# A comparison between functional and object oriented programming approaches in JavaScript

Kim Svensson Sand Tord Eliasson

May 3, 2017

# Contents

# 1 Abstract

## 1.1 Keywords

# 2 List of figures

# 3 List of tables

# 4 List of symbols

# 5 Abbreviations

ES6 = ECMAScript 2015

FP = Functional Programming

OOP = Object-Oriented Programming

# 6 Introduction

Programming paradigms is a way to classify a certain way of programming [1]. Each paradigm has its own set of rules for how the code should be written. Some programming languages make it possible to write code according to one paradigm, while others make it possible to write according to multiple paradigms.

In the 1950's and 60's multiple high-level languages were developed, that today's programming languages are based upon [2]. These were imperative languages such as Fortran and Algol, functional languages such as Lisp and object-oriented languages such as Simula. Originally Lisp had performance issues and has not been used a lot commercially, but versions of it are still used today. However it is mostly used an academic context. In the 70's C was introduced and quickly established itself thanks to it's ability to access low-level functionality, compact syntax and it's good compiler. In the 80's the object oriented language C++ was developed. It was based on C and and took a lot of inspiration from Simula. Java, that is based on C++ was introduced in the 90's. Since imperative languages gained the upper hand in the 60's and 70's new languages were based on those and continued being the most used.

Object-oriented or imperative languages are dominating today with languages such as C, C++, Java and C#. However, functional languages seem to be of interest in the industry again for handling big data and concurrency with languages such as Scala, Erlang or Haskell [3, 4, 5]. There is a performance difference where functional languages tend to be slower, but there are also positive aspects that could compensate for those, such as memory usage and less code [6, 7]. Some are also arguing that a more functional approach will give you more readable code, code that is easier to maintain and easier to test, and that learning it will give you better experience as a programmer [3, 8].

We will compare two programming paradigms, object-oriented programming and functional programming. To compare these we have chosen four different algorithms which we each will implement twice. Once according to the object-oriented paradigm and once according to the functional paradigm. The different algorithms we will implement are tree search algorithms, shellsort algorithm, tower of hanoi algorithm and Dijkstra's algorithm. The comparisons we will make between the algorithms are runtime, memory usage, time of development and lines of code. All algorithms will be written in JavaScript.

## 6.1 Research Questions

- Will functional versus object-oriented approaches in JavaScript have an impact on performance, such as runtime and memory usage?
- Can programmers with an object oriented background decrease development time by using practises from functional programming?

With the comparison of programming paradigms, the thing that have to be tested is each of the paradigms performance. Because we are writing both paradigms in the same language, we are able to compare just the the paradigms, without having to compare languages and compilers.

We both have studied at BTH, where we learned about programming only according to OOP. Therefore it would be interesting to see how our programming will change, by learning FP. This could give teachers a hum about introducing functional programming to give students a chance to use different paradigms depending on their needs, instead of always using OOP.

We both came from a background of object-oriented programming, therefore the question how this will affect our programming will be very interesting.

Should we write some kind of hypothesis here?

# 7 Background and Related work

Here we will describe the functional programming paradigm, FP, and also the object oriented programming paradigm, OOP. We will also describe JavaScript and it's support for functional and object-oriented methods. The related work described are articles that further explains the positive aspects of functional programming or studies similar to this one.

## 7.1 Functional programming

Functional programming, FP, is a programming paradigm that at its core is based on lambda-calculus [2]. Programs are constructed using functions and by avoiding changing the state. By not modifying the state side effects are avoided. Computation is done by changing the environment, rewriting the functions, rather than changing variables. Multiple functions can be composed into larger and more complex functions, and should be reduced to its simplest state following mathematical rules. The following are concepts used in FP.

**Lazy evaluation** - Lazy evaluation is when a value is not calculated until it is needed [6].

**Static type checking** - Pure functional languages usually have static type checking [2]. This means that variables are of a certain type, for example int or char. In such languages it is not allowed to use functions with the wrong types. So if there is for example a function taking an int as parameter it is illegal to call that function with a String as its parameter. In dynamic type checked-languages this would be legal, since variables are not specified as types, and might cause an unexpected error. For example in JavaScript:

```
1 //JavaScript has dynamic type checking
2 function double(nr) {
3 return nr * 2;
4 }
5
6 var word = "string";
7 double(word); //Will return NaN but still continue
      running
```

A similar program in golang [9], that is a statically type checked language:

```
1 func main() {
2   var word string = "string"
3   fmt.Printf("%d", double(word)) //Error: cannot use
      word (type string) as type int in argument to double
```

```
}

func double(nr int) int {
  return nr * 2
}
```

This will give an error when compiling.

There are however pure functional languages with dynamic type checking, such as Clean [10]. There are also languages that are mostly functional with dynamic type checking, such as Common Lisp [11] or Scheme [12] [13].

**Side effects** - Side effects are for example changing a variable or any interaction outside the function [3]. This is avoided in functional programming since it may result in incorrect and unexpected behaviour.

**Pure functions** - A pure functions always returns the same result, given the same input, and does not have side effects [3]. See the following example:

```
var sum = 0;

//Impure
function add(a, b) {
  sum = a + b;
}

//Pure
function add(a, b) {
  var tmp = a + b;
  return tmp;
}
```

**Higher order functions** - Higher order functions are an important concept in functional programming [6]. First class functions mean that functions are treated as values, which means that they can be stored in variables, stored in arrays or created if needed. A higher order function is a first class function that either takes a function as a parameter, returns a function as a result or both. This makes it possible to compose larger and more complex functions.

```
function add(a, b) {
  return a + b;
}

//Functions can be placed in variables.
var thisFunc = add;

//And also put in other variables.
var sameFunc = thisFunc;
sameFunc(1, 2); //Will give the output 3.
```

```
11
12 //Functions can also be used as parameters or return
       values.
13 function applyFunc(f, a, b) {
14   return f(a, b);
15 }
16
17 applyFunc(function(a, b) {
18   return a * b;
19 }, 3, 2); //Will give output of 6
20
21 //Returns a function that returns a string
22 function getStringCreator(category, unit) {
23   return function(value) {
24     return category + ': ' + value + unit;
25   }
26 }
27
28 var weightStringCreator = getStringCreator('Weight', 'kg
       ');
29 weightStringCreator(5); //Outputs "Weight: 5kg"
```

**Recursion** - Recursive functions are functions that call themselves and are used as loops [6]. It is important in functional programming since it can hide mutable state and also implement laziness. In functional programming recursion is used rather than loops.

```
1  //Adds all numbers from start to end with a loop
2  function iterativeAdd(start, end) {
3    var sum = 0;
4    while(start <= end) {
5      sum += start;
6      start++;
7    }
8
9    return sum;
10 }
11
12 //Adds all numbers from start to end recursively
13 function recursiveAdd(start, end) {
14   if (start == end) {
15     return end;
16   }
17   else {
18     return start + recursiveAdd(start + 1, end);
19   }
20 }
21 iterativeAdd(1, 6); //Outputs 21
22 recursiveAdd(1, 6); //Also outputs 21
```

**Currying** - Currying means a function can be called with fewer arguments than it expects and it will return a function that takes the remaining arguments [3].

```
1  //Returns a new function that takes the remaining
       argument or the sum of a and b if both are provided.
2  function addWithCurrying(a, b) {
3    if(b) {
4      return a + b;
5    }
6    else {
7      return function(b) {
8        return a + b;
9      }
10   }
11 }
12
13 var curryAdd = addWithCurrying(4);
14 curryAdd(6); //Outputs 10
15 addWithCurrying(4, 6); //Also outputs 10
```

**Immutable data structures** - In functional programming mutations, that are side effects, are avoided [6]. Hidden side effects can result in a chains of unpredicted behaviour in large systems. Instead of mutating the data itself a local copy is mutated and returned as a result, as seen in the pure functions example.

## 7.2 Object-oriented programming

Object-oriented programming is a programming paradigm which is built around "objects". These objects may contain data, in form of fields, often referred as attributes and code in form of procedures, often referred to as methods [14].

These objects are created by the programmer to represent something with the help of its attributes and methods. What kind of variables and functions an object should contain is defined in a class, which works like a blueprint for the object. For example, a class can represent an employee. An employee has the attributes assignment and salary. The employee also has a method for doing work. Then the employee has to work somewhere, so we create another object for a company where our employee can work. The company has the attributes income and number of employees. It also has methods to hire employees and fire employees. Since there are more employees who works at this company, we can add more employees by creating new objects of the information in employee class.

By building objects together like this, you can build programs by the object oriented paradigm.

**Class** - A class is a model for a set of objects, which the object oriented paradigm is built around [2]. The class establishes what the object will contain, for example variables and functions, and signatures and visibility of these. To create a object of any kind, a class must be present.

```
1  //Creates a class Animal with the properties name and
       age and a function for logging the properties to the
        screen
2  class Animal {
3    constructor (name, age) {
4      this.name = name;
5      this.age = age;
6    }
7
8    logAnimal() {
9      console.log('Name: ' + this.name + '\nAge: ' + this.
       age);
10   }
11 }
12
13 var animal = new Animal('Buster', '9');
14 animal.logAnimal();
15 //Outputs Name: Buster
16 //Age: 9
```

**Object** - An object is a capsule that contains the variables and functions established in the class. While objects created from the same class contains the same variables and functions, since the information it contains is handled specifically for every object, the information in the class can vary much. All data and functions can be accessible from outside the object.

**Inheritance** - When an object acquires all properties and functionality of another object [15]. it is called inheritance. This provides code reusability.

```
1  //Based upon the Animal class with an added property
       race
2  //The original logAnimal() is overidden so the new
       property is also logged to the screen.
3  class Dog extends Animal {
4    constructor(name, age, race) {
5      super(name, age);
6      this.race = race;
7    }
8
9    logAnimal() {
10     super.logAnimal();
11     console.log('Race: ' + this.race);
12   }
13 }
14
```

```
15 var dog = new Dog('Buster', '9', 'Shitzu');
16 dog.logAnimal();
17 //Outputs Name: Buster
18 //Age: 9
19 //Race: Shitzu
```

**Encapsulation** - Encapsulation can be used to refer to two different things [2, 15]. A mechanism to restrict direct access to some object components and the language construct that facilitates the bundling of data with methods. The access part is done by making the different parts of an object public or private and the bundling is made with objects.

**Polymorphism** - When a task is used in different ways it is called polymorphism [2, 15]. This is achieved with the help of overloading and overriding.

**Overloading** - Refers to creating a function with the same name as another function, often very similar, with either different types of variables in the parameters or different number of parameters [15].

**Overriding** - Refers to overwrite a function written in a superclass to make it do something else than first intended, without changing the superclass [15].

## 7.3 JavaScript

JavaScript is a prototype based language with first class functions that is dynamically type checked. This makes it a multi-paradigm language with basic support for object-oriented, imperative and functional paradigms. There are multiple libraries to tweak JavaScript with certain functions or better support for certain paradigms, for example Undescore.js [16], for functonal programming, or flow [17], for static type checking. There are also a lot of languages available that are transpiled into JavaScript, for example TypeScript [18]. JavaScript can be run in most browsers and also in servers and is implemented to follow the ECMA standards [19]. It has a garbage collection system for handling memory [20]. The garbage collection works so that objects or variables that are unreachable are removed from memory.

Objects in JavaScript are treated as references. When an object is initiated to a variable it is created and placed in memory and the variable is given a reference to it. This means that if another variable is assigned this object it gets a copy of the reference to it, rather than a copy of the object itself. Arrays are objects in JavaScript and will be treated the same way.

```
1 //JavaScript Object handling
2 var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
3 var obj = {
```

```
4     text: 'Some text'
5   }
6
7   var arrayRef = array;
8   var objRef = obj;
9
10  // Gets references to original array and object.
11  console.log(arrayRef); // [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
12  console.log(objRef); // { text: 'Some text' }
13
14  // Change stuff
15  arrayRef.pop();
16  objRef.text = 'Some other text';
17
18  // Mutating new variables mutates original objects.
19  console.log(array); // [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
20  console.log(obj); // { text: 'Some other text' }
```

### 7.3.1 Functional programming

JavaScript is not a functional language, but it is possible to write functional code with it [3]. There are also libraries that makes functional programming in JavaScript easier, such as Underscore.js [**?**, 6]. However, we will use ECMA 2015, ES6, that already provide many of the functions provided by Underscore.js. This is also to use the same environment for our different implementations in this experiment.

In JavaScript it is possible to treat functions as any other variable, pass them as function parameters or store them in arrays, so called first class functions [3]. There are also higher order functions such as map(), filter() and reduce() that might replace loops [21]:

**map()** - Calls a provided function on every element in an array and returns an array with the outputs [22].

**filter()** - Returns a new array with all the elements that passes a test in a provided function.

**reduce()** - Reduces a array to a single value.

However there is no automatic currying or immutable data structures in JavaScript. JavaScript is also dynamically type checked. All of these can be added to JavaScript with libraries.

There is a call stack limit in JavaScript that varies depending on the runtime environment. This limits the number of function calls that can be made, which

also limits recursion. In ES6 there is tail call optimization that makes it possible to make certain function calls without adding to the call stack, that would allow for better recursion. However this is currently not implemented for most servers and browsers [19].

### 7.3.2 Object oriented programming

JavaScript has always had support for OOP. But with ES6, OOP starts to look more like classical OOP languages like for example Java [23].

JavaScript do not fully support encapsulation, since you can't make variables and functions private or public. Otherwise there is full support for OOP in JavaScript.

As a workaround to this, many programmers add an underscore to the variable or function before the name of it. This underscore symbolizes that it should be handled as a private instead of a public.

## 7.4 Related work

In "Curse of the excluded middle" [8] Erik Meijer argues with multiple examples that the industry should not focus on a combination between functional and objected oriented methods to counter handling big data with concurrency and parallelism. He concludes that it is not good enough to avoid side effects in imperative or object oriented languages. It is also not good enough to try to ignore side effects in pure functional languages. Instead he thinks that people should either accept side effects or think more seriously about using the functional paradigm.

In "Functional at scale" [4] by Marius Eriksen he is explaining why Twitter uses methods from functional programming to handle concurrent events that arises in large distributed systems in cloud environments. In functional programming it is possible build complex parts out of simple building blocks, thus making systems more modular. He concludes that the functional paradigm has multiple tools for handling the complexity present in modern software.

Eriksson and Ärleryd are looking at how to use functional practises, such as immutable data structures, pure functions and currying, when developing front end applications by taking inspiration from Elm in their master's thesis [5]. They have researched each practise in Elm to see if it is possible to use these practises in JavaScript together with tools and libraries. Their conclusion was that it is possible to replicate functional practises from Elm in JavaScript, but that they prefer working with Elm. In JavaScript multiple libraries had to be

used to use the same practises. They also concluded that even though functional programming is not widely used within the industry, functional practises can still be used in all projects.

In "Improving Testability and Reuse by Transitioning to Functional Programming" [24], Benton and Radziwill state that functional programming is better suited for test driven development (TDD) and concludes that a shift toward the functional paradigm benefits reuse and testability of cloud-based applications.

Alic, Omanovic and Giedrimas has made a comparative analysis of functional and object-oriented programming languages [7]. They have compared four languages, C#, F#, Java and Haskell based on performance, runtime and memory usage. Their conclusion is that Java is the fastest while Haskell uses much less memory, and that programming paradigms should be combined to increase execution efficiency.

Dobre and Xhafa writes in "Parallel Programming Paradigms and Frameworks in Big Data Era" that we now are in a big data era [25]. They also review different frameworks, programming paradigms and more in a big data perspective. Around paradigms they state, "functional programming is actually considered today to be the most prominent programming paradigm, as it allows actually more flexibility in defining big data distributed processing workflows."

In "Comparing programming paradigms: an evaluation of functional and object-oriented programs" R. Harrison, et al. compares the quality of code in functional and object oriented programming [26]. To compare these, they use C++ and SML. While their discussion states that they would probably use OOP, since C++ is a better language, the standard list functions were of great help, the debugging was better and reusability was much higher in SML.

Guido Salvaneschi, et al. have conducted an empirical evaluation of the impact of Reactive Programming (RP), with functional programming concepts, on program comprehension [27]. Their experiment involved 127 subjects and the results suggests that RP is better for program comprehension when compared with OOP. They conclude that with RP the subjects produced more correct code without the need of more time, and also that the comprehension of RP programs is less tied to programming skills.

In "Using Functional Programming within an Industrial Product Group: Perspectives and Perceptions" [28] David Scott et al. presents a case-study of using FP in the multiparadigm language OCaml in a large product development team. They found that the team's project was a success even though there were some drawbacks to using OCaml, such as lack of tool support. The engineers believed that OCaml enabled them to be more productive than if they would have used one of the mainstream languages, such as C++ or Python.

# 8 Method

In this experiment each person will implement four different algorithms, once with FP and once with OOP. We have chosen algorithms that are well known, so that our focus will be on the different implementations, rather than on how the algorithms work.

## 8.1 Algorithms

We will implement tree search algorithms, the shellsort algorithm, the tower of hanoi-algorithm and Dijkstra's algorithm. In the search tree we will implement tree traversal, which is a recursive algorithm, as is the tower of hanoi algorithm. These algorithms fit well for FP since it uses recursion. The implementation of the binary tree structure can however be implemented by using classes and objects, that is used in OOP. Shellsort uses state and iteration which is used in OOP and avoided in FP. Dijkstra's algorithm also uses state and iteration, and the graph can also be implemented using classes and objects.

We chose algorithms that use both recursion and iteration to not give advantage to any paradigm. When implementing these algorithms and data structures we can also make use of OOP methods, such as classes and objects. The algorithms' purpose are also different to avoid for example implementing four different sorting algorithms.

### 8.1.1 Tree search algorithms

A binary tree is a tree consisting of nodes, where a node can have a maximum of two children [29]. These children can be described as the left and the right subtree and the parent is the root. The property that differs a binary search tree from a standard binary tree, is the order of the nodes. In a binary tree the order can be undecided, but in a search tree the nodes are stored in an order based on some property. For example in our binary search tree the left subtree of a root will contain smaller numbers than the root, and the right subtree will contain larger numbers, see example in figure 1. Using tree traversal it is possible to find certain nodes or get a list of sorted objects. Our binary tree will contain random numbers and the functions:

**findNode(comparable, rootNode)** - Finds a specific key in the tree.

**inOrderTraversal(root)** - Returns an array of all numbers in the tree, sorted smallest to biggest.

**insert(comparable, rootNode)** - Inserts number into the tree. Returns true or false depending on success. If comparable is already in the tree, false should be returned. Note that in the functional implementation this function will return the resulting tree instead of mutating the tree and returning true or false.

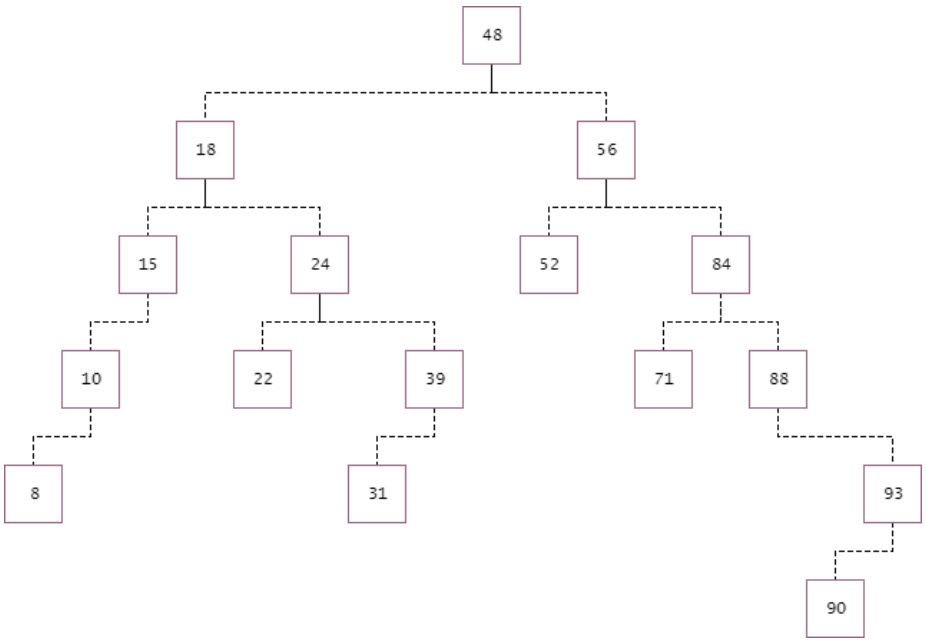


Figure 1: Binary tree example

**Algorithm descriptions:**

**algorithm findNode(comparable, root)**

```
1  if root is undefined then
2    return null
3  else if comparable is equal to comparable in root then
4    return root
5  else if comparable is larger than comparable in root
       then
6    return findNode(comparable, rightSubtree in root)
7  else if comparable is smaller than comparable in root
       then then
8    return findNode(comparable, leftSubtree in root)
9  end if
```

**algorithm inOrderTraversal(root)**

```
set array to empty array
if comparable in root is undefined then
  return array
else then
  add inOrderTraversal(leftSubtree in root) to end of
    array
  add comparable in root at end of array
  add inOrderTraversal(rightSubtree in root) at end of
    array
  return array
end if
```

Running this in the tree in figure 1 would return the array [8, 10, 15, 18, 24, 22, 31, 39, 48, 52, 56, 71, 84, 88, 90, 93].

**algorithm insert(comparable, root)**

```
if comparable is equal to comparable in root then
    return false
else if comparable is larger than comparable in root
    then
  if rightSubtree in root is undefined
    create newNode
    set comparable in newNode to comparable
    set rightSubtree of root to newNode
    return true
  else then
      return insert(comparable, rightSubtree in root)
  end if
else if comparable is smaller than comparable in root
    then
  if leftSubtree in root is undefined then
create newNode
    set comparable in newNode to comparable
    set leftSubtree in root to newNode
    return true
  else then
    return insert(comparable, leftSubtree in root)
  end if
end if
```

### 8.1.2 Shellsort algorithm

The shellsort algorithm is named after its creator Donald Shell [29]. It was one of the first algorithms to break the quadratic time barrier. The algorithm sorts by sorting items using insertion sort with a gap. For each run the gap is decreased until the gap is 1 and the items are sorted. How the gap is decreased is

decided with a gap sequence. Different gap sequences gives shellsort a different worst-case running time.

We will use one of Sedgewick's gap sequences that has one of the fastest worst-case running times $O(n^3/4)$.

The sequence is

$\{1, 5, 19, 41, 109\}$, where the terms are of the form

$4^k - 3 * 2^k + 1$, or

$9 * 4^k - 9 * 2^k + 1$

In our implementation the sequence is put in an array and not calculated during execution, since we do not want to get different results between our implementation because of the calculation of the gap sequence.

This algorithm will sort an array filled with randomized numbers. Our implementation uses this algorithm in a function:

**shellsort(array)** - Takes an unsorted array as an input and returns a sorted copy of the array.

**Algorithm description:**

**algorithm algorithm shellsort(array of size n)**

```
1  set sortedArray to array
2  set gapSequence to Sedgewicks gap sequence
3  set currentGapIndex to 0
4  set currentGap to the largest gap i gapSequence where
       gap is smaller than n divided by 2
5  set currentGapIndex to the index of currentGap in
       gapSequence
6  while currentGap is larger than 0 do
7    for i = currentGap to n
8      set currentValue to array[i]
9      set currentIndex to i
10     while currentIndex - currentGap is larger or equal
       to 0 and sortedArray[currentIndex - currentGap] is
       larger than currentValue do
11       set sortedArray[currentIndex] to sortedArray[
       currentIndex - currentGap]
12       set currentIndex to currentIndex - currentGap
13     end while
14     set sortedArray[currentIndex] to currentValue
15   end for
16   set currentGapIndex to currentGapIndex - 1
```

```
17    set currentGap to gapSequence[currentGapIndex]
18  end while
19  return sortedArray
```

### 8.1.3 The Tower of Hanoi algorithm

The Tower of Hanoi is a game invented by mathematician douard Lucas in 1883 [30]. The game consists of three pegs and a number of disks stacked in decreasing order on one of the pegs, see figure 2. The goal is to move the tower from one peg to another by moving one disk at a the time to one of the other pegs. A disk can not be placed on a peg on top a smaller disk.

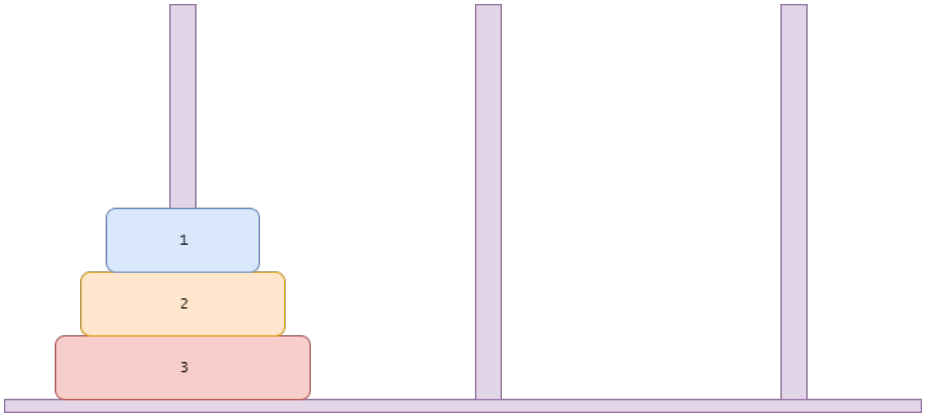


Figure 2: Tower of hanoi image

This algorithm will move a tower from one peg to another in the function:

**hanoi(tower, start, dest, aux)** - Moves tower from start to dest following the rules of the tower of hanoi problem. Returns the start, dest and aux pegs with the repositioned tower.

**Algorithm description:**

**algorithm hanoi(tower, start, dest, aux)**

```
1  (*Pseudo code based on that tower is the number of the
       largest disk, where 1 is the smallest disk in the
       tower.*)
2  if tower is equal to 1
3    move tower from start to dest
4  else
```

```
5     hanoi(tower - 1, start, aux, dest)
6     move tower from start to dest
7     hanoi(tower - 1, aux, dest, start)
8   end if
```

### 8.1.4 Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest path in a graph consisting of a number of nodes connected by edges [29], where the weight of the edges is known, see figure 3. The algorithm will find the shortest path from the start node to the end node in a graph. It is initiated by:

1. setting the distance to the start node to 0.
2. setting the distance to all other nodes to infinity.
3. mark all nodes as unvisited.

The algorithm will then find the shortest path using the following steps:

1. Set current node to the node with the smallest distance that has not already been visited.
2. For all neighbors to current node that has not already been visited, check if their distance is smaller than the distance of current node + the distance to the neighbor. If so, update the distance of the neighbor.
3. Mark current node as visited.
4. Repeat until all nodes have been visited or the end node has been visited.

See the result in figure 4.

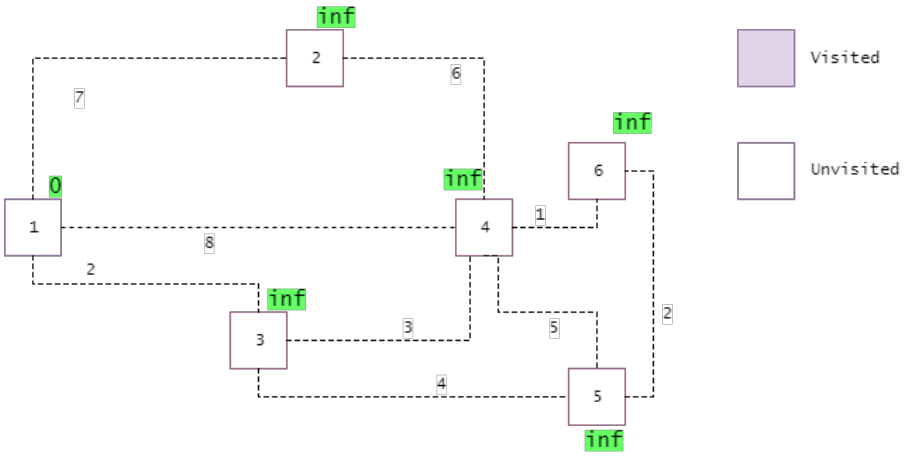


Figure 3: After initiation of Dijkstra's algorithm

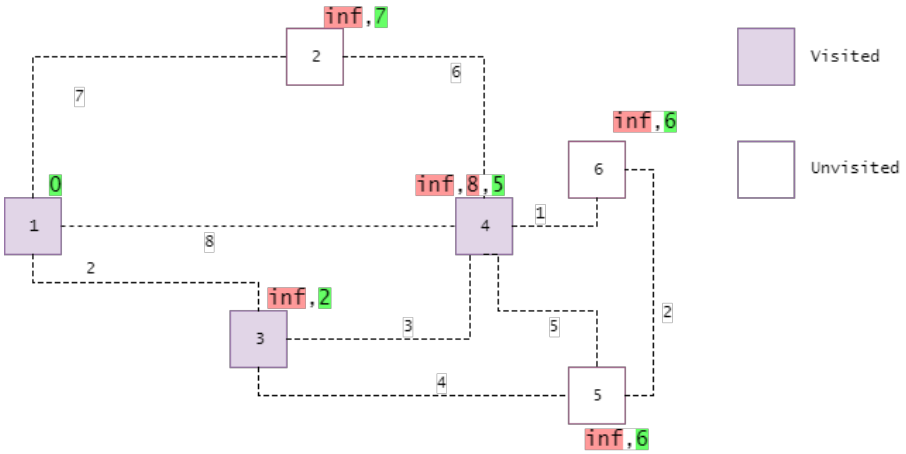


Figure 4: After three nodes have been visited with Dijkstra's algorithm

This algorithm will be used in a function:

**dijkstras(graph, startNode, endNode)** - That takes graph, startNode and endNode and returns an array with the shortest path from startNode to destNode.

**Algorithm description:**

**algorithm dijkstras(graph, startNode, endNode)**

```
1  for each node in graph
2    set dist[node] to infinity
3    set path[node] to undefined
4  end for
5  set dist[startNode] to 0
6  set unvisitedNodes to nodes in graph
7  while unvisitedNodes is not empty or endNode is not in
       unvisitedNodes do
8    set current to node where dist[node] is smallest and
       node is in unvisitedNodes
9    remove current from unvisitedNodes
10     for each neighbor of current where neighbor is in
       unvisited do
11         set temp to dist[current] + weight of edge in
       graph, where edge is from current to neighbor
12     if temp is smaller than dist[neighbor] then
13       set dist[neighbor] to temp
14       set path[neighbor] to path[current] + current
15     end if
16   end for
17 end while
18 return path[endNode]
```

## 8.2 Environment

We will implement the algorithms using Atom [31], a text editor, and compile and run our tests and our code with Node.js [32]. Tail call optimization is not available in Node.js, see 7.3.1. To manage our dependencies we are using npm [33]. For automation we are using Grunt [34]. Mention github and google docs?

Since node.js does not support ES6-modules we are using babel to transpile our ES6-modules to node-modules that are supported. We are doing this since we have prior experience with ES6 modules and not with node-modules. Since node.js support almost everything else in ES6 we will only be transpiling the modules, leaving our code in an ES6 standard.

To measure memory and run time we are using node.js process, that is a global and therefore always available within node.js applications [32]. We are using the following functions:

**process.hrtime(time)** - Returns the current high resolution time in a [seconds, nanoseconds] tuple Array. If the optional time-parameter, an earlier hrtime, is used, it will return the difference between that time and the current time.

**process.memoryUsage()** - Returns an object describing the memory usage

of the Node.js process measured in bytes.

## 8.3 Testing

Our code will be tested through code reviews and unit testing.

### 8.3.1 Code reviews

We will review each other's implementations and our implementations will also be reviewed by a third party. The reviews should help us to find bugs and also to confirm that we have used FP or OOP methods according to our guidelines, see 8.5.1.

### 8.3.2 Unit testing

Unit testing will be done using the JavaScript libraries Mocha [35] for writing tests, Chai [36] for evaluating expressions, and Karma [37] for automated tests.

Our implementations will have to pass the tests in appendix A to be accepted as done.

## 8.4 Implementation

We will proceed from a template with the following structure:

```
|---dist
|   \---<Files generated from babel>
|---src
|   |---js
|        |---<function and class files>
|   \---index.js
|---es6-test
|   \---<Test files>
|---test
|   \---<Test files generated from babel>
| Gruntfile.js
| package.json
```

To run tests use the command:

**npm test** - Will transpile the test files in the es6-test-folder, place the generated files in the test-folder and run the tests.

To run the algorithm use the command:

**npm start** - Will transpile the JavaScript-files in src-folder, place the generated files in the dist-folder and run index.js in the dist-folder.

For an implementation of an algorithm to be considered done the following has to be implemented and provided:

- Tests for the algorithm that are defined in appendix A placed in (projectName)/es6-test
- A complete implementation of the algorithm that has passed the tests placed in (projectName)/src/js
- A measurement implementation that must run the algorithm and measure time and memory usage using node.js process. The implementation must have a function for creating randomized data of a certain size according to table 1
- A timelog describing how long the implementation took.

## 8.5 Measurements

We will measure runtime, memory usage, code length and the development time of each implementation. The measurements are done on the algorithms described in this chapter and the algorithms will run on randomized data, described in table 1. We are not measuring the initiation of the randomized data, except for the tree search algorithms that include an insert-function. The algorithms are measured with three different sizes of data, described in table 1 to get more accurate results. We have chosen measurement sizes such as to avoid exceeding the call stack limit for our functional implementations.

| | |
|---|---|
| Algorithm | Binary search tree algorithms |
| Description | Should measure insert(), inOrderTraversal() and findNode(), where findNode() is run three times with a random value between 1 and 10 * n as parameter. |
| Measurement data | A binary tree with n nodes filled with randomised values between 1 and 10 * n. |
| Measurement sizes | n = 1000, n = 5000 and n = 10000. |
| Algorithm | Shellsort algorithm |
| Description | Should measure the shellsort-function. |
| Measurement data | An array of length n filled with randomised values between 1 and 2 * n |
| Measurement sizes | n = 1000, n = 3000 and n = 6000. |
| Algorithm | Tower of Hanoi algorithm |
| Description | Should measure the hanoi-algorithm where a tower is moved from one peg to another. |
| Measurement data | A tower of n disks. |
| Measurement sizes | n = 10, n = 20 and n = 25. |
| Algorithm | Dijkstra's algorithm |
| Description | Should measure Dijkstra's algorithm from the first node in the randomised graph to the last node in the randomised graph. |
| Measurement data | A graph with n nodes where each node is connected to the last two nodes, if those nodes exist. The edges between the nodes should be of a randomised weight between 1 and 100. |
| Measurement sizes | n = 10, n = 15 and n = 20. |

Table 1: Table describing measurements to be implemented

#### 8.5.1 Guidelines for implementing in the different paradigms

Functional programming guidelines:

- Do not use classes
- Treat functions as variables
- Compose functions to build programs
- Use already provided pure functions when possible
- Write pure functions
- Do not change variables
- Use recursion instead of iteration

Object-Oriented Programming guidelines:

- Use classes and objects to build programs
- Use inheritance when possible
- Use encapsulation
- Each class should have only one job
- Prefer iteration over recursion

# 9 Result

Our implementations have been reviewed by each other and also by Emil Folino that is a lecturer at BTH. The reviews worked well, but a few bugs still slipped through and had to be fixed. They were found when we noticed some strange results that we could not explain. The tests for Dijkstra's algorithm were not good enough for the functional implementation, so some manual testing had to be done.

Should we add our code here? Or should we add the code as appendices? Or should we link to github? Should we have a link to github somewhere anyway or is that not necessary?

### 9.0.1 Development time and lines of code

The total results for the development time were:

**Kim OOP** - 20 hours and 32 minutes.

**Kim FP** - 18 hours and 20 minutes.

**Tord OOP** - 24 hours and 8 minutes.

**Tord FP** - 26 hours and 13 minutes.

Kim spent less time on the functional implementations in total, while Tord spent less time on the object-oriented implementations. For Kim she had to spend more time on the OOP solution when implementing the recursive algorithms and vice versa. Tord's results are following this pattern for the tree search algorithms and the shellsort algorithm implementations. However, he spent significantly less time on the OOP tower of hanoi algorithm than the FP implementation, and spent about as much time for both dijkstra's algorithm implementations.

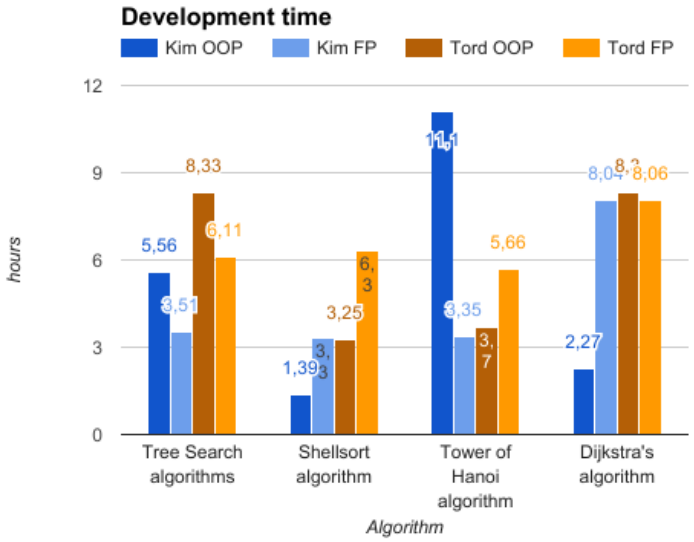


Figure 5: Development time for our implementations

The total results for the lines of code were:

**Kim OOP** - 930 lines of code.

**Kim FP** - 618 lines of code.

**Tord OOP** - 706 lines of code.

**Tord FP** - 530 lines of code.

For both Kim and Tord the functional implementations had less code in total and individually for all implementations except for shellsort. Here both OOP implementations had the least code. We can see for Kim's tower of hanoi algorithm that there is a large difference in code length between the OOP and the FP implementations.

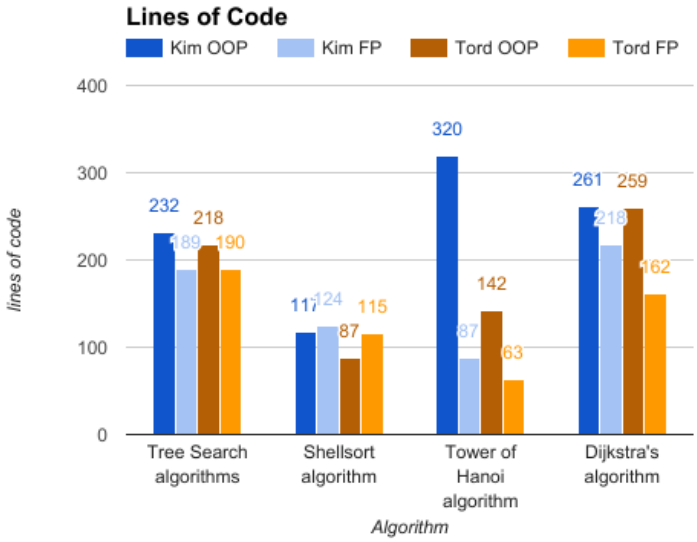


Figure 6: Lines of code in our implementations

### 9.0.2 Search Tree Algorithms

We can see that for Kim the functional implementation has a better runtime, but still uses a lot more memory than the object-oriented implementation. Tord's object-oriented implementation performs a lot better than his functional, both for memory usage and runtime. Tord's functional implementation is a lot slower and uses a lot more memory than all the other implementations.

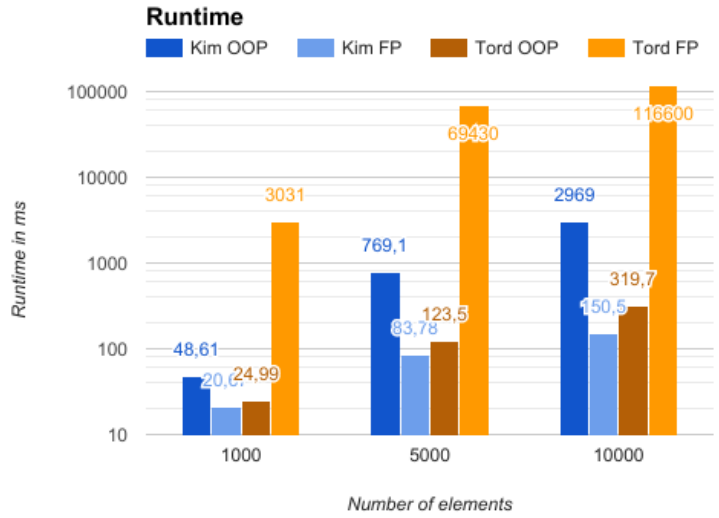


Figure 7: Runtime for tree search implementations.

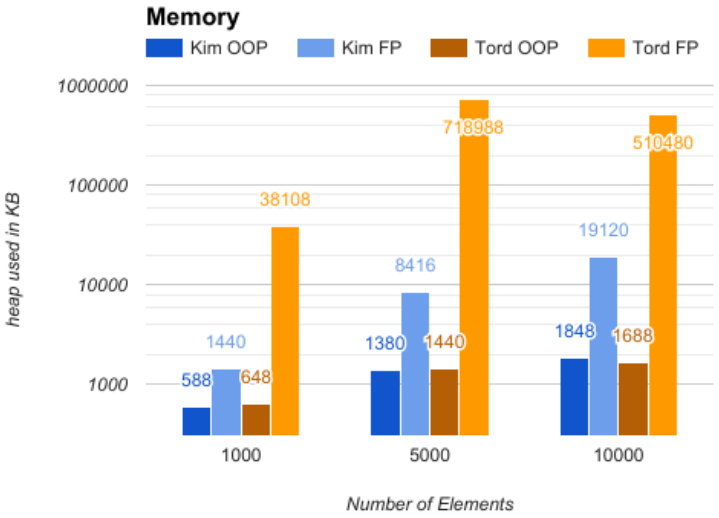


Figure 8: Memory usage for tree search implementations.

### 9.0.3 Shellsort

For shellsort Kim's and Tord's results are following the same trend. Both of their OOP implementations performed a lot better with faster runtimes and lower memory usage. We can also see that Kim's functional implementations is slower than Tord's, while Tord's is using more memory.

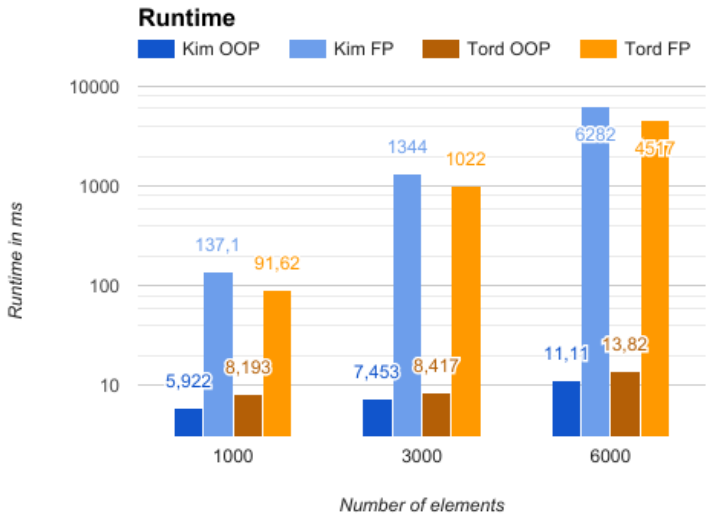


Figure 9: Runtime for shellsort implementations.

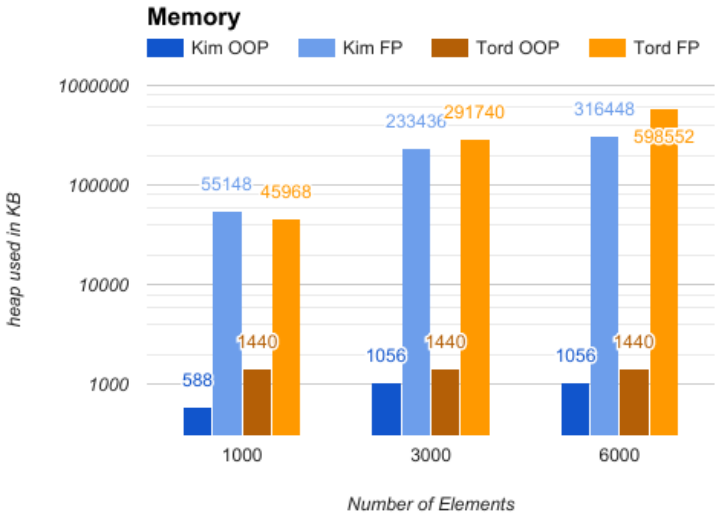


Figure 10: Memory usage for shellsort implementations.

#### 9.0.4 Tower of Hanoi

For the Tower of Hanoi-implementations Tord's OOP implementation could not handle 25 discs because it ran out of memory. His FP implementation was faster than his OOP implementations for small tower sizes, but was the slowest of all implementations for larger. Kim's functional implementation was both faster and used less memory than her object-oriented implementation. However, there was not as big of a difference in performance between the two compared to results for other implementations.

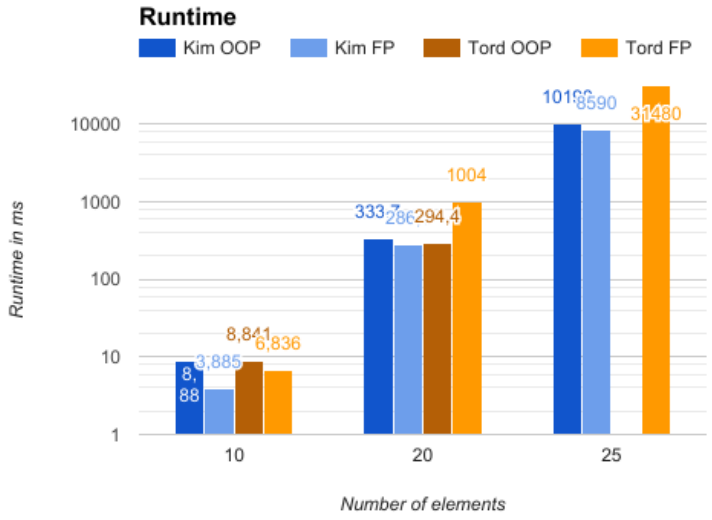


Figure 11: Runtime for hanoi implementations.

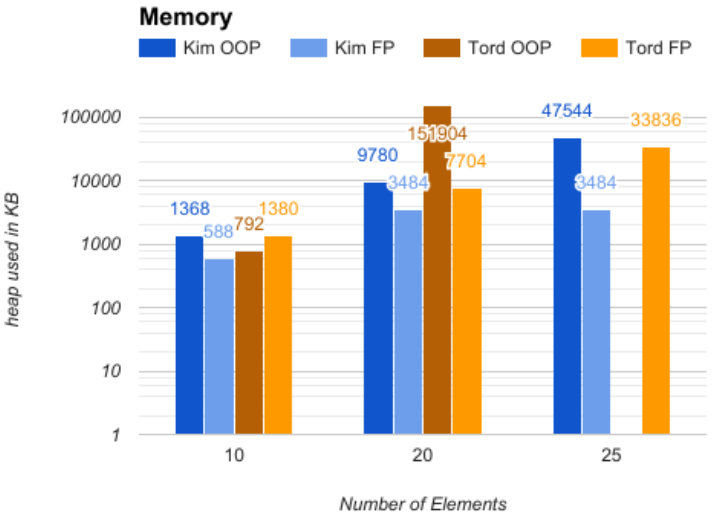


Figure 12: Memory usage for hanoi implementations.

### 9.0.5 Dijkstra's algorithm

Both Kim's and Tord's OOP implementations performed better than their FP implementations for Dijkstra's algorithm. Kim's FP implementation was a lot slower than all the others, while Tord's FP implementation only performed slightly worse than the OOP implementations.

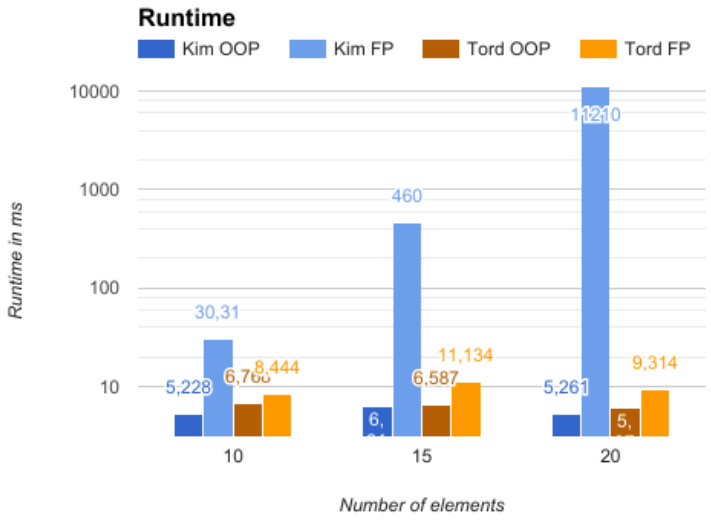


Figure 13: Runtime usage for Dijkstra's algorithm implementations.

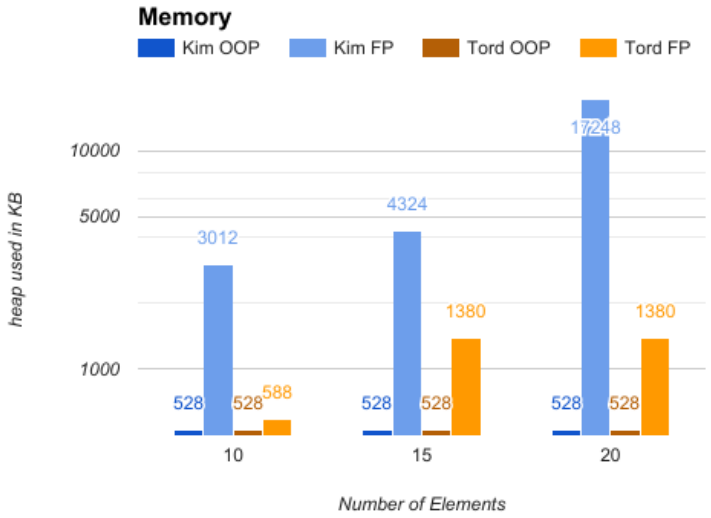


Figure 14: Memory usage for Dijkstra's algorithm implementations.

# 10 Analysis

Mostly the implementations are faster when implemented in their expected way. For example the tree search algorithms are faster recursively and the shellsort algorithm is faster iteratively. This is also clear when looking at the development time where the results suggest that we spent more time when implementing, for example, shellsort recursively or the tower of hanoi iteratively.

We expected the development time to be shorter for the object-oriented implementations, since we both have prior experience with it and since we have nearly no prior experience with functional programming. The results show that this was not necessarily so. Neither of the paradigms seem faster to implement, however the functional paradigm might have a slight upper hand since we both had no prior experience with it. Tord's results are partly deviant from the earlier statement that we spent less or more time implementing certain algorithms depending on if it's more suited for recursion or iteration. This is clear when looking at the results for the tower of hanoi algorithm and Dijkstra's algorithm. This can be explained by that he had some trouble with tower of hanoi and used the internet for help and also that he had to rewrite and complement both of the Dijkstra's algorithms.

Over all the functional solutions had less lines of code in total and individually, except for both of our shellsort implementations. However, shellsort is a short

algorithm where neither of us could add a lot of classes or methods. We both felt that this does not mean that the code is easier to read or faster to write. Rather that we simply did not implement classes with methods for the functional implementations. We both found that the lack of classes made the code messier to implement which is apparent when looking at our development time results.

For the runtime results we did expect the object-oriented implementations to be faster than the functional implementations based on [26] and our research about JavaScript, see section 7.3. Overall the functional implementations performed worse than the object-oriented implementations, even though it performed better for the recursive algorithms. We both implemented very slow implementations when using functional approaches, compared to when we implemented slow implementations according to OOP. This could however be because of JavaScript. Since JavaScript does not support immutable variables we have instead treated variables as immutable by copying. This seems to take a lot of time and also use a lot of memory because of JavaScript's garbage collection. When looking at our shellsort implementations we can see that if we try to remove some of our copying our runtimes decrease a lot. Optimizing functional solutions is an option, but that would take time and would have to be considered when developing functional JavaScript. Perhaps using one of the functional libraries available would solve this problem.

Since we only had two different test groups with one person in each, it was a bit hard to analyze our results when algorithms were hard to compare. For example when an algorithm was much slower than all the others. This might not be because of the paradigm itself, but rather because the programmer has made a mistake. However, it can still give us some results as to if it is easier to make mistakes in one paradigm, but that would only be valid for that particular language. We think that it would be possible to draw more conclusions if there were more test subjects taking part in this experiment.

# 11 Conclusion

Overall we both appreciated trying out functional programming, but we don't think that we will use pure functional programming or functional languages in the future if we don't have to. We will however use a lot of the approaches that we have been using in this experiment. In our experience we found that we had less bugs when avoiding side effects and the functions were modular and easy to combine. We also liked using classes since there was a layer of abstraction on top of JavaScript's native types. We think that combining the two and using recursion or iteration when it fits best would benefit both the performance, development time and code readability.

Based on our results different paradigms are good for different things and one

is not clearly better than the other. According to our research most languages perform well, and there are also a lot of multiparadigm languages out there. Based on our results it is best to use a combination of them, especially since functional programming and object-oriented programming are not opposites. To us it makes a lot of sense that a lot of the object-oriented languages, that are based on the imperative paradigm, are starting to implement support for functional approaches.

# 12 Future Work

In this experiment we have implemented the algorithms ourselves and compared our own work. It would be interesting to see a similar experiment with multiple test subjects that are not involved in the thesis work.

A lot of our sources argue that functional programming should be really good for handling data concurrently. Therefore an experiment to compare different paradigms from both a performance, development and maintainability perspective would be interesting.

We also did these implementations in JavaScript. It would be interesting to see this kind of comparison in another language that better support full functional or full OOP approaches. Perhaps comparing other paradigms and combined paradigms as well.

# References


[1] Programming paradigms. [Online]. Available: http://cs.lmu.edu/ ray/notes/paradigms/

[2] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*, ser. Undergraduate Topics in Computer Science. London: Springer, 2010.

[3] drboolean. (2017) Gitbook, proffesor frisby's mostly adequate guide to functional programming. [Online]. Available: https://drboolean.gitbooks.io/mostly-adequate-guide/content/

[4] M. Eriksen, "Functional at scale," *Queue*, vol. 14, no. 4, pp. 60:39–60:55, Aug. 2016. [Online]. Available: http://doi.acm.org.miman.bib.bth.se/10.1145/2984629.3001119

[5] N. Eriksson and C. Ärleryd, "Elmulating javascript," Master's thesis, Linkping University, Human-Centered systems, 2016.

[6] M. Fogus, *Functional JavaScript*, 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., may 2013.

[7] D. Alic, S. Omanovic, and V. Giedrimas, "Comparative analysis of functional and object-oriented programming," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2016, pp. 667–672. [Online]. Available: http://ieeexplore.ieee.org.miman.bib.bth.se/document/7522224/citations

[8] E. Meijer, "The curse of the excluded middle," *Commun. ACM*, vol. 57, no. 6, pp. 50–55, Jun. 2014. [Online]. Available: http://doi.acm.org.miman.bib.bth.se/10.1145/2605176

[9] The go programming language. [Online]. Available: https://golang.org/

[10] (2016, dec) Clean. [Online]. Available: http://clean.cs.ru.nl/Clean

[11] (2015, may) Common lisp. [Online]. Available: https://common-lisp.net/

[12] (2015, jan) Scheme reports. [Online]. Available: http://www.scheme-reports.org/

[13] (2016, dec) Comparison of functional programming languages. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_functional_programming_languages

[14] E. Kindler, "Object-oriented simulation of simulating anticipatory systems," *International Journal of Computer Science*, pp. 163–171, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.8133

[15] J. Skansholm, *C++ direkt*, 3rd ed. Lund: Studentlitteratur AB, 2012.

[16] (2015, apr) Underscore.js. [Online]. Available: http://underscorejs.org/

[17] F. Inc. (2017) Flow: A static type checker for javascript. [Online]. Available: https://flow.org/

[18] Microsoft. (2016) Typescript - javascript that scales. [Online]. Available: https://www.typescriptlang.org/

[19] kangax and zloirok. (2017) Ecmascript 6 compability table. [Online]. Available: http://kangax.github.io/compat-table/es6/

[20] M. D. Network and individual contributors. (2017) Memory management - javascript — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

[21] M. Grady, "Functional programming using javascript and the html5 canvas element," *J. Comput. Sci. Coll.*, vol. 26, no. 2, pp. 97–105, Dec. 2010. [Online]. Available: http://dl.acm.org.miman.bib.bth.se/citation.cfm?id=1858583.1858597

[22] M. D. Network and individual contributors. (2017) Javascript reference - javascript — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

[23] R. S. Engelschall. (2016) Ecmascript 6: New features: Overview and comparison. [Online]. Available: http://es6-features.org/#ClassDefinition

[24] M. C. Benton and N. M. Radziwill, "Improving testability and reuse by transitioning to functional programming," *CoRR*, vol. abs/1606.06704, 2016. [Online]. Available: http://arxiv.org/abs/1606.06704

[25] C. Dobre and F. Xhafa, "Parallel programming paradigms and frameworks in big data era," *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 710–738, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10766-013-0272-7

[26] R. Harrison, L. G. Smaraweera, M. R. Dobie, and P. H. Lewis, "Comparing programming paradigms: an evaluation of functional and object-oriented programs," *Software Engineering Journal*, vol. 11, no. 4, pp. 247–254, July 1996.

[27] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini, "On the positive effect of reactive programming on software comprehension: An empirical study," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[28] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy, "Using functional programming within an industrial product group: Perspectives and perceptions," *SIGPLAN Not.*, vol. 45, no. 9, pp. 87–92, Sep. 2010. [Online]. Available: http://doi.acm.org.miman.bib.bth.se/10.1145/1932681.1863557

[29] M. A. Weiss, *Data Structures and Algorithm Analisys in C++*, 4th ed. Edinburgh Gate, Harlow, Essex CM20 2JE, England: Pearson, 2014.

[30] P. K. Stockmeyer. (1998, aug) Rules and practice of the game of the tower of hanoi. [Online]. Available: http://www.cs.wm.edu/ pkstoc/page_2.html

[31] (2017) Atom. [Online]. Available: https://atom.io/

[32] N. Foundation. (2017) Node.js. [Online]. Available: https://nodejs.org/en/

[33] I. npm. (2017) npm. [Online]. Available: https://www.npmjs.com/

[34] (2016) Grunt: The javascript task runner. [Online]. Available: https://gruntjs.com/

[35] (2017, jan) Mocha - the fun, simple, flexible javascript test framework. [Online]. Available: https://mochajs.org/

[36] (2017, apr) Chai. [Online]. Available: http://chaijs.com/

[37] F. Ziegelmayer. Karma - spectacular test runner for javascript. [Online]. Available: https://karma-runner.github.io/1.0/index.html

# A Appendix

## A.1 Test cases

| | |
|---|---|
| ID | T1 |
| Algorithm | Binary search tree algorithms |
| Function | insert(comparable, rootNode) |
| Description | Inserted items should be added at the correct place in the tree. If number already exists in the binary search tree, it should not be inserted. |
| Preconditions | See figure 15 |
| ID | T1.1 |
| Input | 9, node(13) |
| Expected values | FP: None, OOP: See figure 15 |
| Expected output | FP: See figure 15, OOP: false |
| ID | T1.2 |
| Input | 8, node(13) |
| Expected values | FP: None, OOP: See figure 16 |
| Expected output | FP: See figure 16, OOP: true |
| ID | T1.3 |
| Input | 1, node(13) |
| Expected values | FP: None, OOP: See figure 17 |
| Expected output | FP: See figure 17, OOP: true |
| ID | T1.2 |
| Input | 33, node(13) |
| Expected values | FP: None, OOP: See figure 18 |
| Expected output | FP: See figure 18, OOP: true |

| | |
|---|---|
| ID | T2 |
| Algorithm | Binary search tree algorithms |
| Function | findNode(comparable, rootNode) |
| Description | findNode should return the node containing comparable. If comparable is not in the tree undefined should be returned. |
| Preconditions | See figure 15 |
| ID | T2.1 |
| Input | 5, node(13) |
| Expected values | None |
| Expected output | undefined |
| ID | T2.2 |
| Input | 13, node(13) |
| Expected values | None |
| Expected output | node(13) |
| ID | T2.3 |
| Input | 2, node(13) |
| Expected values | None |
| Expected output | node(2) |
| ID | T2.4 |
| Input | 32, node(13) |
| Expected values | None |
| Expected output | node(32) |
| ID | T2.5 |
| Input | 20, node(13) |
| Expected values | None |
| Expected output | node(20) |

| | |
|---|---|
| ID | T3 |
| Algorithm | Binary search tree algorithms |
| Function | inOrderTraversal() |
| Description | Should return a sorted array of the elements in the tree. If the tree is it should return an empty array. |
| ID | T3.1 |
| Preconditions | See figure 15 |
| Input | FP: node(13), OOP: none |
| Expected values | None |
| Expected output | [2, 3, 6, 7, 9, 13, 16, 20, 24, 32] |
| ID | T3.2 |
| Preconditions | Empty tree |
| Input | FP: none, OOP: none |
| Expected values | None |
| Expected output | [] |

| | |
|---|---|
| ID | T4 |
| Algorithm | Shellsort |
| Function | shellsort(array) |
| Description | If an array of numbers is input a sorted array should be returned. If an empty array is input an empty array should be returned. |
| Preconditions | None |
| ID | T4.1 |
| Input | [] |
| Expected values | None |
| Expected output | [] |
| ID | T4.2 |
| Input | [ 9, 8, 1, 15, 3, 4, 11, 2, 7, 6] |
| Expected values | None |
| Expected output | [ 9, 8, 1, 15, 3, 4, 11, 2, 7, 6] |

| | |
|---|---|
| ID | T5 |
| Algorithm | The Tower of Hanoi |
| Function | hanoi(tower, start, dest, aux) |
| Description | Should return start, dest and aux pegs with moved tower. If tower has zero discs it should return empty start, dest and aux. The algorithm should take $2^n - 1$ moves. |
| Preconditions | None |
| ID | T5.1 |
| Input | tower size 8, peg1 with tower, peg3 empty, peg2 empty |
| Expected values | FP: none, OOP: nrOfMoves=255 |
| Expected output | peg1 empty, peg2 empty, peg3 with tower |
| ID | T5.2 |
| Input | tower size 0, peg1 empty, peg3 empty, peg2 empty |
| Expected values | FP: none, OOP: nrOfMoves=0 |
| Expected output | peg1 empty, peg2 empty, peg3 empty |

| | |
|---|---|
| ID | T6 |
| Algorithm | Dijksta's algorithm |
| Function | dijkstras(graph, startNode, endNode) |
| Description | Should return the shortest path from startNode to endNode. If startNode is same as endNode it should return empty path. If graph is empty it should return empy path. |
| Preconditions | None |
| ID | T6.1 |
| Input | See figure 19, node1, node6 |
| Expected values | None |
| Expected output | See figure 20 |
| ID | T6.2 |
| Input | See figure 19, node1, node4 |
| Expected values | None |
| Expected output | See figure 21 |
| ID | T6.3 |
| Input | See figure 19, node2, node5 |
| Expected values | None |
| Expected output | See figure 22 |
| ID | T6.4 |
| Input | See figure 19, node1, node1 |
| Expected values | None |
| Expected output | empty path |
| ID | T6.5 |
| Input | nodes=[], edges[], noNode, noNode |
| Expected values | None |
| Expected output | empty path |

## A.2 Test case images

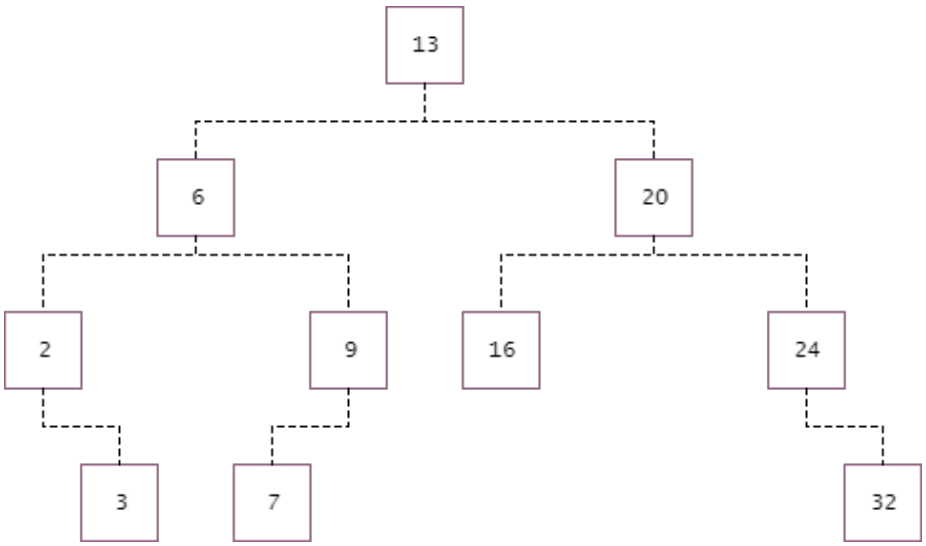


Figure 15: Input tree

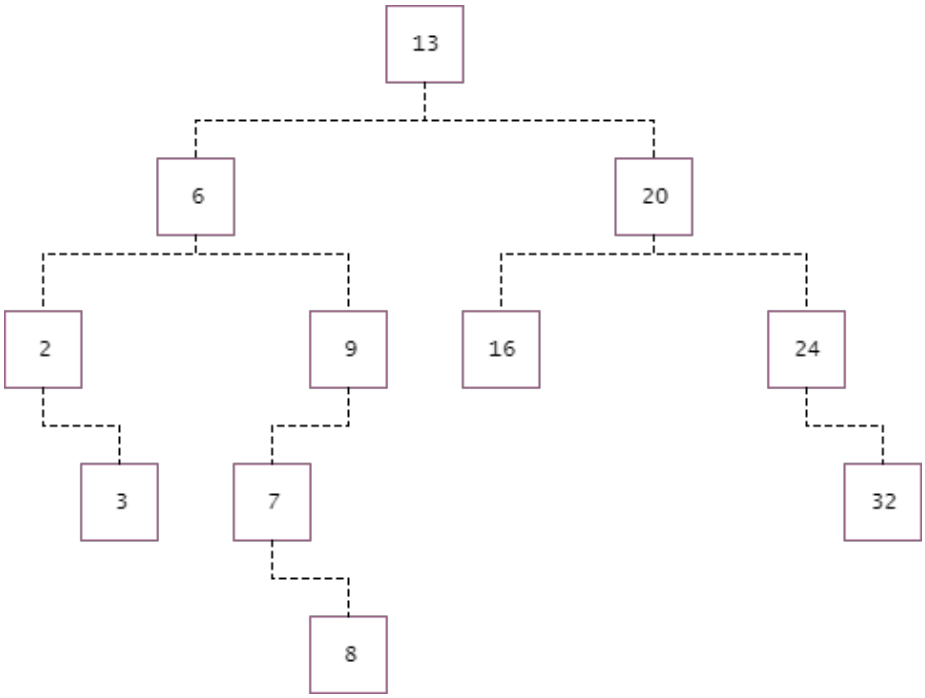


Figure 16: Output tree after inserting 8

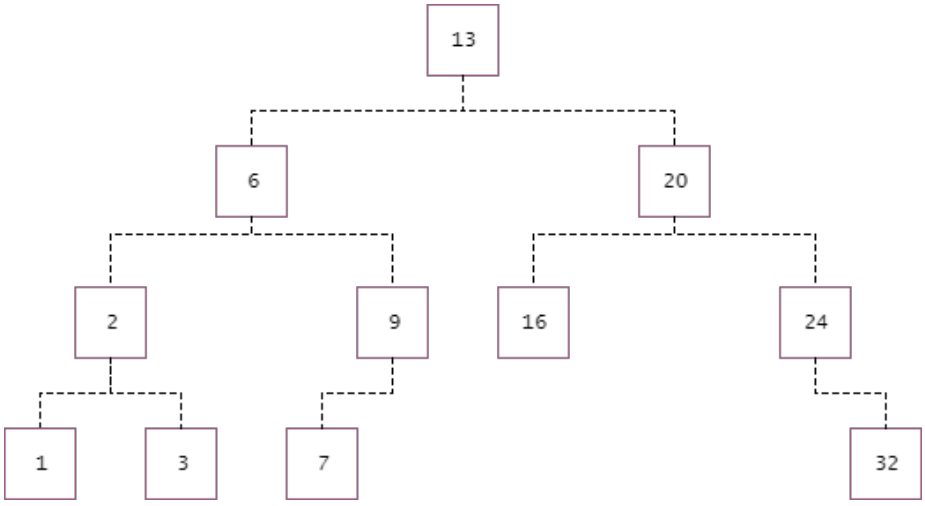


Figure 17: Output tree after inserting 1

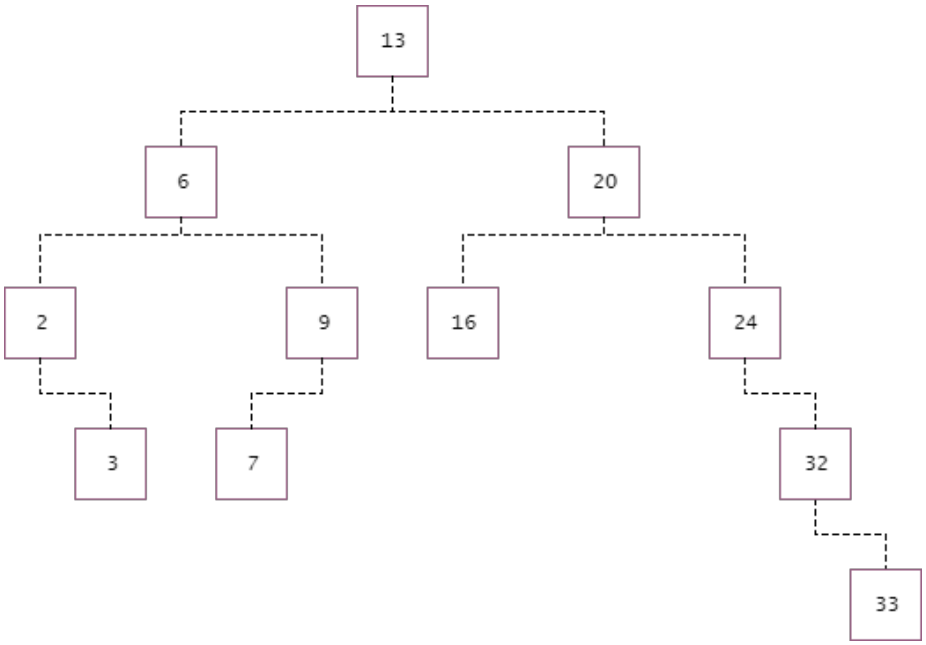


Figure 18: Output tree after inserting 33

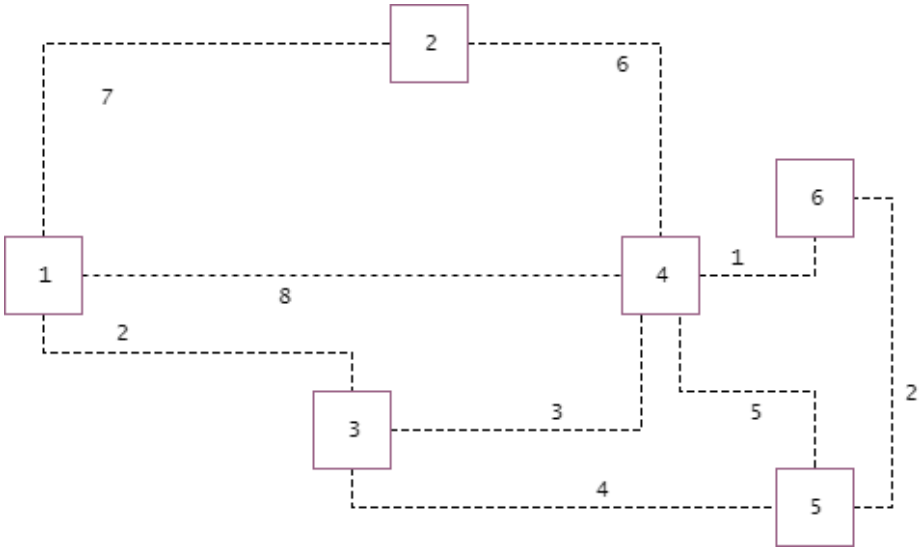


Figure 19: Input graph

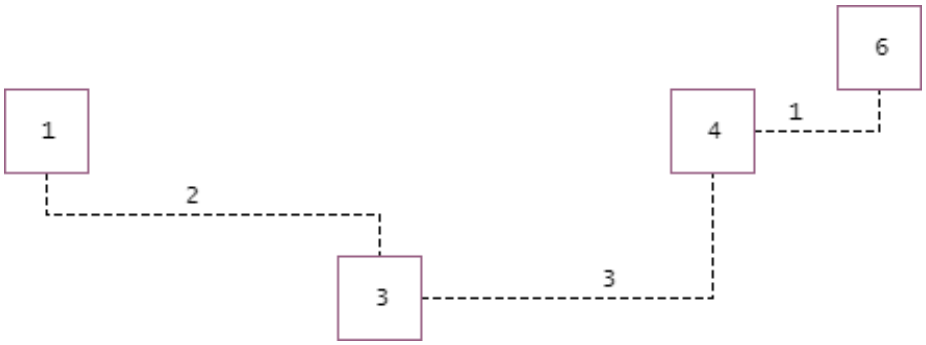


Figure 20: Output graph 1

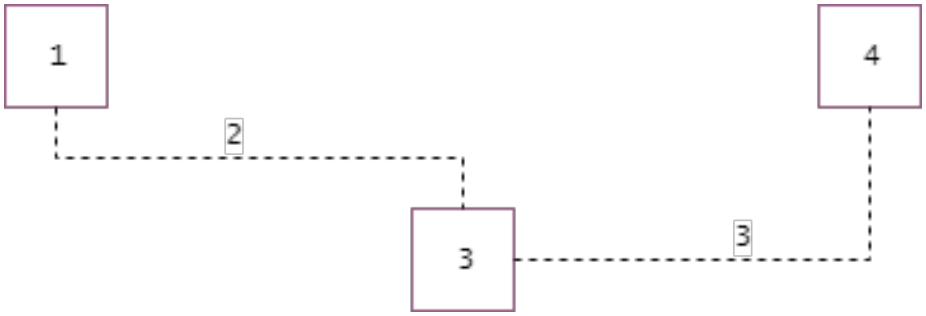


Figure 21: Output graph 2

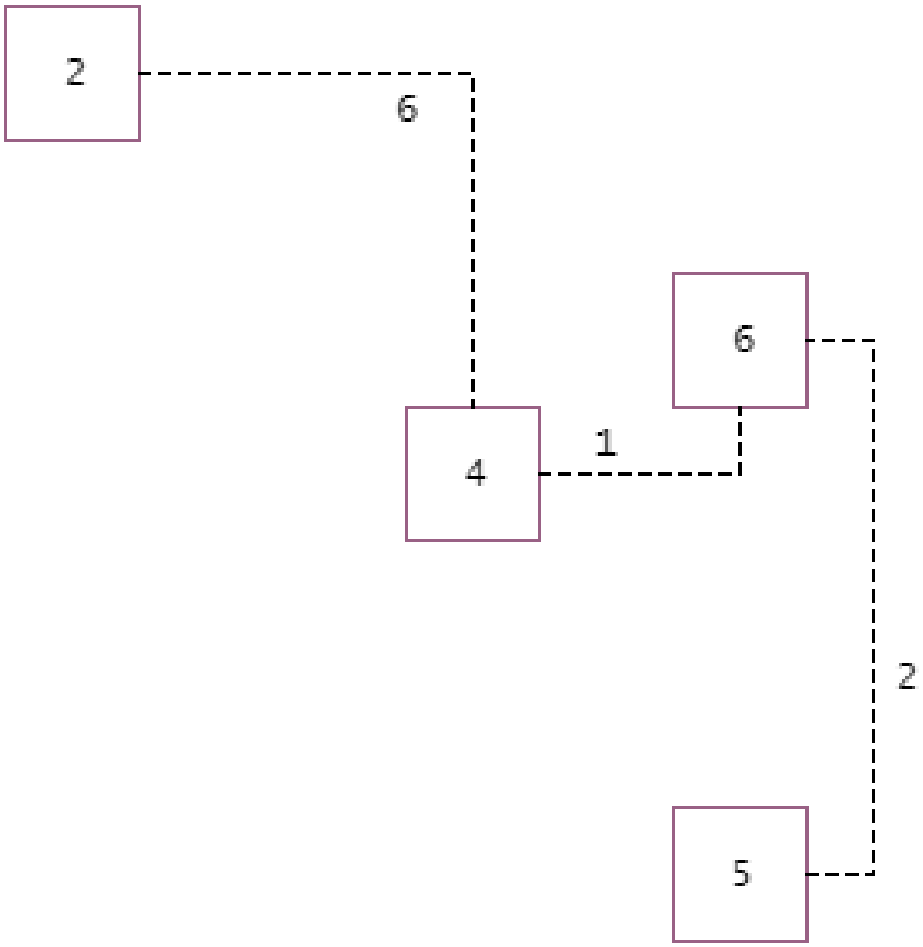


Figure 22: Output graph 3