Algorithms and Data Structures II
Course Notes

Felipe Balbi

April 23, 2020

Contents

Week 1 **3**
1.001 What is analysis of algorithms? . . . 3
1.002 What is analysis of algorithms? . . . 4
1.004 How to measure/estimate time and space requirements . . . 4
1.006 The RAM model . . . 5
1.007 The Ram Model . . . 6
1.009 Counting up time and space units, part 1 . . . 6
1.010 Counting up time and space units, part 2 . . . 6
1.011 Counting up time and space units . . . 7
1.101 Growth of functions, part 1 . . . 7
1.103 Growth of functions, part 2 . . . 10
1.105 Growth of functions . . . 11
1.106 Faster computer versus faster algorithm . . . 11
1.108 Faster computer versus faster algorithm . . . 12

Week 2 **13**
1.201 Worst and best cases . . . 13
1.202 Worst and average cases . . . 13
1.301 Introduction to asymptotic analysis . . . 13
1.303 Big-$\mathcal{O}$ notation . . . 14
1.305 Omega notation . . . 16
1.307: Theta notation . . . 18
1.309 Asymptotic notation . . . 19

Week 3 **21**
2.001 Introduction to recursion . . . 21
2.002 The structure of recursive algorithms . . . 21
2.004 Tracing a recursive algorithm . . . 22
2.101 From iteration to recursion . . . 22
2.103 Writing a recursive algorithm, part 1 . . . 23
2.104 Writing a recursive algorithm, part 2 . . . 23

Week 1

Key Concepts

- Determine time and memory consumption of an algorithm described using pseudocode
- Determine the growth function of the running time or memory consumption of an algorithm
- Use Big-O, Omega and Theta notations to describe the running time or memory consumption of an algorithm. Learning objectives:

1.001 What is analysis of algorithms?

Analysis allows us to select the best algorithm to perform a given task.

There are three main aspects we generally use to analyse algorithms:

Correctness whether the algorithm performs the given task according to a given specification

Ease of understanding how difficult is it to understand the algorithm

Resource consumption how much memory and how much CPU time does an algorithm consume

Algorithms who perform a given correctly consuming minimum ammount of resources are better candidates than those requiring more resources.

During this cource, emphasis is given to computational resource consumption of algorithms, that is, the amount of memory, CPU time and, perhaps, bandwidth necessary to complete a computation.

Processing requirements (i.e. CPU time) is measured in terms of the number of operations that must be carried out in order to execute the algorithm. This number is important because with lower number operations, naturally, the algorithm executes faster.

Memory requirements, conversely, are measured in terms of the number of memory units required by the algorithm during its execution. This number is important because we can't compute on data that doesn't fit our memory.

In summary, we learn how to analyse algorithms in terms of its CPU and Memory requirements. Based on such analysis, we will be equipped to select the best algorithm given a specific task.

1.002 What is analysis of algorithms?

Please read paragraph 1 of Section 2.2 (p.23) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.004 How to measure/estimate time and space requirements

Suppose we're given the following pseudocode:

```
1: function F(arrays)
2:     for 1 ≤ j <LENGTH(s) do
3:         key ← s[j]
4:         i ← j − 1
5:         while i ≥ 0 ∧ s[i] > key do
6:             s[i + 1] ← s[i]
7:             i ← i − 1
8:         end while
9:         s[i + 1] ← key
10:    end for
11: end function
```

Now we're asked to say how much time and space algorithm needs to execute. How do we go about answering that question?

One may consider an empirical approach of implementing the algorithm in a specific programming language and run it in a specific computer, then measure its runtime and memory consumption in a specific scenario.

One can also consider a more theoretical approach by making some assumptions about the number of operations for each instruction the CPU executes, multiplying by the time required by each instruction and, with that obtaining an estimate for the runtime. For memory requirements, we could look at all new variables created during the execution of the algorithm.

There are pros and cons for either approach:

Approach	Pros	Cons
Empirical	Real/Exact result	Machine-dependant results
	No Need for calculations	Implementation effort
Theoretical	Universal results	Approximate results
	No implementation effort	Calculations effort

During this course, we work with the theoretical approach. There are three aspects we need to understand very well:

The Machine Model know its characteristics well as they affect the results we can obtain.

Assumptions And Simplifications know where assumptions and simplifications cause a deviation from the real world and why.

Calculations calculations will be necessary. Usually simple additions and multiplications.

1.006 The RAM model

The Random-Access Machine Model is a simplified version of a computer machine.

Because a real machine is a very complex structure, we use a model to simplify our work. The model must be simple and yet complete enough to capture enough details as to be relevant. Figure 1 has a visual representation of the model.

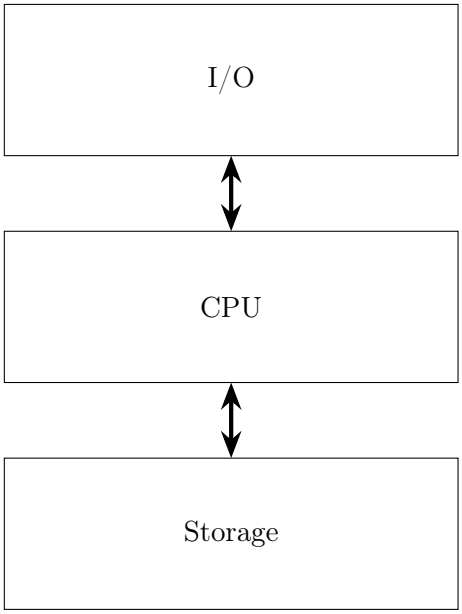

Figure 1: Random-access Machine Model

There are a few assumptions made for this model to work:

Single CPU With a single CPU, all instructions are executed sequentially.

Single Cycle Every simple operation take one time unit (or one cycle) to complete.

Loops/Functions Are Not Simple They are made up of several simple operations.

No Memory Hierarchy Every memory access takes one time unit (or one cycle) to complete. Also we always have exactly as much memory as is needed to run the computation.

We also have one assumption regarding memory consumption:

Simple Variables Uses 1 Memory Position One integer uses 1 memory position while an array of N elements uses N memory positions.

1.007 The Ram Model

Please read pp.23–4 of Section 2.2 from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.009 Counting up time and space units, part 1

We're going the analyse the function shown in listing with the analysis of each line typeset as a comment on that line. In order to get our total, we just add all simple operations together.

```
1: function F1(a, b, c)
2:     max ← a                          ▷ 1 memory read, 1 memory write
3:     if b > max then                  ▷ 1 conditional, 1 comparison, 2 memory reads
4:         max ← b                      ▷ 1 memory read, 1 memory write
5:     end if
6:     if c > max then                  ▷ 1 conditional, 1 comparison, 2 memory reads
7:         max ← c                      ▷ 1 memory read, 1 memory write
8:     end if
9:     return max                       ▷ 1 memory read, 1 return
10: end function
```

Adding up all our memory reads, memory writes, conditionals and conditionals, we get a total of 16 time units. In terms of space, there's only one new variable created, *max*. We have a requirement of only 1 space unit.

1.010 Counting up time and space units, part 2

Let's analyse the linear search algorithm. The algorithm takes 3 arguments, A, N, and x, where A is a 1D array, N is the number of elemnts in A, and x is an integer. The pseudocode is found in algorithm .

```
1: function F2(A, N, x)
2:     for 0 ≤ i < N do
3:         if A[i] = x then      ▷ 1 cond., 1 array access, 1 comparison, 2 memory reads
4:             return i                                  ▷ 1 return, 1 memory read
5:         end if
6:     end for
7:     return −1                                                    ▷ 1 return
8: end function
```

Because the *for* loop is not a simple instruction, we must break it down into simple instructions. A for loop is composed of three main components:

```
1: i ← 0                                                  ▷ 1 memory write
2: if i < N then                          ▷ 1 cond., 2 memory reads, 1 comparison
3:     <instructions>
4:     i ← i + 1                  ▷ 1 memory write, 1 memory read, 1 addition
5: end if
```

Note that the **If** part of the loop takes 4 time units, but runs $N+1$ times, therefore it takes $4 \cdot (N+1)$ time units. Also the increment part of the loop, takes 3 time units and runs N times, therefore it takes $3N$ time units. The total here is $4(N+1)+3N = 7N+5$ time units.

Continuing, we have another 5 time units running N times. Assuming the worst case, only outter-most return statement will execute for exactly 1 time unit.

Adding up all terms we have $7N + 5 + 5N + 1 = 12N + 6$ time units.

In terms of space units, we create a single new variable, i, and therefore our space requirement is 1 space unit.

1.011 Counting up time and space units

Please read about the analysis of insertion sort on pp.24–7 of the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.101 Growth of functions, part 1

Counting up every single time unit is not necessary. After making such large simplifications by using the RAM model, trying to get an exact number of time units is a pointless exercise when all we want to do is compare different algorithms and choose the fastest.

We can look at the running time of two different algorithms for solving the same problem. Figure 2 shows the graph of the running time as the size of the input grows.

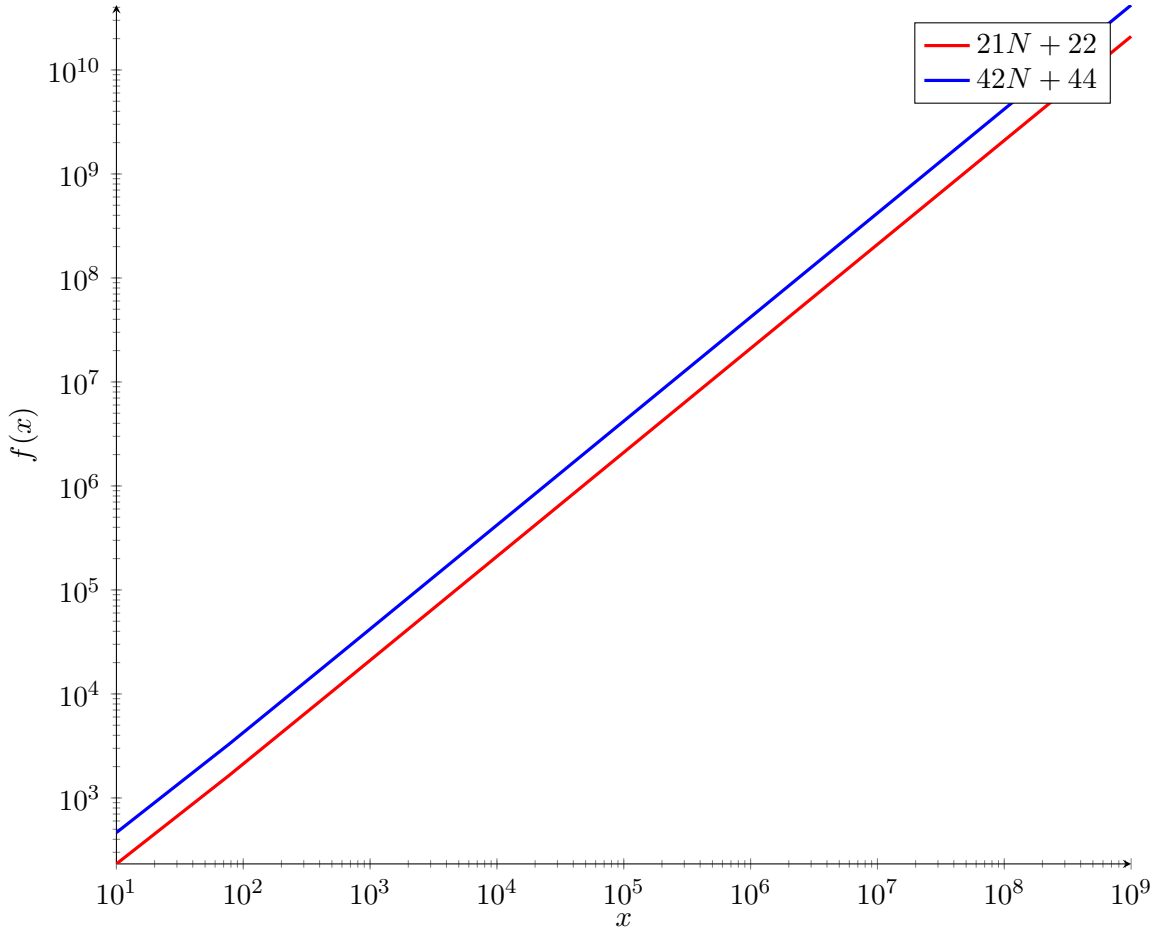

Figure 2: Running Time

Note that the running time grows linearly with the input size. That is, if the input grows 10 times, the running time grows about 10 times as well.

If someone proposes a third algorithm for solving the same problem with running time of $10N^2 + 30$, plotting the new function, we have the graph shown in figure 3.

We can see that the new curve, the one for $10N^2+30$, grows much faster than the other two. The difference is so large that the coefficients are not going to affect the difference as the input size grows.

When comparing algorithms, the growth of the running is sufficient, we don't need to specify coefficients. When analysing asymptotic growth of functions, lower order terms of the function also doesn't affect the function's growth.

For example $N^2 + N \approx N^2$ as N gets larger and larger.

Therefore, when comparing algorithms, we will do the following:

Use Generic Constants e.g. $T(N) = C_1N + C_2$

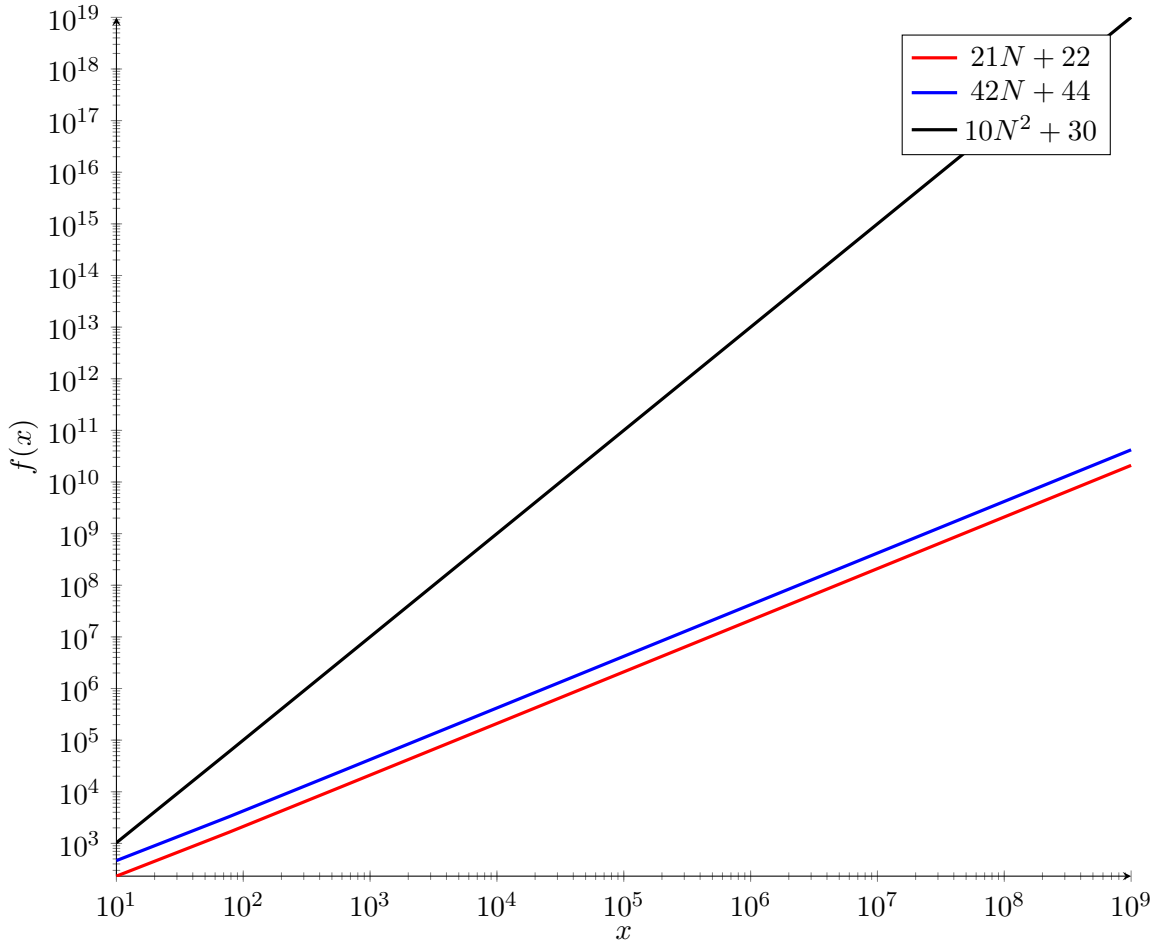

Figure 3: Running Time

Growth Of Running Time Ignore constants and lower-order terms

Below, we can find a listing of the most common growth functions:

- 1 (constant time)
- $\log N$
- N
- $N \log N$
- N^2
- N^3

- 2^N

Figure 4 depicts each of the growth functions above.

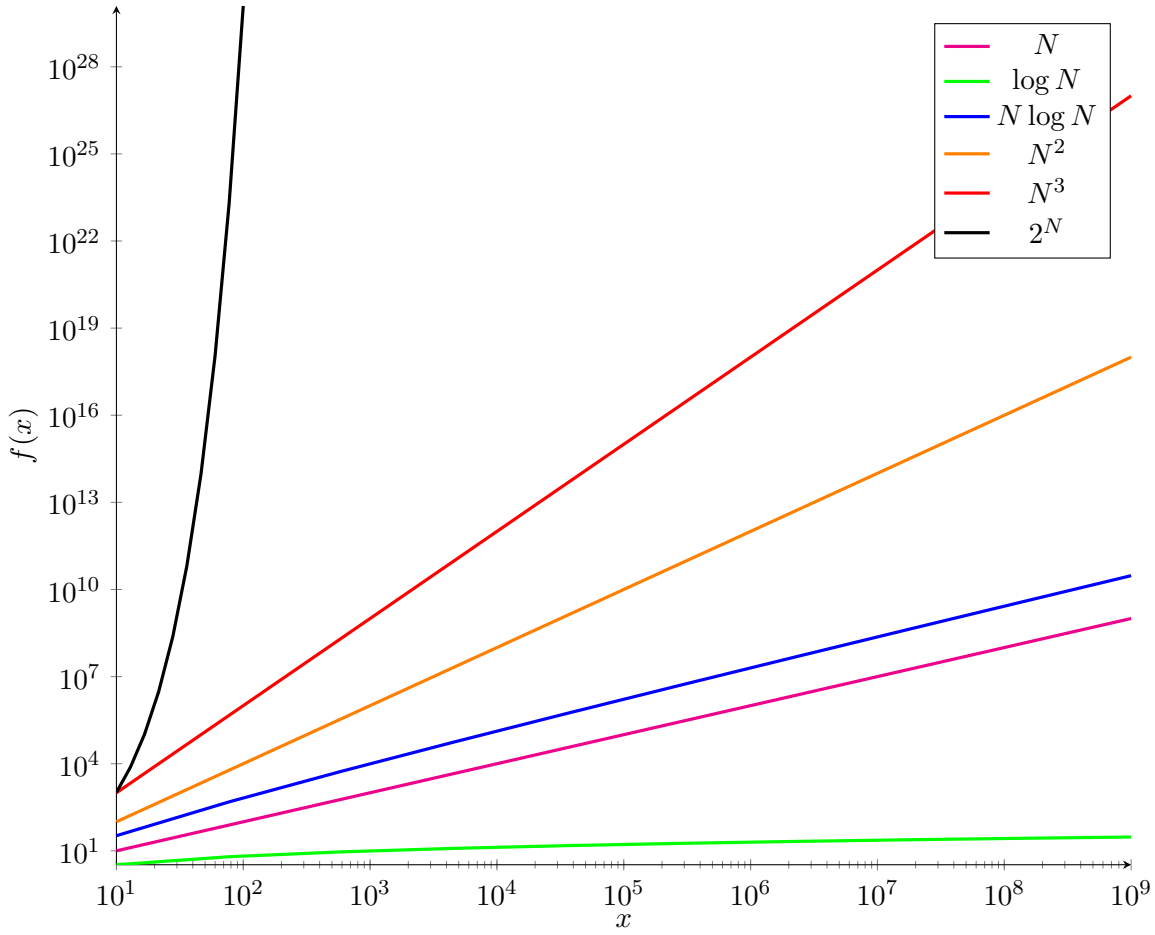

Figure 4: Running Time

1.103 Growth of functions, part 2

The following pseudocode in listing , computes the sum of the diagonal of a square matrix. Instead of counting every memory access and numerical operation, we are checking if the instruction takes constant time or not.

```
1: function SumDiag(A)
2:     sum ← 0                                          ▷ C_0
3:     N ← Length(A[0])                                 ▷ C_1N + C_2
4:     for 0 ≤ i < N do                                 ▷ C_3N + C_4
5:         sum ← sum + A[i, i]                          ▷ C_5N
6:     end for
7:     return sum                                       ▷ C_6
8: end function
```

Adding up all the terms, we get the following expression:

$$\begin{aligned} T(N) &= (C_1 + C_3 + C_5)N + (C_0 + C_2 + C_4 + C_6) \\ &= C_7N + C_8 \\ &= N \end{aligned}$$

1.105 Growth of functions

Please read the sub-section titled 'Order of growth' in Section 2.2 (pp.28–9) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.106 Faster computer versus faster algorithm

Assuming we designed an algorithm to solve a particular problem with a quadratic growth (i.e. $T(N) = N^2$). We will also assume that we have a computer where 1 time unit = 1ns.

The table below shows the running time for different input sizes:

N	N^2
10^1	$0.1\mu S$
10^2	$10\mu S$
10^3	$1mS$
10^4	$100mS$
10^5	$10S$
10^6	$16.7min$
10^7	$27.8hr$
10^8	$116days$

Because of that, we buy a computer which is 10 times faster, which will give us the following table:

N	N^2	N^2 (10x)
10^1	$0.1\mu S$	$0.01\mu S$
10^2	$10\mu S$	$1\mu S$
10^3	$1mS$	$0.1mS$
10^4	$100mS$	$10mS$
10^5	$10S$	$1S$
10^6	$16.7min$	$1.7min$
10^7	$27.8hr$	$2.8hr$
10^8	$116days$	$11.6days$

If we manage to design a new algorithm with a linear growth (i.e. $T(N) = N$), we will get the following table:

N	N^2	N^2 (10x)	N
10^1	$0.1\mu S$	$0.01\mu S$	$10nS$
10^2	$10\mu S$	$1\mu S$	$100nS$
10^3	$1mS$	$0.1mS$	$1\mu S$
10^4	$100mS$	$10mS$	$10\mu S$
10^5	$10S$	$1S$	$0.1mS$
10^6	$16.7min$	$1.7min$	$1mS$
10^7	$27.8hr$	$2.8hr$	$10mS$
10^8	$116days$	$11.6days$	$0.1S$

It's clear that investing in Algorithmic development pays off.

1.108 Faster computer versus faster algorithm

Please read Section 1.2 (p.11–14) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 2

Key Concepts

- Determine time and memory consumption of an algorithm described using pseudocode
- Determine the growth function of the running time or memory consumption of an algorithm
- Use Big-$\mathcal{O}$, Omega and Theta notations to describe the running time or memory consumption of an algorithm.

1.201 Worst and best cases

While computing the running time $T(N)$ of an algorithm as a function of the input size is sufficient for some classes of algorithms, there are other algorithms where the *nature* of the input can also change the running time of the algorithm.

One such example is the **Linear Search** algorithm. Its running time will change according to the input size and the nature of the input.

For example if the value we're looking for is **always** in the first index of the input array, Linear search will run in constant time $\mathcal{O}(1)$ regardless of the input size. If, however, the value we're looking is **never** in the input array, Linear search running grows linearly with the input size.

We can say that the case where the number we're looking is in the first position of the array is the *Best Case* scenario. Conversely, the case where the number we're looking is not in the array is called the *Worst Case* scenario.

1.202 Worst and average cases

Please read p.27 of the guide book, on worst case and average case analysis:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.301 Introduction to asymptotic analysis

Asymptotic analysis is the analysis of the growth of a function as the input size grows larger and larger.

As the input size tends to infinity, the constants and lower-order terms are irrelevant as they provide a very small impact in the function growth behavior.

1.303 Big-$\mathcal{O}$ notation

Big-$\mathcal{O}$ Notation gives us an upper bound to a function growth. For any given function, there is a set of functions that can be considered an upper bound. This is exactly what Big-$\mathcal{O}$ notation defines: a set of functions $g(N)$ that can act as a upper bound for the growth of a function $T(N)$.

More formally, Big-$\mathcal{O}$ is defined as:

$$T(N) \in \mathcal{O}(g(N)) \rightarrow C \cdot g(N) \geq T(N) \forall N \geq n_0$$

Where both C and n_0 are positive constants. In figure 5 we show an example function $10N^2 + 15N + 5$ and two possible upper bounds N^2 and $25N^2$.

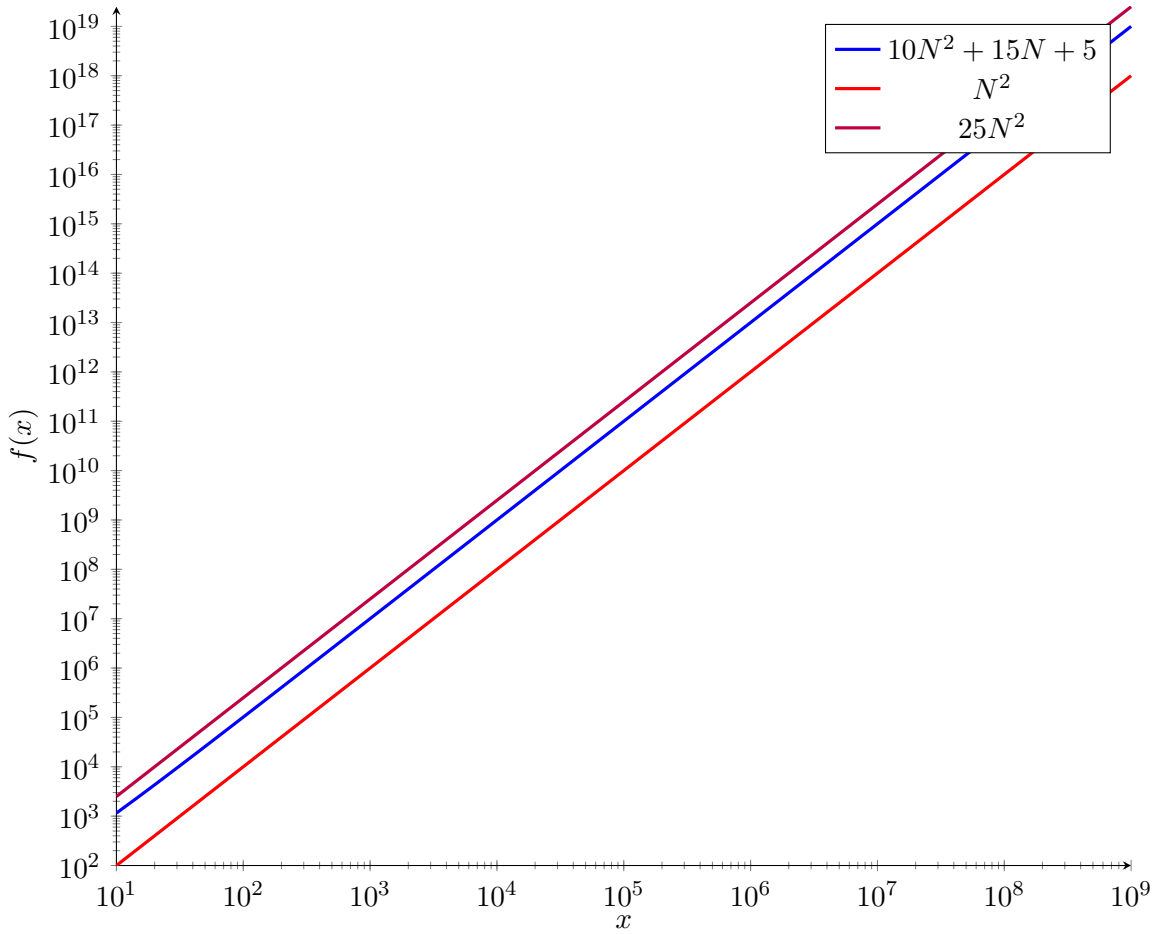

Figure 5: Big-$\mathcal{O}$

We can show the same thing with N^3, N^4, and 2^N. See figure 6 below.

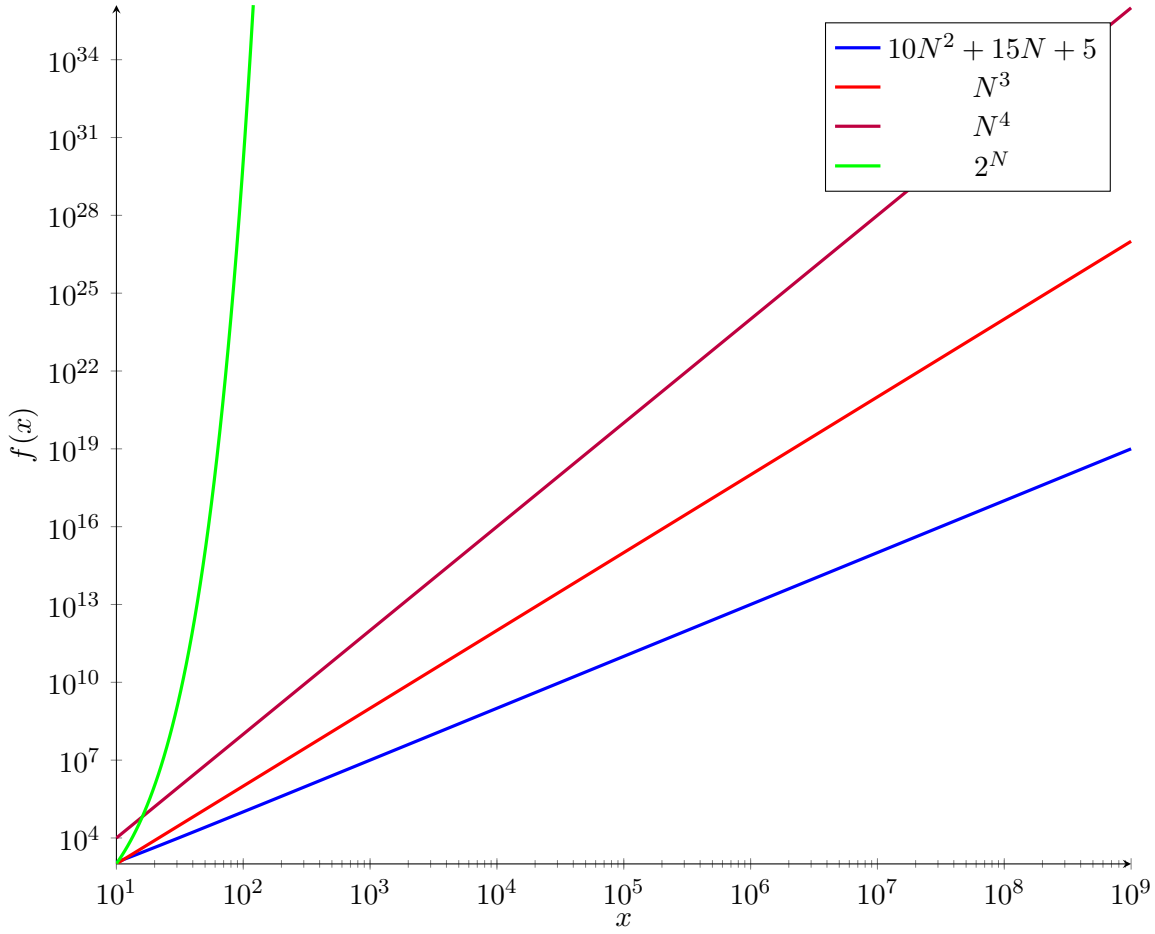

Figure 6: Big-$\mathcal{O}$: N^3, N^4, 2^N

1.305 Omega notation

Big-Ω notation is analogous to Big-$\mathcal{O}$ notation, however instead of looking for upper bounds, we're looking for lower bounds.

Much like Big-$\mathcal{O}$ notation, there are a set of functions that can act as lower bound for a given function. More formally, Big-Ω is defined as:

$$T(N) \in \Omega(g(N)) \rightarrow C \cdot g(N) \leq T(N) \forall N \geq n_0$$

We can produce a similar graph as with Big-$\mathcal{O}$ notation. It's show in figure 7 below.

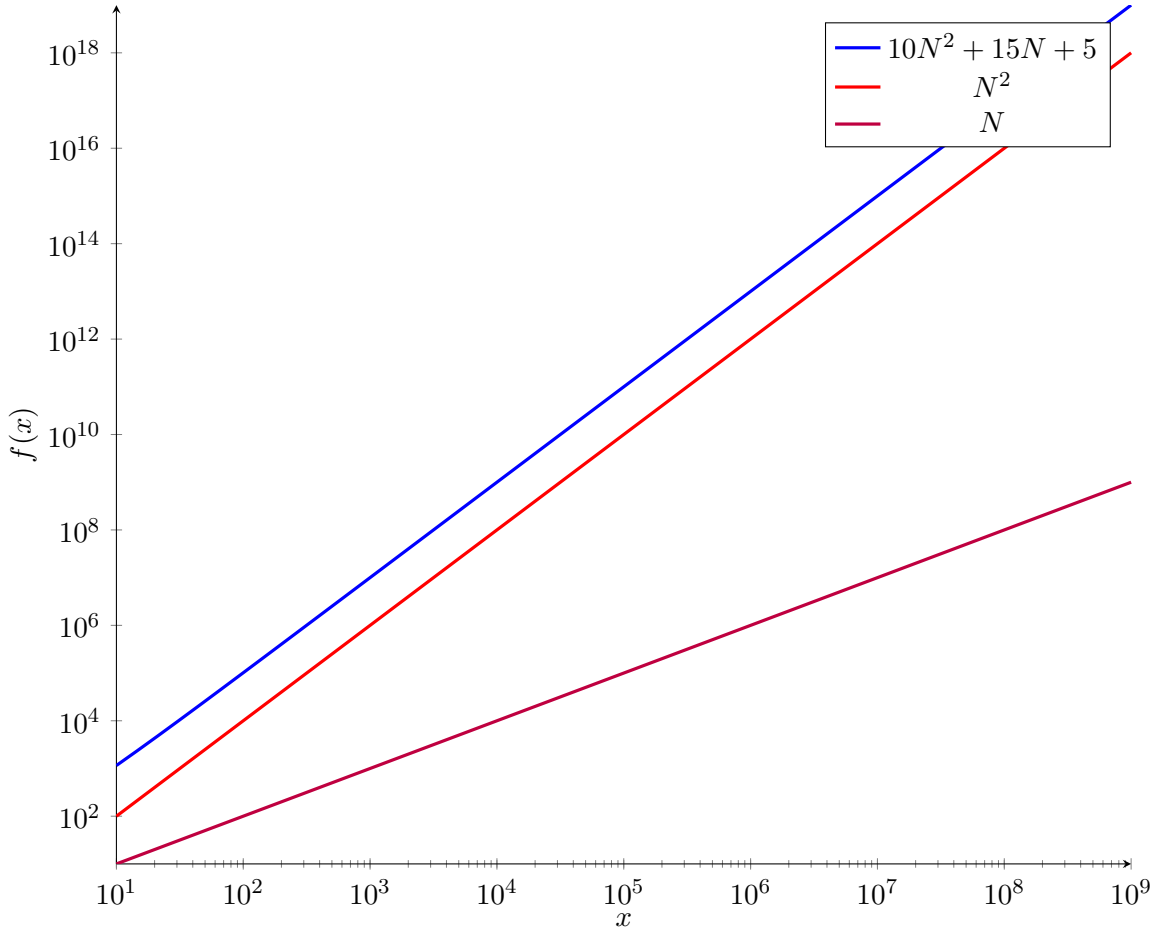

Figure 7: Big-Ω

We can also show that the function $T(N) = 10N^2 + 15N + 5$ is $\Omega(\log N)$ and $\Omega(1)$. See figure 8 below.

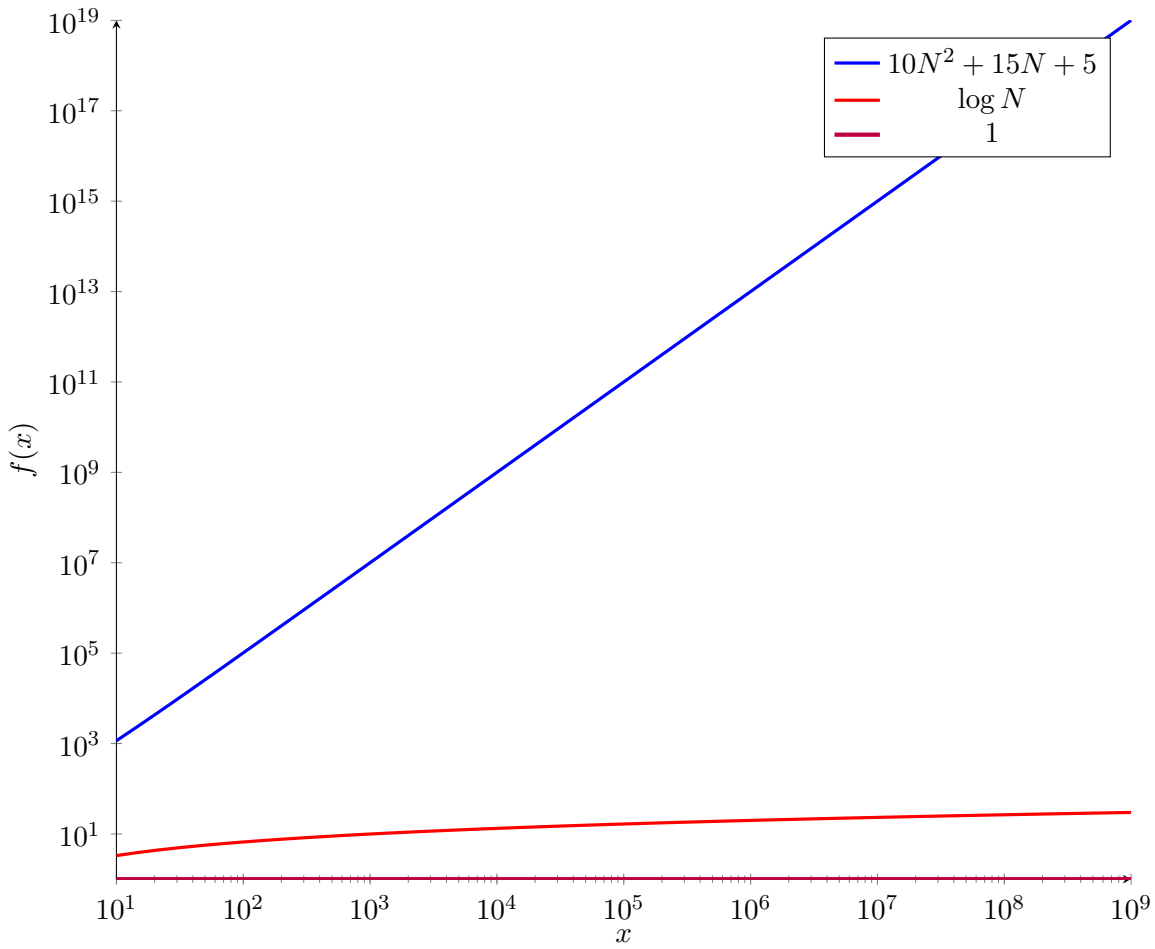

Figure 8: Big-Ω: $\Omega(\log N)$ and $\Omega(1)$

1.307: Theta notation

One drawback of both Big-$\mathcal{O}$ and Big-Ω is that they both refer to a set of functions. This means that when we say that the running time of an algorithm is $\mathcal{O}(N^4)$ it might be that the algorithm grows with N^2 much faster than with N^4, however $\mathcal{O}(N^4)$ is still correct.

With Θ notation, we find a single function that acts as both upper-bound and lower-bound for running time or memory consumption. What we do, in practice, is that we find two different constants c_1 and c_2 such that $c_1 \cdot g(N)$ is a lower bound and $c_2 \cdot g(N)$ is an upper bound. Naturally, $c_1 \leq c_2$.

Figure 9 depicts this:

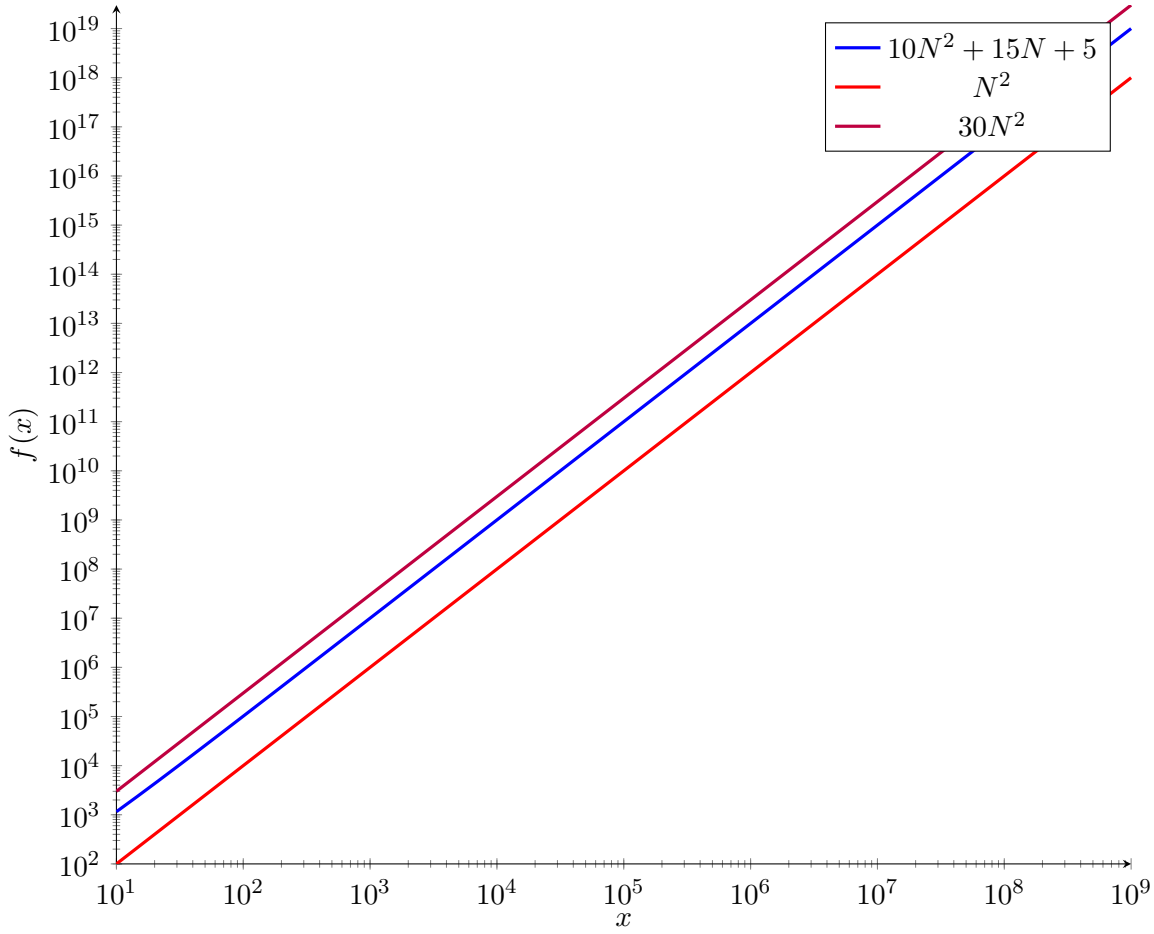

Figure 9: Big-Θ

What we can see is figure 9 is that if $g(N) = N^2$ is multiplied by 1, then it can act as a lower-bound, while if it's multiplied by 30, then it can act as an upper-bound. Therefore $c_1 = 1$ and $c_2 = 30$.

More formally, Big-*Theta* notation is defined as follows

$$T(N) \in \Theta(g(N)) \rightarrow \begin{cases} c_1 \cdot g(N) \geq T(N) \forall N \geq n_0 \\ c_2 \cdot g(N) \leq T(N) \forall N \geq n_0 \end{cases}$$

1.309 Asymptotic notation

Please read Section 3.1 (pp.43–52) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 3

Key Concepts

- Trace and write recursive algorithms
- Write the recursive version of an iterative algorithm using pseudocode
- Calculate the time complexity of recursive algorithms.

2.001 Introduction to recursion

During this week we learn about recursion. The topic is divided into three parts:

1. Understanding Recursion
2. Creating Recursion
3. Analysing Recursion

Recursion happens when an algorithm calls itself. For example, listing 1 is a recursive algorithm:

Algorithm 1 A Simple Recursive Algorithm

```
1: function HELLO
2:     PRINT("hello")          ▷ Print "hello" on the screen
3:     HELLO                   ▷ Recursive call
4: end function
```

The algorithm shown in listing 1 is infinitely recursive, meaning it will never stop with the recursive calls. This is the result of a badly designed recursive algorithm.

2.002 The structure of recursive algorithms

We can modify the previous algorithm so it doesn't recurse infinitely. Algorithm 2 shows the new version of the algorithm.

Algorithm 2 A Better Recursive Algorithm

```
1: function HELLO(n)
2:     if n = 0 then                          ▷ If n = 0...
3:         return                             ▷ We're done
4:     end if
5:     PRINT("hello")                         ▷ Print "hello" on the screen
6:     HELLO(n − 1)                           ▷ Recursive call approaching base case
7: end function
```

The `if` statement in algorithm 2 is called the *Base Case*. We use it to stop the recursion.

As a rule of thumb, recursive algorithms should always include at least one base case and a recursive call approaching the base case.

2.004 Tracing a recursive algorithm

Tracing a recursive algorithm lets us understand what task is accomplished by the algorithm. Algorithm 3 below will be used to demonstrate this.

Algorithm 3 Tracing a recursive algorithm

```
1: function F(a, b)
2:     if b = 0 then
3:         return a
4:     end if
5:     return F(a + 1, b − 1)
6: end function
```

It's clear from the code listing that the base case triggers when b is equal to 0. We can also see that in the recursive call, we're getting closer to 0 by decrementing b by 1 unit. At the same time b is decremented, a is incremented by the same amount.

We can start tracing this algorithm with inputs $2, 2$ respectively for a and b. The first time the algorithm runs, it checks if $b = 0$. Because that check evaluates to false, we move on to the recursive call and change a to 3 and b to 1.

In the recursive call we check if $b = 0$; it isn't, then we move to the recursive call by changing a to 4 and b to 0.

In this new recursive call we check if $b = 0$, it is, then we return the value of a which is 4. That value trickles all the way back to the first call.

In summary, this recursive algorithm calculates $a + b$.

2.101 From iteration to recursion

An iterative algorithm is one that uses a loop to repeat a set of instructions. A recursive algorithm repeats a set of instructions by calling itself.

Algorithm 4 and 5 achieve the same thing, that is printing the numbers from n down to 0. One is iterative while the other is recursive.

Algorithm 4 Iterative Count Down

```
1: function IterCountDown(n)
2:     for i ← n downto 0 do
3:         Print(n)
4:     end for
5: end function
```

Algorithm 5 Recursive Count Down

```
1: function RecCountDown(n)
2:     if n < 0 then
3:         return
4:     end if
5:     Print(n)
6:     RecCountDown(n − 1)
7: end function
```

Both of these algorithms need an initial value, a condition to stop or continue repetition, and a method for updating the value of the variable we're using otherwise we will never stop repeating.

2.103 Writing a recursive algorithm, part 1

When writing a recursive algorithm, we should first treat the recursive call as a black box, for which we only know the result.

By doing that, we limit the amount of information we need to keep track of in order to understand what's happening.

This means that each call is responsible for a small part of the job, with everything being delegated to the recursive call.

2.104 Writing a recursive algorithm, part 2

Applying the technique from the previous section in a recursive linear search algorithm.

The small part the algorithm is going to execute is checking if the value we're looking for is in the last element of the array, if it is we're done, if it isn't, we'll delegate the search in the remaining part of the array.

This would result in an implementation like the one shown in algorithm 6.

Note that we if the value of N is less than 0, we know that we have consumed the entire array or we received an empty array to start with. Therefore, the item wasn't in the array, so we return *FALSE*.

Algorithm 6 Recursive Linear Search

```
1: function RecLinearSearch(A, N, x)
2:     if N < 0 then
3:         return FALSE
4:     end if
5:     if A[N − 1] = x then
6:         return TRUE
7:     end if
8:     return RecLinearSearch(A, N − 1, x)
9: end function
```

Moreover, we're always checking the final value of the array, pointed to by $A[N-1]$. If the value we're searching for is in that position, we return it.

If, however, the value is not there, we recursively call ourselves to process the remaining part of the array. This causes us to reduce N by one at least recursive call and, thus, approximate the base case of an empty array.