CM1035: Algorithms and Data Structures I

Summary

Arjun Muralidharan

9th May 2020

Contents

List Of Figures

List Of Tables

List Of Algorithms

1 Problems, Algorithms And Flowcharts

Learning Outcomes

- ✓ Explain in broad strokes what are problems and algorithms in Computer Science.
- ✓ Explain the basic elements and construction of flowcharts.
- ✓ Express elements of simple algorithms as flowchart.

1.1 Problems

A **computational problem** is a statement of a desired input/output relationship. An **algorithm** is a tool for solving such a problem, i.e. a computational procedure for achieving that input/output relationship. A problem is well-defined when there is a clearly stated input and a description of the desired output.

A specific input needed to compute a solution is called an **instance** of a problem.

Algorithms predate computers and have existed since almost 3000. They are not specific to computer science. Multiple algorithms can solve the same problem. It is therefore important to be able to decide on the best algorithm. A good metric is to check if an algorithm can cover a wide variety of inputs. Another metric is the time it takes to execute an algorithm.

1.1.1 Heron's Method

In computation, we are faced the limitation that computers can only store finite numbers. Hence, irrational numbers need to be approximated.

$$\left|\sqrt{2} - \frac{x}{y}\right| \leq \eta$$

where $\frac{x}{y}$ is an approximation and η is the remaining error.

Heron's method approximates the value of a square root $\sqrt{x}$ by the following method.

1. Make a guess g for the square root of a number x. A good starting point is the *mean* between 1 and x. Also define n as the number of decimal points desired for precision.
2. Calculate $\frac{x}{g}$.
3. If $\frac{x}{g} - g < 10^{-(n-1)}$, return g. Otherwise, carry on.
4. Calculate the mean between $\frac{x}{g}$ and g. Set this to be the new g.
5. Repeat from step 2.

The flowchart for Heron's algorithm is shown in Figure 1.

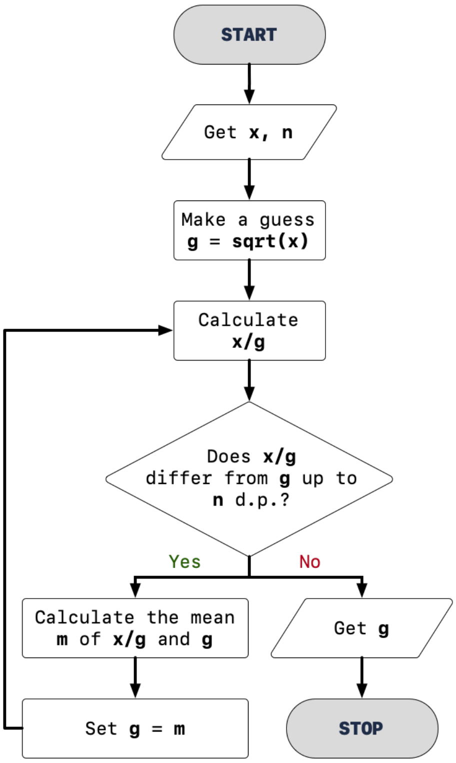


Figure 1. *Heron's Algorithm*

1.1.2 Euclid's Algorithm

Another algorithm is **Euclid's Algorithm**, which finds the greatest common divisor of two non-zero integers.

The flowchart for Euclid's algorithm is shown in Figure 2.

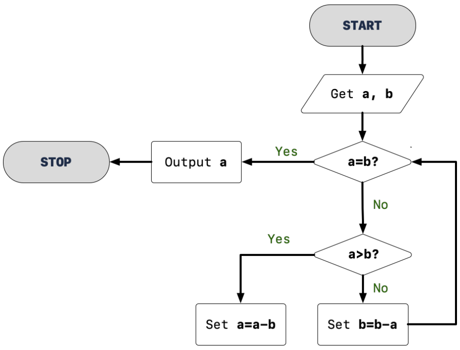


Figure 2. *Euclid's Algorithm*

2 Pseudocode

Learning Outcomes

✓ Explain the necessity and concept of pseudocode.

✓ Describe the concept of iteration and how it is represented in pseudocode.

✓ Convert flowchart (if possible) to pseudocode

Pseudocode allows for notation of an algorithm in a more readable format than flowcharts. You should be able to translate a flowchart into Pseudocode. Further, it allows for abstracting from any specific programming language. The notation for pseudocode is shown in Table 1. An example is shown in Algorithm 1

Operator	**Notation**
Assignment	$\leftarrow$
Arithmetic	$+ \quad - \quad \times \quad /$
Comparison	$= \quad \neq \quad < \quad > \quad \leq \quad \geq$
Logic	$\vee \quad \wedge \quad \neg$ **if** . . . **then**

Table 1. *Notation in Pseudocode*

Example The following algorithm is shown using pseudocode.

Algorithm 1 Simple example of pseudocode

```
$t \leftarrow 18$
$T \leftarrow 20$
if $t < T$ then
    $t \leftarrow t + 0.5$
end if
```

2.1 Functions

Functions are noted with the **function** keyword and contains a body. The body is bookended by **end function**. An example is shown in Algorithm 2.

Algorithm 2 Functions

```
function EVEN($n$)
    if $n \mod 2 = 0$ then
        return true
    end if
end function
```

2.2 Loops

We generally work with two kinds of loops: **for** loops and **while** loops. **for** loops execute a fixed number of iterations and generally increment a counter for each iteration. **while** loops execute as long as a certain condition is true.

Loops can be exited using the **break** or **continue** statements. However, these should be avoided if possible.

An example of **for** and **while** loops are provided in Algorithm 3.

Algorithm 3 Loops

```
$x \leftarrow 1$
for $2 \leq i \leq 10$ do
    $x \leftarrow x + i$
end for
$x \leftarrow 1$
$y \leftarrow 0$
while $x < 11$ do
    $x \leftarrow x + y$
end while
```

3 Abstract Data Types & Structures

Learning Outcomes

- ✓ Describe the basic elements of an abstract data structure
- ✓ Explain queues, stacks and vectors in terms of their structure and operations
- ✓ Compare these three different abstract data structures of vectors, stacks and queues

3.1 Booleans And Integers

We have so far already worked with two abstract data types. These are not data structures, as they are not defined by their operations.

1. **Booleans**, which can take the values **TRUE** and **FALSE**.
2. **Integers**, which can take a value from the set of whole numbers $\mathbb{N}$.

3.2 Vectors

Vectors are an abstract data type and an arrangement of elements with the following properties:

1. They have a *fixed* length.
2. Elements cannot be *deleted*.
3. Elements cannot be *added*.

4. All elements can be accessed (using a $SELECT$ operation).

Table 2 lists valid operations on a vector.

Operation	Pseudocode
length	LENGTH[v]
select[k]	$v[k]$
store![o,k]	$v[k] \leftarrow o$
Construct new vector of length n	**new** Vector $w(n)$

Table 2. *Vector operations*

We assume that vectors are indexed starting at 1. This is because their length is stored at index 0 of an array.

3.3 Queues

A **queue** is a kind of vector. It is a sequential arrangement of ordered elements, but with the following properties:

1. They have a *variable* length.

2. Elements can only be *deleted* from the *head*.

3. Elements can only be *added* at the *tail*.

4. Elements can only be accessed at the head of the queue. To access other elements, first the preceding elements need to be removed from the head the queue.

Table 3 lists valid operations on a queue. A queue follows a *FIFO* (first-in, first-out) policy, meaning that elements added first (and have been in the queue the longest) are accessed first.

3.4 Stacks

A **stack** is another kind of vector. It is a sequential arrangement of ordered elements with the following properties:

1. They have a *variable* length.

2. Elements can only be *deleted* from the *top*.

3. Elements can only be *added* at the *top*.

4. Elements can only be accessed at the top of the stack. To access other elements, first the preceding elements need to be removed from the top of the stack.

Operation	**Pseudocode**	**Description**
head	HEAD$[q]$	Returns the element at the head of the queue
dequeue!	DEQUEUE$[q]$	Returns and removes the element at the head
enqueue![o]	ENQUEUE$[o, q]$	Adds an element to the tail of the queue with the value o
empty?	EMPTY$[q]$	Asks if the queue has any elements, and returns TRUE if empty
Construct new (empty) queue	**new** Queue q	Creates a new, empty stack

Table 3. *Queue operations*

Table 4 lists valid operations on a stack. A stack follows a *LIFO* (last-in, first-out) policy, meaning that elements added most recently (and have been in the queue the shortest amount of time) are accessed first.

4 Data Structures & Searching Data I

Learning Outcomes

✓ Explain the difference between an abstract data structure and a concrete data structure

✓ Explain how abstract data structures can be implemented by arrays and linked lists

✓ Describe the linear search algorithm

4.1 Arrays

An **array** is a set of elements that a computer can manipulate. Arrays are a way to implement abstract data structures such as vectors, stacks and queues. It is **not** an abstract data structure, but rather an implementation thereof.

An array allocates a fixed amount of memory for a given data structure and therefore cannot change in size. This is reflected in the abstract data structure of a vector. An array only has two possible operations since it's size is fixed:

1. $read[k]$ to access an array element at index k

Operation	**Pseudocode**	**Description**
push![o]	PUSH$[o, s]$	Adds an element to the top of the stack
top	TOP$[o, s]$	Returns the element at top of the stack
pop!	POP$[s]$	Returns and removes the element at the top of the stack
empty?	EMPTY$[s]$	Asks if the stack has any elements, and returns TRUE if empty
Construct new (empty) stack	**new** Stack q	

Table 4. *Stack operations*

2. $write![o, k]$ to write or overwrite the value o at index k.

Arrays are indexed starting at 0. Usually, the length of an abstract data structure (such as a vector) is stored at this index.

4.2 Dynamic Arrays

A **dynamic array** is an abstract data structure that allows for extensibility. It is similar to the vector with the operations shown in Table 5.

The dynamic array operations are all implementable using an array with its basic *read* and *write* operations.

4.3 Linear Search

A linear search solves the problem of answering whether a specific search term is present on an abstract data structure. It returns either the index at which a search term was found, or false.

In a *vector*, linear search needs to systematically look at all the elements one after the other to determine if the value is in the data structure. A pseudocode implementation of linear search is shown in Algorithm 4.

[1] The first operation shifts the last element to the left by one. Repeat this for further elements on the right to be shifted. The second operation overwrites the last element with an empty value. The third operation updates the array length stored at index 0.

Operation	**Pseudocode**	**Array implementation**
length	LENGTH$[d]$	read[0]
select[k]	$d[k]$	read[k]
store![o,k]	$d[k] \leftarrow o$	write![o,k]
removeAt![k]	$d[k] \leftarrow \emptyset, k \leq LENGTH[d]$	write![d[k+1], k], write![,d[length[d]]], write![2,0][1]
insertAt![o,k]	$d[k] \leftarrow o, k \leq LENGTH[d] + 1$	new Array[length[d]+1], write![0,length[d]+1], write! elements on left to old locations, write![o,k], write! elements on right to new locations

Table 5. *Dynamic array operations*

Algorithm 4 Linear Search — $\mathcal{O}(n)$

```
function LinearSearch($v$, $item$)
    for $1 \leq i \leq$ LENGTH[v] do
        if $v[i] = item$ then
            return $i$
        end if
    end for
    return FALSE
end function
```

A linear search using **stacks** or **queues** needs to prevent data loss.

1. For **stacks**, we `push` elements to a second vector and then `pop!` the elements to access elements beneath
2. For **queues**, we `dequeue!` elements and `enqueue!` them to the end of the queue. In order to prevent an infinite loop, we also first `enqueue!` a new "end of queue" item to the stack, which we check for.

4.4 Linked Lists

The **linked list** is another data structure that can implement abstract structures.

Pointers A pointer stores the address of a piece of information in memory. In an array, all data is stored sequentially. In a linked list, this is not the case. Hence, we need pointers to tell us where to look next. We retrieve underlying data from a pointer by *dereferencing* the pointer.

Nodes The collection of a *key* and a *pointer* is called a **node**. A linked list consists of nodes.

Singly And Doubly Linked A linked list is **singly linked** if the nodes only contain a pointer $x.next$ to the next node. In a **doubly linked** list, nodes contain pointers $x.next$ and $x.prev$ to additionally point to the previous node.

Head A special pointer exists in a singly linked list, called the **head**. This is a pointer that is not associated with a node. If we dereference this pointer, we retrieve the the first node of the linked list.

Tail The final node has a pointer to `NULL`. If we see a pointer pointing to the `NULL` address, we know that we have reached the end of the pointer.

Operations in a linked list cannot access any arbitrary element. Rather, we need to reference elements using pointers.

1. $read[pointer]$ to access a list element at a specific pointer
2. $write![value, pointer]$ to write or overwrite the value at a given pointer.

Searching A Linked List The best way to search a linked list is to traverse the list with a *temporary pointer* that is initially set to the value `head`. We can then traverse the list until we find the key or the pointer reads `NULL`. This algorithm is shown in Algorithm 5.

Algorithm 5 Searching a linked list

```
function ListSearch($L$,$k$)
    $x \leftarrow L.head$
    while $x \neq$ NULL and $x.key \neq k$ do
        $x \leftarrow x.next$
    end while
    return $x$
end function
```

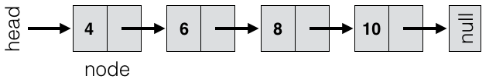
Queues and Stacks with Linked Lists

Algorithms and Data Structure I

In these notes we will go over how we can implement stacks and queues with the linked list data structure. We will quickly review linked lists, in particular how elements can be added and removed from the front node (or first node) of the linked list, as well as adding elements at the end of a linked list. After this, we will show how stacks can be implemented with linked lists, and finally how queues can be implemented.

1 Linked lists

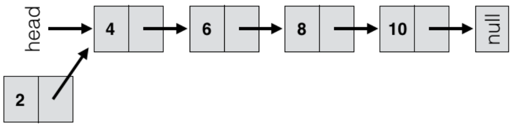
Recall that with linked lists we have nodes, and in each node we have a data field and a field that stores a *next* pointer that points to the next node. In addition to this we have a head pointer and a pointer to null at the beginning and end of the list respectively. Here's the picture we have of a linked list:



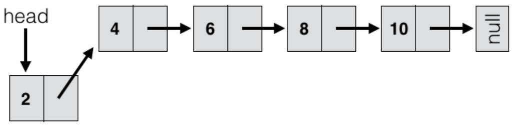
In addition to the list, we are able to create new pointers as variables and have them point to the nodes of linked lists. This will allow us to keep track of certain elements of a linked list, and we can dereference pointers to read (and write) the values stored at where the pointer is pointing. When we come to implement queues, having extra pointers will be very useful.

1.1 Adding and removing elements at the front of a linked list

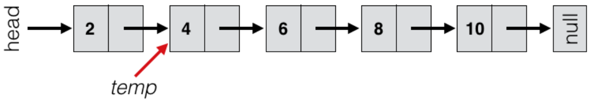
If a node is to be added at the front of the linked list, we first create a node which stores the required value (in the data field) as well as a pointer to the first node of the list, if the list is empty then it will point to null. The situation will look like the example above:



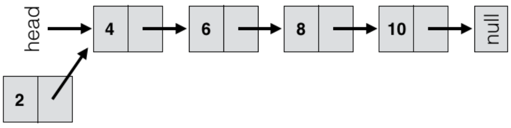
Then after this, the head pointer will be moved to now point at this new node to arrive at the following situation:



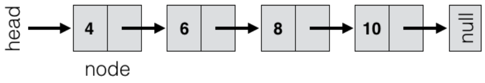
To remove a node from the front of the linked list, we can just, in some sense, reverse this process by first getting the head pointer to point at second node (or null if there is only one element in the list). We can assign the head pointer this value by first creating a new pointer called *temp* and give it the value of the next pointer of the first node. The situation will look like this:



We can then give the head pointer the same value as *temp* to recover this situation:



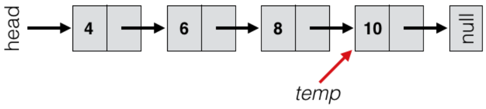
Then it remains to remove what was the first node in the linked list to obtain the following linked list:



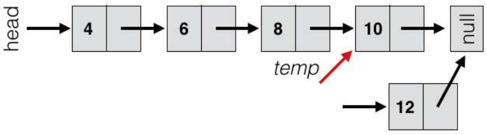
Thus the first node has been removed from the linked list.

1.2 Adding elements at the end of a linked list

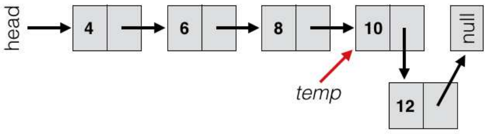
To add an element at the end of a list we need to introduce an extra pointer in addition to the linked list called *temp*. This pointer will traverse the list until it reaches the end of the list. We can check whether we have reached the end by checking the next pointer of each node to see if it points to null, and if it does we do not alter the value of *temp*. Once this pointer has traversed the list to the end we end up with this situation:



To add a new node to the end of the list, we first create a new node with the relevant data in its data field and we keep track of its location with a pointer. Then the next pointer of this new node will point to null to get this:



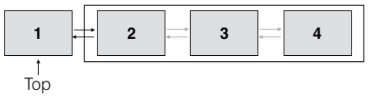
In the next step we take the next pointer of the final node in the original linked list, and give it the same value at the pointer to this new node:

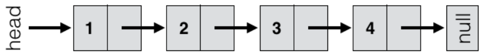


We have now added a new node to the end of the list. If we wanted to keep track of where the end of the list is we would just update *temp* to have the same value as the next pointer of the node at which it is currently pointed. Therefore, instead of having to traverse the list every time we wanted to add a new node at the end, we just update the value of *temp*; this will be useful when implementing queues.

2 Stacks

To implement a stack with a linked list, we just make the elements of the stack be the nodes in a linked list; every element storing a value will correspond to a node storing the same value in its data field. The top of the stack is purely the first node in the linked list. In essence the head pointer in the linked list will point to the top of our stack. The order of elements in a stack will then just be reflected by the order of the nodes in the linked list: if an element is pushed before another in the stack then there will be a pointer from the node corresponding to latter to the node corresponding to the former. So we have the following picture:





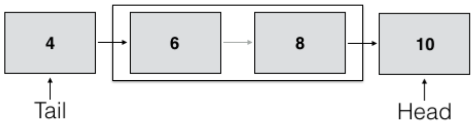
If our stack is empty then the head pointer will point to null, just as the top operation applied to an empty stack will have to return null, or nothing, or something similar. Now, we will show how each of the stack operations can be implemented by the linked list:

- top: The head pointer will be dereferenced and the associated value returned
- empty?: The head pointer will be dereferenced and return TRUE if and only if it points to null, and otherwise return FALSE
- push![o]: A new node storing the value o in its data field will be added to the front of the linked list, and the head pointer will now point to this new node
- pop!: The front node of the linked list will be removed as outlined above, with the head pointer now pointing to what was the second node (or null if there was only one element)

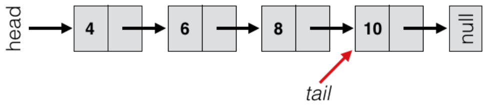
This is why linked lists are an excellent data structure for implementing stacks. The relative ease with which we can add and remove nodes to the front of the linked list without having to assign large amounts of new memory for each operation is a real advantage.

3 Queues

We can see that it is relatively straightforward to implement a stack with a linked list, but what about queues? Well, we can see that the elements in a queue will just correspond to the nodes in the linked list, much as with a stack. However, because a queue has a head and a tail, we need two pointers: the head pointer will point to the node corresponding to the head of the queue, and for the tail of the queue there will be a tail pointer that points to the final node in the linked list. If we are given the following queue:



Then the corresponding linked list will be:



To keep track of the tail of the queue we just keep track of the *tail* pointer. If the queue has just one element then the head and tail pointer will coincide, and for an empty queue, the head and tail pointer will both point to null. Here are the implementations of the queue in full:

- head: The head pointer will be dereferenced and the associated value returned
- empty?: The head pointer will be dereferenced and return TRUE if and only if it points to null, and otherwise return FALSE
- enqueue![o]: A new node storing the value o in its data field will be added to the end of the linked list as outlined above, with the *tail* pointer indicating where the end is. After the node is added, the *tail* pointer will be updated to be the next pointer of the current node, so that it is now pointing to the end node of the linked list

- dequeue!: The front node of the linked list will be removed as outlined above, with the head pointer now pointing to what was the second node (or null if there was only one element)

Again, we can see how linked lists are a natural data structure for implementing queues. The ease with which we can add and remove nodes at the beginning and end of the linked list suits the enqueuing and dequeuing operations of a queue. The only extra technicality is the need for a pointer to keep track of the node corresponding to the tail of the queue.

Exercise: Have a think about how dynamic arrays can be implemented with the linked list data structure. Go through each of the operations of the dynamic array and see if you can describe how it would be implemented with a linked list.

5 Sorting Data I

5.1 Bubble Sort

Bubble sort is popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

It starts at the head of a vector, compares the element to the adjacent elements, and swaps the elements if the first element is larger than the second element.

Passes The Bubble sort algorithm may need multiple passes to sort the entire vector. The maximum number of passes is given by the most difficult vectors to solve - such that are completely in reverse order. In this case, the number of required passes is $n-1$ where n is the number of elements. Therefore, **the maximum number of passes required** is always $n - 1$.

Algorithm 6 Bubble Sort — $\mathcal{O}(n^2)$

```
function BubbleSort(vector)
    $n \leftarrow$ LENGTH[vector]
    for $1 \leq i \leq n-1$ do
        $count \leftarrow 0$
        for $1 \leq j \leq n-1$ do
            if $vector[j+1] < vector[j]$ then
                Swap($vector, j, j+1$)
                $count \leftarrow count + 1$
            end if
        end for
        if $count = 0$ then
            break
        end if
    end for
    return vector
end function
```

5.2 Insertion Sort

Insertion sort works from the left of a vector, starting with the second element. It compares the element to the element on the left. If it needs to move, the element is moved along to the left as far as needed.

As an algorithm, we can store the current element in a temporary variable, while we "move" elements to the right by overwriting the slots to the right and then restoring the original values from the temporary variable.

Algorithm 7 Insertion Sort — $\mathcal{O}(n^2)$

```
function InsertionSort($v$)
    for $2 \leq$ LENGTH[v] do
        $key \leftarrow v[j]$
        $i \leftarrow j - 1$
        while $i > 0 \wedge v[i] > key$ do
            $v[i+1] \leftarrow v[i]$
            $i \leftarrow i - 1$
        end while
        $v[i+1] \leftarrow key$
    end for
end function
```

6 Random-Access Machines, Growth Of Functions And Time Complexity

Learning Outcomes

✓ Explain the model of random access machines.

✓ Explain asymptotic growth of functions and worst-case time complexity

✓ Describe worst-case time complexity of bubble and insertion sort

6.1 Random Access Machines

We generally assume a **random access machine (RAM)** model of computation. In the RAM model, instructions are executed one after another, with no concurrent operations. Memory can be directly access on a memory module.

- A RAM generally consists of a **memory unit** and a **central processing unit (CPU)**
- The CPU consists of a **program counter** that keeps track of where in the program the computer is, a **central unit** that is capable of executing *simple arithmetic* and *boolean logic*, and **registers** that can hold data.
- The central unit can *read* input and programs and *write* outputs from and to the memory unit.
- The program counter stores a non-negative integer.
- The memory unit and registers can store any integer (within the size of the register).

6.2 Worst-Case Time Complexity

Let M be a RAM machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n . If $f(n)$ is the running time of M , we say that M runs in time $f(n)$ and that M is an $f(n)$ time random access machine. Customarily we use n to represent the length of the input.

6.2.1 Big-O Notation

The **running time** of an algorithm on a particular input is the number of steps executed. Machine-independently, we define "time-steps". The total running time of an algorithm is defined as

$$\text{time cost per statement} \times \text{no. of executions per statement} = \text{total running time}$$

Function growth can be described and compared using **Big-O Notation**. In *asymptotic analysis*, we can consider only the highest order terms of a function and ignore the constants. For example, given the function

$$f(n) = 6n^3 + 2n^2 + 20n + 45$$

then the *asymptotic* or *Big-O* notation is given as

$$f(n) = \mathcal{O}(n^3)$$

A table of common functions and their respective Big-O notation is shown in Table 6 and sorted by growth rate from slowest to fastest.

Type	**Notation**
Constant	$\mathcal{O}(1)$
Logarithmic	$\mathcal{O}(\log(n)) = \mathcal{O}(\log(n^c))$
Polylogarithmic	$\mathcal{O}((\log(n))^c)$
Linear	$\mathcal{O}(n)$
Quadratic	$\mathcal{O}(n^2)$
Polynomial	$\mathcal{O}(n^c)$
Exponential	$\mathcal{O}(c^n)$

Table 6. *Notation in Pseudocode*

Formally, this means that any function is asymptotically *upper-bounded* by its respective Big-O class.

Let f and g be functions. It holds that

$$\exists k \exists n_0 \forall n \{|f(n)| \leq k \cdot g(n) \mid k > 0, n > n_0\} \to f(n) \in \mathcal{O}(g(n))$$

stating that the size of the function $f(n)$ will eventually be overtaken by the Big-O class $g(n)$ multiplied by a constant k. It is an upper bound.

Because Big-O classes live inside each other, we always seek to ascertain the *smallest* Big-O class in which a function exists.

7 Searching Data II

7.1 Binary Search

Binary Search is a more efficient way of searching an item in a list. It proceeds by halving the list and comparing the middle element to the searched term. If the middle element is smaller than the term, the search continues on the right sublist, and if larger, on the left sublist. Then the procedure is repeated.

Algorithm 8 Binary Search — $\mathcal{O}(\log(n))$

```
function BinarySearch($v$, $item$)
    $L \leftarrow 1$
    $R \leftarrow$ LENGTH[$v$]
    while $L \leq R$ do
        $M \leftarrow \lfloor \frac{L+R}{2} \rfloor$
        if $v[M] < item$ then
            $L \leftarrow M + 1$
        else if $v[M] > item$ then
            $R \leftarrow M - 1$
        else
            return $M$
        end if
    end while
    return false
end function
```

Binary search is faster than linear search because each additional search step works on a smaller input (half of the previous input). Hence the time to complete one additional step decreases by half with each step. Hence the algorithm is logarithmic.

Searching Linked Lists in JavaScript

Algorithms and Data Structures I

1 Objectives

In this reading material, the objectives are:

1. Give a brief introduction to objects and object constructors in JavaScript
2. Describe how objects can be used to construct a linked list
3. Describe how a linked list in JavaScript can be searched

2 Objects in JavaScript

We have previously covered JavaScript Arrays, and in doing so, we saw that they are *objects* in JavaScript. What is an object? The concept of an object appears in many programming languages, and in its simplest sense it is a collection of data. Indeed, the simplest kinds of objects are data structures. However, in many languages objects can have *methods*, which allow one to manipulate the data within objects; at this point objects cease to be collections of data (or data structures) and become something more complicated.

For our purposes we will focus on the simplest kinds of objects. In this case an object consists of *properties*: each property has a name and a value. For example, we could take Old McDonald's farm and the properties as animals in the farm (with the values being the number of the animals). The corresponding object would look like this:

```
var farm = {
  cows : 20,
  pigs : 10,
  ducks : 6,
  lambs : 5,
  chickens: 40
}
```

Here we create a variable called `farm` to which we will assign an object. The whole object is inside the curly brackets, and we see a comma-separated list of properties of the form `name:value`: the name of the property is on the left and the value is on the right. Thus the number of cows in Old McDonald's farm is 20, and the number of chickens is 40. The values need not be numbers; they could be any of the simple data types in JavaScript: Booleans, strings, undefined, null or even another object. We will be using the null data type later on, which signifies nothing. One of the quirks of JavaScript is that null is an object.

From the object we can extract the values of particular properties. For example, for `farm`, we can obtain the number of ducks by typing `farm.ducks`. If we type `var ducks = farm.ducks`, this will create a new variable called `ducks` and assign the value 6 to it. In general for an object called `obj` with property `prop`, `obj.prop` will store the value associated with property `prop`. We can think of the property name `prop` of `obj` as a *pointer* telling us where a value is stored, and then we *dereference* the pointer by using the syntax `obj.prop`. This will be very useful when we come to think about linked lists.

We can make our own objects from scratch by creating a new variable and assigning it properties as above. This is cumbersome if we wish to create a lot of objects that are essentially the same. For example, Old McDonald might have a friend called Young McKenzie who has a farm with the same species of animals, but with different numbers. To be able to make multiple similar objects we use something called a *constructor*. Note that in your Introduction to Programming course this is called the *factory pattern*, but constructor is the more standard piece of terminology so I will use it.

Imagine we want to create lots of objects like `farm` but with the numbers of animals changing from farm to farm (this number could also be 0). To do this we create a function that takes the numbers of the answers as arguments and makes an object. For the case of the farms, the function will look like this:

```
function Farm(cow, pig, duck, lamb, chicken) {
  this.cows = cow;
  this.pigs = pig;
  this.ducks = duck;
  this.lambs = lamb;
  this.chickens = chicken
}
```

Note that we are using the syntax `this.name` and assigning it the values in the arguments of the function. The syntax `this.name` just says that the object created by this function will have the property `name`, and we can assign it a particular value. To now use the constructor, consider the following:

```
var farm = new Farm(20, 10, 6, 5, 40);
```

This will create the same variable `farm` as above. The important thing to note is the use of `new`, which says that we are creating a new object, followed the kind of object this will be, or which class it belongs to. In our case, the object `farm` will belong to the class `Farm`. By convention, we use the capital letter at the beginning of class names to indicate something is a class.

3 Linked lists with objects

Recall that a linked list consists of a collection of nodes, with each node containing a data field, and another element containing the value of a pointer to the next node. Therefore, each node of a linked list is a collection of data. As previously mentioned we do not have pointers as a basic data type in JavaScript so we have to be more creative. However, as alluded to in the previous section, we can use the names of properties in an object as a pointer.

Each node of a linked list will be implemented in JavaScript by an object with two properties: the first property is `data`, which is the data stored in the data field of a node; the second property is `next`, which will correspond to a pointer to the next node. The property `next` could be an object corresponding to another node, or it could be `null`.

In the following we are going to create some JavaScript code, so you will need to set up the environment:

1. Create a folder in your "Documents" folder and name the folder `linked`
2. Create an empty JavaScript file called `linked.js` and save it to the folder `linked`
3. Go to your CLI and change the directory to the folder `linked`

Now open up the JavaScript file `linked.js` and write the following code to the file:

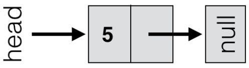
```
var head = null;
head = {data : 5, next : null};
console.log(head);
console.log(head.data);
console.log(head.next);
```

Now save the file and run it by entering `node linked` into the CLI.

You should see the following printed:

```
{ data: 5, next: null }
5
null
```

Let's unpack what is going on here. The variable `head` represents the head of the linked list, which will point to the beginning of the linked list. In this code we initially have `head` being assigned the value `null`, thus it is pointing at nothing. In other words, we have an empty linked list. Then in the next line we will assign a new object to `head`, thus corresponding to a single node. We assign to `head` a new object containing two properties: `data` contains the data stored at the node (in this case the number 5), and `next` is the property that points at the next node. In the second line, `next` is assigned `null`, thus adding the new node to the end of the empty list. Therefore we have an implementation of the following linked list:



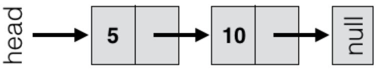
When we return to the code we see that the value stored in `head.data` "dereferences" the head pointer and reveals the value stored in the data field, and then `head.next` stores the pointer to the next node, or null.

Instead of adding a node to the end of a list by hand, we can use a constructor to do this for us. Remove all the current code in the file `linked.js` and put the following:

```
function LLNode(data) {
  this.data = data;
  this.next = null;
}
var head = new LLNode(5);
console.log(head);
console.log(head.data);
console.log(head.next);
```

When you run this from the CLI, you should essentially get the same thing printed as with the previous code (that was deleted) but with `LLNode` added to point out that `head` belongs to this class of objects.

With this constructor, we now have a method for adding elements to the end of a linked list. Consider the following linked list:

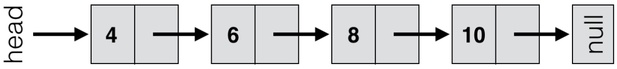


This is the linked list from earlier with a new node added at the end that stores the number 10. We will now implement this linked list based on the code already in `linked.js`. Put the following code under your existing code in `linked.js`:

```
head.next = new LLNode(10);
console.log(head.next.data);
console.log(head.next.next);
```

After running this code, the final two values printed to the console should be `10` and `null` respectively.

Exercise 1: Construct the following linked list through repeated use of the `LLNode` constructor.



4 Searching Linked Lists

To search a linked list we need to *traverse* the list; this is done by updating the value of a temporary pointer (called temp) to be the value of the next pointer as we go along the list. Consider the following piece of JavaScript:

```
var temp = head;
while (temp !== null) {
  temp = temp.next;
}
```

This loop will traverse the linked list by assigning a variable temp the value stored at head, then while head is not pointing to null, temp is updated to have the value of the next pointer. The loop is exited when temp takes the value null. We will use this technique to search for a value in the data fields of the nodes in the linked list.

In the JavaScript file linked.js, put the following code under the existing code:

```
function searchLL(head, item) {
  var temp = head;
  while (temp !== null) {
    if (temp.data === item) {
      return true;
    }
    temp = temp.next;
  }
  return false;
}
```

This will return true if the value item is stored in one of the data properties of one of the nodes, and false otherwise. One of the arguments to searchLL is head, which will be the object stored at head corresponding to the linked list. Test this function on various arguments: on the linked lists constructed above change item to have values that are, and are not, in the list.

Exercise 2: Write a function called numberLL that will take two input arguments: the head of a list called head, and a value called item. The function should return the number of times that item appears in the list, which should be a non-negative number.

8 Recursion

Recursion is the concept of a procedure referring to itself recursively. The main applications for using recursion are

- **Decrease and conquer**: Decreasing the problem to a *base case* and calling a procedure recursively on every decreasing inputs until the base case is reached.
- **Divide and conquer**: Split the problem into smaller partial problems and sort those first.

Algorithm 9 Recursive decrease-and-conquer procedure

```
function FACTORIAL($n$)
    if $n = 0$ then
        return 1
    end if
    return FACTORIAL($n - 1$)
end function
```

8.1 Call Stack

The call stack is a part of programming languages that is used to manage function calls and variables. At the beginning of a function call, the compiler creates a **stack frame** to hold all the variables and arguments relevant to that function. Every time a function terminates, the stack is popped and the memory cleared. Each time a function is called, it is pushed on to the stack, and popped when the function terminates. This ensures that function calls happen in order. If a function has too much recursion, this can cause a **stack overflow** as the call stack is stored in memory, which is finite. If a function has infinite recursion, stack overflow is guaranteed.

9 Sorting Data II

9.1 Quicksort

Quicksort is a recursive algorithm to sort a vector. It partitions the vector into sub-vectors at an arbitrary pivot, and then moves the elements smaller than the pivot to the left of it and elements larger to the right, and then moves the pivot element to the correct location in the partition. It then repeats this process recursively on the sub-vectors.

While Quicksort has a worst-case time-complexity of $\mathcal{O}(n^2)$, this case is unlikely to occur among possible inputs. The typical time complexity, or average-case complexity, is $\mathcal{O}(n\log(n))$

Algorithm 10 Quicksort — $\mathcal{O}(n^2)$

```
function QUICKSORT($A,\ p,\ r$)
    if $p < r$ then
        $q = $ PARTITION($A, p, r$)
        QUICKSORT($A, p, q - 1$)
        QUICKSORT($A, q + 1, r$)
    end if
end function

function PARTITION($A$,$p$,$r$)
    $x \leftarrow A[r]$ // Set the pivot to be the last element
    $i \leftarrow p - 1$ // Set $i$ as the divider between elements smaller or bigger than $x$
    for j = p to $r - 1$ do // Loop through the vector
        if $A[j] \leq x$ then
            $i \leftarrow i + 1$ // Move divider to the right to include an element bigger than $x$
            exchange $A[i]$ with $A[j]$ // Move the $j$th element to the left partition
        end if
    end for
    exchange $A[i + 1]$ with $A[r]$ // Move the pivot to the right of the divider
end function
```

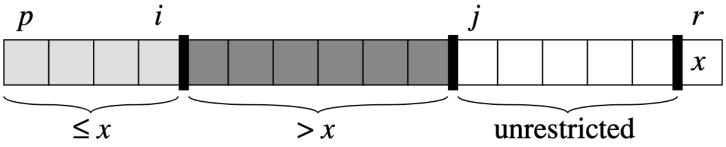


Figure 3. *Partitions in a Quicksort*

9.2 Merge Sort

Merge sort is another recursive algorithm that divides an array into subarrays, sorts them recursively and merges the resulting subarrays into an end result. This procedure relies on a MERGE function that compares two stacks element-by-element and merges them by placing the smaller of two elements into the result stack first. This comparison is then done recursively on ever-smaller subarrays, until the subarray length is one. In order to simplify code, the version below places *Sentinels* at the end of subarrays to terminate the comparison process and not have to track how many elements have been compared.

Algorithm 11 Merge Sort

```
function MERGE-SORT($A$,$p$,$r$ )
    if $p < r$ then// Check if the array is longer than 1
        $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
        MERGE-SORT($A, p, q$)
        MERGE-SORT($A, q + 1, r$)
        MERGE($A, p, q, r$)
    end if
end function

function MERGE($A$,$p$,$q$,$r$ )
    $n_1 \leftarrow q - p + 1$ // Length of left subarray
    $n_2 \leftarrow r - q$ // Length of right subarray
    let $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$ be new arrays // Extend length by 1 for sentinels
    for $i = 1$ to $n_1$ do // Copy left subarray
        $L[i] \leftarrow A[p + i - 1]$
    end for
    for $j = 1$ to $n_2$ do // Copy right subarray
        $R[j] \leftarrow A[q + j]$
    end for
    $L[n_1 + 1] \leftarrow \infty$ // Insert sentinels
    $R[n_2 + 1] \leftarrow \infty$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k = p$ to r do
        if $L[i] \leq R[j]$ then // Pick left element
            $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$ // Pick right element
            $j \leftarrow j + 1$
        end if
    end for
end function
```

9.3 Summary Of Sorting And Search Algorithms

The worst-case and average-case time complexities are shown in

Algorithm	**Worst Case**	**Average Case**
Bubble Sort	$\mathcal{O}(n^2)$	$\Theta(n^2)$
Insertion Sort	$\mathcal{O}(n^2)$	$\Theta(n^2)$
Quick Sort	$\mathcal{O}(n^2)$	$\Theta(n\log(n))$
Merge Sort	$\mathcal{O}(n\log(n))$	$\Theta(n\log(n))$
Linear Search	$\mathcal{O}(n)$	$\Theta(n)$
Binary Search	$\mathcal{O}(\log(n))$	$\Theta(\log(n))$

10 Computational Complexity

A procedure with exponential time-complexity will always be worse than polynomial time-complexity. This helps differentiate efficient from inefficient algorithms.

A problem can be viewed a set of inputs that satisfy some property outlined by the problem. A **complexity class** is a set of problems that can be solved with a specific complexity, i.e. a specific amount of computational resources.

Complexity classes **do not contain algorithms**; they contain inputs (or words). We can classify problems in terms of the algorithms used to solve them, but only the problems are in the same complexity class.

10.1 Complexity Classes

P The class of problems that can be **solved** in polynomial time $\mathcal{O}(n^k)$ and can be considered "easy" in the sense that an efficient algorithm exists to solve them.

NP The class of problems that can be **verified** (using a *certificate* in the form or a result that can be checked) in polynomial time but solved only in non-polynomial, i.e. exponential time.

NP-Hard The class of problems that are at least as hard as any problem in NP and with the property that any problem in NP can be reduced to the NP-hard problem, meaning it can be transformed and then solved, yielding the answer for both the NP-hard problem and the NP problem. Some of these problems extend beyond NP.

NP-Complete The class of problems that are both NP-hard and in NP.

NP-complete and NP-hard problems are unlikely to have a solution in polynomial time. It has not yet been proven to be impossible.

In an NP-complete problem, we need to check *every* possible candidate solution which might be an exponential number of inputs to check. This is similar to linear search, which checks every element of a vector.