understand, reuse or modify. We can't stress enough the importance of writing your definitions with styles, even if it costs extra time to start with. You should always color code, label everything, give meaningful names to variables, break repeated operations into modules and preview your input and output.



Figure (33): Poor organization in visual programming make the code hard to read or debug

## **1\_9: Algorithms tutorials**

### **1\_9\_1: Unioned circles tutorial**

Use the 4-step process to design an algorithm that unions 2 circles, given the following: both are located on the XY-Plane. The first circle (Cir1) has a center ( $C_1 = (2,2,2)$ ) and radius ( $R_1$ ) = some random number between 3 and 6. The second circle (Cir2) has a center ( $C_2$ ) shifted to the right of ( $C_1$ ) by an amount equal to  $R_1$  along positive X-Axis.  $R_2 = R_1 * 1.2$

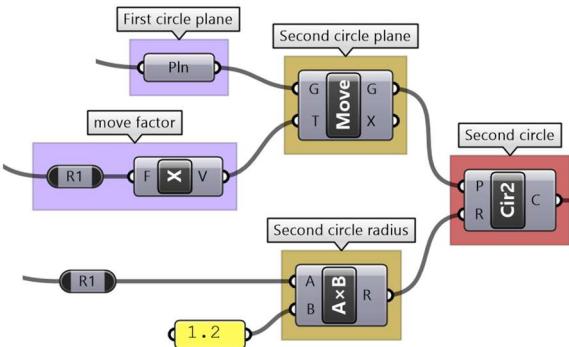
Analyze the question and the flow of the solution

<b>Output</b> Curve for the region union	
<b>Key Process: union of 2 circles</b> Use the <b>Region Union</b> component that takes curves and a plane	
<b>Input to the region union</b> Identify the input needed and use given input when relevant.  The plane for region union has been given. The 2 circles need their own plane and radius. The center of the plane is the center of the circle.	
<b>Intermediate process to generate the center and plane of the 1st circle</b>  Construct a center from the given coordinates. Create a plane using <b>Plane Origin</b> component and use the constructed center and XY-Plane  The radius is from a random number between 3 and 6. Use <b>Random</b> component to create the radius	

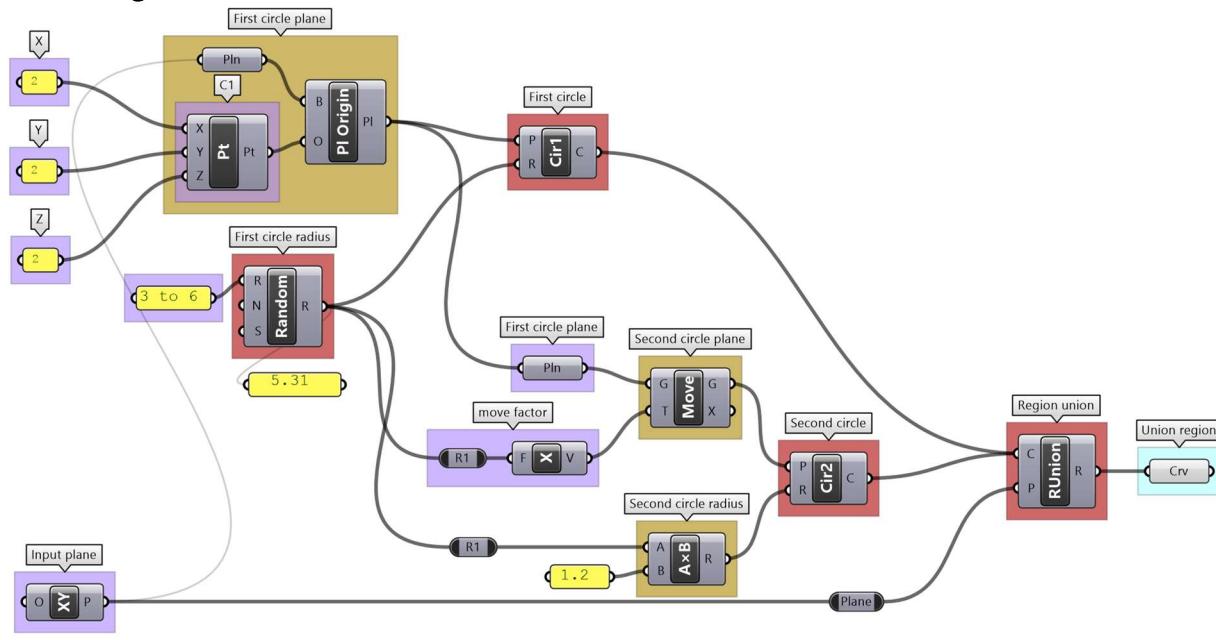
### Intermediate process to generate the center and plane of the 2nd circle

Calculate the 2nd circle plane by moving the first circle plane along the x-axis by an amount = first radius

Calculate the 2nd circle radius by multiplying the first radius by 1.2



### Put it all together



### 1\_9\_2: Sphere with bounds tutorial

Use the 4-step process to draw a sphere with a radius between 2 and 6. If input is less than 2, then set the radius to 2, and if input radius is greater than 6, set the radius to 6. Use a number slider to input the radius and set between 0 and 10 to test. Make sure your solution is well organized, color-coded and labeled properly

#### Use the 4-step process to solve the algorithms

##### Output

The sphere as geometry

Output sphere

Geo

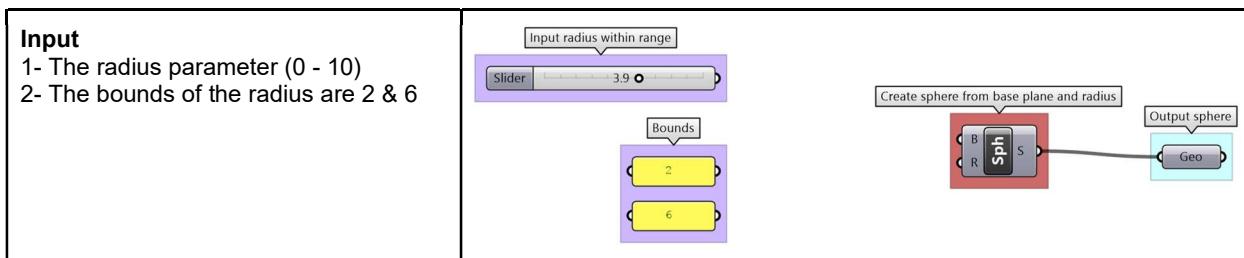
##### Key Process: create a sphere

Map the coordinates list from its current domain to a new domain 3 to 9  
Use **ReMap** component

Create sphere from base plane and radius

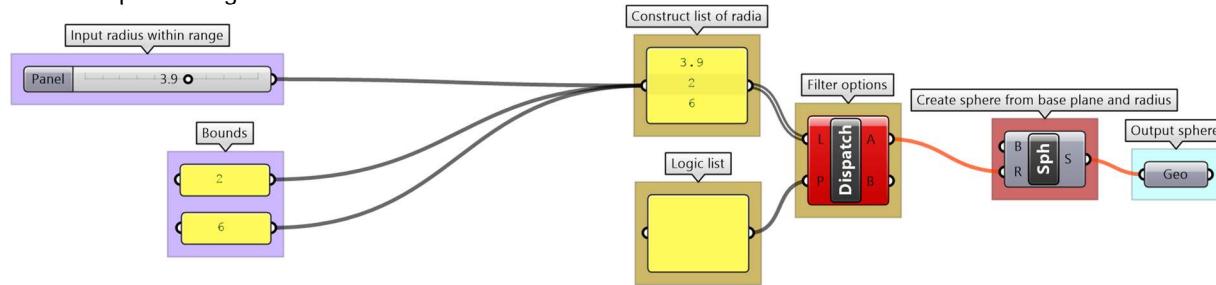
Sph

Output sphere  
Geo



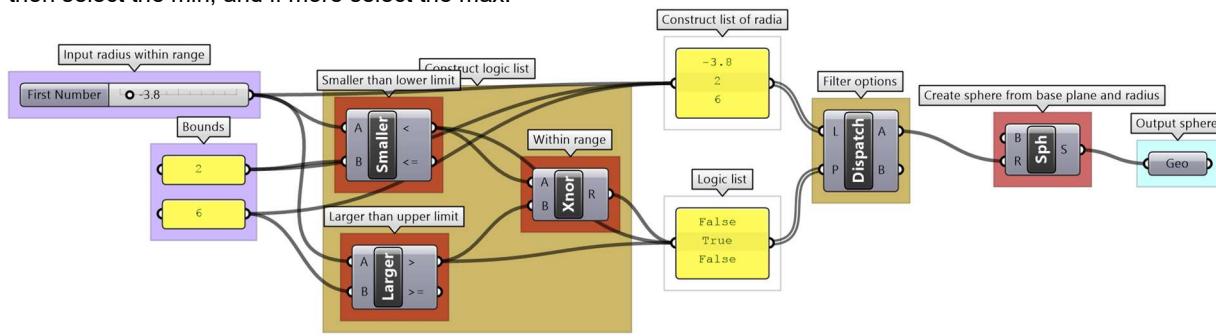
### Intermediate processes #1

Construct a selection logic of radii and pattern. The radii is a list of the values from the slider, min and max. The list of pattern is generated to select the correct radius value



### Intermediate processes #2

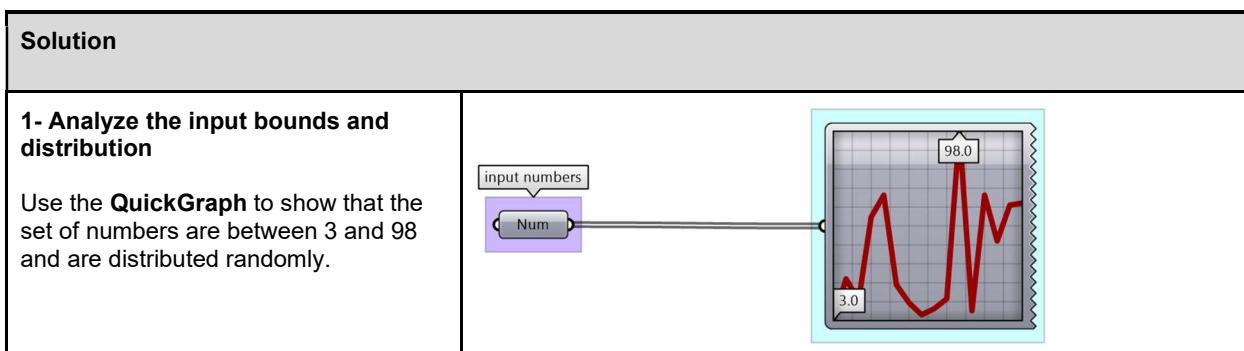
The selection logic checks if the radius from the slider is between the bounds, then set it to be selected, if less, then select the min, and if more select the max.



## 1\_9\_3: Data operations tutorial

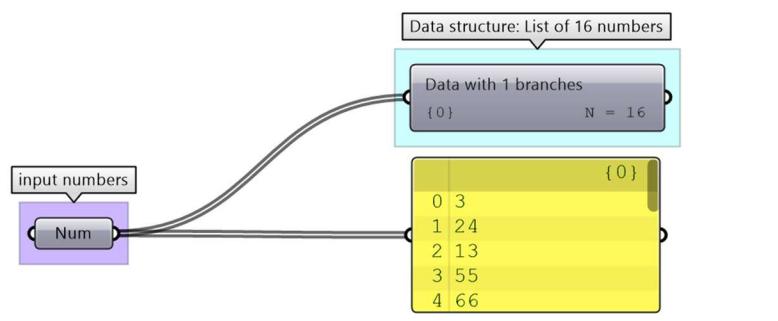
Given the numbers embedded in the Number parameter below:

- 1- Analyze input in terms of bounds and distribution
- 2- View the data and how it is structured
- 3- Extract even numbers
- 4- Sort numbers descending
- 5- Remap sorted numbers to (100 to 200)



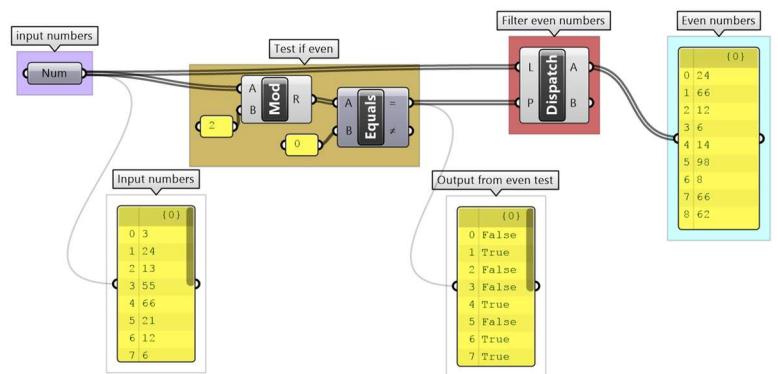
## 2- Analyze the input data structure and values

Use the **Panel** and **Parameter Viewer** to show there are 16 elements organized in a list



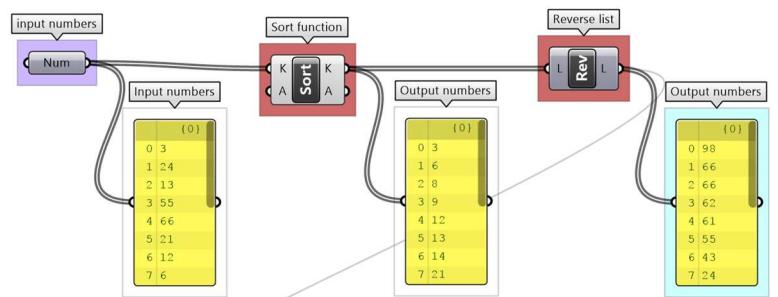
## 3- Extract Even numbers

Create the logic to test if a number is even (divisible by 2 without a remainder) and use **Dispatch** to extract even numbers



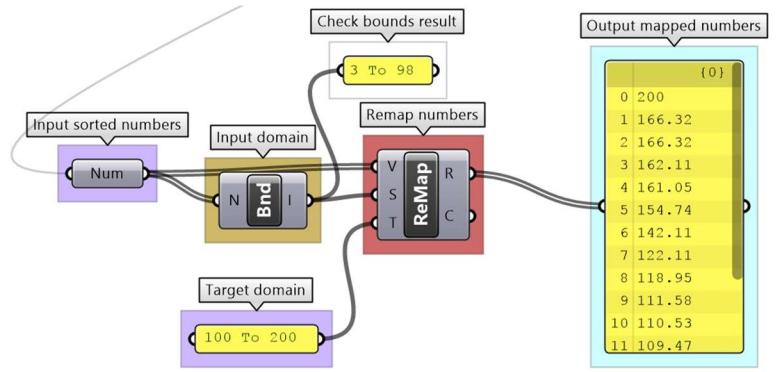
## 4- Sort numbers descending

The **Sort List** component sorts numbers in ascending order. Use **Reverse List** component to further process the list to order descending



## 5- Remap to 100-200

Check the input range and use Remap component to scale data to be between 100-200

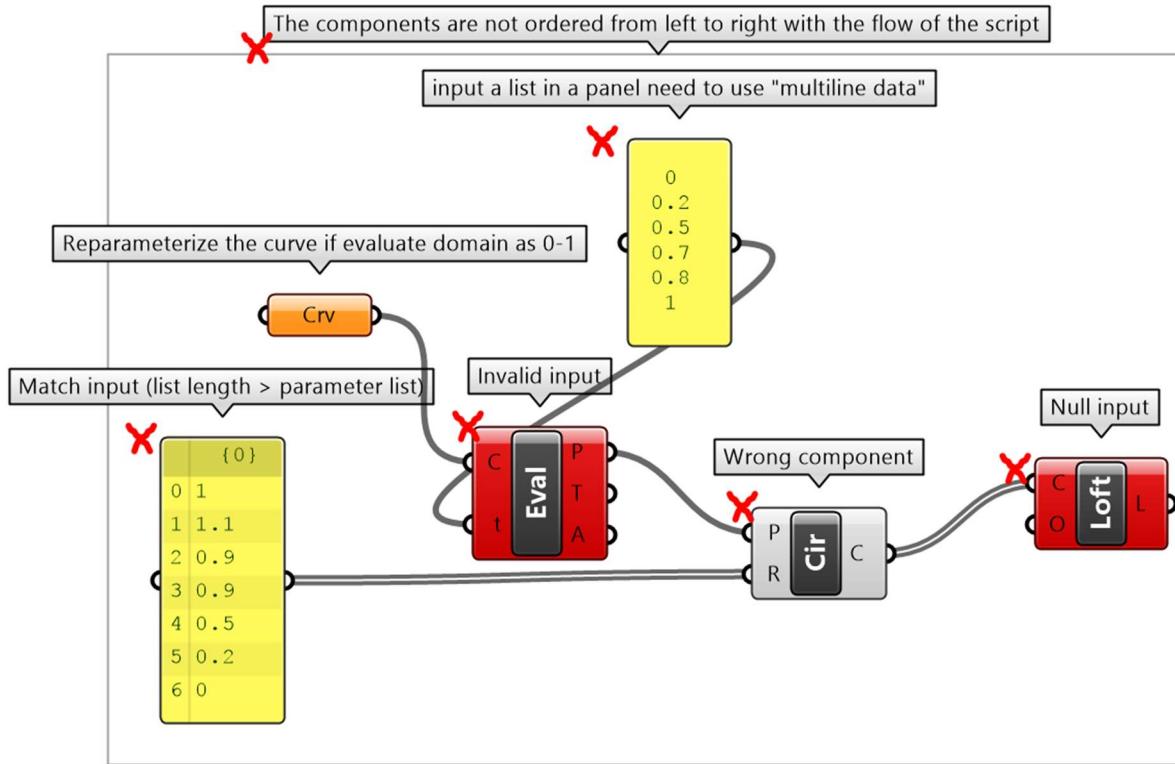


## 1\_9\_4: Pitfalls tutorial

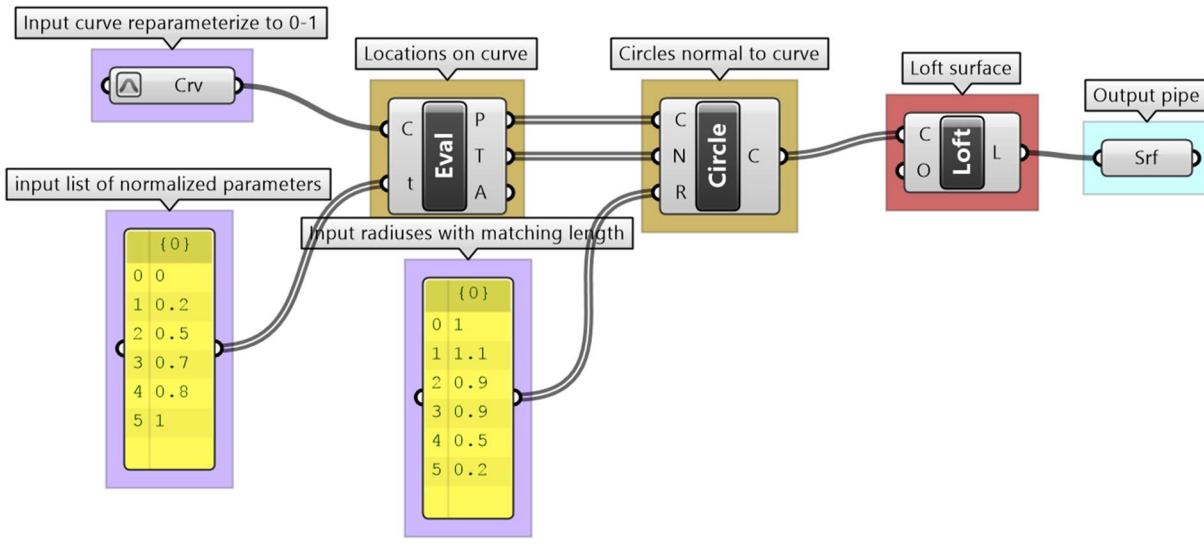
Analyze what the following algorithm is intended to do, identify the errors that are preventing it from working as intended, then rewrite to fix the errors. Organize to reflect the algorithm flow, label and color-code.

### Solution

### Mark the errors:



### Fix the errors and rewrite the solution with labels:



# Chapter Two: Introduction to Data Structures

All algorithms involve processing input data to generate a new set of data as output. Data is stored in well-defined structures to help access and manipulate efficiently. Understanding these structures is the key for successful algorithmic designs. This chapter includes an in-depth review of the basic data structures in Grasshopper.

## 2\_1: Overview

Grasshopper has three distinct data structures: **single item**, **list** of items and **tree** of items. GH components execute differently based on input data structures, and hence it is essential to be fully aware of the data structure before using. There are tools in GH to help identify the data structure. Those are **Panel** and **Param Viewer**.

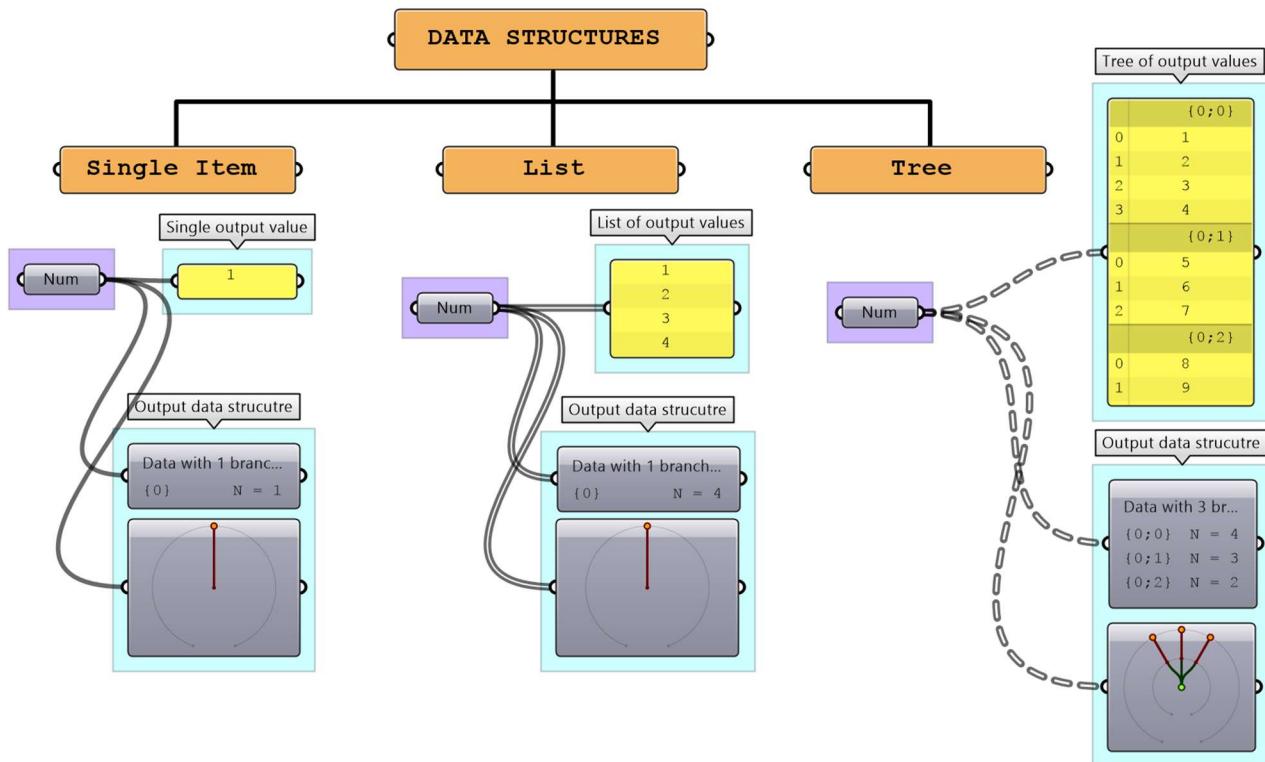


Figure (34): Data structures in Grasshopper

Processes in GH execute differently based on the data structure. For example, the **Mass Addition** component adds all the numbers in a list and produces a single number, but when operating on a tree, it produces a list of numbers representing the sum of each branch.

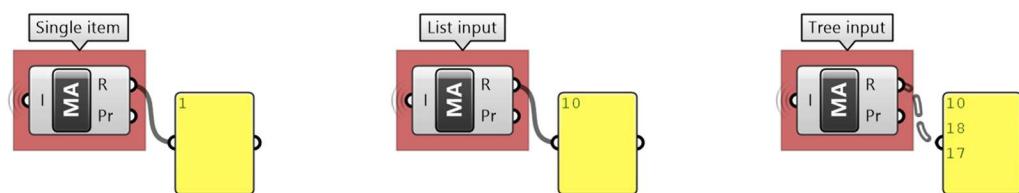
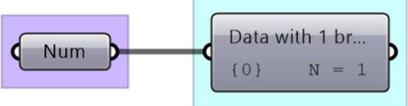
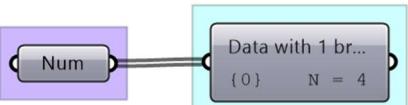
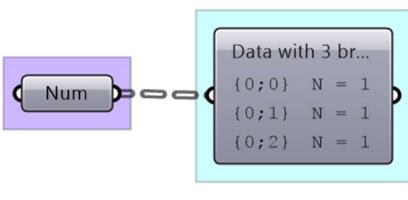


Figure (35): Components execute differently based on the data structures. Result of adding numbers from Figure(34)

The wires connecting the data with components in GH offer additional visual reference to the data structure. The wire from a single item is a simple line, while the wire connecting a list is drawn as a double line. A wire output from a tree data structure is a dashed double line. This is very useful to quickly identify the structure of your data.

Display the data structure	Example
<b>Item:</b> single branch with single item <b>Wire display:</b> single line	
<b>List:</b> single branch with multiple items <b>Wire display:</b> double line	
<b>Tree:</b> multiple branches with any number of items per branch <b>Wire display:</b> double dashed line	

## 2\_2: Generating lists

There are many ways to generate lists of data in GH. So far we have seen how to directly embed a list of values inside a parameter or a panel (with multiline data). There are also special components to generate lists. For example, to generate a list of numbers, there are three key components: **Range**, **Series** and **Random**. The **Range** component creates equally spaced range of numbers between a min and max values (called domain) and a number of steps (the number of values in the resulting list is equal to the number of steps plus one).

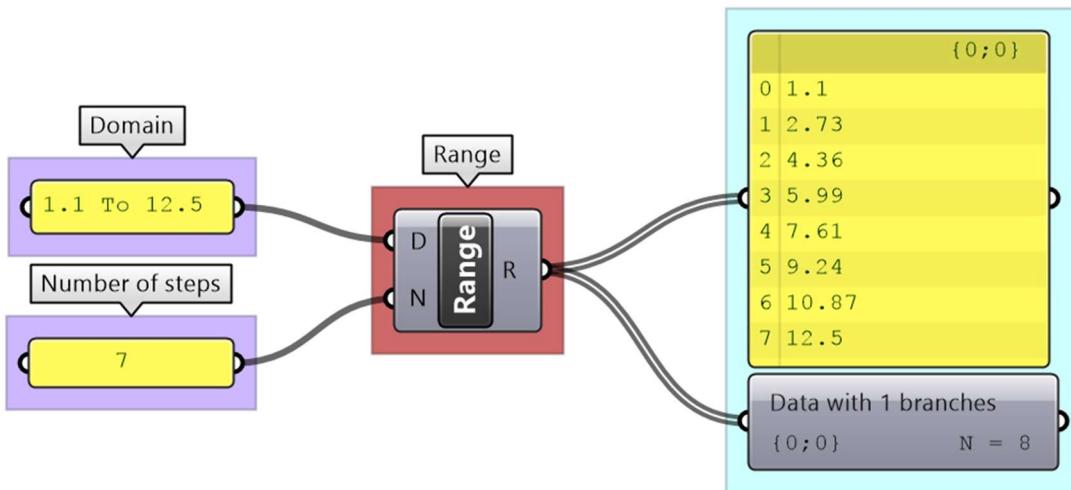


Figure (36): Generate a list of 8 numbers using the **Range** component in Grasshopper

The **Series** component also creates an equally spaced list of numbers, but here you set the starting number, step size and number of elements.

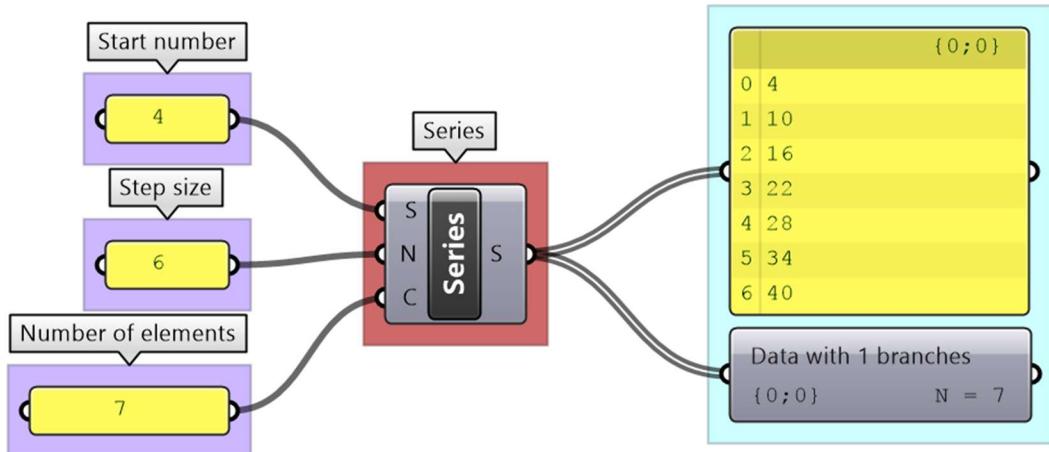


Figure (37): Generate a list of 7 numbers using the **Series** component in Grasshopper

The **Random** component is used to create random numbers using a domain and a number of elements. If you use the same seed, then you always get the same set of random numbers.

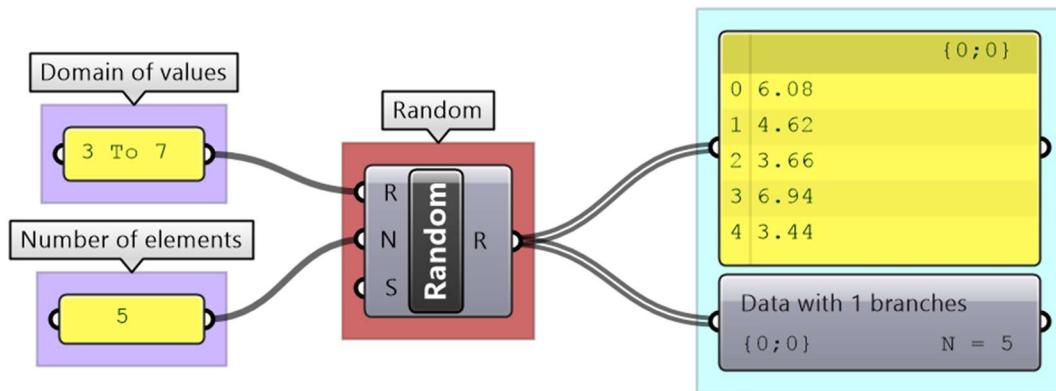


Figure (38): Generate a list of numbers using the **Random** component in Grasshopper

Lists can be the output of some components such as **Divide** curve (the output includes lists of points, tangents and parameters). Use the **Panel** component to preview the values in a list and **Parameter Viewer** to examine the data structures.

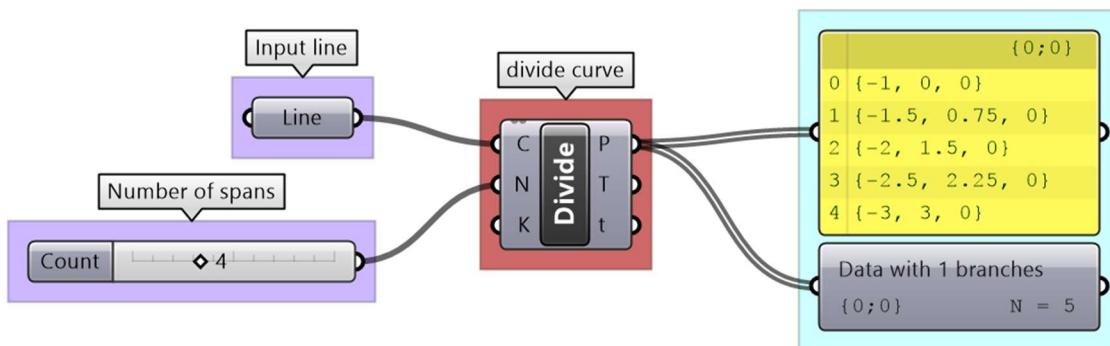


Figure (39): **Divide Curve** takes a single input (curve) and generate lists of output

## 2\_2\_1 Generating lists tutorial

Explore 4 different ways to create circles. Use different data sources and data structures.

Description	Grasshopper solution
Directly set a circle in a parameter	<p>Directly set</p>
Set the plane input to the default XY-Plane (internal). Supply a list of radii using <b>Range</b> component	<p>List input</p> <p>Plane:internal, Radius:Supplied range</p>
Supply one value for the center. Normal is set to default (internal). List of radii using <b>Random</b> component	<p>Center:Supplied, Radius:Supplied random list</p>
Circle from 3 points: A: set internally to one value B: Supply one value C: Supply a list of values using the <b>Series</b> component to set a list of Z coordinates	<p>A:Internal, B:Supplied, C:Supplied list</p>

## 2\_3: List operations

GH offers an extensive list of components for list operations and list management. We will review a few of the most commonly used ones.

You can check the length of a list using the **List Length** component, and access items at specific indices using the **List Item** component.

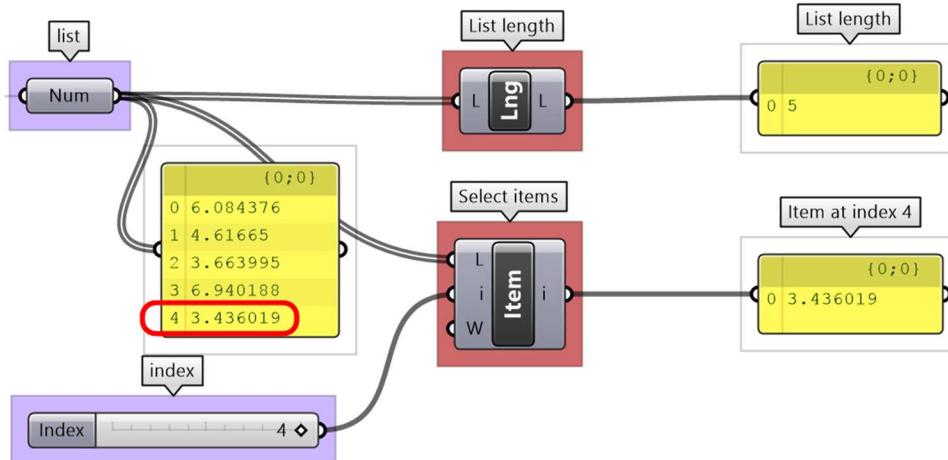


Figure (40): Examples of list operations in Grasshopper

Lists can be reversed using the **Reverse List** component, and sorted using the **Sort List** component.

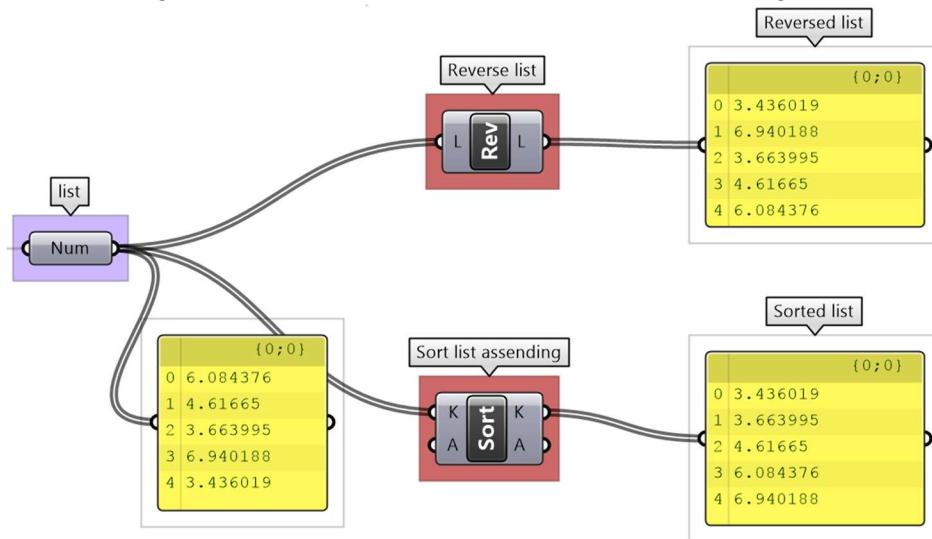


Figure (41): Lists can be reversed or sorted using designated components in Grasshopper

Components such as **Cull Patterns** and **Dispatch** allow selecting a subset of the list, or splitting the list based on a pattern. These components are very commonly used to control data flow and select a subset of the data.

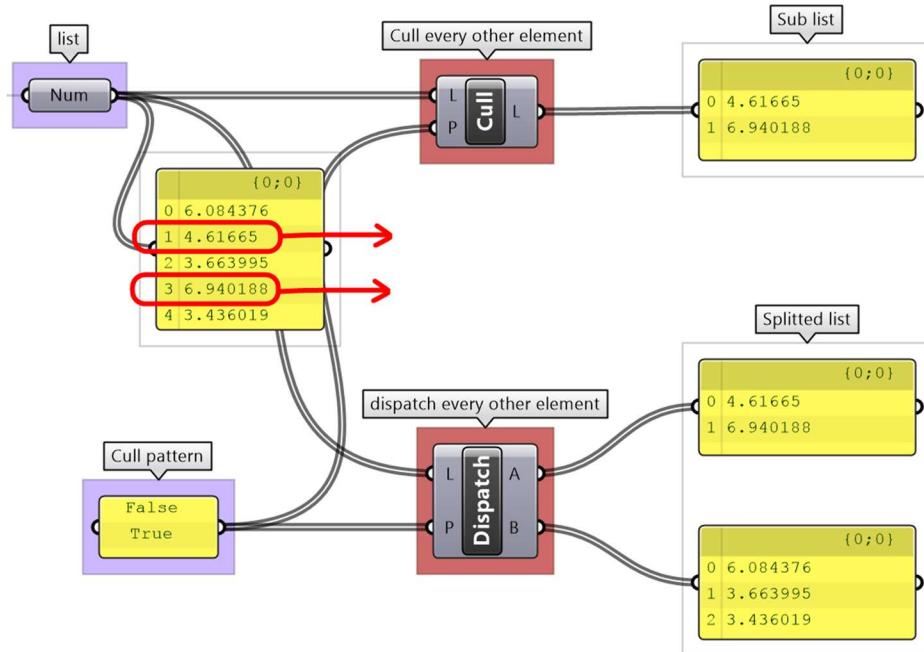


Figure (42): Cull part of a list using components such as **Cull Pattern** and **Dispatch**

The **Shift List** component allows shifting a list by any number of steps. That helps align multiple lists to match in a particular order.

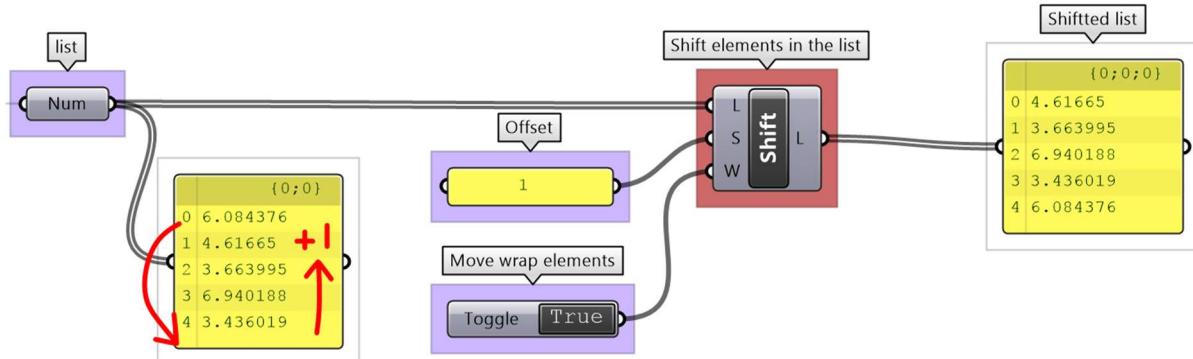


Figure (43): **Shift** operation in Grasshopper

The **Subset** component is another example to select part of a list based on a range of indices.

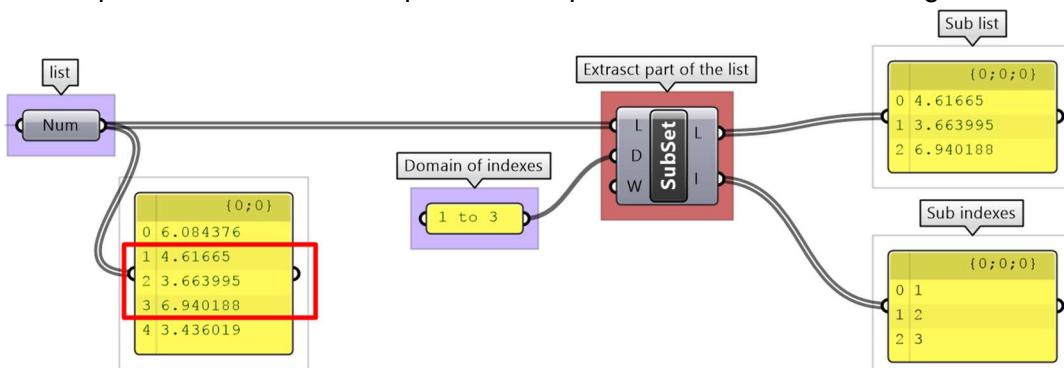
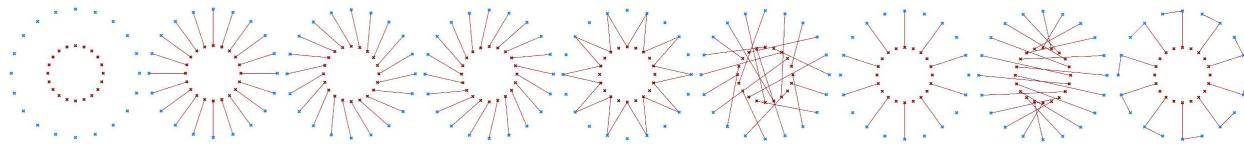


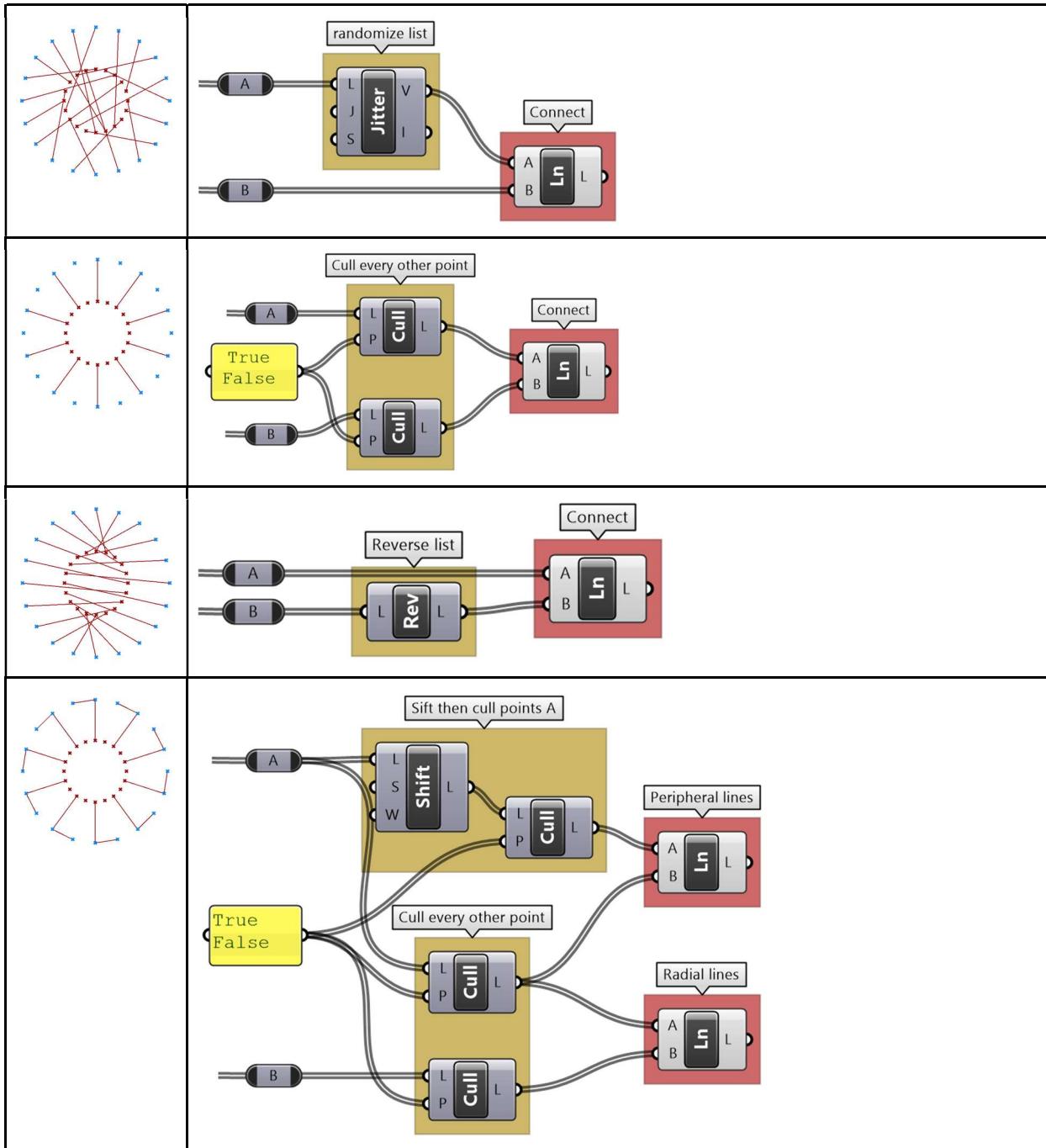
Figure (44): Example to select a subset of the list using a range of indices

## 2\_3\_1 List operations tutorial

Use the two given lists of points to generate the following images.



Output image	Grasshopper solution
	<p>Input lists A and B</p>
	<p>Connect input lists</p>
	<p>Shift forward</p> <p>Shift</p> <p>1</p> <p>Connect lists</p>
	<p>Shift backward</p> <p>Shift</p> <p>-1</p> <p>Connect lists</p>
	<p>Select every other line</p> <p>Sifted lines forward</p> <p>True False</p> <p>Shifted lines backwards</p> <p>Crv</p> <p>Output</p>



## 2\_4: List matching

When the input is a single item or has equal number of elements in a simple list, it is easy to imagine how the data is matched. The matching is based on corresponding indices. Let's use the **Addition** component to examine list matching in GH.

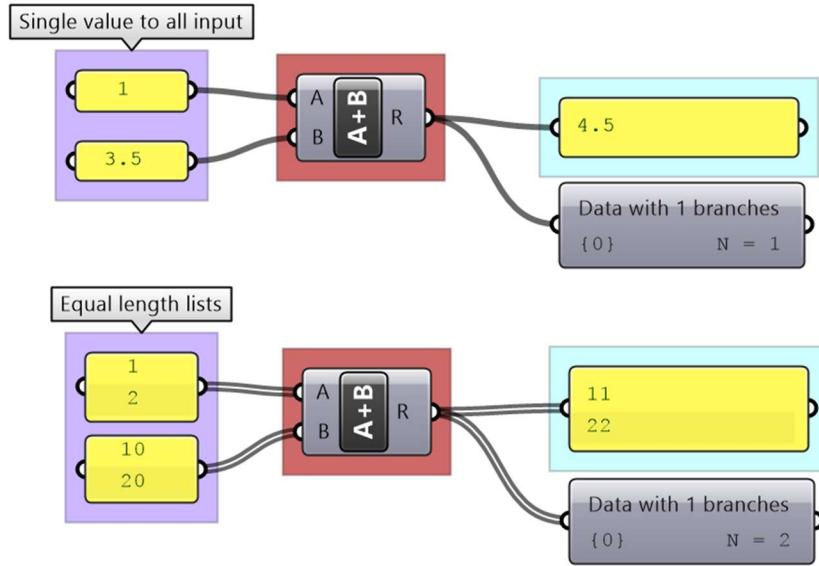


Figure (45): Matching equal length lists is based on matching corresponding indices

There are times when input has variable length lists. In this case, GH reuses the last item on the shorter list and matches it with the next items in the longer list.

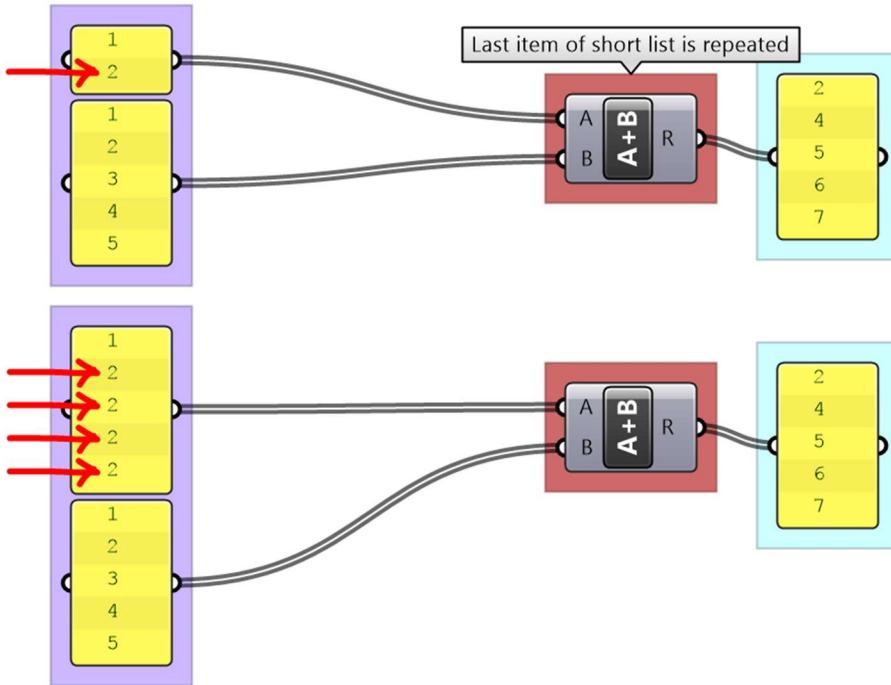


Figure (46): The default list matching in Grasshopper reuses the last element of the shorter list

Grasshopper offers alternative ways of data matching: **Long**, **Short** and **Cross** reference that the user can force to use. The **Long** matching is the same as the default matching. That is the last element of the shorter list is repeated to create a matching length.

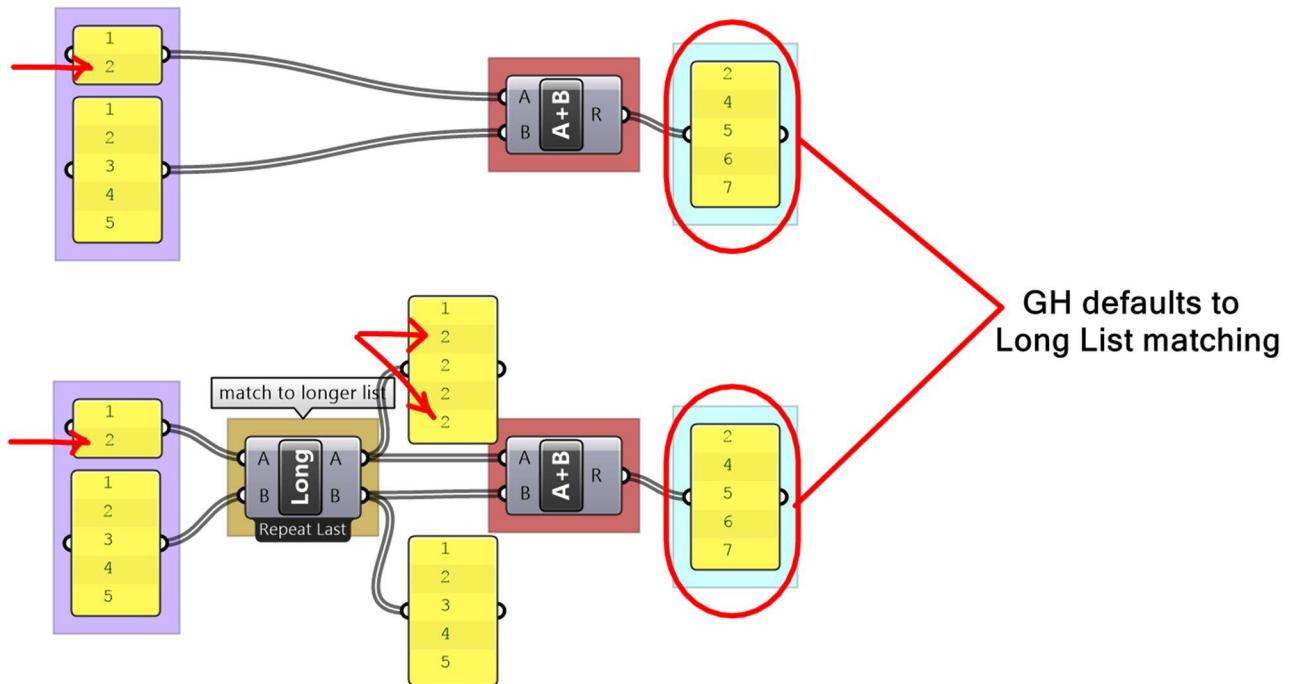


Figure (47): Long list matching is the default matching mode in Grasshopper

The **Short** list matching truncates the long list to match the length of the short list. All additional elements are ignored and the resulting list has a length that matches the shorter list.

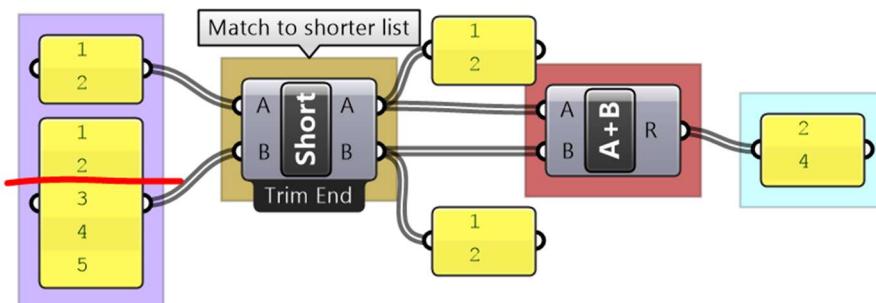


Figure (48): Short matching of lists omits additional values in longer lists

The **Cross Reference** matches the first list with each of the elements in the second list. The resulting list has a length equal to the multiplication product of the length of input lists. Cross reference is useful when trying to produce all possible combinations of input data. The order of input affects the order of the result as shown in Figure (49).

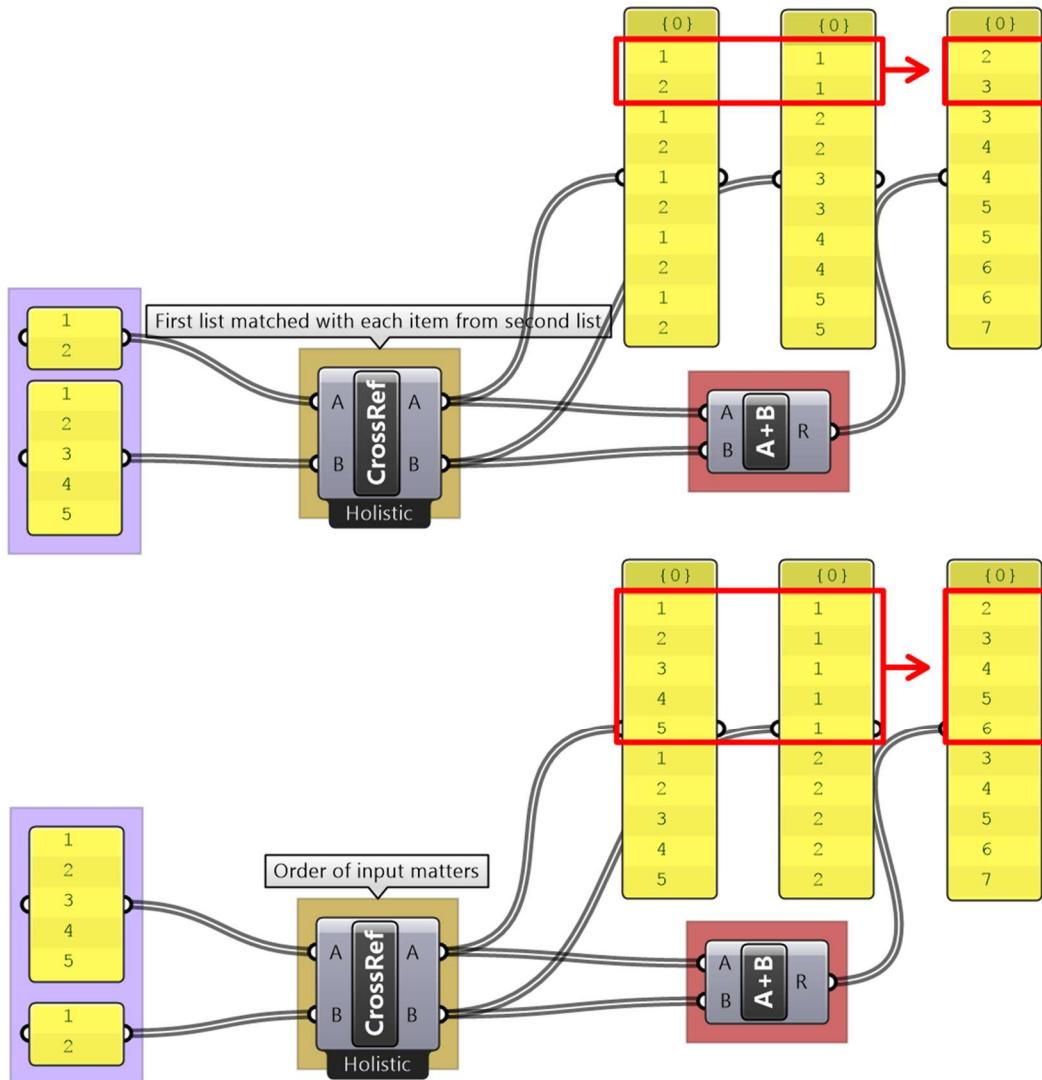


Figure (49): Cross reference matching creates longer lists to account for all possible permutations

If none of the matching methods produce the desired result, you can explicitly adjust the lists to match in length based on your requirements. For example, if you like to repeat the shorter list until it matches the length of the longer list, then you'll need to create the logic to achieve that as in the following example.

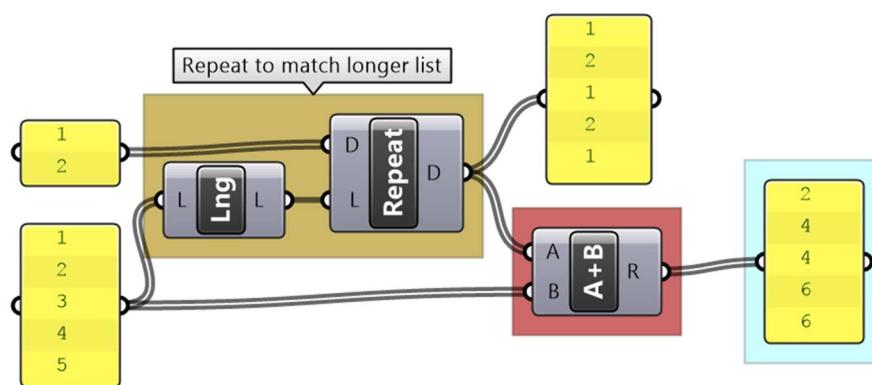
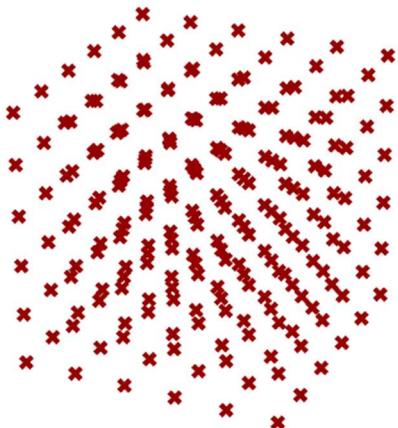


Figure (50): Need to create custom script to generate custom matching

## 2\_4\_1 List matching tutorial

Use the input list of 6 numbers to construct the points in the image

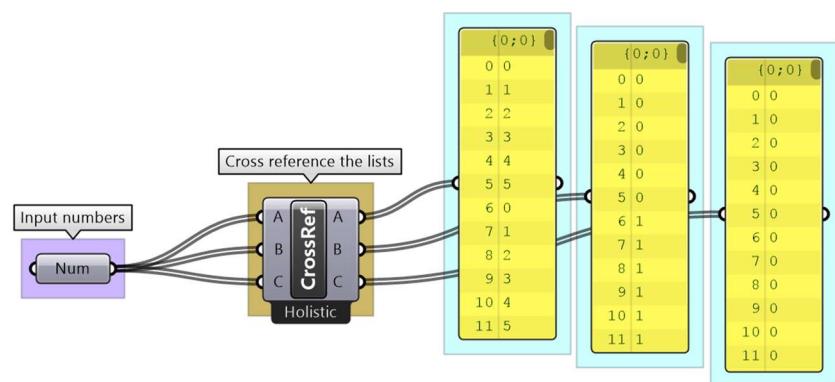


Solution	
<b>Output:</b> A list of $6 \times 6 \times 6 = 216$ points constructed from a list of X, Y, Z coordinates	<p>Cube points → Pt → Empty</p>
<b>Key process:</b> Use the <b>Construct Point</b> component to generate the list of points	<p>X list → Y list → Z list → Construct points → Pt → Empty</p>
<b>Input:</b> Examine input using the <b>Parameter Viewer</b> and <b>Panel</b> components.  The given list has 6 points representing each coordinate along each axis	<p>Input numbers → Num → Data with 1 branches            {0;0} N = 6            {0;0}            0 0            1 1            2 2            3 3            4 4            5 5</p>

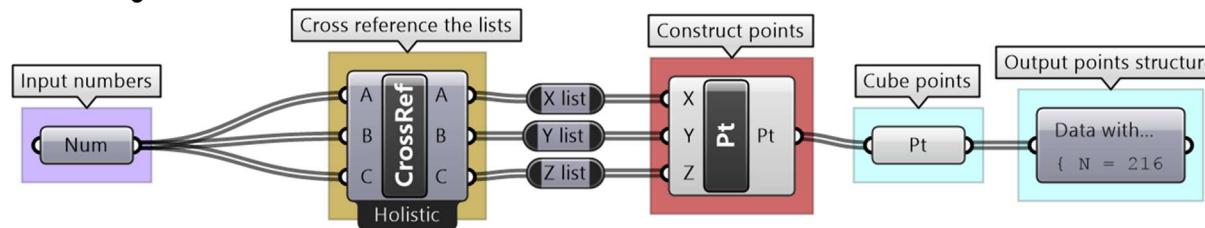
### Intermediate process:

Need to find all possible permutations for the coordinates to create the cube of 216 points along all 3 axes

Use **Cross Reference** matching to generate lists of coordinates that have all possible permutations



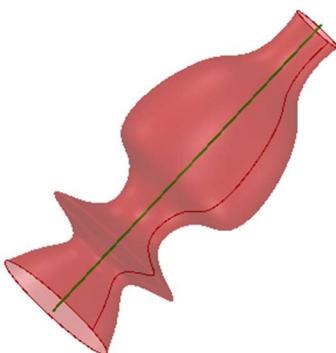
### Put it all together



## 2\_5: Data structures tutorials

### 2\_5\_1: Variable thickness pipe tutorial

Create a surface similar to the one in the image with thickness that changes in 10 locations random along the curve. Thickness variations are random between 1 and 3.



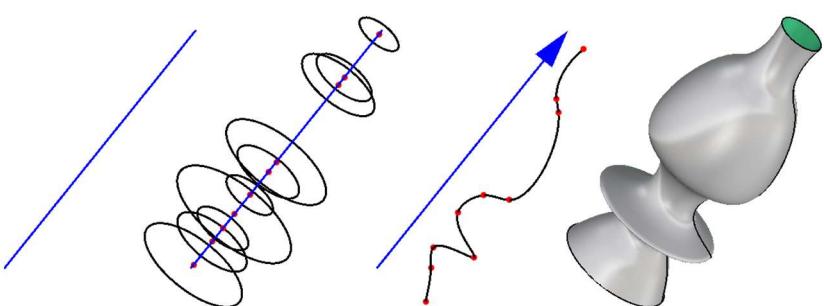
### Algorithm analysis

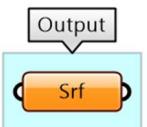
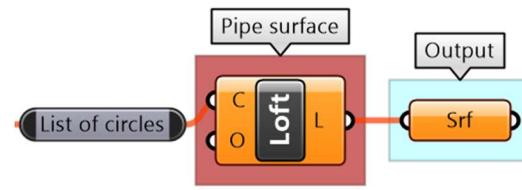
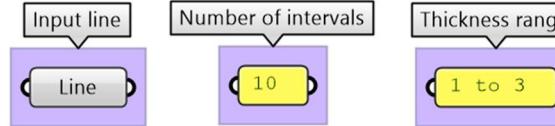
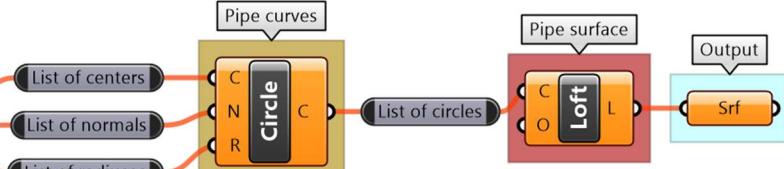
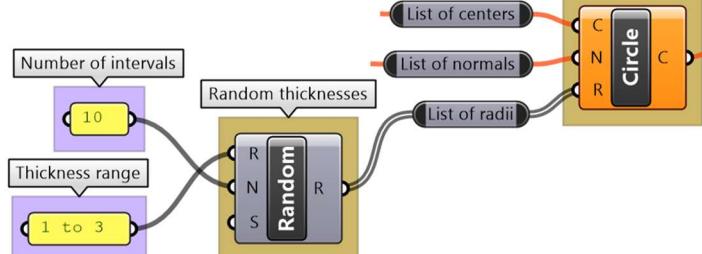
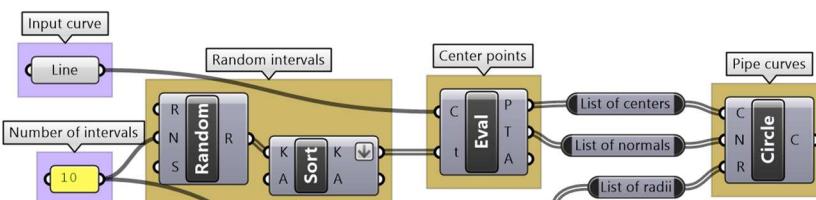
To figure out an algorithm, it is useful to think in terms of 3D modeling. There are 2 ways to generate this surface:

- 1- Create **circles** along the line at random locations with random radius, then **loft** the result.
- 2- Figure out the **profile curve** and **revolve** along the line

The first process goes like this:

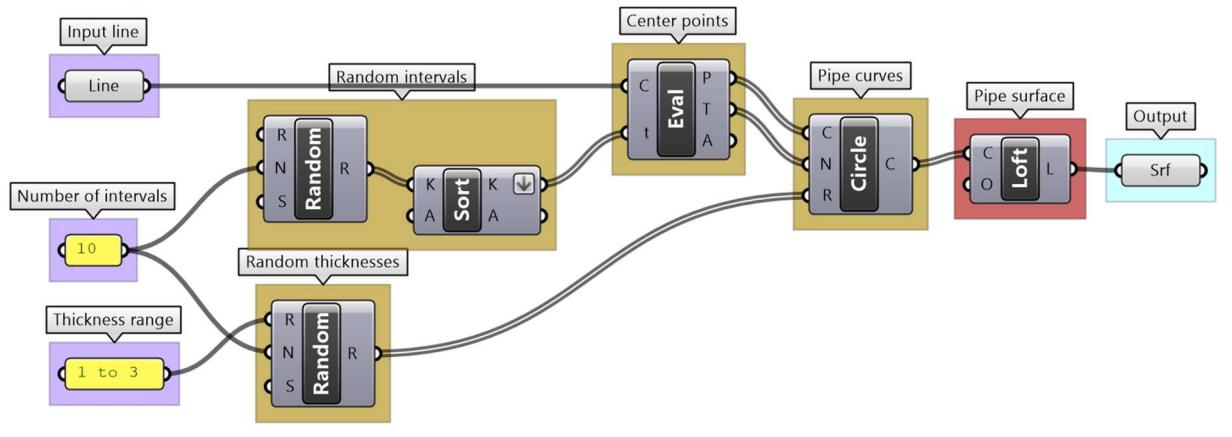
- 1- **Divide** the line at random locations
- 2- Orient to the **planes** at locations (line normal to planes)



<p>3- Create the <b>circles</b> (or points) for the profile curve          4- Select the circles (in order) to <b>Loft</b> (or <b>Interpolate Curve</b> then <b>Revolve</b>)</p>	
<b>Solution steps</b>	
<b>Output:</b> The surface	
<b>Key process:</b> Use the <b>Loft</b> component to generate the surface	
<b>Input:</b> Line, Number of intervals and Thickness range	
<b>Intermediate process #1:</b> The <b>Loft</b> is created from a list of circles. Use the <b>Circle</b> component that takes centers, normals and radii lists. We can use the default <b>Loft</b> options.	
<b>Intermediate process #2:</b> List of radii is created randomly. Use the <b>Random</b> component and the input thickness range.	
<b>Intermediate process #3:</b> Evaluate the line at random intervals. Use the <b>Evaluate Curve</b> component to extract points and normals, and use the <b>Random</b> component to generate the parameters along the curve.  Problem: the random parameters are not ordered and hence produce unordered points. Use the <b>Sort List</b> component to order the parameters before feeding	

into the **Evaluate Curve**.

### Put it all together



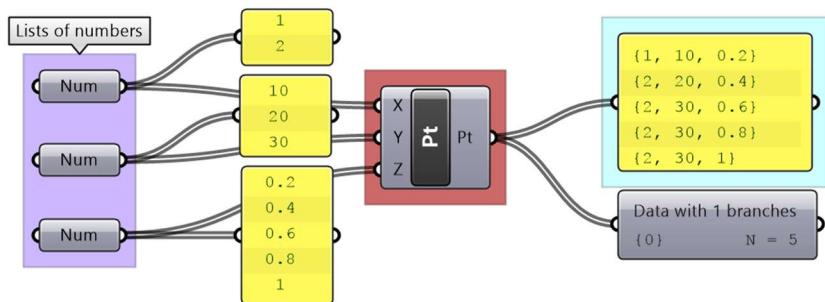
### 2\_5\_2: Custom matching tutorial

Explain the default GH list matching in the following example. Compare the result with "Shortest List" matching, then try to create a custom matching that repeats the pattern of the shorter lists. E.g. [1,2] becomes [1,2,1,2,...] until it matches the length of the longer list.

### Solution

#### Construct default GH matching:

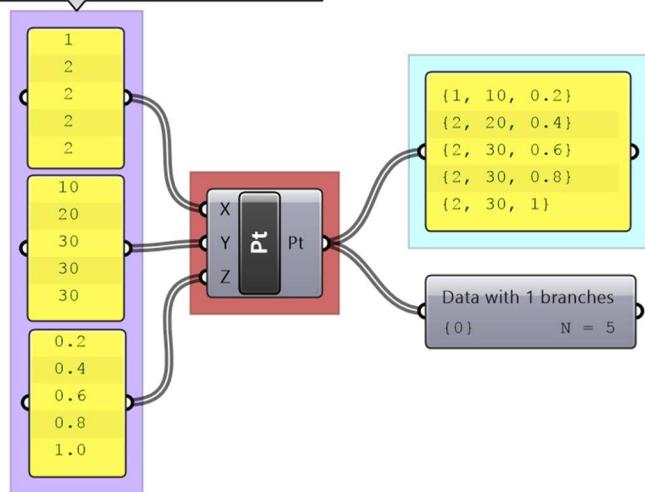
To test the matching, fill the lists as coordinates to a **Construct Point** component and observe the result.



#### Analysis of GH default matching:

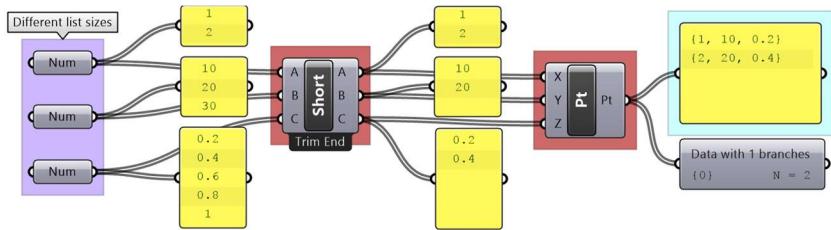
The last element of shorter lists is repeated until all lists have the same length, then elements are matched by indices

Last element in short lists is repeated to match longest list



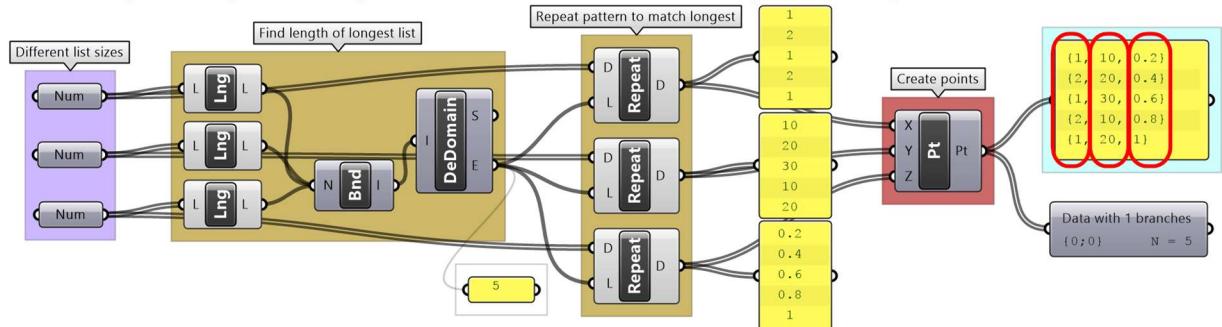
### Shortest List matching:

Omit additional values in longer lists so that the length of all lists equals the length of the shortest list.



### Custom matching:

Use the **Repeat** component to repeat the elements until match the length of the longest list.

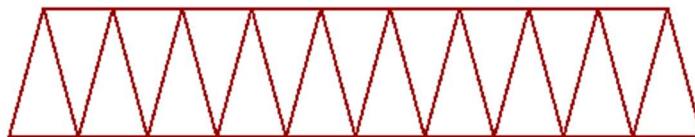


### 2\_5\_3: Simple truss tutorial

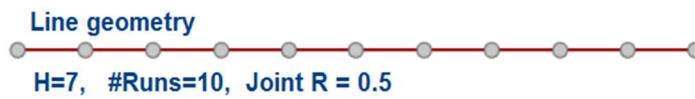
Create simple truss as in the image. Use given baseline, height, number of runs and joint radius.

#### Algorithm analysis

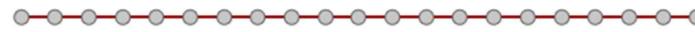
Identify the desired output for the truss



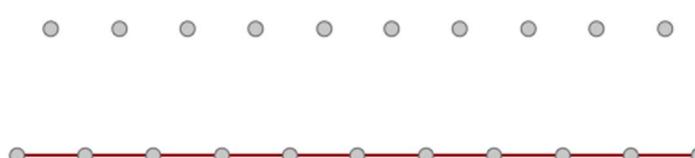
Define the input  
L= line geometry on xy-plane  
H= height  
R= number of runs  
J= joint radius



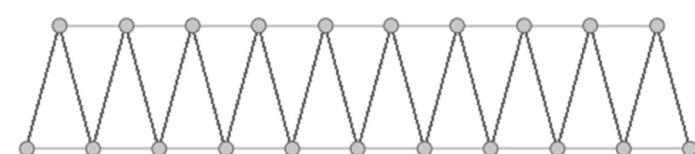
Divide curve by  $2 \times R$



Move every other point in the Z direction by height



Create 3 sets of ordered points for the bottom beams, top beams and middle beams, then connect each of the 3 sets with a polyline



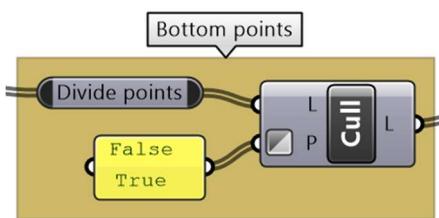
#### Algorithm implementation in Grasshopper

<p><b>Output:</b> There are 2 outputs, the beams as curves (polylines) and joints as spheres (surfaces)</p>	<p>Beams Joints</p> <p>Crv Srf</p>
<p><b>Key processes:</b> Need to create the polylines for the top, middle, and bottom beams. Use the <b>Polyline</b> component with relevant set of points for each.  Use the <b>Sphere</b> component to create joints. Use middle points and joint radius as input.</p>	<p>Top Middle Bottom Join spheres List of planes List of radii</p> <p>Beams Crv Joints Srf</p>
<p><b>Given input:</b> Four given input: line, number of runs, height and joint radius</p>	<p>Input line Number of runs Height Joint radius</p> <p>Line 10 7 0.5</p>
<p><b>Intermediate process #1</b> Divide the curve with twice the number of runs. Use <b>Divide Curve</b> component and <b>Multiply</b> the number of runs</p>	<p>Input line Divide line to double number of runs Line Number of runs 10</p>
<p><b>Intermediate process #2</b> To create top points, select every other point from the list of all divide points, then move vertically by the height amount. Use <b>Cull Pattern</b> component to select points and <b>Move</b> component to shift vertically</p>	<p>Top points Divide points False True Height Cull Move G T G X</p>

### Intermediate process #3

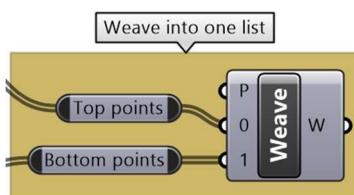
To create bottom points, select every other point, in the invert pattern used to select top points.

Use **Cull Pattern** component to select points (set *invert* flag for the pattern input)

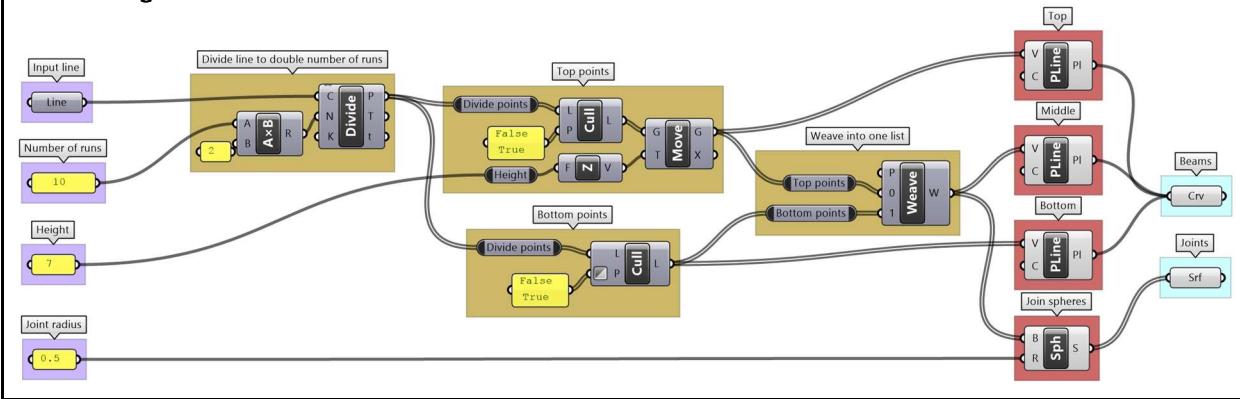


### Intermediate process #4

To create middle points, **Weave** the top and bottom points.

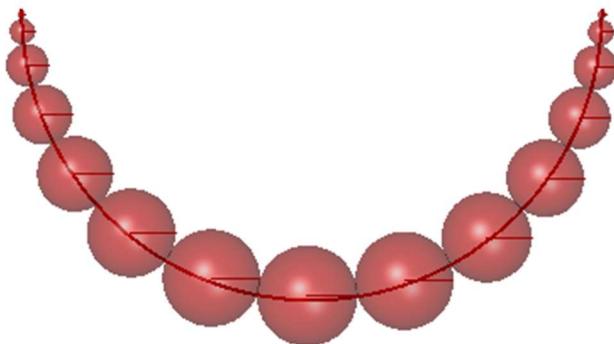


### Put it all together



### 2\_5\_4: Pearl necklace tutorial

Create a necklace with one big pearl in the middle, and gradually smaller size pearls towards the ends as in the image. Make the number of pearls parametric between 15-25.



### Algorithm analysis

<p>The workflow to create the necklace follows these general lines:</p> <ol style="list-style-type: none"> <li>1- Divide the curve into segments of variable distances (widest in the middle and narrow towards the ends).</li> <li>2- Find midpoints for each segment and its length</li> <li>3- Create spheres at centers using half the length as radius</li> </ol>																																			
<b>Solution steps</b>																																			
<p><b>Output:</b> The surfaces</p>																																			
<p><b>Key process:</b> Use the <b>Sphere</b> component to generate the surfaces</p>																																			
<p><b>Given input:</b> Necklace curve, Number of pearls as a parameter (can be changed by the user)</p>																																			
<p><b>Intermediate process #1:</b> The <b>Range</b> component creates equal distances. We need to change to variable distances and for that we can use the <b>Graph Mapper</b> component to control the spacing.</p>	<table border="1"> <thead> <tr> <th></th> <th>Distances</th> </tr> </thead> <tbody> <tr><td>0</td><td>{0;0}</td></tr> <tr><td>1</td><td>0.01</td></tr> <tr><td>2</td><td>0.04</td></tr> <tr><td>3</td><td>0.08</td></tr> <tr><td>4</td><td>0.15</td></tr> <tr><td>5</td><td>0.23</td></tr> <tr><td>6</td><td>0.33</td></tr> <tr><td>7</td><td>0.44</td></tr> <tr><td>8</td><td>0.55</td></tr> <tr><td>9</td><td>0.66</td></tr> <tr><td>10</td><td>0.76</td></tr> <tr><td>11</td><td>0.84</td></tr> <tr><td>12</td><td>0.91</td></tr> <tr><td>13</td><td>0.96</td></tr> <tr><td>14</td><td>0.99</td></tr> <tr><td>15</td><td>1</td></tr> </tbody> </table>		Distances	0	{0;0}	1	0.01	2	0.04	3	0.08	4	0.15	5	0.23	6	0.33	7	0.44	8	0.55	9	0.66	10	0.76	11	0.84	12	0.91	13	0.96	14	0.99	15	1
	Distances																																		
0	{0;0}																																		
1	0.01																																		
2	0.04																																		
3	0.08																																		
4	0.15																																		
5	0.23																																		
6	0.33																																		
7	0.44																																		
8	0.55																																		
9	0.66																																		
10	0.76																																		
11	0.84																																		
12	0.91																																		
13	0.96																																		
14	0.99																																		
15	1																																		

<p><b>Intermediate process #2:</b> Since we have normalized distances from the start of the curve (parameters are between 0 to 1), we can use the <b>Evaluate Length</b> component to find the divide points.</p>	<p>Divide curve with variable distances</p> <p>Input curve Normalized lengths C L N P T t</p>
<p><b>Intermediate process #3:</b> Generate the segments. Use <b>Polyline</b> and <b>Explode</b> components to turn the points into segments</p> <p>Center points are calculated at the middle of the segments. Use <b>Evaluate Length</b> at mid length</p> <p>Radius is calculated as half of each segment length. Use <b>Length</b> and <b>Division</b> components.</p>	<p>Extract segments as lines Find mid point of each line Divide points PLine Explode Evaluate Length 0.5 Diameter lines Len A/B R Find radius (len/2)</p>
<p><b>Put it all together</b></p> <p>Input curve Number of pearls Steps ◊ 15 Range Create a list of normalized distances Divide curve with gradually increasing then decreasing distances Extract subdivided lines Find mid point on each line (normalize length) Evaluate Length 0.5 PLine Explode Len A/B R Create pearls B R Sph S</p>	

# Chapter Three: Advanced Data Structures

This chapter is devoted to the advanced data structure in GH, namely the data trees, and different ways to generate and manage them. The aim is to start to appreciate when and how to use tree structures, and best practices to effectively use and manipulate them.

## 3\_1: The Grasshopper data structure

### 3\_1\_1 Introduction

In programming, there are many data structures to govern how data is stored and accessed. The most common data structures are variables, arrays, and nested arrays. There are other data structures that are optimised for specific purposes such as data sorting or mining. In Grasshopper, there is only one structure to store data, and that is the **data tree**. Hold on, what about what we have learned so far: **single item** and **list** of items? Well, in GH, those are nothing but simple trees. A single item is a tree with one branch, that has one element, and a list is a tree with one branch that has a number of elements. It is actually pretty elegant to be able to fit all data in one unifying data structure, but at the same time, this requires the user to be aware and vigilant about how their data structure changes between operations, and how that can affect intended results. This chapter attempts to demystify the data tree of Grasshopper.

### 3\_1\_2 Processing data trees

We used the **Panel** and **Parameter Viewer** components to view the data structure. We will use both extensively to show how data is stored. Let's start with a single item input. The **Parameter Viewer** has two display modes, one with text and one that is graphical. You can see that the single item input is stored in one branch that has only one item.

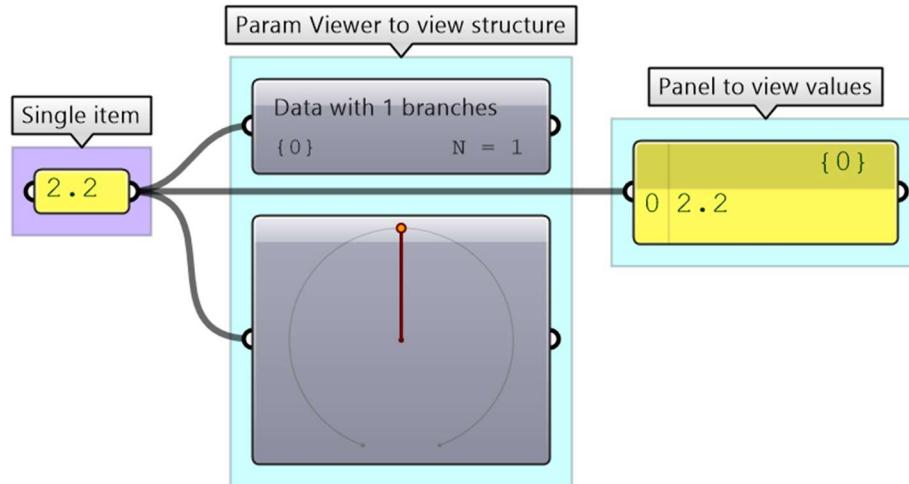


Figure (51): Different ways to preview the data structure in Grasshopper

The **Parameter Viewer** shows each branch address (called “Path”), and the number of elements in that branch as shown in Figure (52).

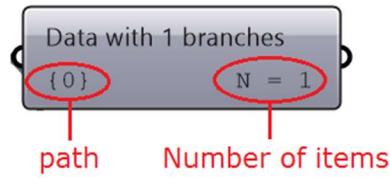


Figure (52): The **Parameter Viewer** indicates the path address and the number of elements in each branch

A list of items is typically stored in a tree with one branch. Figure (53). However, the three items can also be stored in three different branches. Figure (54).

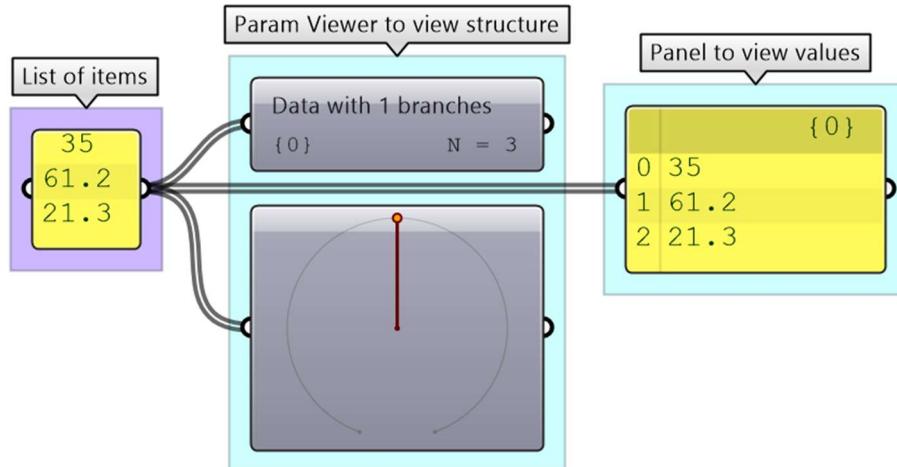


Figure (53): A list is a tree that has one branch with multiple elements

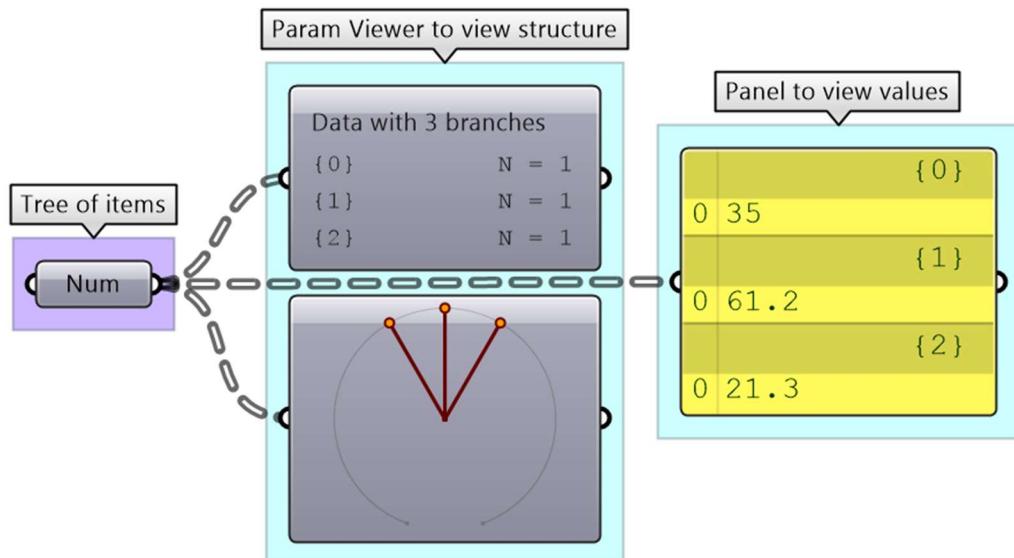


Figure (54): A tree contains any number of branches with any number of elements in each branch

The key to understanding the Grasshopper data structure is to be able to answer the following question: **What is the significance of storing the 3 numbers in one branch vs 3 branches?**

The data structure informs GH components about how to match input values. In other words, components may process data differently based on their structure. The following example illustrates how different data structures for the same set of values can affect the result.

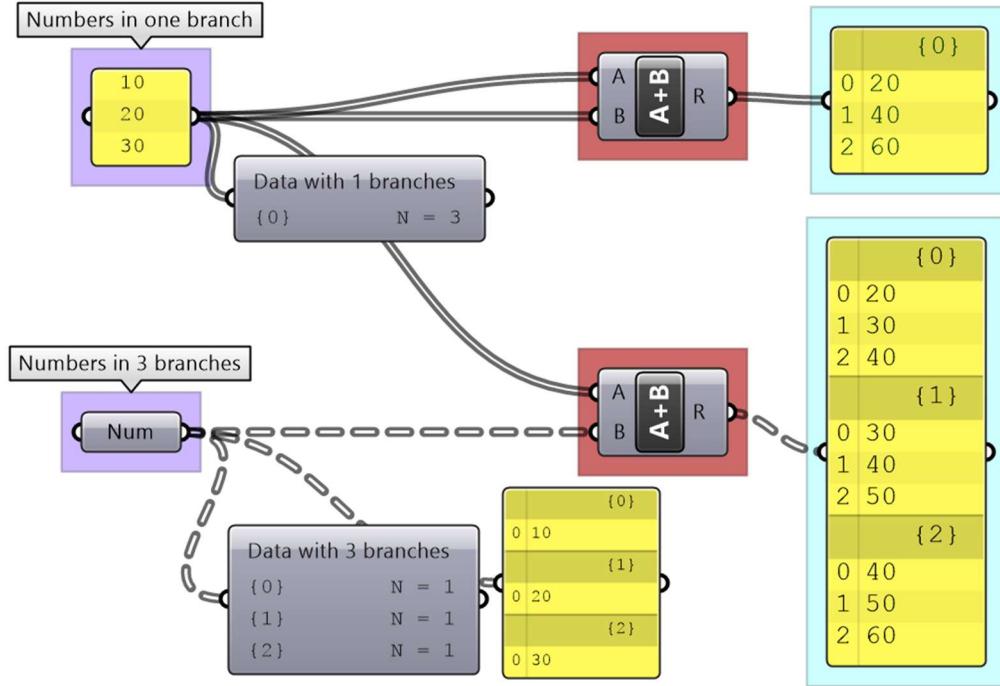


Figure (55): Organizing same set of values in different data structures affect the output

We will elaborate on data tree matching later, but you can already see that GH components do pay attention to the data structure and the result can vary considerably based on it. This is one of the complications inherited in using one unifying data structure in a programming language.

### 3\_1\_3 Data tree notation

The first step to understanding data trees is to learn the GH notation of trees. As we have seen from the examples, trees consist of branches, and each branch holds a number of elements. The address or path of each branch is represented with integers separated by semicolons and enclosed in curly brackets. The index of each element is enclosed by square brackets. This diagram shows a breakdown of the address of elements in trees.

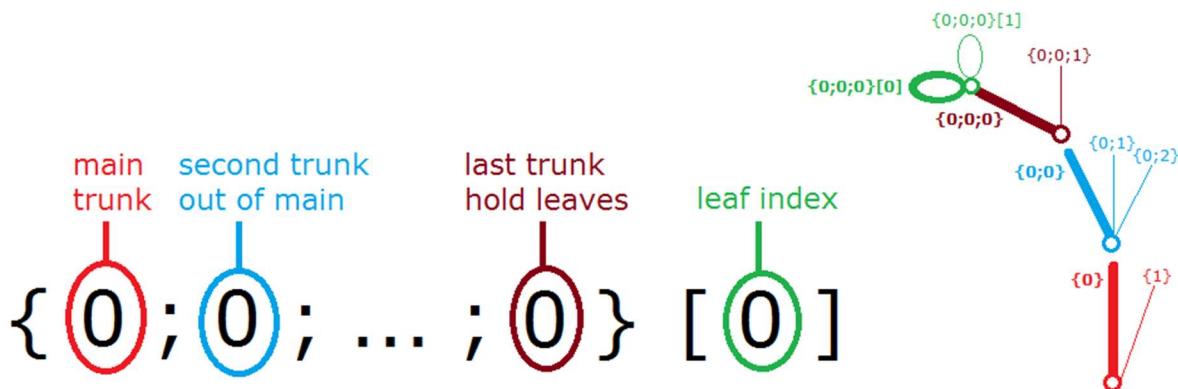


Figure (56): Address of elements include the address of the branch and the index of the element in the branch

Here are a few examples of various trees structures and how they show in the **Paramster Viewer** and the **Panel**.

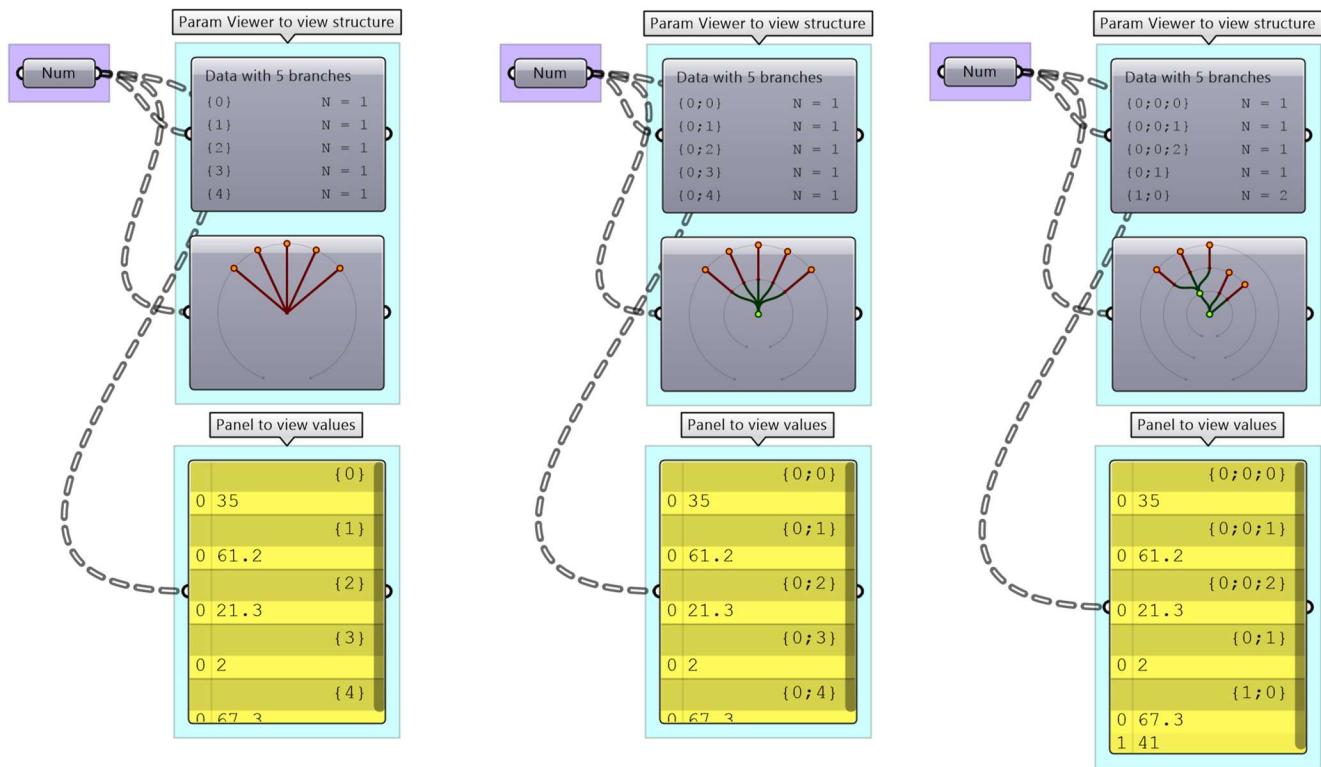
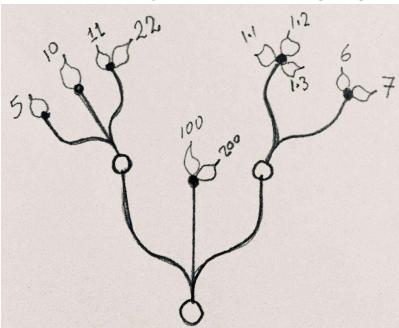


Figure (57): Same set of values held in different structures.

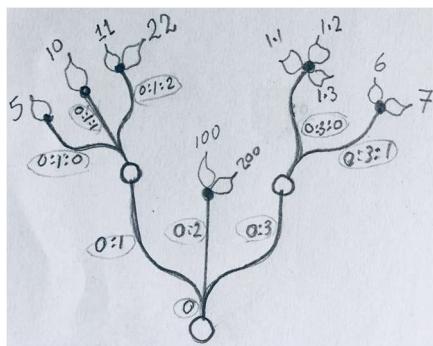
Left: 5 trunks (5 trees) with one item in each. Middle: 5 branches out of one trunk (1 tree), and each branch holds a single item. Right: two trunks (2 trees), the first has 2 branches with the first branching into 3 branches, each holds one item, the second holds 1 item. The second trunk holds 2 items.

### **3\_1\_1 Data tree tutorial:**

Construct a tree of numbers using the **Number** parameter, that look similar to the image when viewed in the **Param Viewer** and contains the numbers indicated. Then type in a **Panel** the full address to the item "1.2" . Note that order of branches and leaves is always from left to right going clockwise

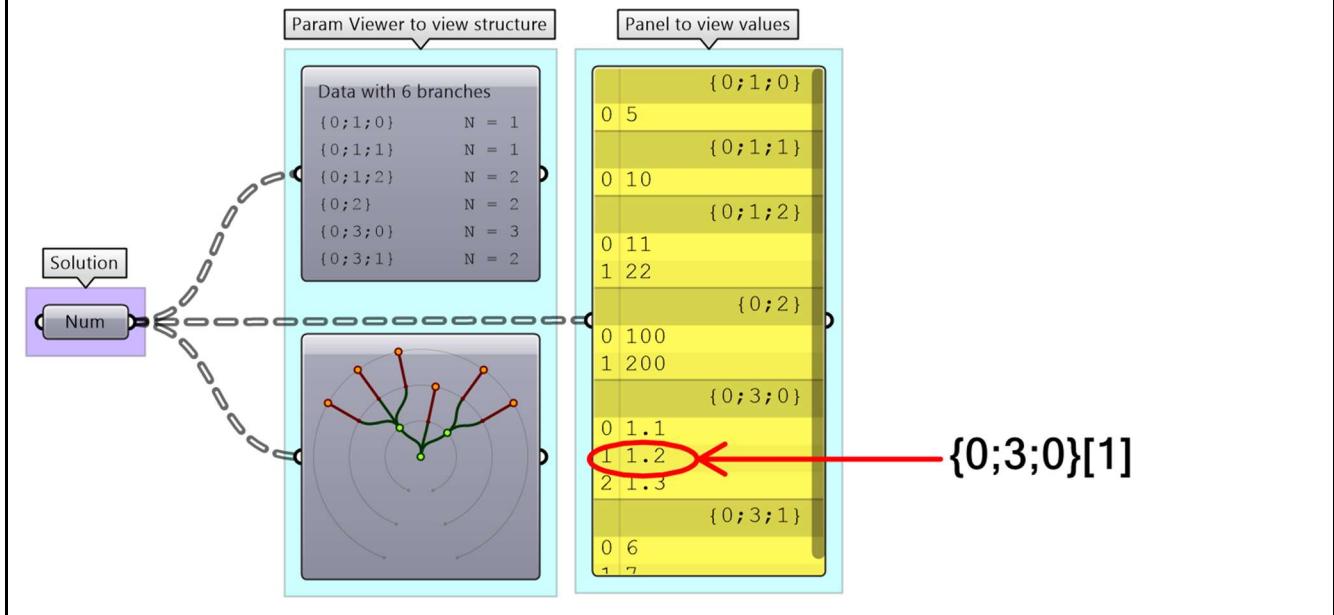


## Solution



The path for "1.2" is: { 0 ; 3 ; 0 } [ 1 ]

Note: The three branches from the main trunk are set here to 0:1, 0:2, and 0:3. They also could have been 0:0, 0:1 and 0:2. Both are correct.



### 3\_2: Generating trees

There are many ways to generate complex data trees. Some explicit, but mostly as a result of some processes, and this is why you need to always be aware of the data structures of output before using it as input downstream. It is possible to enter the data and set the data structure directly inside any Grasshopper parameter. Once set, it is relatively hard to change and therefore is best suited for a constant input. The following is an example of how to set data tree directly inside a parameter.

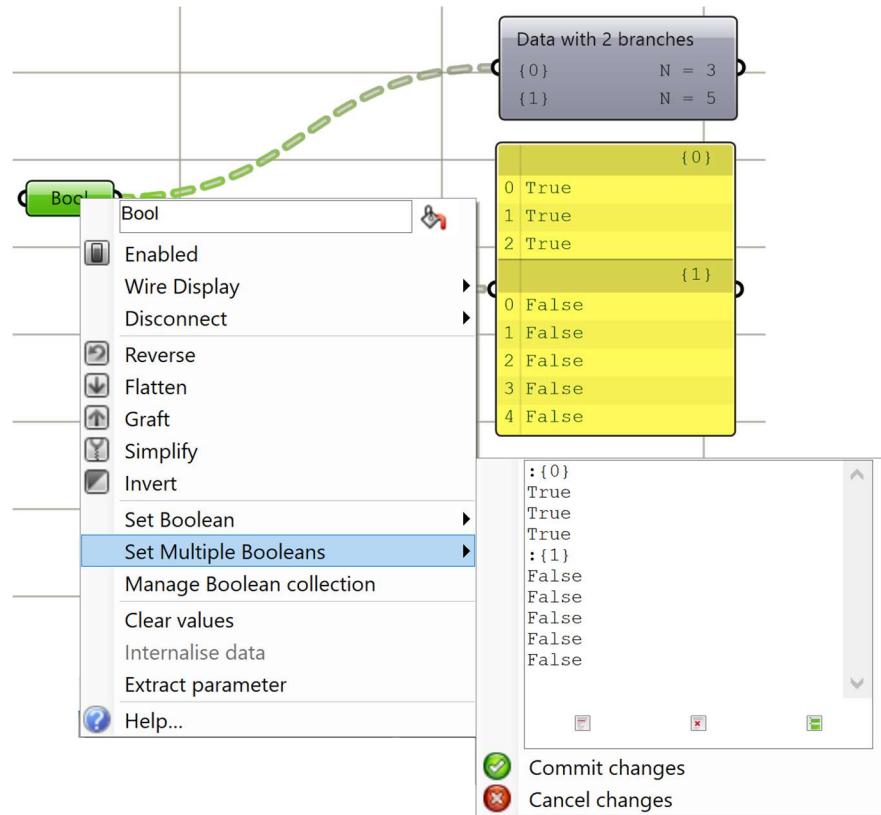
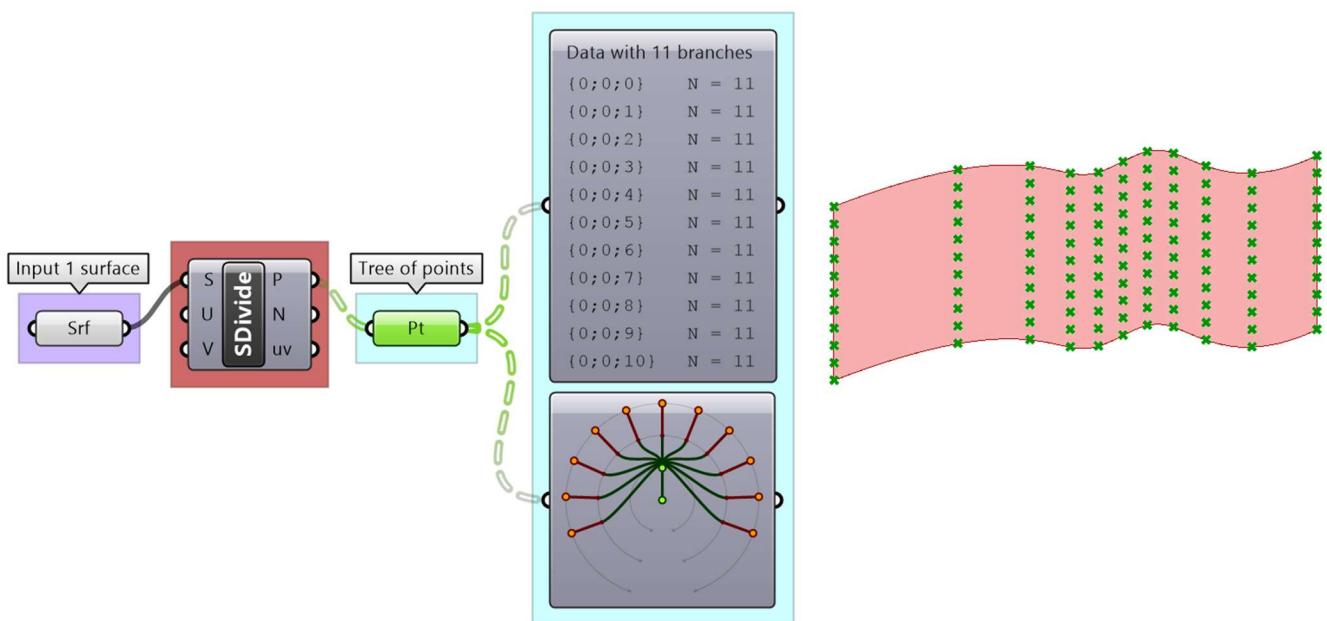


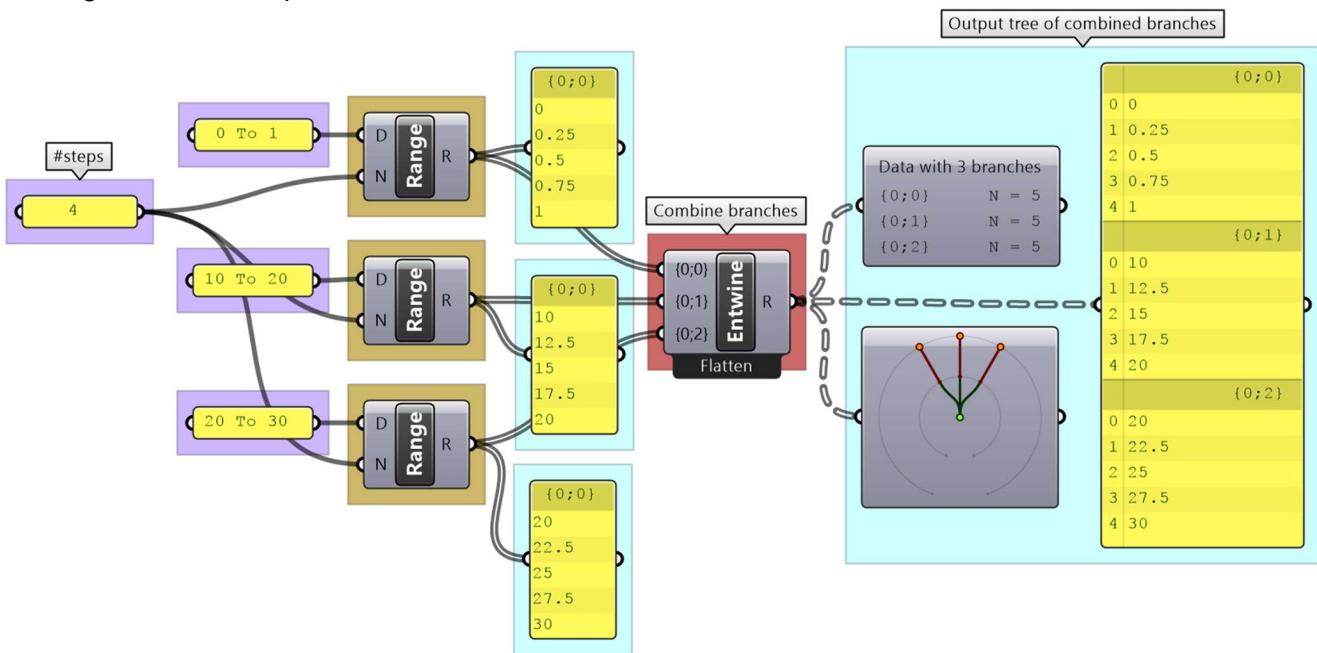
Figure (58): Set data trees directly inside the parameter

There are many components that generate data trees such as **Grid** and **DivideSrf**, and others that combine lists into a tree structure such as **Entwine**. Also all the components that produce lists can also create tree if the input is a list. For example, if input more than one curve into the **DivideCrv** component, we get a tree of points.



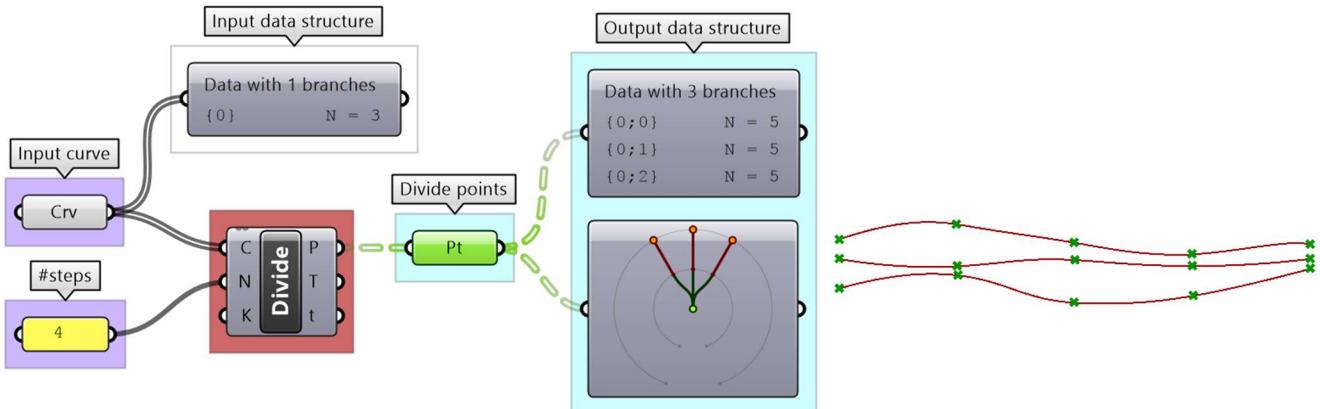
Figure(59): **SDivide** component takes one input (surface) and outputs a data tree (grid).

All components that generate lists of numbers (such as **Range** and **Series**) can also generate trees when given a list of input.



Figure(60): **Entwine** component takes any number of lists and combine them into a tree structure.

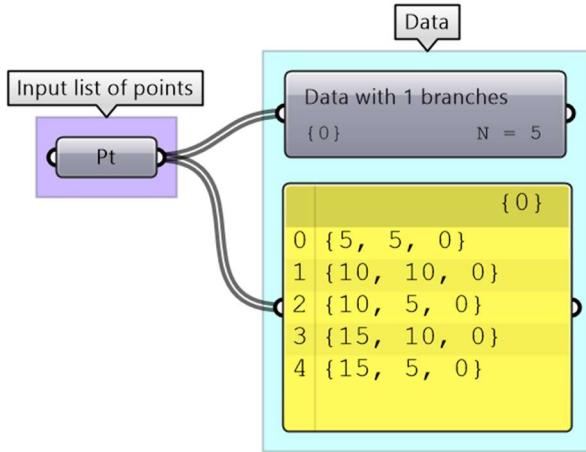
Perhaps one of the most common cases to generate a tree is when dividing a list of curves to generate a grid of points. So the input is one list and the output is a tree.



Figure(61): **Divide** component takes any list (curves) and generates a tree structure (grid).

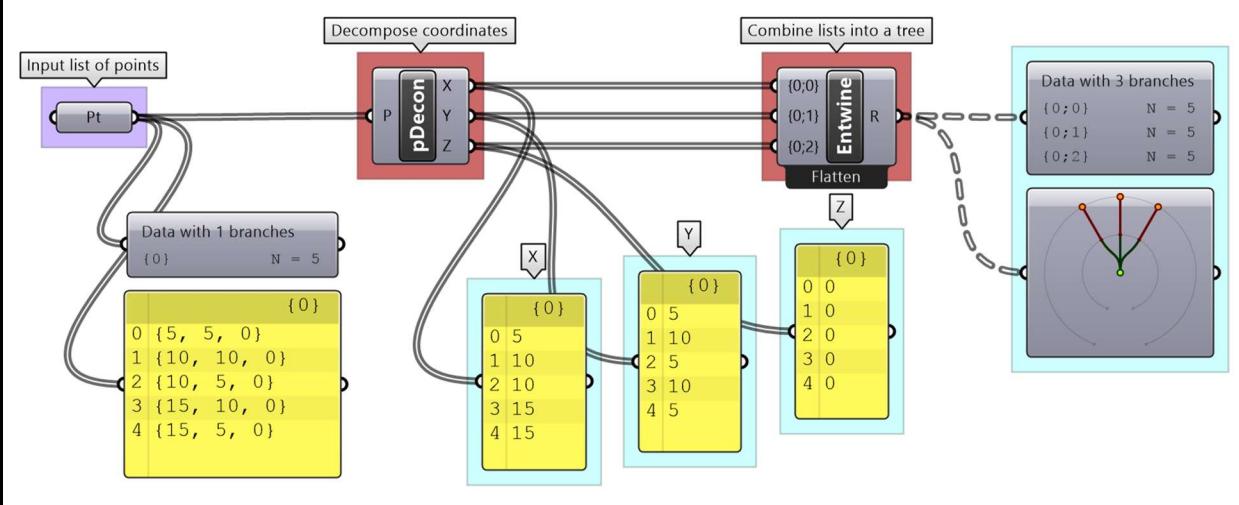
### 3-2-1 Generating trees tutorial

Given the following list of points, construct a number tree with 3 branches, one for each coordinate.



### Solution

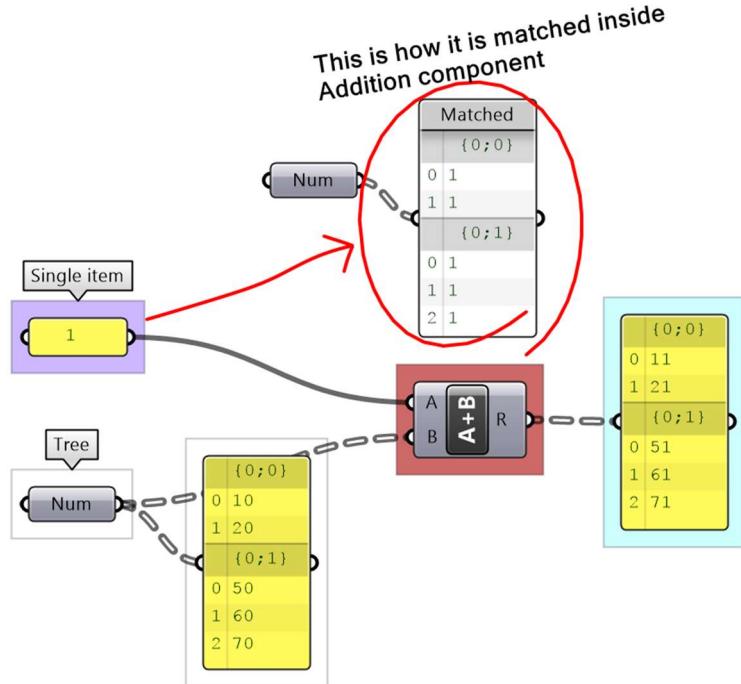
Discussion: each input point is a single data item that contains 3 numbers (coordinates). We know we would like to isolate each coordinate into a separate list, then combine them into one data structure. Hence we need to first deconstruct input points (use *Deconstruct* of *pDecon* component), then combine the lists into one structure (use *Entwine* component).



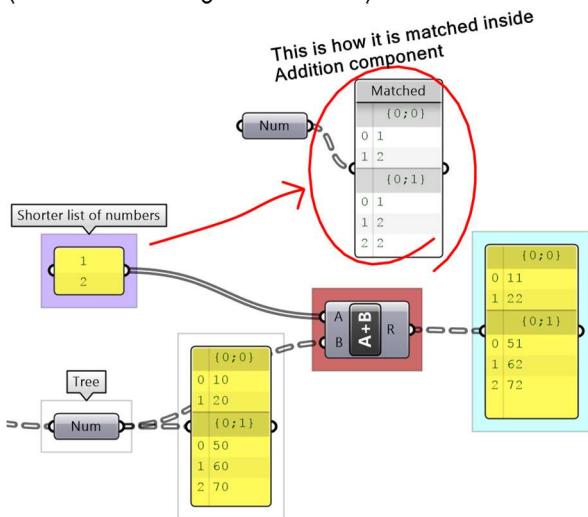
### 3\_3: Tree matching

We explained the **Long**, **Short** and **Cross** matching with lists. Trees follow similar conventions to expand the shorter branches by repeating the last element to match. If one tree has less branches, the last branch is repeated. The following illustrates common tree matching cases.

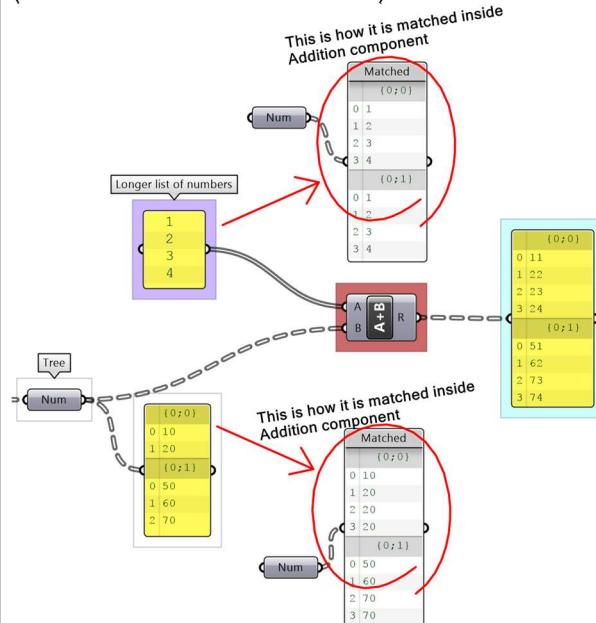
Match an item with a tree:



Match a shorter list with a tree  
(tree branches longer than the list):

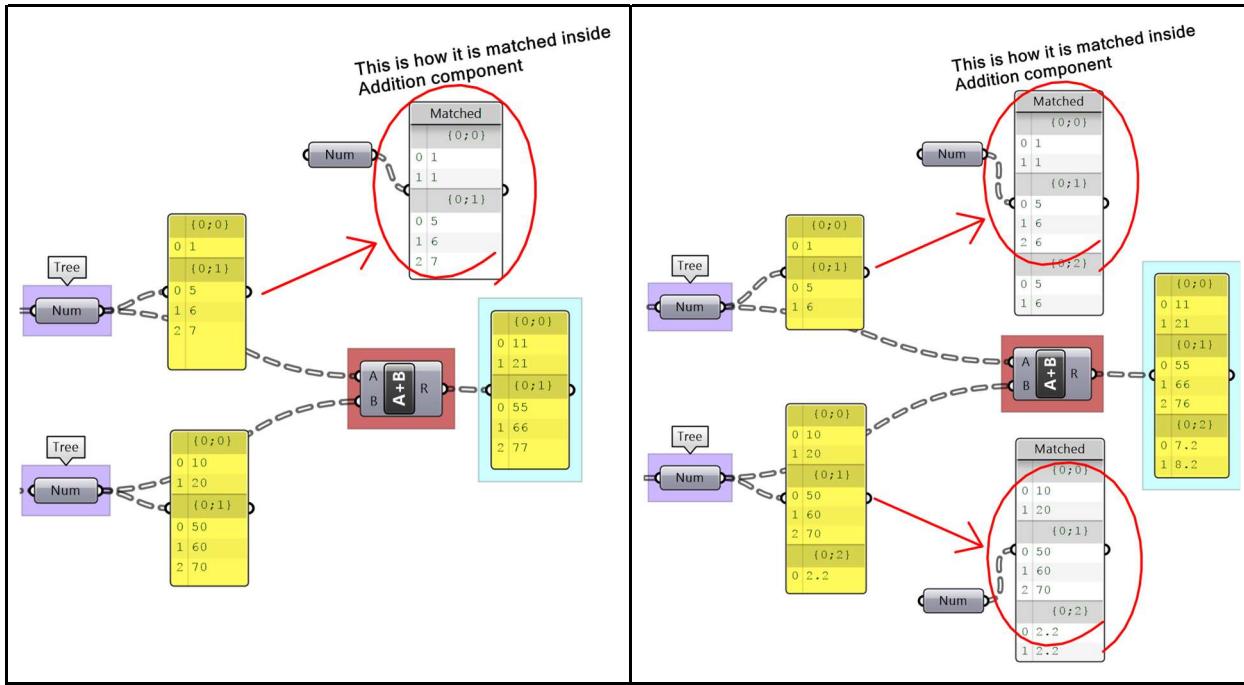


Match a longer list with a tree  
(tree branches shorter than the list):



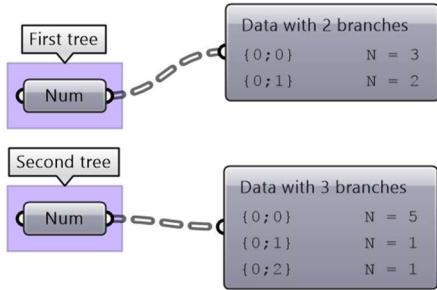
Match 2 trees with same number of branches:

Match 2 trees with different number of branches:



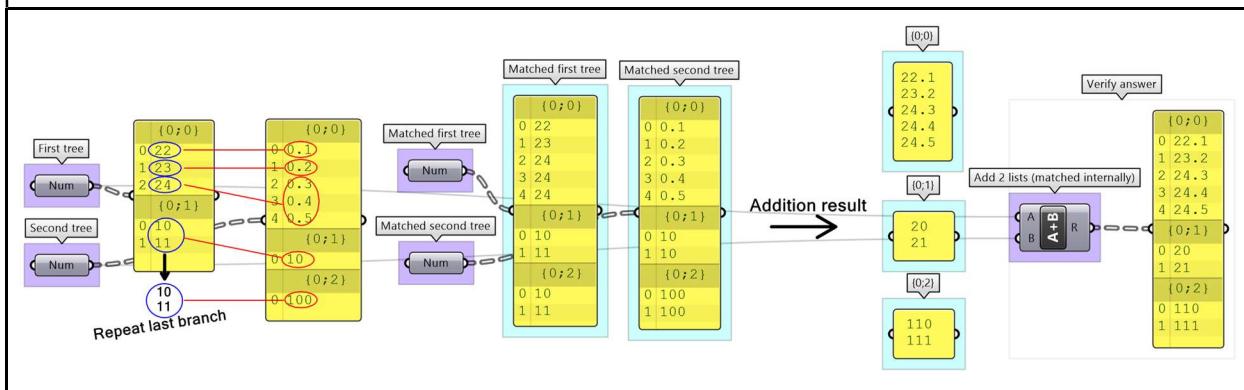
### 3\_3\_1 Tree matching tutorials

Inspect the following 2 number structures, then predict the structure and result of adding them (with default Grasshopper matching). Verify your answer using **Addition** components.



#### Solution

Key solution idea: The two input trees have different number of branches and different number of elements in each branch. The last branch of the shorter tree is repeated to match the number of branches, then corresponding branches are matched by repeating the last element of the shorter branch.



## 3\_4: Traversing trees

Grasshopper provides components to help extract branches and items from trees. If you have the path to a branch or to an item, then you can use **Branch** and **Item** components. You need to check the structure of your input so you can supply the correct path.

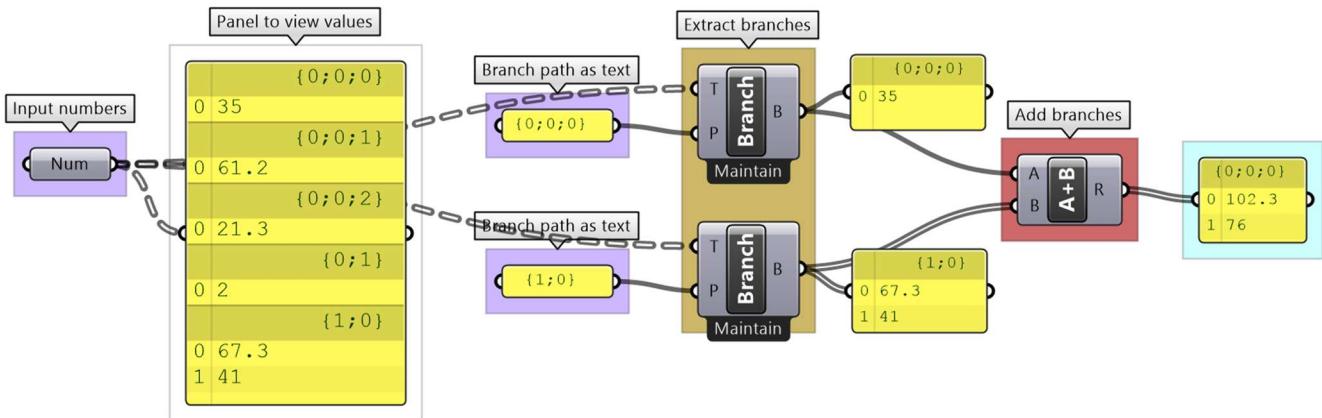


Figure (62): Select branches from a tree

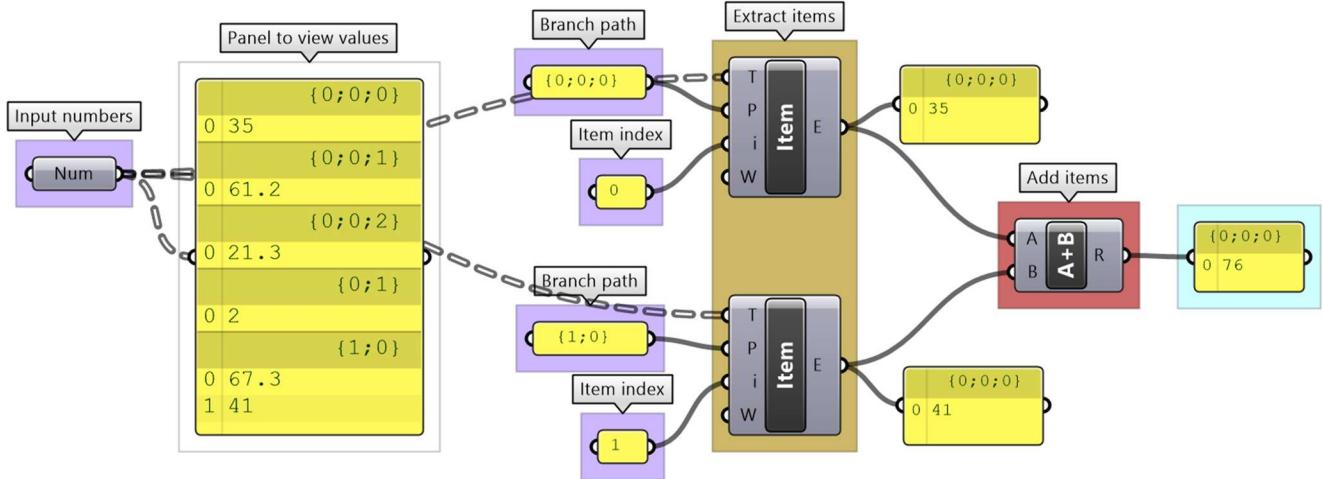


Figure (63): Select items from a tree

If you know that your structure might change, or you simply do not want to type the path, you can extract the path using the **Param Viewer** and **List Item** components.

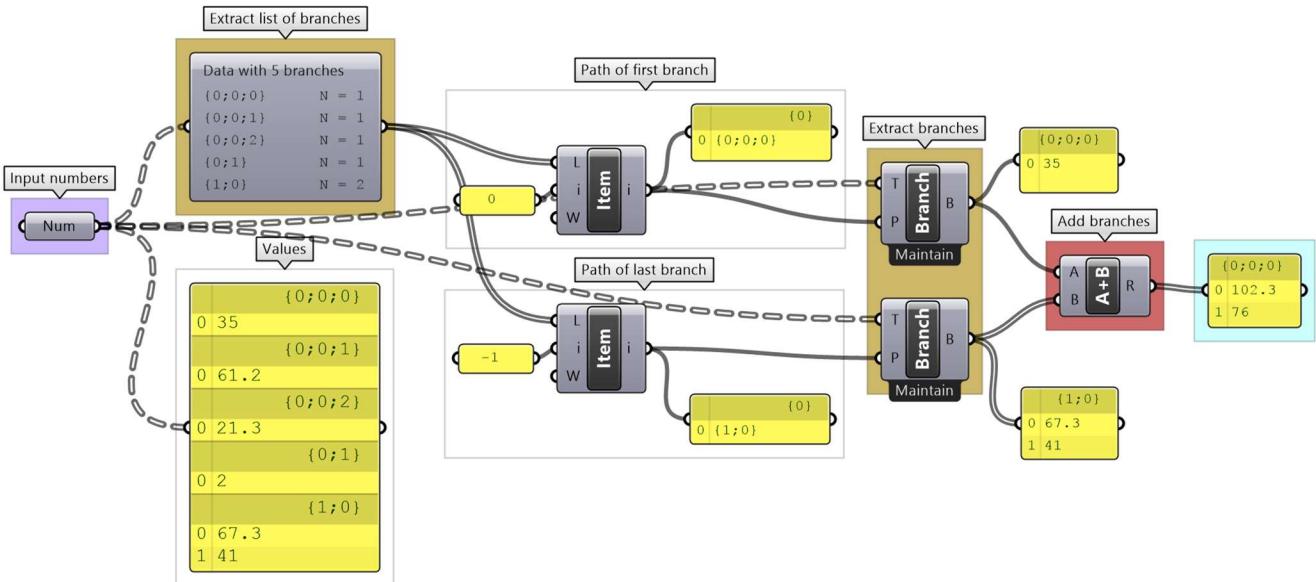
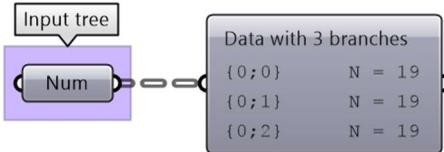


Figure (64): Example of how to extract data paths dynamically

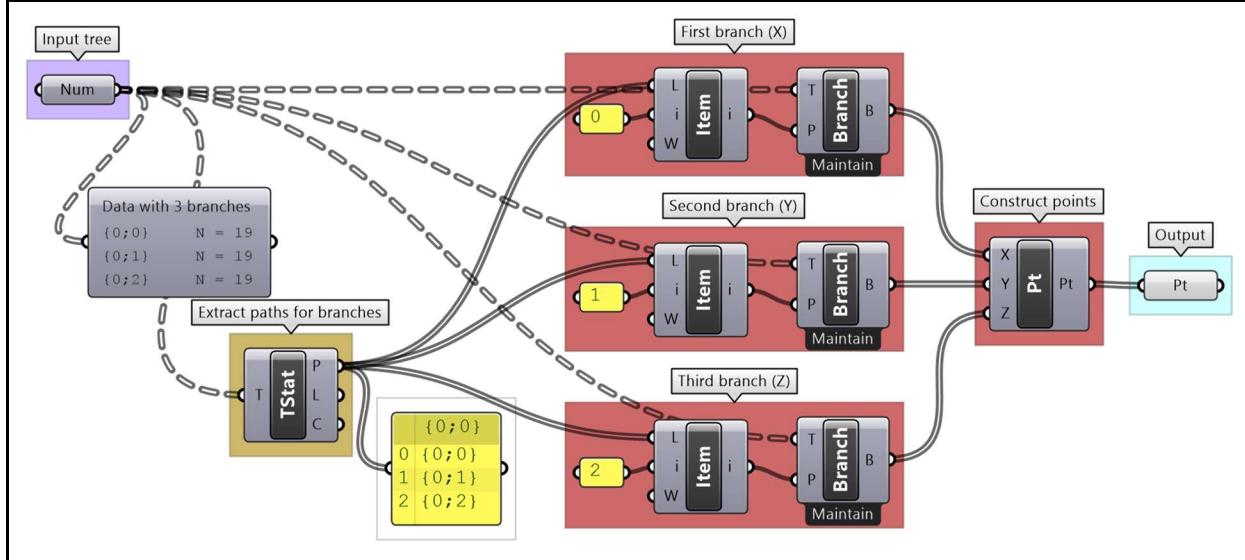
### 3\_4\_1 Traversing trees tutorial

The following tree has 3 branches for each one of the coordinates (x, y, z) of some list of points. Use that tree to construct a list of these points.



#### Solution

Key solution idea: We can construct a point list using as input 3 lists representing X, Y and Z values. If we can isolate the 3 branches of the input tree, then we will be able to feed them to the point construction component.



## 3\_5: Basic tree operations

Basic tree operations are widely used and you will likely need them in most solutions. It is very important to understand what these operations do, and how they affect the output.

### 3\_5\_1: Viewing the tree structure

As we have seen in the data matching, different data structures of the same set of elements produce different results. Grasshopper offers three ways to view the data structure, the **Parameter Viewer** in text or diagram format, and the **Panel**.

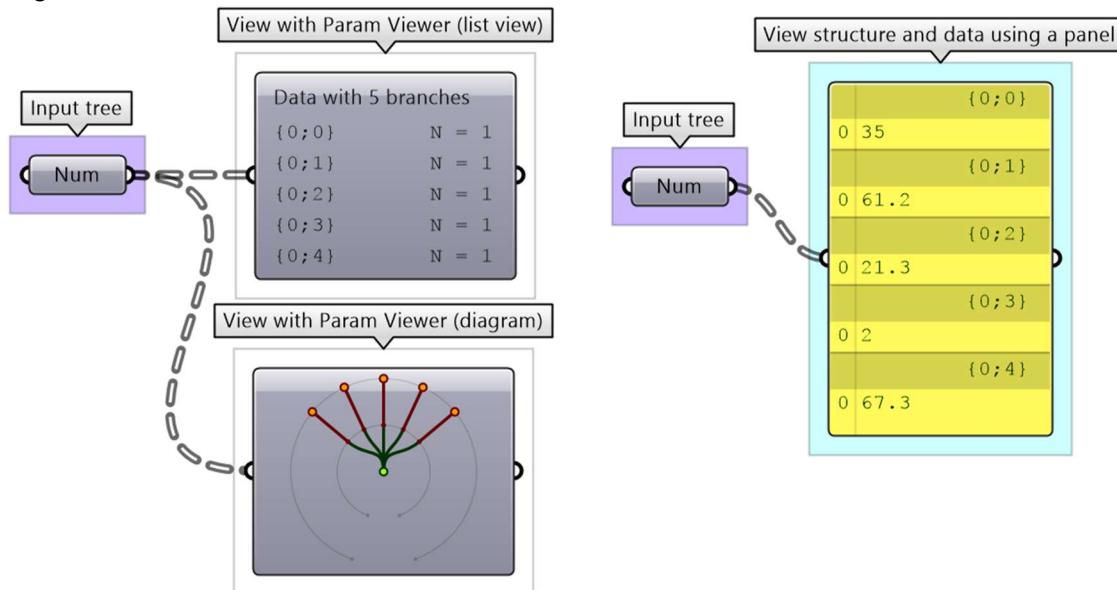


Figure (65): View trees using the **Parameter Viewer** and the **Panel** components

Tree information can be extracted using the TStats component. You can extract the list of paths to all branches, number of elements in each branch and the number of branches.

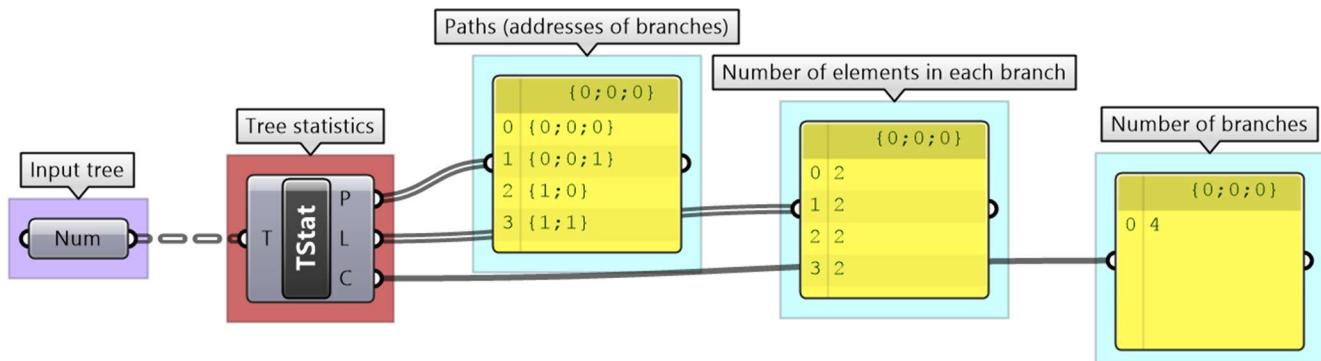


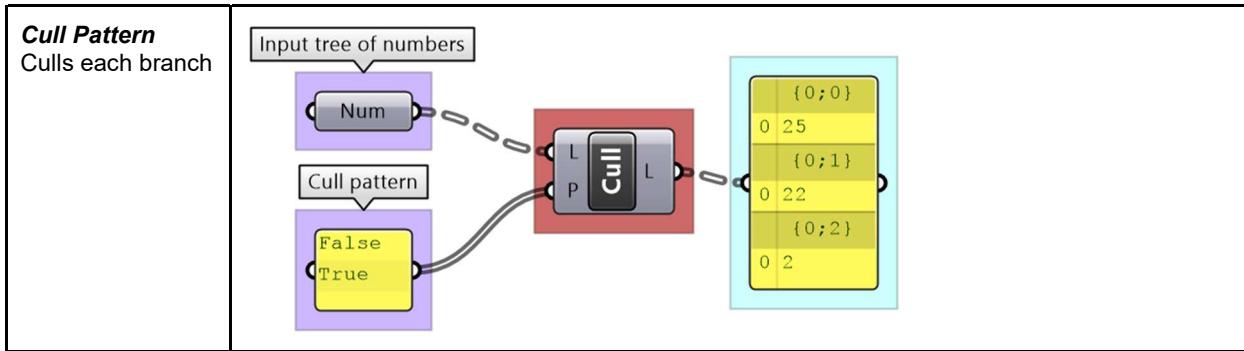
Figure (66): Extract trees structure using **TStats** component

### 3\_5\_2: List operations on trees

Trees can be thought of as a list of branches. When using list operations on trees, each branch is treated as a separate list and the operation is applied to each branch independently. It is tricky to predict the resulting data structure and therefore it is always important to check your input and output structures before and after applying any operation.

To illustrate how list operations work in trees, we will use a simple tree, a grid of points, and apply different list operations on it. We will then examine the output and its data structure.

Operations	Example of how the list operation apply to trees
<p><b>List Item</b> Select items at specific index in each branch</p>	<p>Tree structure and elements</p> <p>Data with 3 branches (0;0) N = 3 (0;1) N = 3 (0;2) N = 3</p> <p>Select elements at each branch at index 0</p> <p>Input tree of numbers</p> <p>Num</p> <p>Item</p> <p>A+B</p> <p>0 15 0 11 0 1</p> <p>0 25 0 22 0 2</p> <p>0 40 0 33 0 3</p>
<p><b>List Item</b> Select multiple indices to isolate part of the tree and perform one operation on such as <b>Mass Addition</b></p>	<p>Input tree of numbers</p> <p>Num</p> <p>Item</p> <p>MA</p> <p>0 15 1 25 0 11 1 22 0 1 1 2</p> <p>0 40 0 33 0 3</p>
<p><b>Split List</b> Split the elements of branches into 2 trees at the specified index</p>	<p>Input tree of numbers</p> <p>Num</p> <p>Split</p> <p>0 15 0 11 0 1 1 22 1 2</p> <p>0 25 0 22 0 2 1 35 1 33 1 3 2 15 2 11 2 1</p>
<p><b>Shift List</b> Shifts the elements of each branch</p>	<p>Input tree of numbers</p> <p>Num</p> <p>Shift</p> <p>0 25 1 35 2 15</p> <p>0 22 1 33 2 11</p> <p>0 2 1 3 2 1</p>



### 3\_5\_3: Grafting from lists to a trees

In some cases you need to turn a list into a tree where each element is placed in its own branch. Grafting can handle complex trees with branches of variable depths.

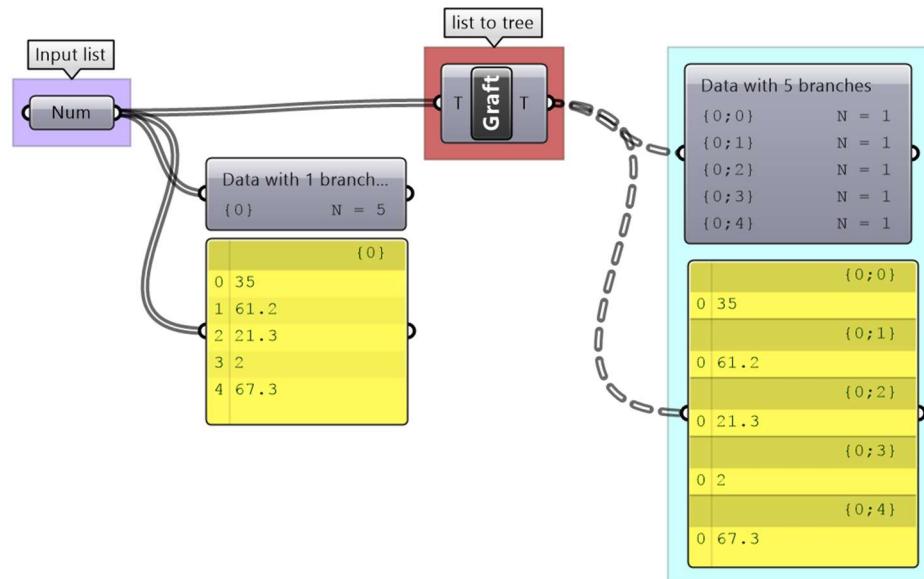


Figure (67): Grafting a tree create a new branch for each element

It might feel unintuitive to complicate the data structure (from a simple list to a tree), but grafting is very useful when trying to achieve certain matching. For example if you need to add each element of one list with all the elements in the second list, then you will need to graft the first list before inputting to the addition process.

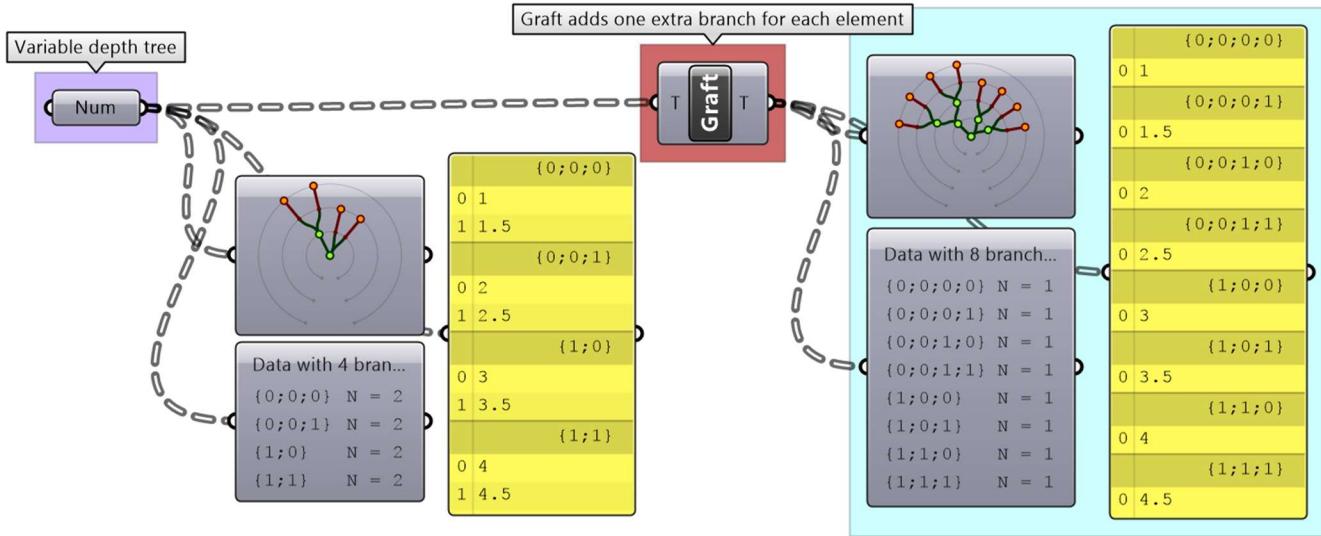


Figure (68): Grafting complex trees

### 3\_5\_4: Flattening from trees to lists

Other times you might need to turn your tree structure into a simple list. This is achieved with the flattening process. Data from each branch is extracted and sequentially attached to one list.

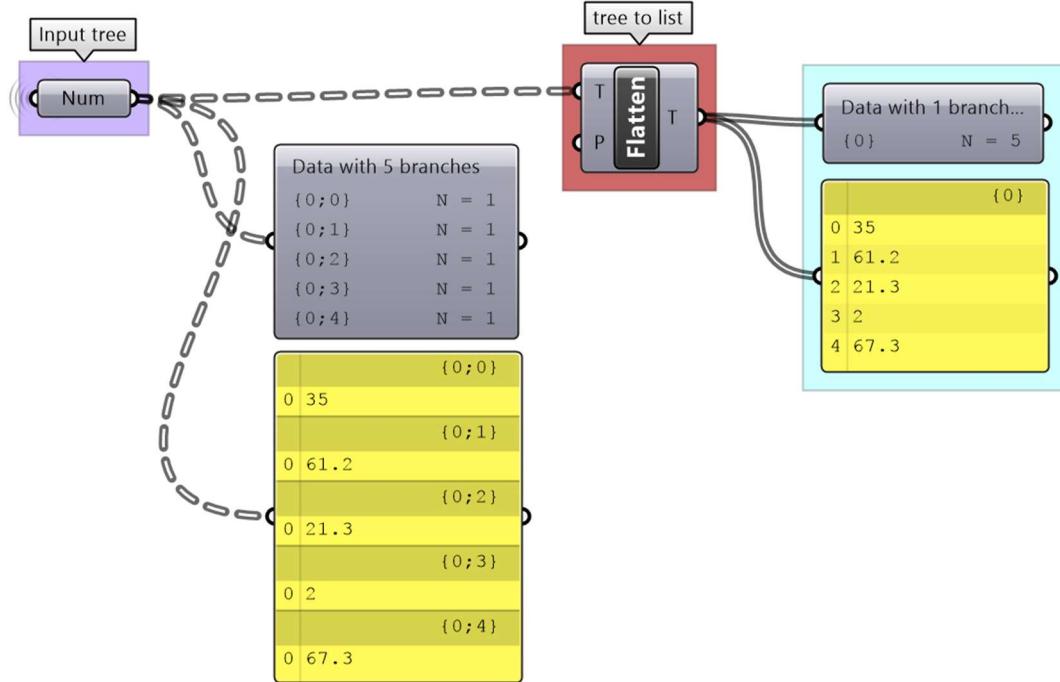


Figure (69): Flattening place all tree elements in one list

Flatten also can handle any complex tree. It takes the branches in order starting with the lowest index trunk and put all elements in one list.

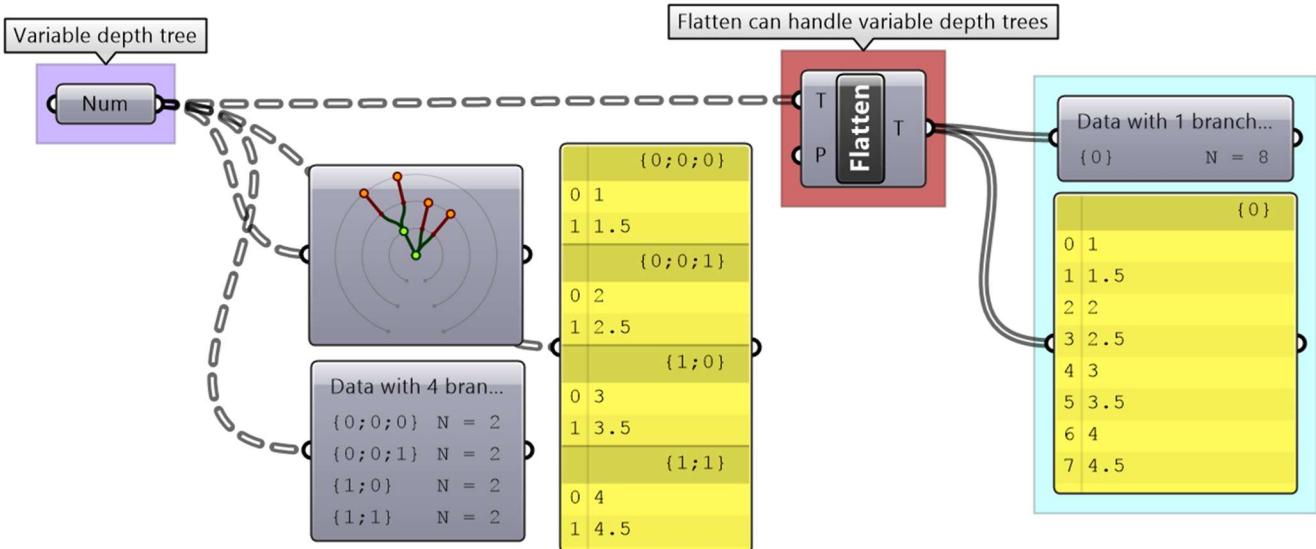


Figure (70): Flattening complex trees

### 3\_5\_5: Combining data streams

It is possible to compose a number of lists into a tree where each list becomes a branch in a new tree. It is different from the merging of lists where simply one bigger list is created.

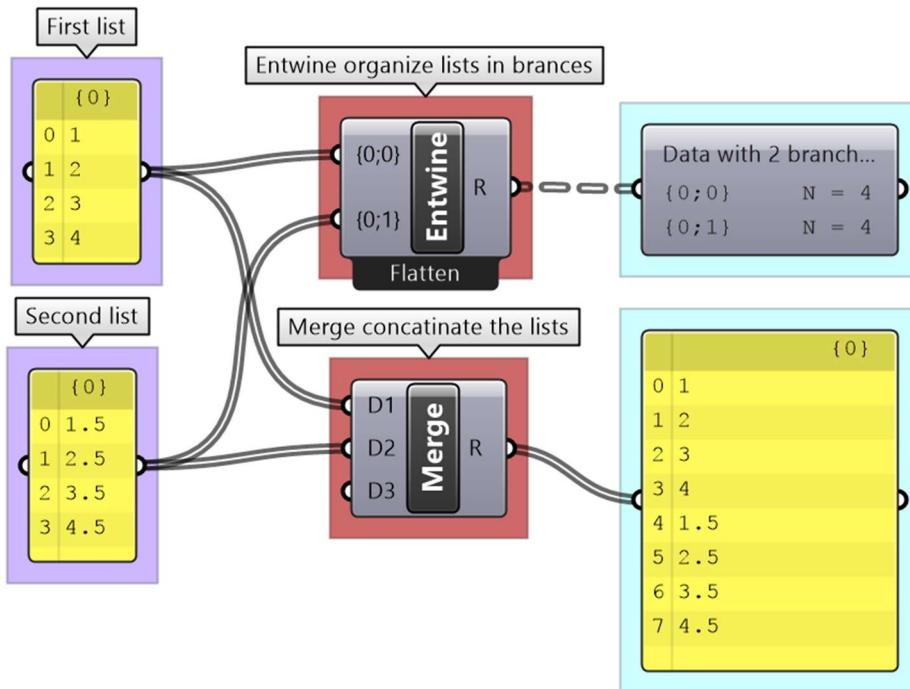


Figure (71): **Entwine** and **Merge** components combine lists into trees or bigger lists

### 3\_5\_6: Flipping the data structure

It is logical in some cases to flip the tree to change the direction of branches. This is specially useful in grids when points are organized in rows and columns (similar to a 2 dimensional array structure).

Flipping causes corresponding elements across branches (have same index in their branch) to be grouped in one branch. For example, a data tree that has 2 branches and 4 items in each branch, can be flipped into a tree with 4 branches and 2 elements in each branch.

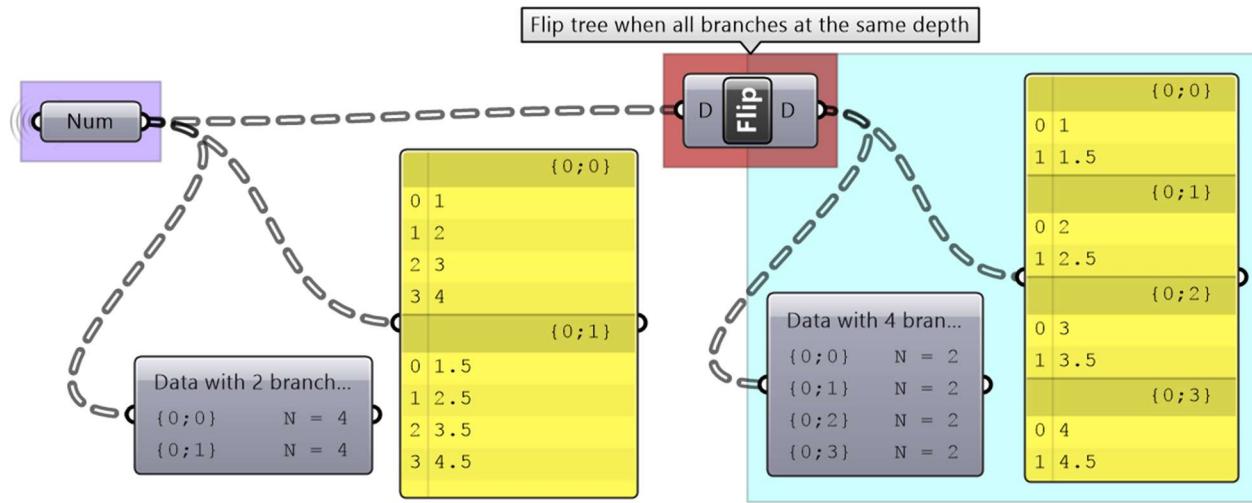


Figure (72): **Flip** helps reorganize data trees

If the number of elements in the branches are variable in length, some of the branches in the flipped tree will have “null” values.

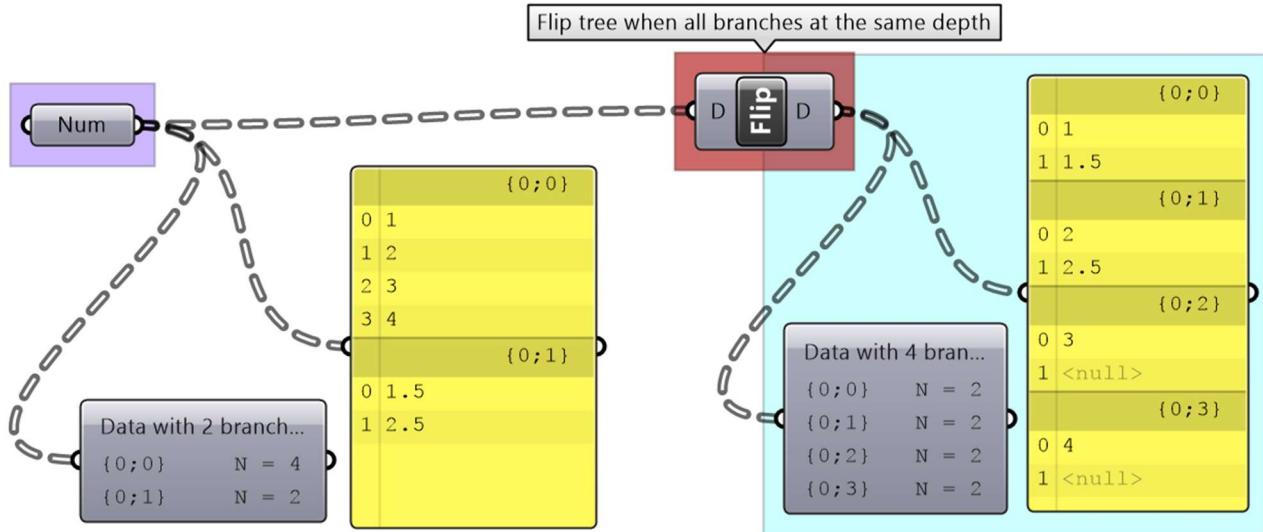


Figure (73): Add “null” when flipping trees with variable length branches

Flipping is one of the operations that cannot handle variable depth branches, simply because there is no logical solution to flip.

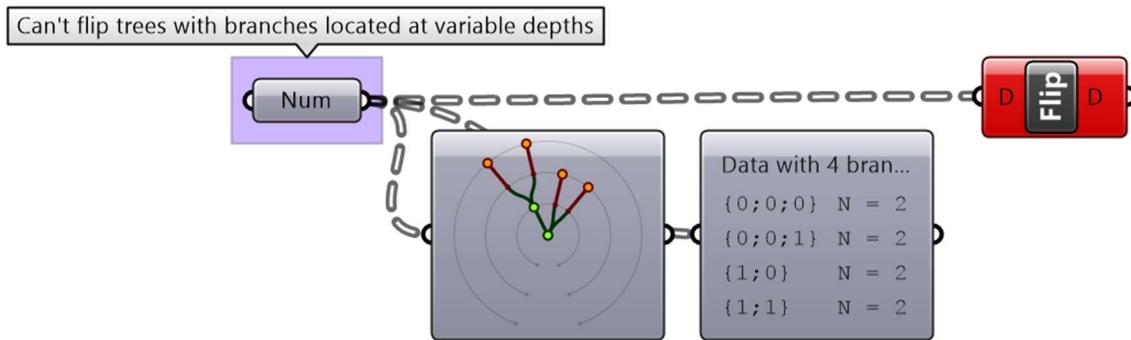


Figure (74): **Flip** fails when the input tree has variable depth branches

### 3\_5\_7: Simplifying the data structure

Processing data through multiple components can add unnecessary complexity to the data structure. The most common form is adding leading or trailing zeros to the paths addresses. Complex data structures are hard to match. **Simplify Tree** process helps remove empty branches. There are other operations such as **Clean Tree** and **Trim Tree** to help remove null elements, empty branches and reduce complexity. It is also possible to extract all branches as separate lists using **Explode Tree** operation.

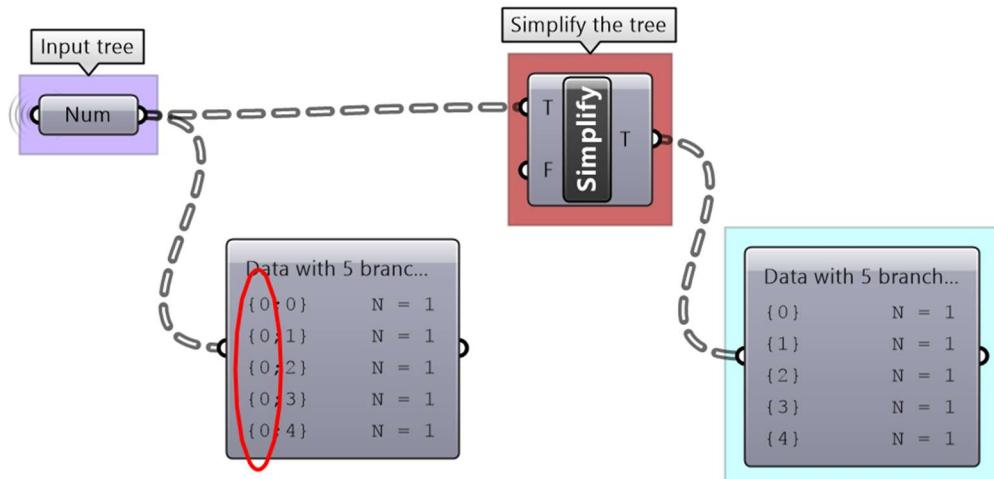
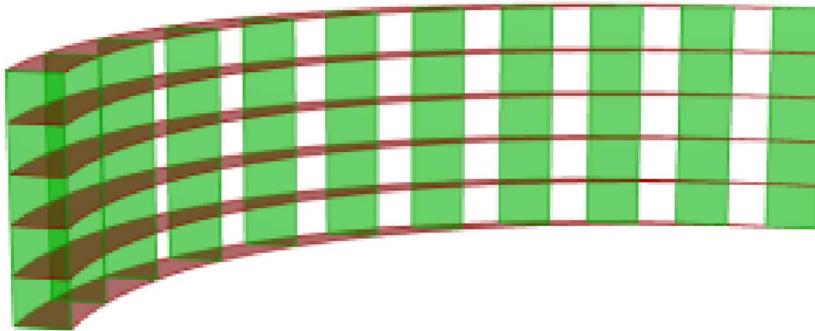


Figure (75): Paths can increase in complexity as more operations are applied to the data. **Simplify** helps remove empty branches

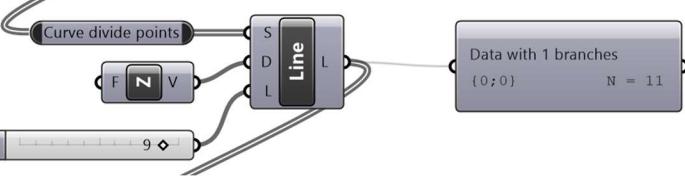
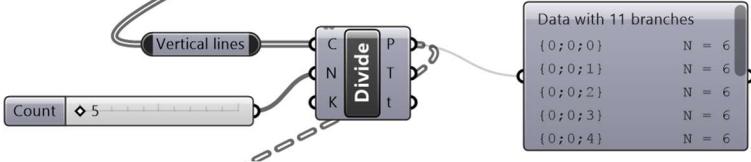
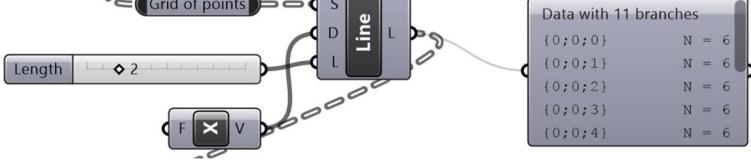
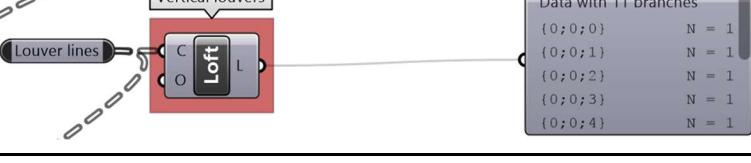
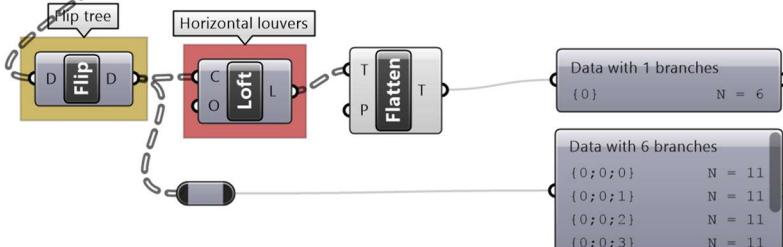
### 3\_5\_8 Basic tree operations tutorial #1

Given one curve on XY-Plane, create horizontal and vertical louvers as in the image



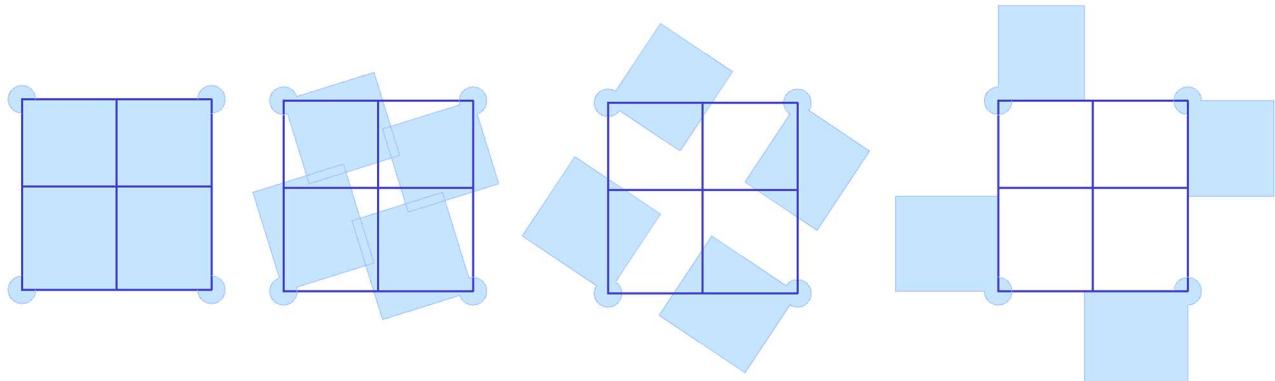
#### Solution

<b>Input curve</b> <u>Data structure:</u> single item (one branch and one item in the branch)	
<b>Divide curve to extract points.</b> <u>Data structure:</u> list (one branch with 11 items). Note that the path has added leading "0". This indicates the next layer of calculation.	

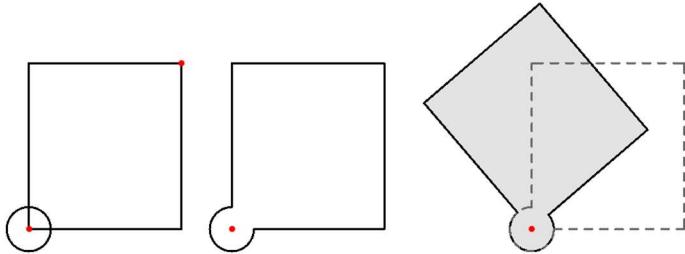
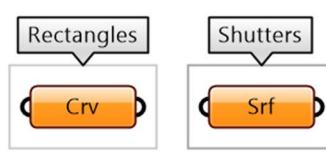
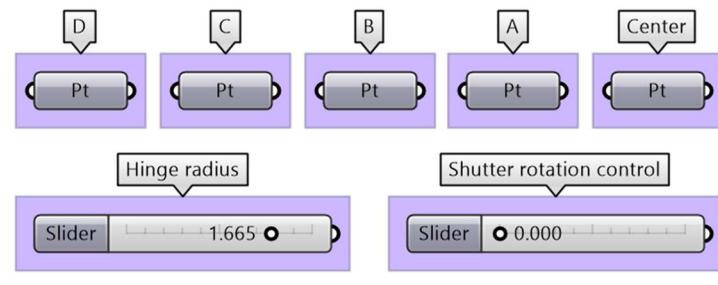
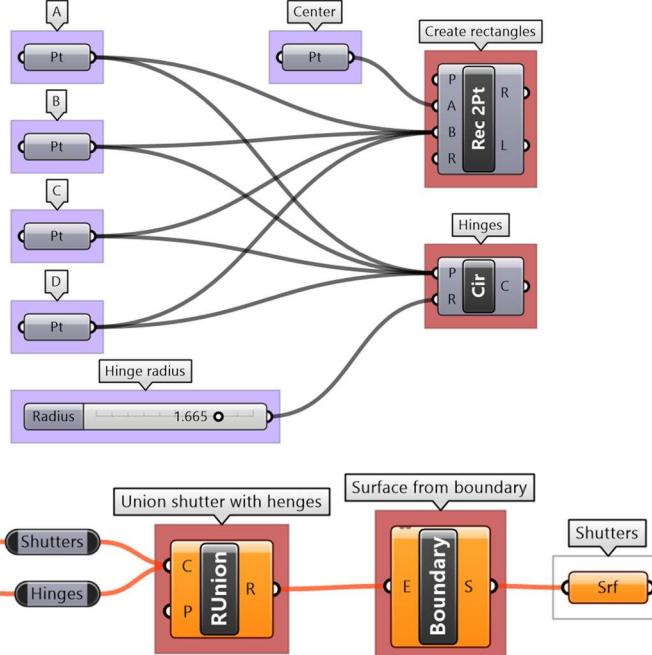
<p>Create vertical <b>lines</b> at each point.  <u>Data structure:</u> list (one branch with 11 items). Note that the path did not increase in complexity.</p>	 <p>Curve divide points → Line → Data with 1 branches  {0;0} N = 11</p>
<p>Divide vertical lines to create a <b>grid of points</b>.  <u>Data structure:</u> Tree (11 branches with 6 items). Note that the path has added leading "0".</p>	 <p>Vertical lines → Divide → Data with 11 branches  {(0;0;0), (0;0;1), (0;0;2), (0;0;3), (0;0;4)} N = 6</p>
<p>Create horizontal <b>lines</b> at each point.  <u>Data structure:</u> Tree (11 branches with 6 items). Note that the path did not increase in complexity.</p>	 <p>Grid of points → Line → Data with 11 branches  {(0;0;0), (0;0;1), (0;0;2), (0;0;3), (0;0;4)} N = 6</p>
<p>Create lofted <b>surfaces</b> through branches of lines.  <u>Data structure:</u> Tree (11 branches with 1 item each). Note that the path did not increase in complexity.</p>	 <p>Louver lines → Loft → Data with 11 branches  {(0;0;0), (0;0;1), (0;0;2), (0;0;3), (0;0;4)} N = 1</p>
<p>Flip the tree matrix and then create lofted <b>surfaces</b> through branches of lines.  <u>Data structure:</u> Tree (11 branches with 1 item each). Note that the path did not increase in complexity.</p> <p>You can <b>flatten</b> the tree to create one list of horizontal louvers.</p>	 <p>Flip tree → Loft → Flatten → Data with 1 branches  {0} N = 6</p> <p>Horizontal louvers → Loft → Flatten → Data with 6 branches  {(0;0;0), (0;0;1), (0;0;2), (0;0;3)} N = 11</p>

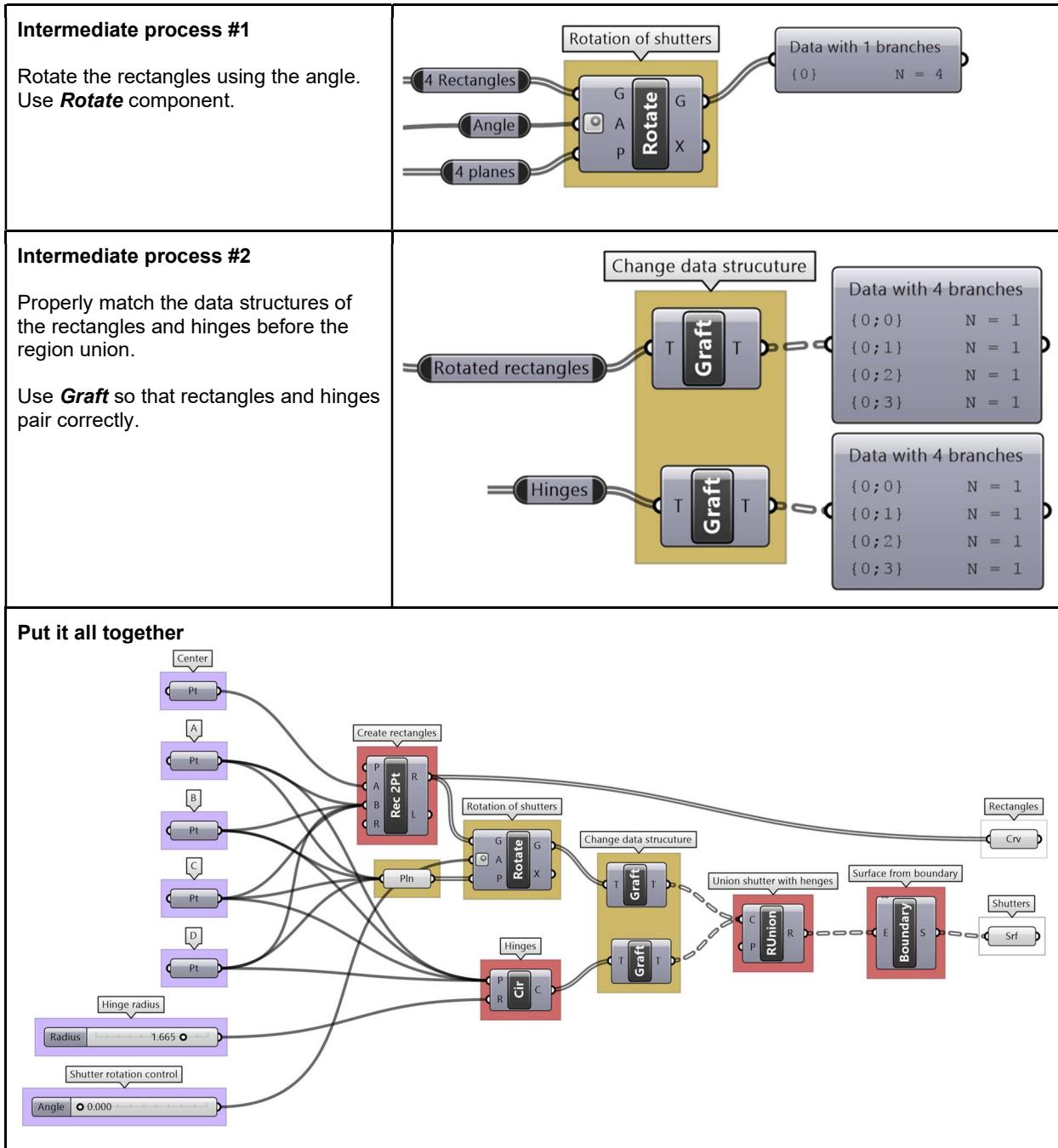
### 3\_5\_9 Simple tree operations tutorial #2

Given four corner points on a plane and a radius for the hinge, create a shutter that can open and shut as in the image using a rotation parameter.



#### Algorithm analysis

<p>For each shutter there are two parts: the rectangle and the hinge.</p> <p>Union the rectangle and hinge, then allow rotating around the hinge.</p> <p>There is one rotation control to move all shutters together</p>	
<h3>Grasshopper implementation</h3>	
<p><b>Output</b> Surface of the shutters Curves for the frame</p>	
<p><b>Input</b> 4 corner points (and center) Hinge radius Rotation parameter</p>	
<p><b>Key processes</b>  Create rectangle and hinges. Use <b>Rectangle</b>  Union the curves. Use <b>RUnion</b> Create a surface from the boundary. Use <b>Boundary</b> component</p>	



## 3\_6: Advanced tree operations

As your solutions increases in complexity, so will your data structures. We will discuss three advanced tree operations that are necessary to solve specific problems, or are used to simplify your solution by tabbing directly into the power of the data tree structure.

### 3\_6\_1: Relative items

The first operation has to do with solving the general problem of connectivity between elements in one tree or across multiple trees. Suppose you have a grid of points and you need to connect the points

diagonally. For each point, you connect to another in the  $+1$  branch and  $+1$  index. For example a point in branch  $\{0\}$ , index  $[0]$ , connects to the point in branch  $\{1\}$ , index  $[1]$ .

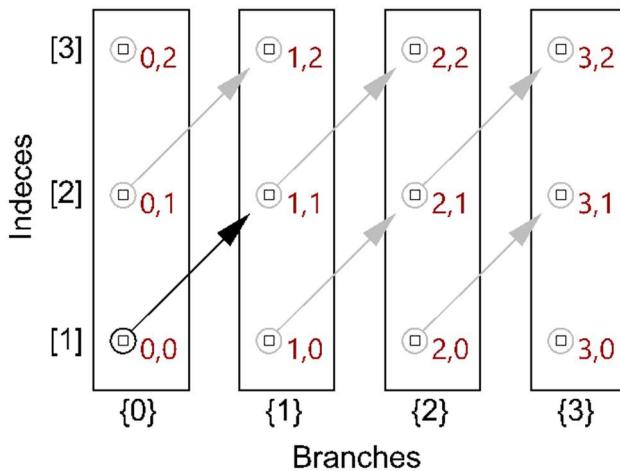


Figure (76): Relative Item mask  $\{+1\}[+1]$  create positive diagonal connectivity

In Grasshopper, the way you communicate the offset is expressed with an offset string in the format “ $\{\text{branch offset}\}[\text{index offset}]$ ”. In our example, the string to connect points diagonally is “ $\{+1\}[+1]$ ”. Here is an example that uses relative tree component in Grasshopper. Notice that the relative item component creates two new trees that correlate in the manner specified in the offset string.

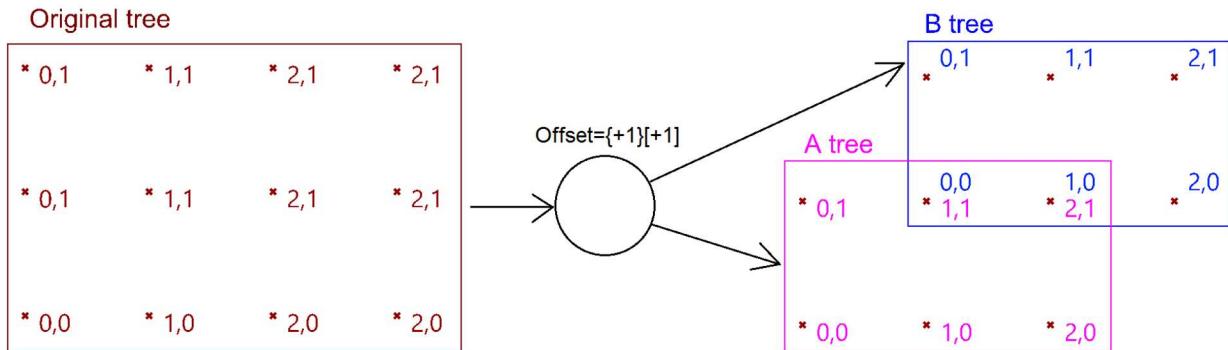


Figure (77): Relative Item mask  $\{+1\}[+1]$  breaks the original tree into 2 new trees with diagonal connectivity

Here is an example implementation in Grasshopper where we define relative items in one tree, then connect the two resulting trees with lines using the **Relative Item** component.

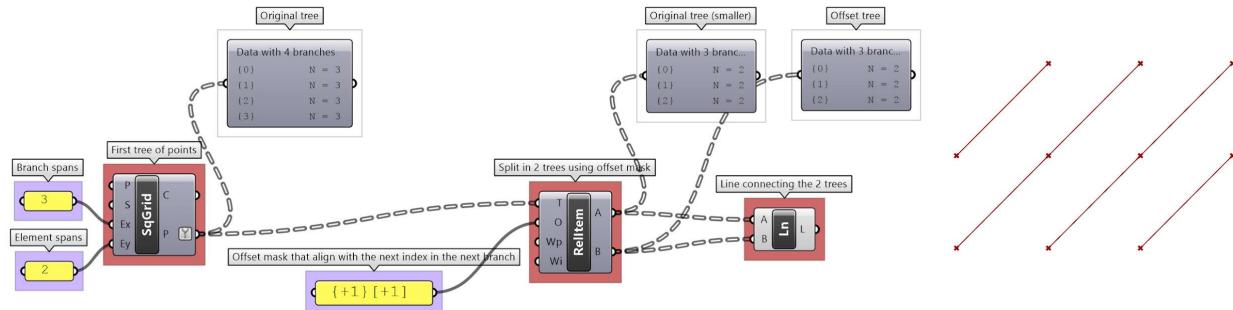
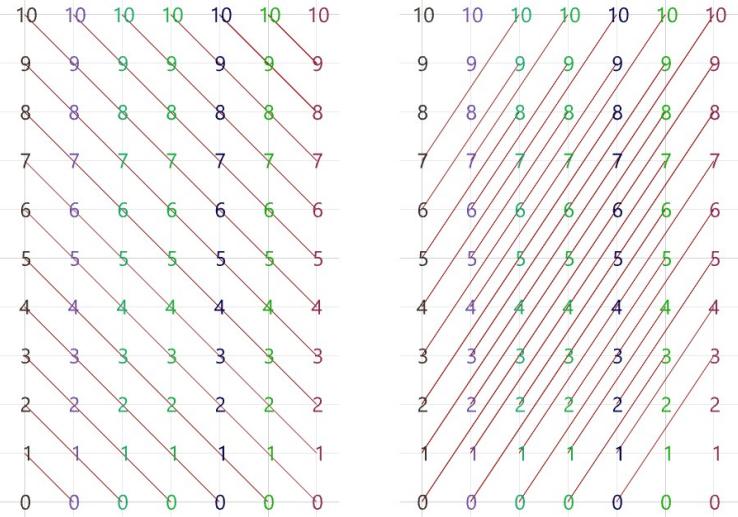


Figure (78): **Relative Item** with mask  $\{+1\}[+1]$  in Grasshopper

### 3\_6\_1\_1 Relative item tutorial #1

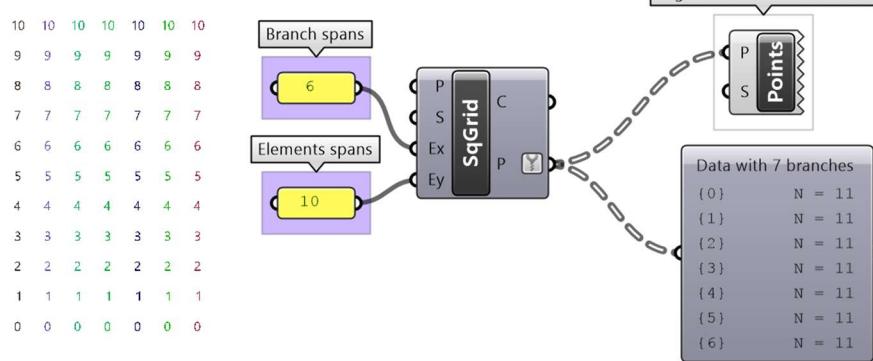
Create the pattern shown in the image using a square grid of 7 branches where each branch has 11 elements.



## Solution

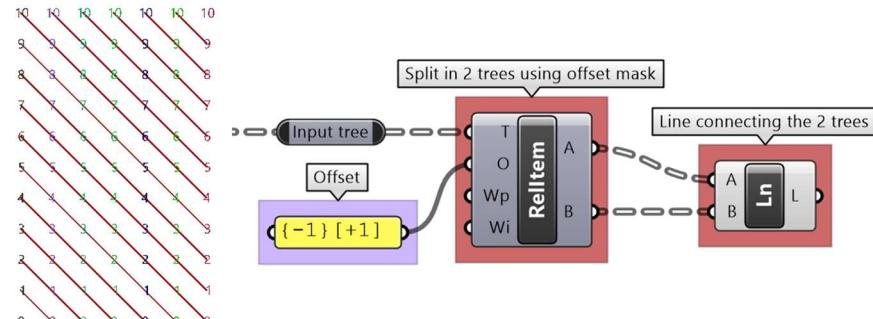
### Define the {branch\_offset} [index\_offset]

Create the grid

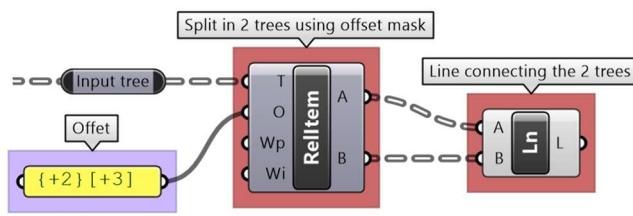
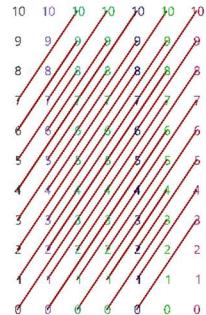


Create relative trees that connect each element with -1 branch and +1 index: {-1}{+1}

Create lines to connect the 2 relative trees.



Change the offset to  $\{+2\}[+3]$  to create the second connections



We showed how to define relative items in one tree, but you can also specify relative items between 2 trees. You'll need to pay attention to the data structure of the two input trees and make sure they are compatible. For example, if you connect each point from the first tree with another point from a different tree with the same index, but +1 branch, then you can set the offset string to be  $\{+1\}[0]$ .

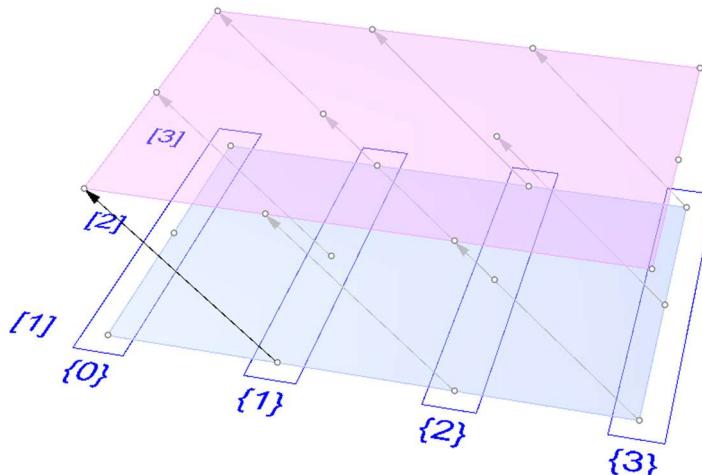


Figure (79): **Relative Items** create connections across multiple trees

The input to the **Relative Items** component is two trees and the output is two trees with corresponding items according to the offset string.

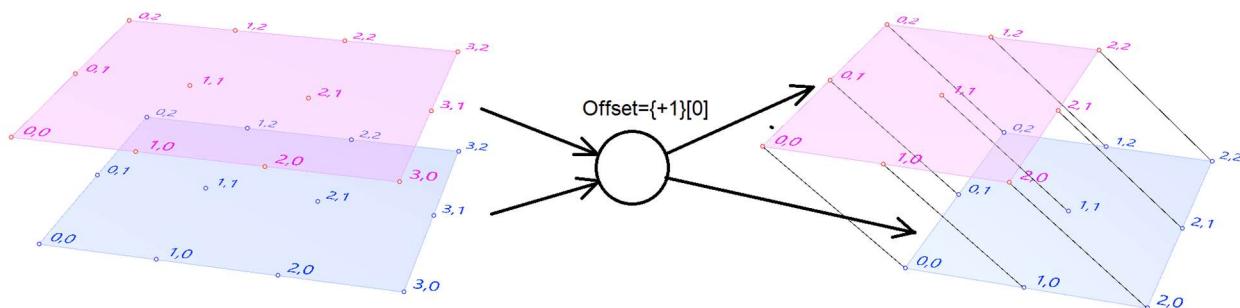


Figure (80): The offset mask of the **Relative Items** generates new trees with the desired connections

The following GH definition achieves the above:

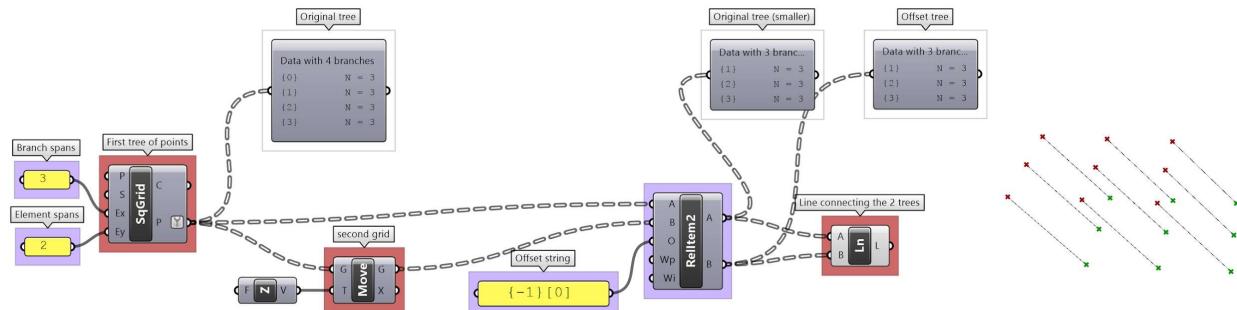
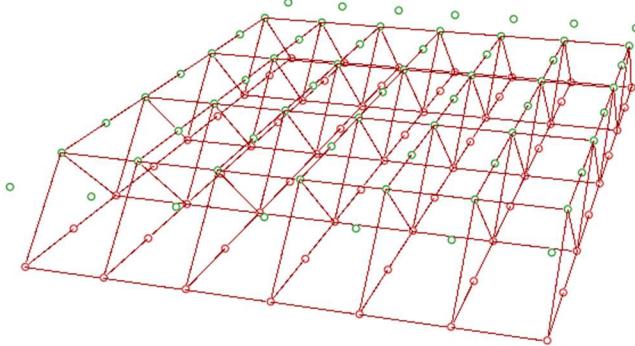


Figure (81): **Relative Items** implementation in Grasshopper

### 3\_6\_1\_2 Relative item tutorial #2

Use relative items between 2 bounding grids to generate the structure shown in the image.

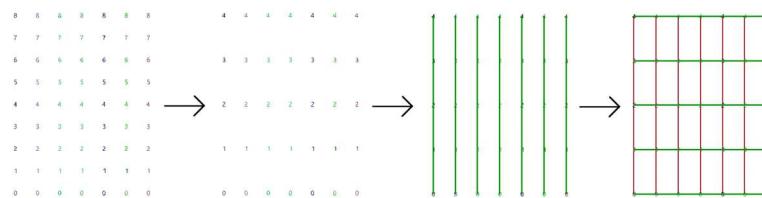


#### Solution

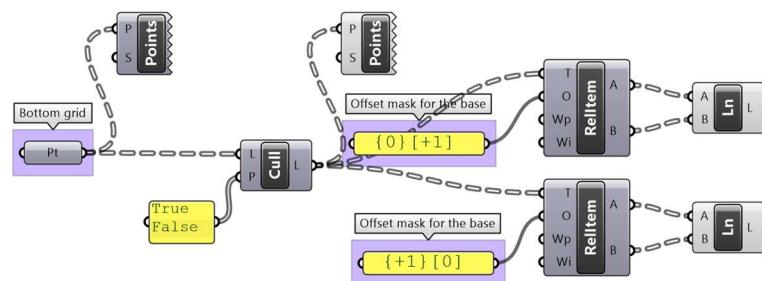
##### Bottom tree connections

Cull every other index and keep the same number of branches (cull indices 1, 3,...)

Define the offset strings for *Relativelitem* components to create the vertical and horizontal connections



##### Grasshopper definition



##### Top tree connections

<p>Cull every other index and keep the same number of branches. (cull indices 0, 2,...)</p> <p>Define the offset strings for <i>RelativeItem</i> components to create the vertical and horizontal connections</p>	
Grasshopper definition	
<b>Connections between the 2 trees</b>	
Use culled grids, then define first offset string for <i>RelativeItems</i> component to create the first set of cross lines: {0}[0]	
Define second offset string for <i>RelativeItems</i> component to define the second set of cross lines: {0}[-1]	

### 3\_6\_2: Split trees

The ability to select a portion of a tree, or split into two parts is a very powerful tree operation in Grasshopper. You can split the tree using a string mask that specifies the positive output of your tree, and what's left is called the negative tree and is given as an output. Since all trees are made out of branches and indices, the split mask should include information about which branches and indices within these branches to split. Here are the rules of the split mask

Split tree mask: syntax and general rules	
{;;}	Use curly brackets to enclose the mask for the tree branches.
[]	Use square brackets to enclose the mask for the elements (leaves), inside square brackets. Can omit if select all items or use [*]
()	Round brackets are used for organizing and grouping
*	Any number of integers in a path. The asterisk also allows you to include all branches, no matter what their paths look like

?	Any single integer
6	Any specific integer
!6	Anything <u>except</u> a specific integer
(2,6,7)	Any one of the specific integers in this group.
!(2,6,7)	Anything <u>except</u> one of the integers in this group.
(2 to 20)	Any integer in this range (including both 2 and 20).
!(2 to 20)	Any integer <u>outside of</u> this range.
(0,2,...)	Any integer part of this infinite sequence. Sequences have to be at least two integers long, and every subsequent integer has to be bigger than the previous one (sorry, that may be a temporary limitation, don't know yet).
(0,2,...,48)	Any integer part of this finite sequence. You can optionally provide a single sequence limit after the three dots.
!(3,5,...)	Any integer <u>not</u> part of this infinite sequence. The sequence doesn't extend to the left, only towards the right. So this rule would select the numbers 0, 1, 2, 4, 6, 8, 10, 12 and all remaining even numbers.
!(7,10,21,...,425)	Any integer <u>not</u> part of this finite sequence.
{ * }[ (0 to 4) or (6,11,41) ]	It is possible to combine two or more rules using the boolean and/or operators. The example selects the first five items in every list of a tree and also the items 7, 12 and 42, then the selection rule

Here are some examples of valid split masks.

<b>Split by branches</b>	
{ * }	Select all (the whole tree output as positive, and negative tree will be empty)
{ *, 2 }	Select the third branch
{ *, (0,1) }	Select the first two end branches
{ *, (0, 2, ...) }	Select all even branches
<b>Split by branches and leaves</b>	
{ * }[(1,3,...)]	Select elements located at odd indices in all branches
{ *; 0 }[(1,3,...)]	Select elements located at odd indices in the first branch
{ *; (0, 2) }[(1,3,...)]	Select elements located at odd indices in the first and third branches

<code>{*, (0,2,...) } [ (1,3,...) ]</code>	Select elements located at odd indices in branches located at even indices
<code>{*, (0,2,...) } [(0) or (1,3,...)]</code>	Select elements located at odd indices, and index “0”, in branches located at even indices

One of the common applications that uses split tree functionality is when you have a grid of points that you like to transform a subset of it. When splitting, the structure of the original tree is preserved, and the elements that are split out are replaced with null. Therefore, when applying transformation to the split tree, it is easy to recombine back.

Suppose you have a grid with 7 branches and 11 elements in each branch, and you'd like to shift elements between indices 1-3 and 7-9. You can use the split tree to help isolate the points you need to move using the mask: `{*}[ (1,2,3) or (7,8,9) ]`, move the positive tree, then recombine back with the negative tree.

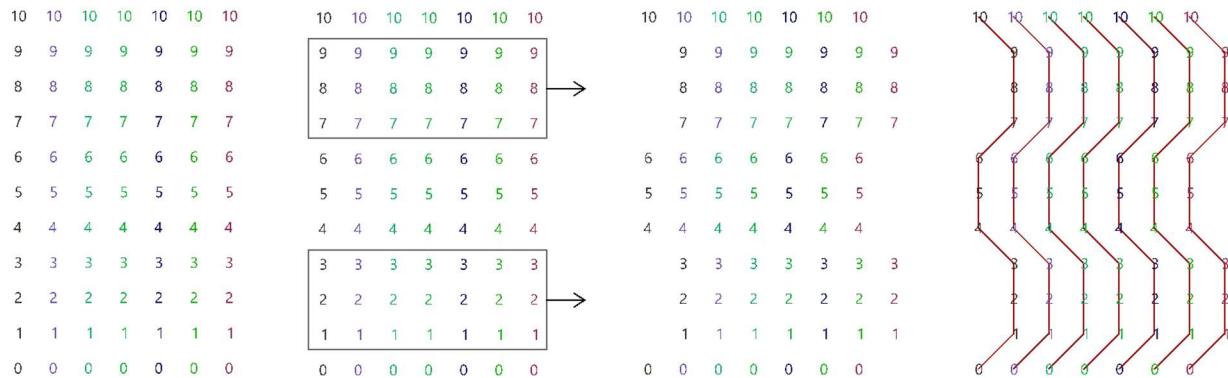


Figure (82): **Split tree** allows operating on a subset of the tree with the possibility to recombine back

This is the GH definition that does the above using the **Split Tree** component.

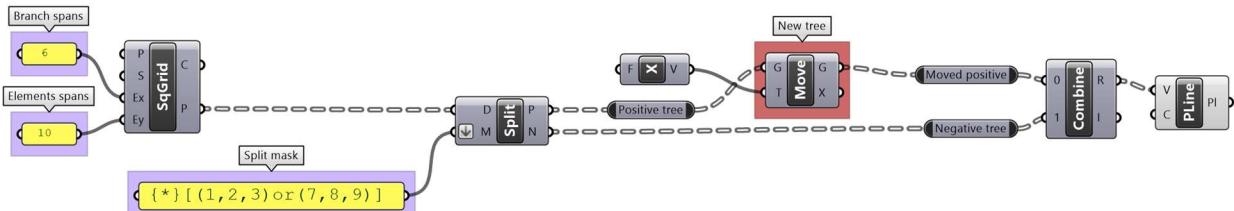
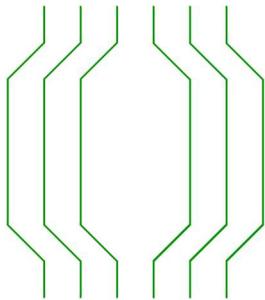


Figure (83): **Split tree** Grasshopper implementation of Figure (73)

One of the advantages of using **Split Tree** over relative trees is that the split mask is very versatile and it is easier to isolate the desired portion of the tree. Also the data structure is preserved across the negative and positive trees which makes it easy to recombine the elements of the tree after processing the parts.

### 3\_6\_2\_1 Split tree tutorial #1

Given a 6x9 grid, use the split tree to generate the following form.



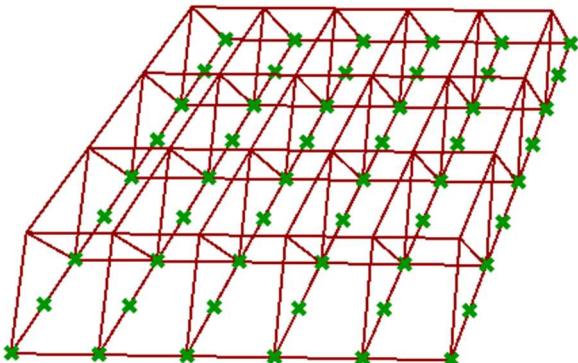
#### Solution steps

Create the grid	
Split the tree to isolate the middle part	
Split the middle part into two new parts	

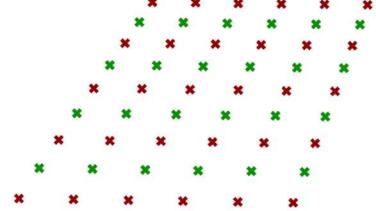
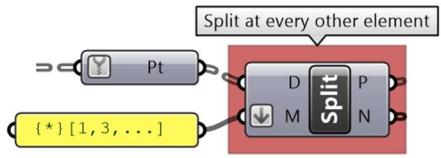
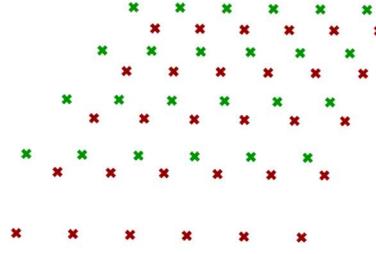
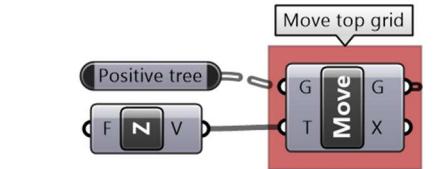
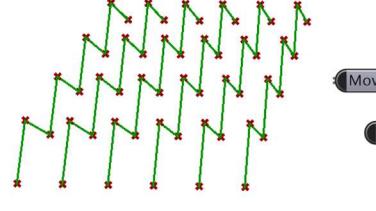
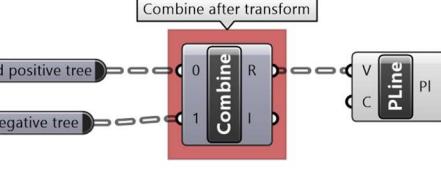
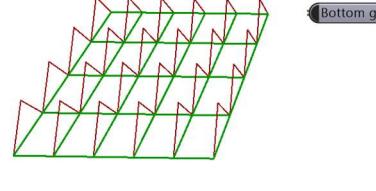
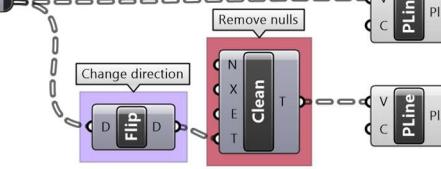
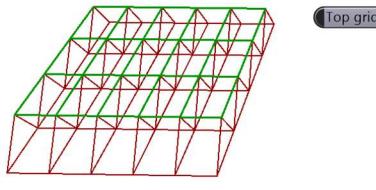
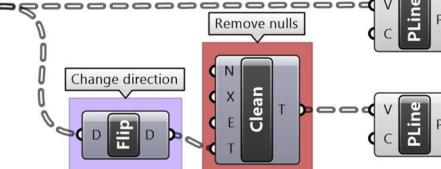
<p>Move the two middle parts in opposite directions then recombine them</p>	<p>Left tree → Shift → New tree</p> <p>Right tree → Shift → New tree</p> <p>Combine middle part</p>
<p>Recombine the middle part with the rest of the tree and create polylines through each branch elements</p>	<p>Middle tree → Combine trees → PLine</p> <p>Negative tree</p>

### 3\_6\_2\_2 Split tree tutorial #2

Given a grid, create the following truss system using split tree functionality.



<p><b>Solution</b></p> <p>Create the 6x9 grid</p>	<p>Branch spans: 5</p> <p>Element spans: 8</p> <p>Create the grid</p>
---	---

Split at every other element		
Move positive tree vertically		
Combine positive and negative trees And create a polyline through each branch elements		
Create bottom curves using negative tree		
Create top curves using positive tree		

### 3\_6\_3: Path mapper

When dealing with complex data structures such as the Grasshopper data trees, you'll find that you need to simplify or rearrange your elements within the tree. There are a few components offered in Grasshopper for that purpose such as **Flatten**, **Graft** or **Flip**. While very useful, these might not suffice when operating on multiple trees or needing custom rearrangement. There is one very powerful component in Grasshopper that helps with reorganizing elements in trees or change the tree structure called the **Path Mapper**. It is perhaps the least intuitive to use and can cause a loss of data, but it is also the only way to find a solution in some cases, and hence it pays to address here.

The **Path Mapper** maps data between source and target paths. The source path is fixed, and is given by the input tree. You can only set the target path. There is a set of constants that help with the mapping. Here is a list of those.

item_count	Number of items in the current branch
------------	---------------------------------------

path_count	Number of paths (branches) in the tree
path_index	Index of the current path

Let's start by familiarizing ourselves with the syntax using built-in mappings inside the **Path Mappers**.

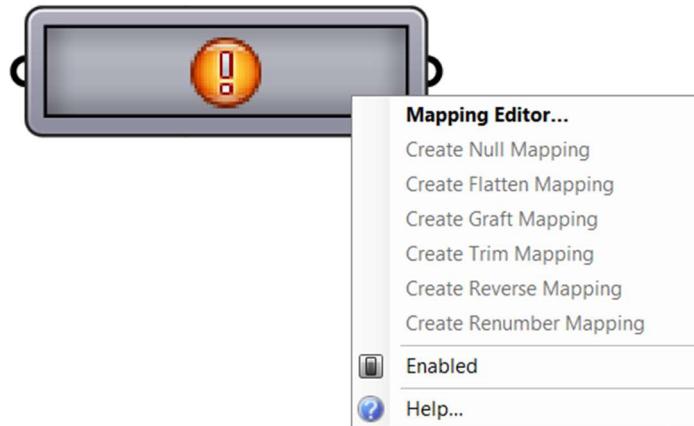
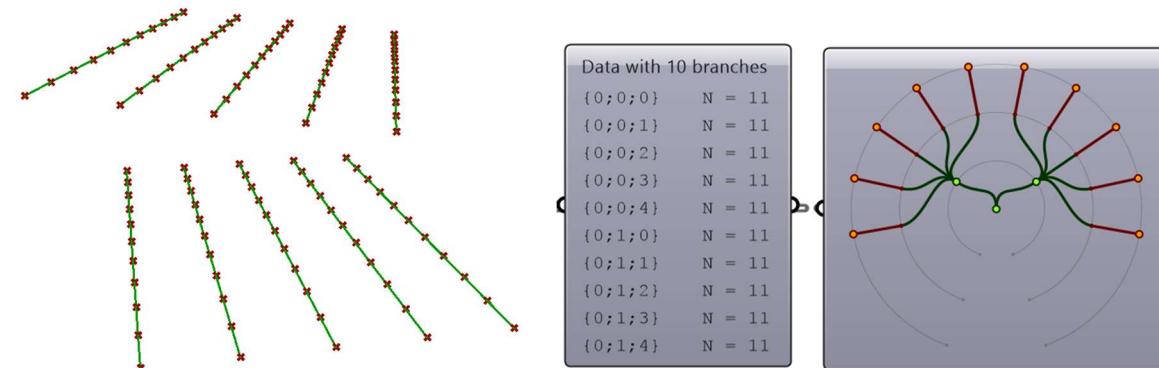
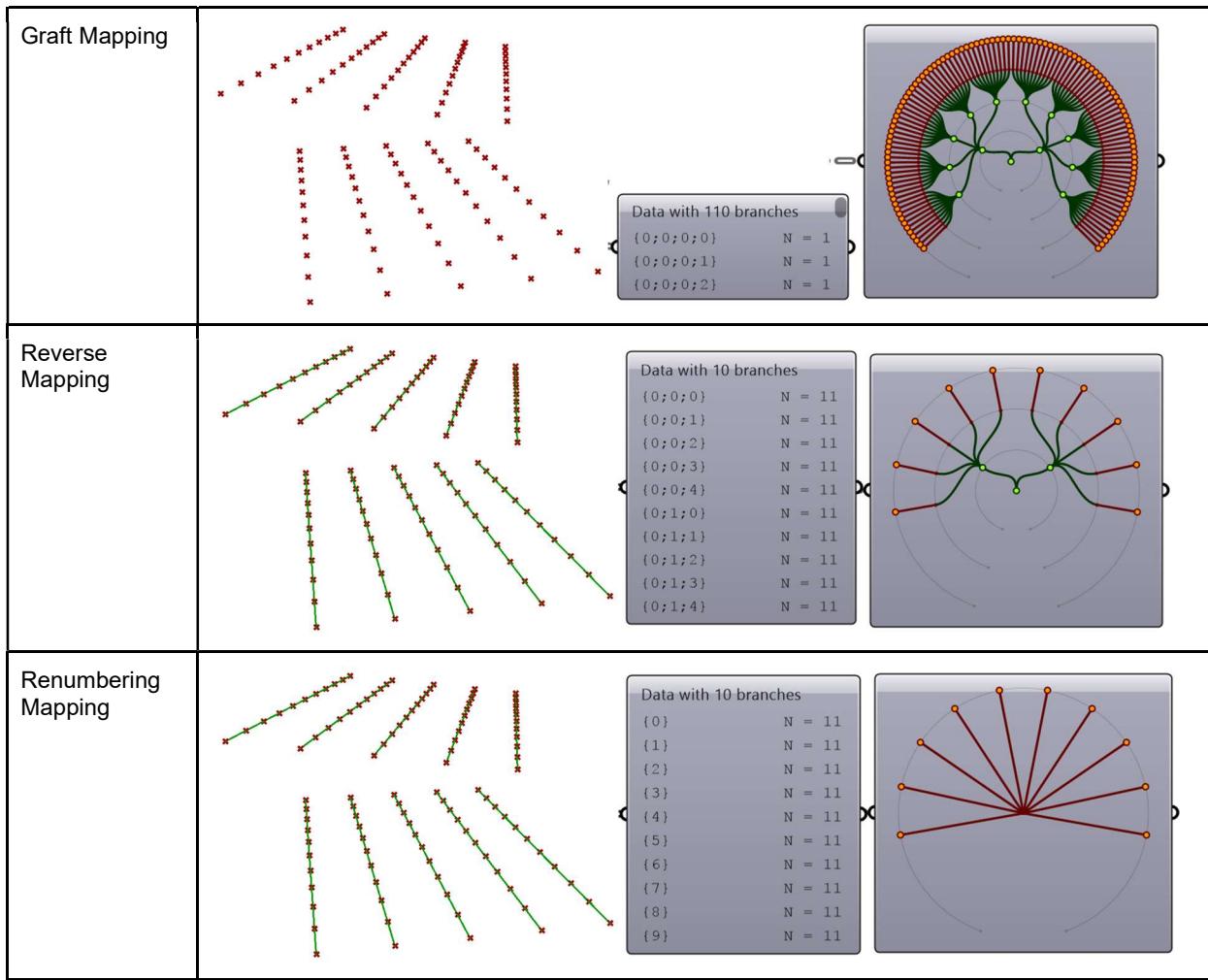


Figure (84): **Path Mapper** built-in mappings

In the following example, the input tree has two grids of points (2 trees). The data structure becomes clear when using a **Polyline** which creates one polyline through each branch. We will examine the effect of applying the built-in map on the structure and connection of points.

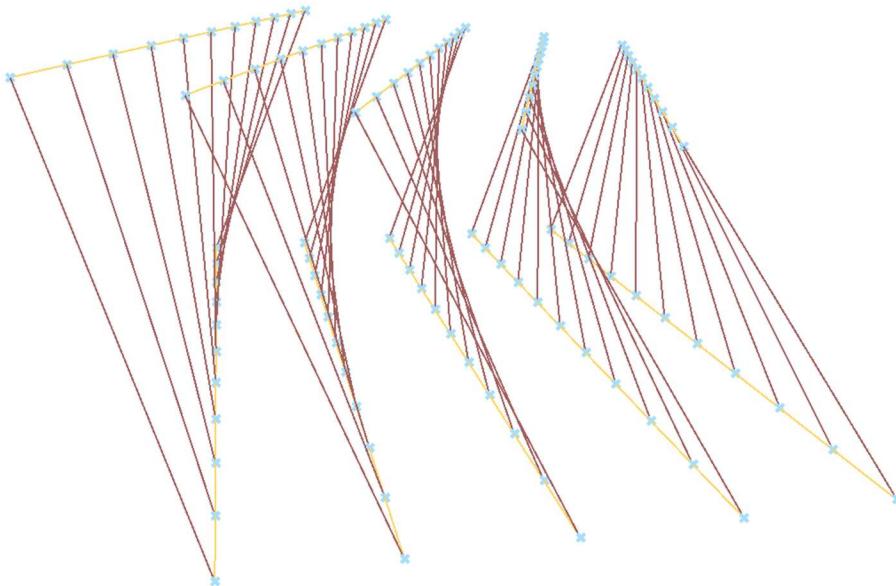


Built-in mappings inside the Path Mapper	
Null Mapping	Does not change anything.
Flatten Mapping	<p>The diagram shows the effect of 'Flatten Mapping' on a tree structure. It consists of two parts: a grid of points on the left and a circular diagram on the right. The grid shows multiple parallel lines of points. The circular diagram shows a single vertical red line connecting points from different branches, indicating that all branches have been flattened into a single vertical polyline.</p> <p>Data with 1 branches (0) N = 110</p>



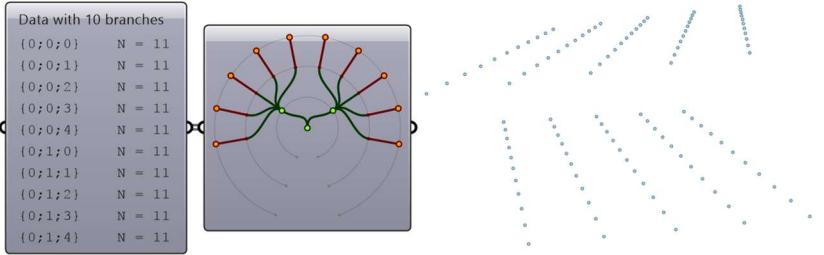
### 3\_6\_3\_1 Path mapper tutorial #1

Given the tree structures of points, create the following connections.

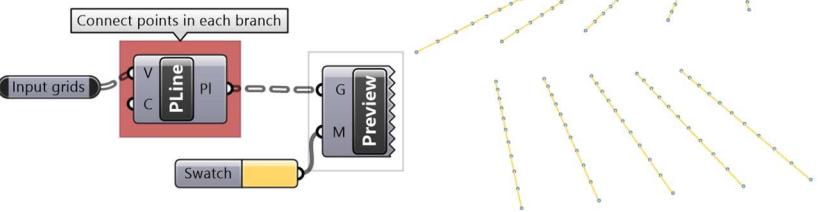


Solution

The input has two trees, and each has 5 branches with 11 elements in each branch, a total of 10 branches.



A **Polyline** can be used to connect the elements in each branch

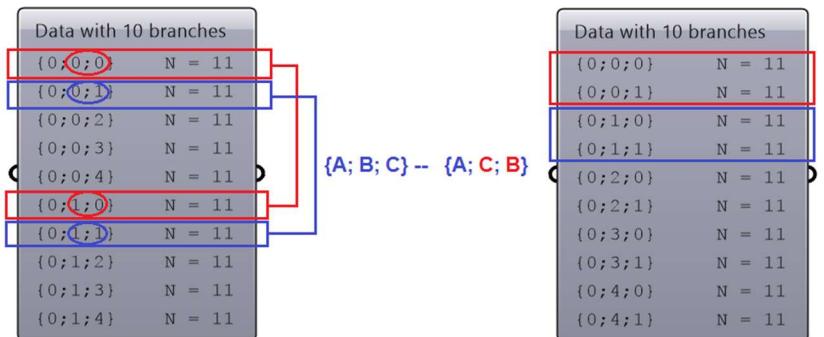


To create the vertical connections, you need to create a branch for each 2 corresponding elements across the 2 trees, then use *Polyline* to connect them

- 1- Analyze the paths of the trees
- 2- Come up with a mapping that generates the desired grouping

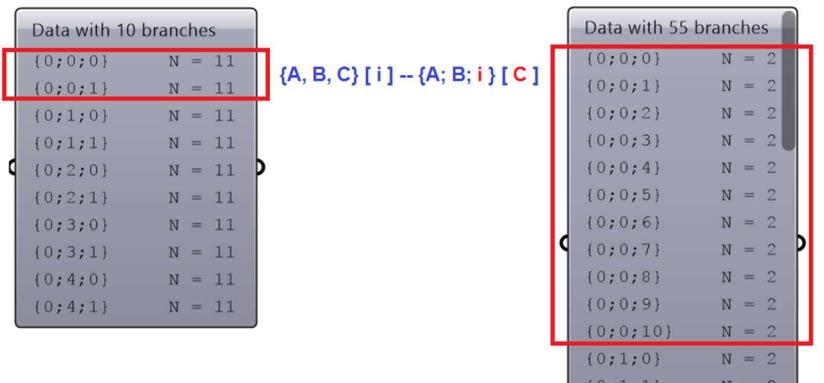
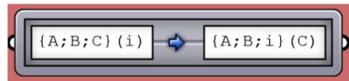
First, group corresponding branches across the 2 trees.

That can be achieved by switching the last two integers in the paths:



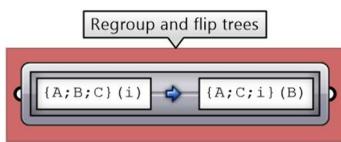
Second, Flip each of the 5 trees. Since the branches has 11 elements each, flipping each tree will create 11 branches with 2 elements in each branch. Total of 55 branches.

You flip by switching the last integer of the path with the element index:

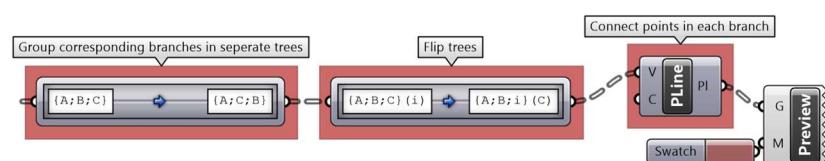
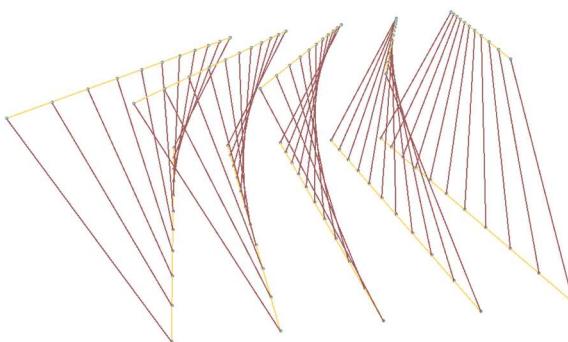


Finally, a **Polyline** makes the vertical connections.

Note: You can combine the 2 mappings in one step as in the following:

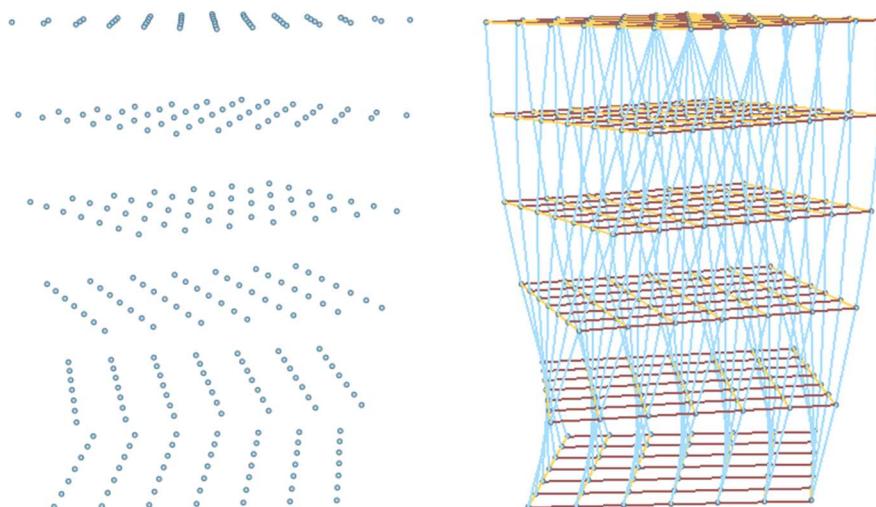


Combining is not always possible, but it can save processing time and size.



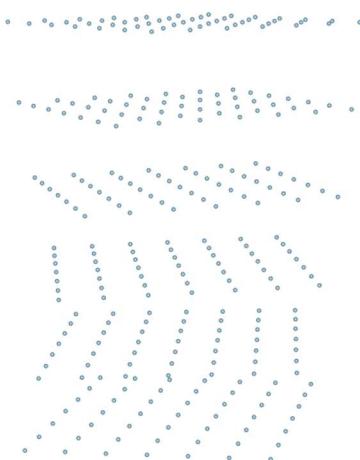
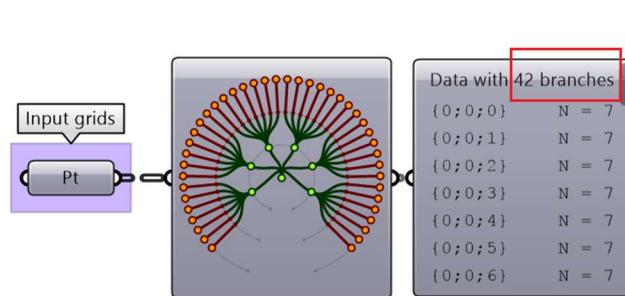
### 3\_6\_3\_2 Path mapper tutorial #2

Given the input tree of points, create the following structure.

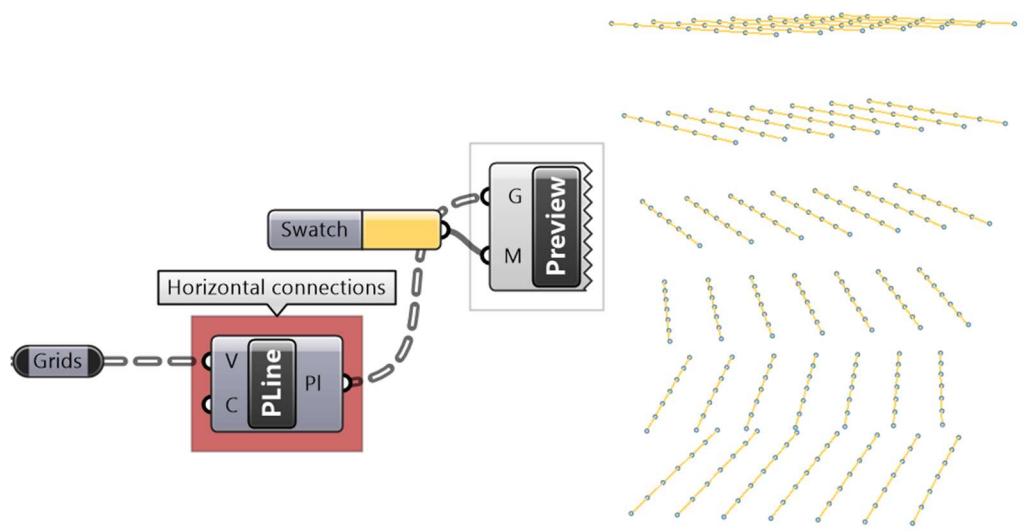


#### Solution

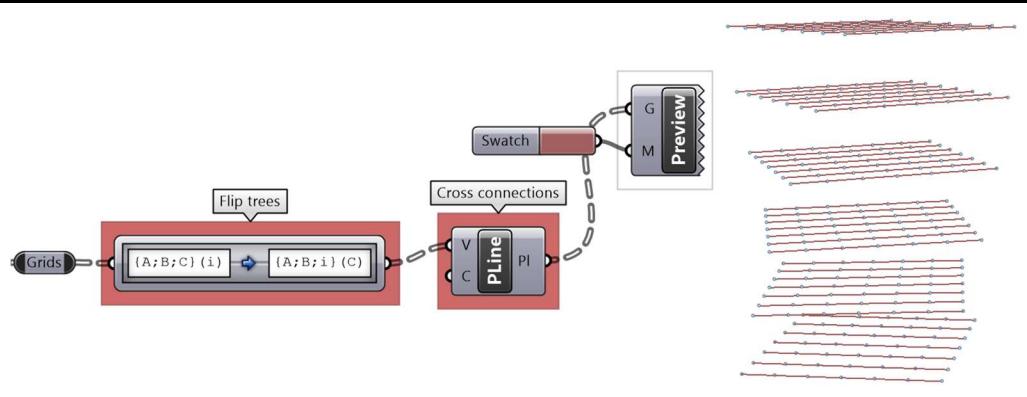
The initial tree has 42 branches, 7 branches in each of the 6 trees



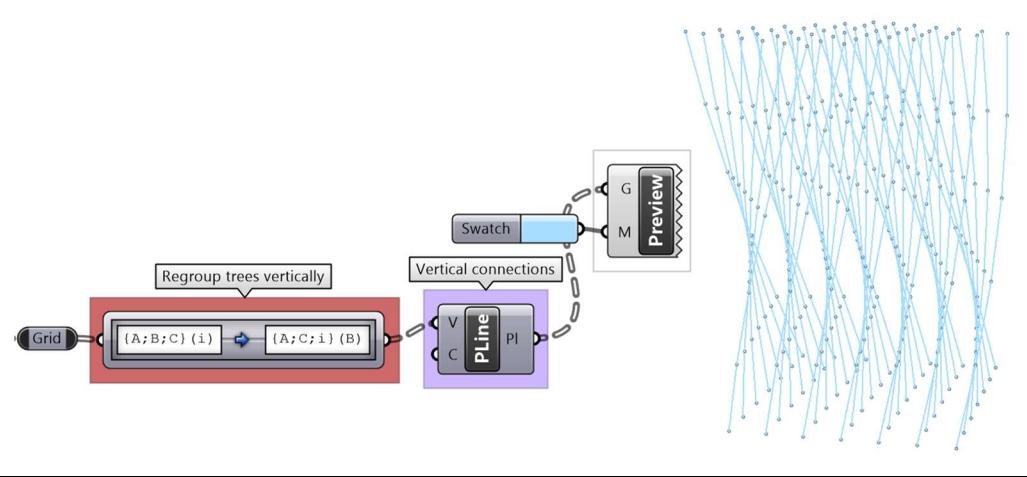
The *Polyline* component connects the elements in each branch

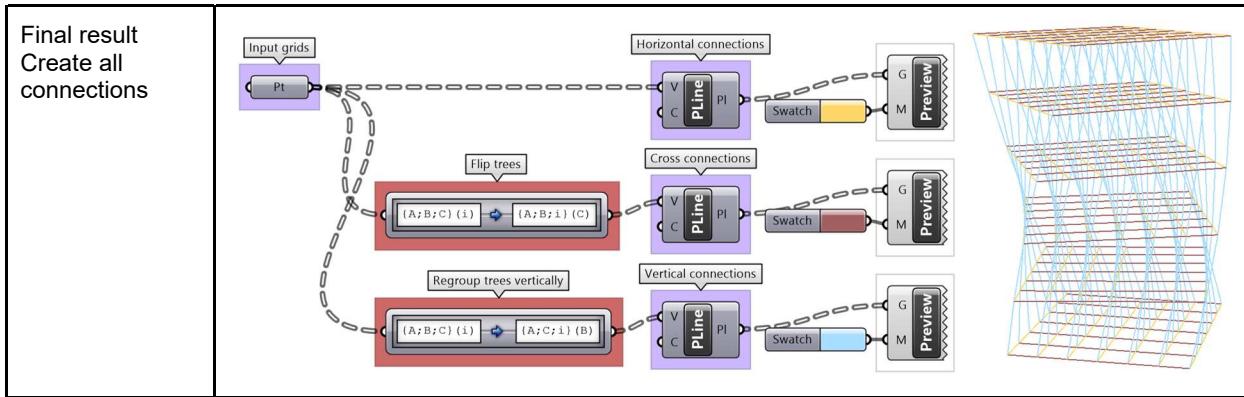


Flip the trees using *Path Mapper* by switching branch and element indices



Regroup the elements of corresponding branches in all trees using the *Path Mapper*

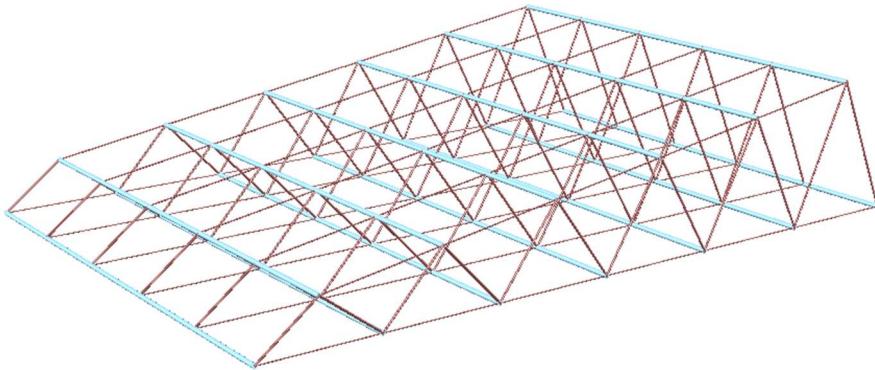




### 3\_7: Advanced data structures tutorials

#### 3\_7\_1: Sloped roof tutorial

Create a parametric truss system that changes gradually in height as shown in the image.



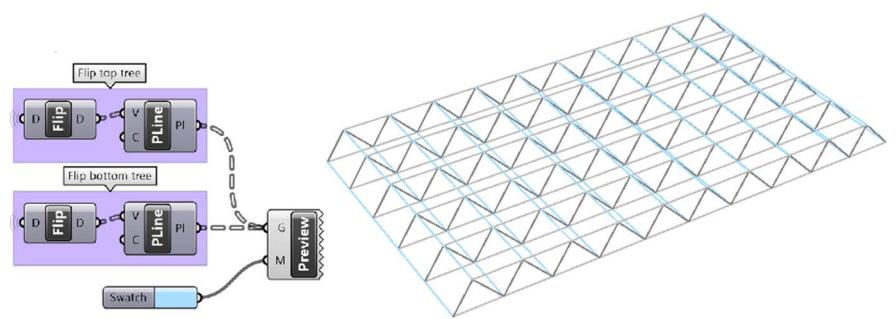
#### Solution

##### Algorithm analysis: First, solve for one simple truss

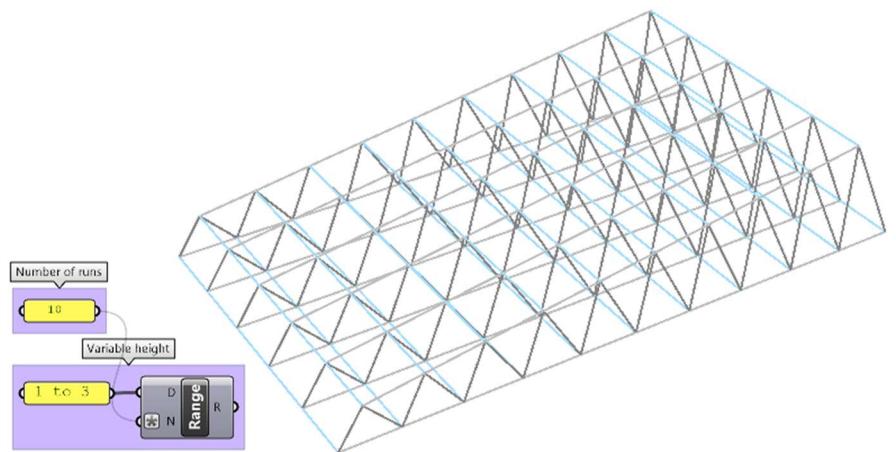
Identify desired output for a single truss	
Define initial input 1- Base line on XY-Plane 2- Number of runs 3- Height	
Identify algorithms steps:	
Create input (L=line, H=height and R= #runs)	$H=7$ 
Divide curve by $2 \times R$	

<p>Move every other point in the Z direction by height</p>	
<p>Create 3 sets of ordered points for the bottom beams, top beams and middle beams, then connect each of the 3 sets with a polyline</p>	
<p>Implement the algorithm In Grasshopper</p>	
<p>Resolve for multiple trusses with variable height</p>	
<p>Create a series of base lines using the initial line and copy in Y-Axis direction</p>	
<p>Use the list of lines as input instead of a single line. Notice that instead of a list of points for each of the 3 sets (bottom, top and middle), we now have a tree or grid of points with a number of branches equal to the number of trusses</p>	

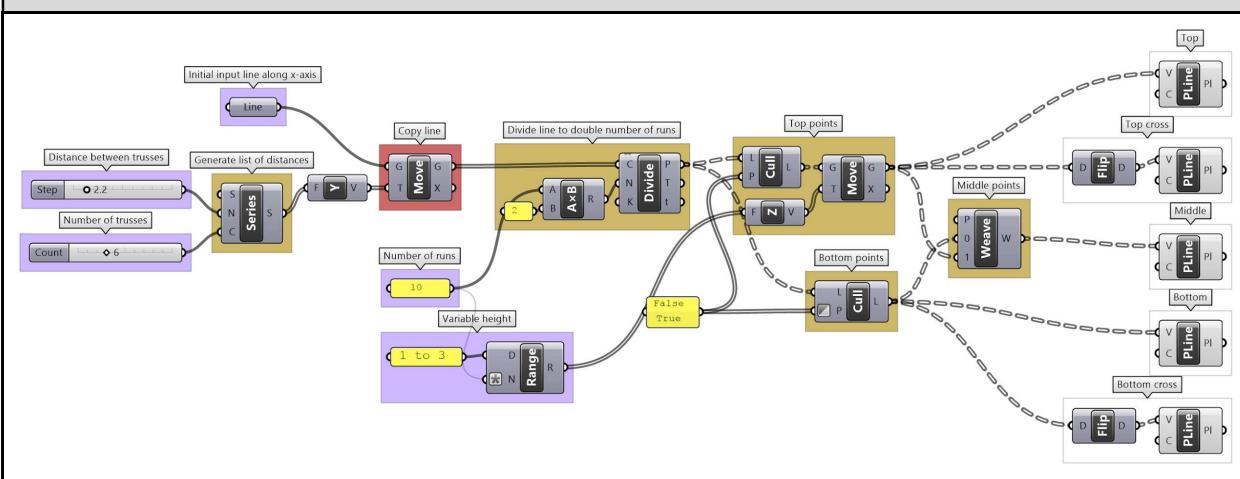
Create cross connections using *Flip* tree operation for the bottom and top trees



Create variable height

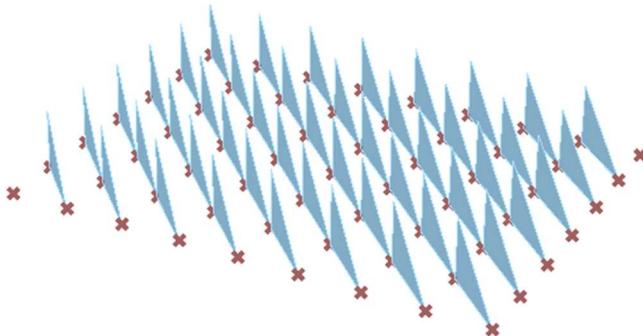


### The complete solution implementation in Grasshopper



## 3\_7\_2: Diagonal triangles tutorial

Given the input grid, use the *RelativeItem* component to create diagonal triangles



### Solution

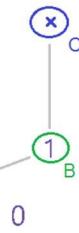
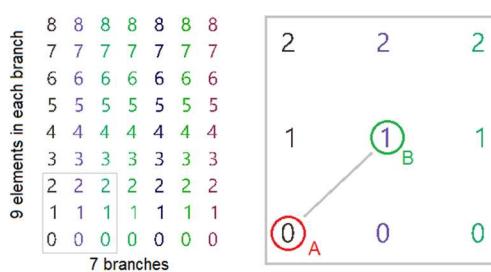
#### Algorithm analysis

To generate the triangles, we need 3 sets of corner points.

Two of the point sets (A, B) are within the grid. B is diagonal from A (relative index is +1 branch and +1 element)

The third point set (C) is a copy of set (B) moved vertically.

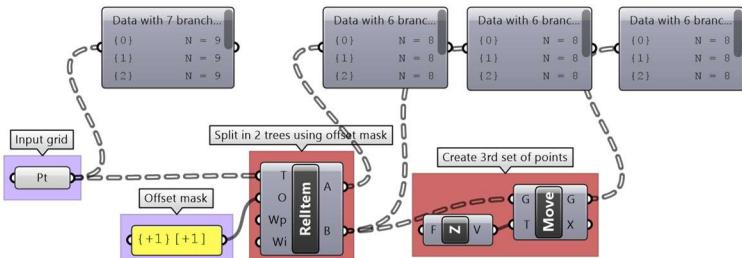
Group corners to connect into boundaries then generate surfaces



#### Grasshopper implementation

Use *RelativeItem* to create set A and set B (use “{+1}[+1] mask”)

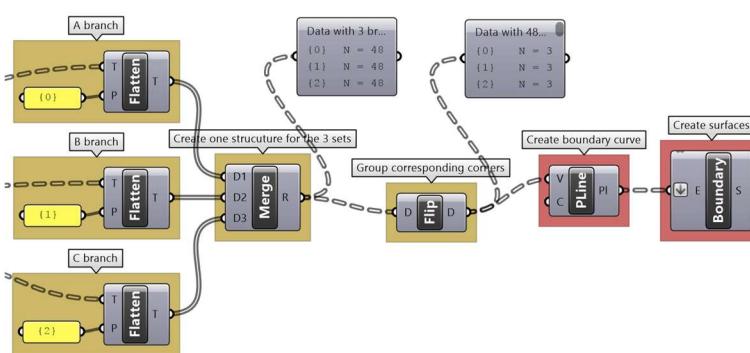
Move set B vertically.



Create a tree with 3 branches for sets A, B and C.

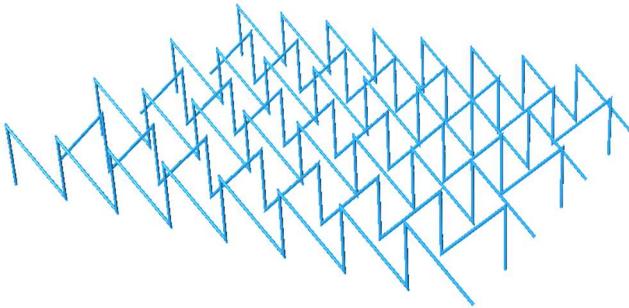
Flip the tree to group corresponding points.

Use *Polyline* and *Boundary* to generate the surfaces.



### 3\_7\_3: Zigzag tutorial

Create the structure shown in the image using a base grid as input.



#### Algorithm analysis

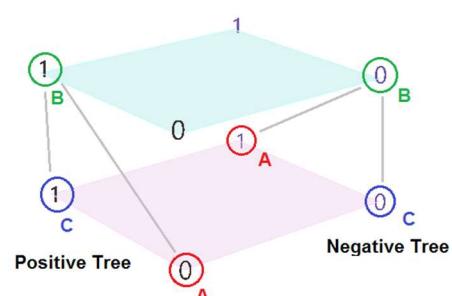
Since the zigzags alternate directions, it is best to split the grid into 2 parts, positive and negative.

Find 3 sets of points in the positive tree and order

Reverse the elements in the branches of the negative tree, then find the 3 sets of points and order

Merge back the 2 trees to create geometry through points

- {1}	{3}	{5}
8 7 6 5 4 3 2 1 0	8 7 6 5 4 3 2 1 0	8 7 6 5 4 3 2 1 0
+ {0}	{2}	{6}



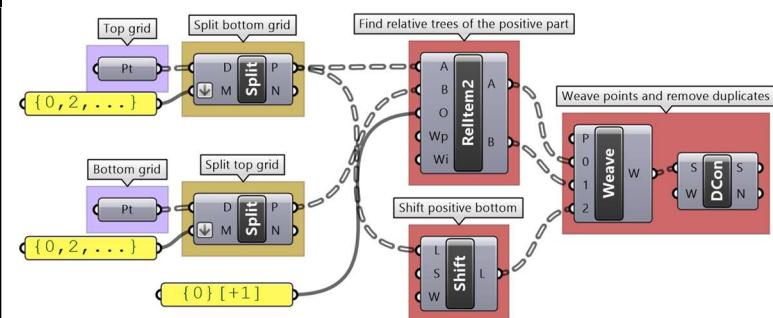
#### Grasshopper implementation

Use the **Split Tree** component to generate positive and negative trees for both bottom and top grids. Use  $\{0,2,\dots\}$  split mask.

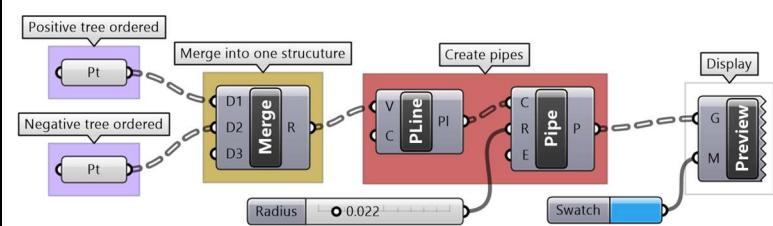
Use **RelativeItems2** to create A and B trees, use  $\{0\}[+1]$  relative mask.

Use **Shift** to create the C tree.

Weave data together and remove duplicates.

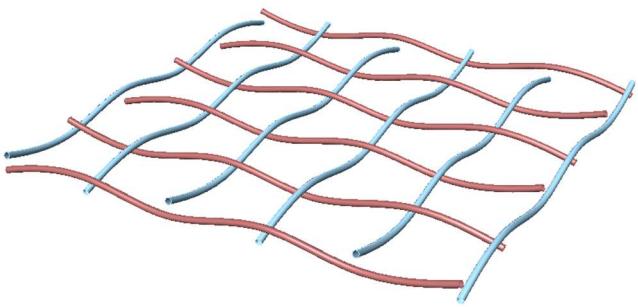


**Merge** ordered positive and negative trees to generate geometry using **Polyline** and **Pipe** components.



### 3\_7\_4: Weaving tutorial

Create flat weaved threads using a rectangular grid as an initial input. Set your desired density and size. Bonus: Make the weaving go along any surface



### Algorithm analysis

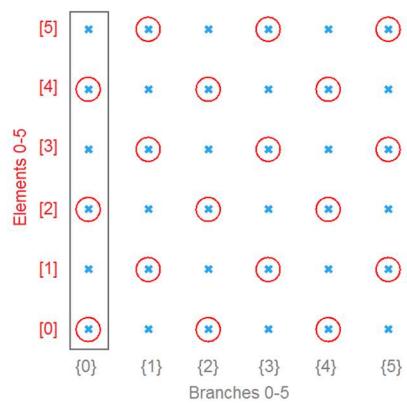
The input is a planar square grid with vertical branches. For vertical threads:

Split the grid into two parts alternating elements in each branch.

Move the first part up, and the second down, then recombine the parts into one set

Draw a curve through the points in each branch.

Flip the grid, then repeat to create horizontal curves

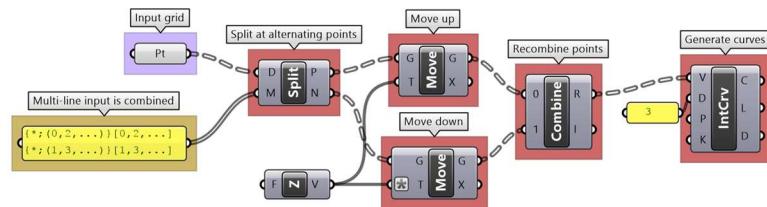


### Grasshopper implementation

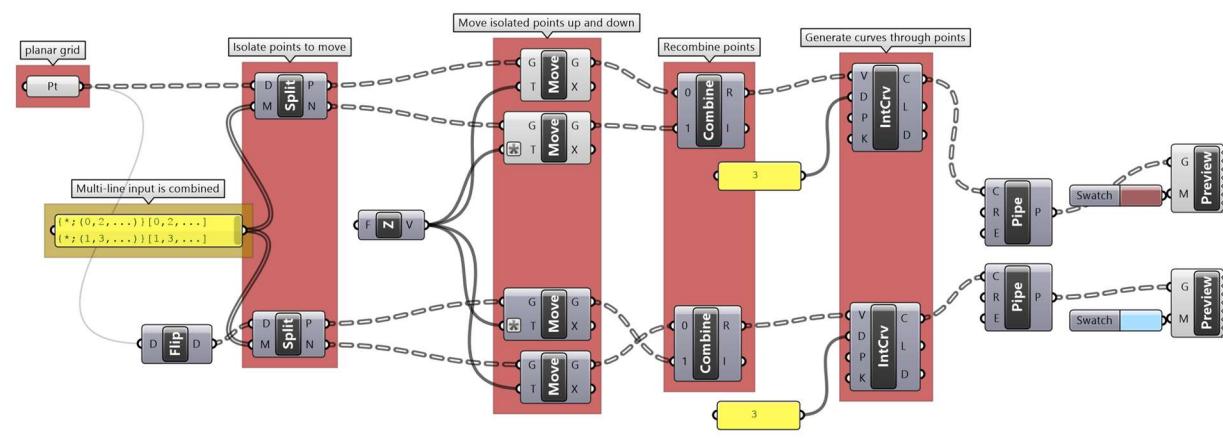
Use **Split** tree to separate alternating points and move up and down

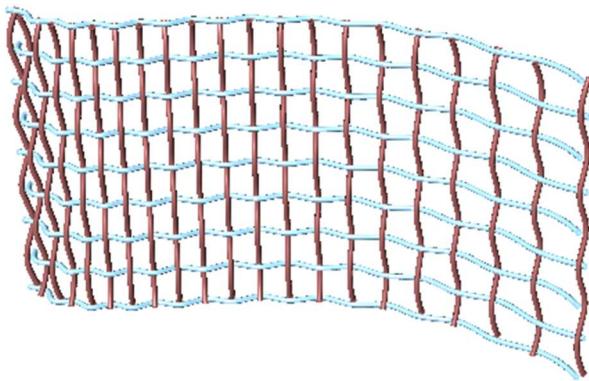
**Combine** points and use **IntCrv** to interpolate through points of each branch

**Flip** the tree, and repeat **Split**, **Combine** and **IntCrv** to create curves in the other direction



### The full Grasshopper solution





### Bonus solution

Instead of using the Z-Axis to move points up and down, use the surface normal direction at each point  
**Note:** Make sure the data structure of normals and points match

