

Algorithms and Data Structures II

Course Notes

Felipe Balbi

May 6, 2020

Contents

Week 1	4
1.001 What is analysis of algorithms?	4
1.002 What is analysis of algorithms?	5
1.004 How to measure/estimate time and space requirements	5
1.006 The RAM model	6
1.007 The Ram Model	7
1.009 Counting up time and space units, part 1	7
1.010 Counting up time and space units, part 2	7
1.011 Counting up time and space units	8
1.101 Growth of functions, part 1	8
1.103 Growth of functions, part 2	11
1.105 Growth of functions	12
1.106 Faster computer versus faster algorithm	12
1.108 Faster computer versus faster algorithm	13
Week 2	14
1.201 Worst and best cases	14
1.202 Worst and average cases	14
1.301 Introduction to asymptotic analysis	14
1.303 Big- \mathcal{O} notation	15
1.305 Omega notation	17
1.307 Theta notation	19
1.309 Asymptotic notation	20
Week 3	22
2.001 Introduction to recursion	22
2.002 The structure of recursive algorithms	22
2.004 Tracing a recursive algorithm	23
2.101 From iteration to recursion	23
2.103 Writing a recursive algorithm, part 1	24
2.104 Writing a recursive algorithm, part 2	24
Week 4	26
2.201 Time complexity of recursive algorithms	26
2.203 Solving recurrence equations	27
2.301 The master theorem	27
2.303 Recursive algorithms and their analysis	28

Contents

Week 5	29
3.001 Comparison and non comparison sorting algorithms	29
3.004 Bubble sort: Pseudocode	29
3.102 Insertion sort: Pseudocode	29
3.104 Insertion sort	31
3.105 Selection sort: Pseudocode	31
Week 6	32
3.202 Quicksort: Pseudocode	32
3.204 Quicksort	33
3.302 Mergesort: Pseudocode	33
3.305 Mergesort	34
Week 7	36
4.001 The limits of comparison sorts	36
4.002 Lower bounds for commparison sorts	38
4.101 Counting sort: Introduction	38
4.102 Counting sort: Pseudocode	40
4.104 Counting sort	40

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 amount of resources are better candidates than those requiring more resources.

During this course, 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  $1 \leq j < \text{LENGTH}(s)$  do
3:      $key \leftarrow s[j]$ 
4:      $i \leftarrow j - 1$ 
5:     while  $i \geq 0 \wedge s[i] > key$  do
6:        $s[i + 1] \leftarrow s[i]$ 
7:        $i \leftarrow i - 1$ 
8:     end while
9:      $s[i + 1] \leftarrow key$ 
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.

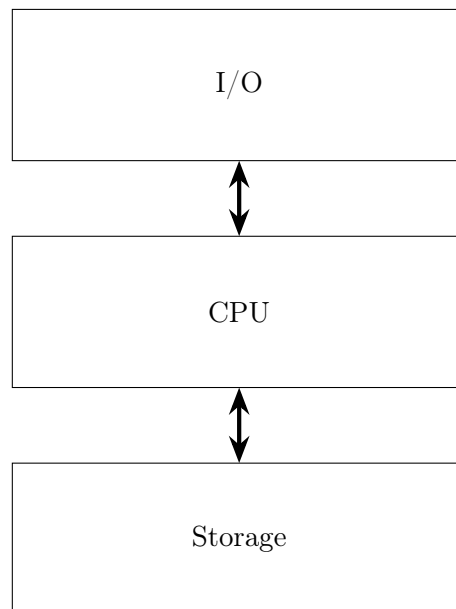


Figure 1: Random-access Machine Model

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 to 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: $max \leftarrow a$	▷ 1 memory read, 1 memory write
3: if $b > max$ then	▷ 1 conditional, 1 comparison, 2 memory reads
4: $max \leftarrow b$	▷ 1 memory read, 1 memory write
5: end if	
6: if $c > max$ then	▷ 1 conditional, 1 comparison, 2 memory reads
7: $max \leftarrow c$	▷ 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 elements in A , and x is an integer. The pseudocode is found in algorithm .

```

1: function F2( $A, N, x$ )
2:   for  $0 \leq i < N$  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:  $i \leftarrow 0$                                 ▷ 1 memory write
2: if  $i < N$  then                                ▷ 1 cond., 2 memory reads, 1 comparison
3:   <instructions>
4:    $i \leftarrow i + 1$                         ▷ 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.

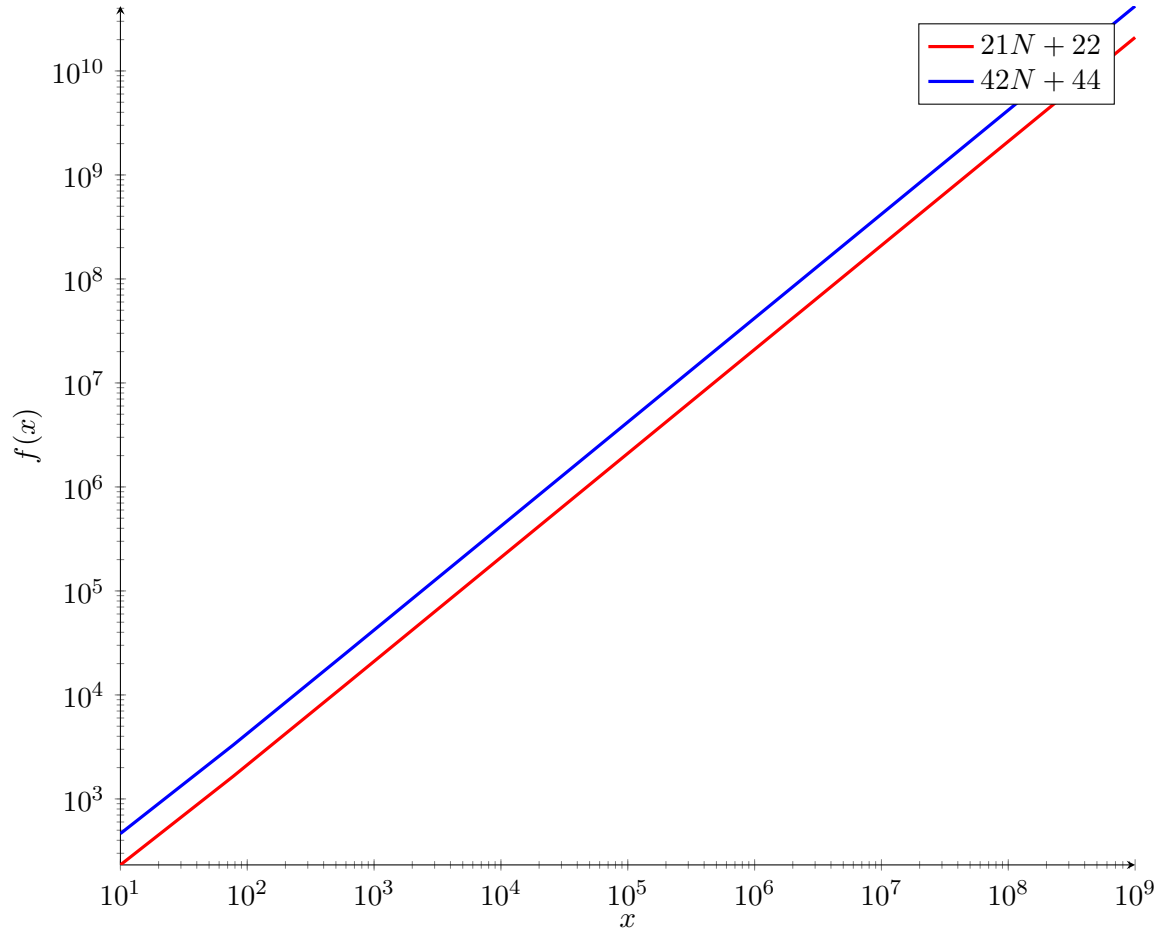


Figure 2: Running Time

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. $T(N) = C_1N + C_2$

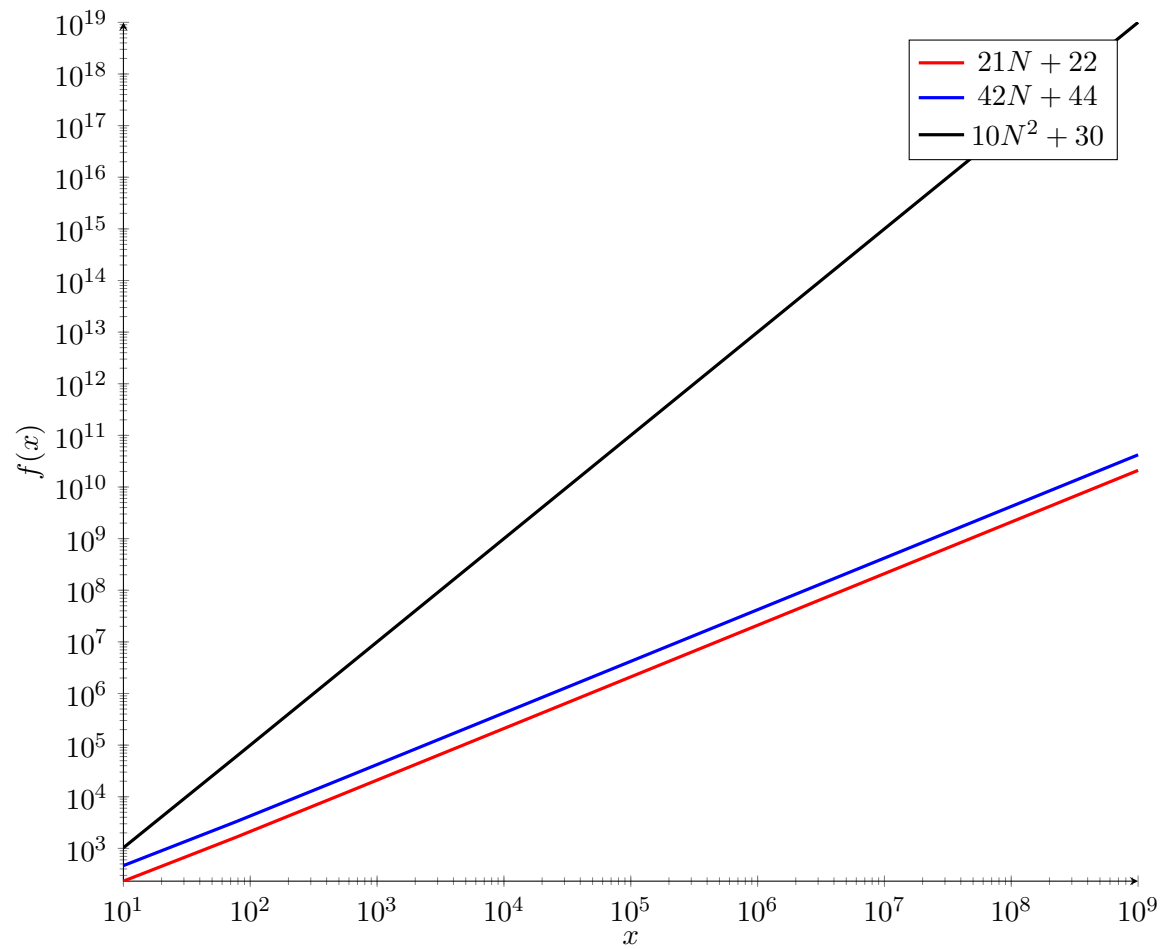


Figure 3: Running Time

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.

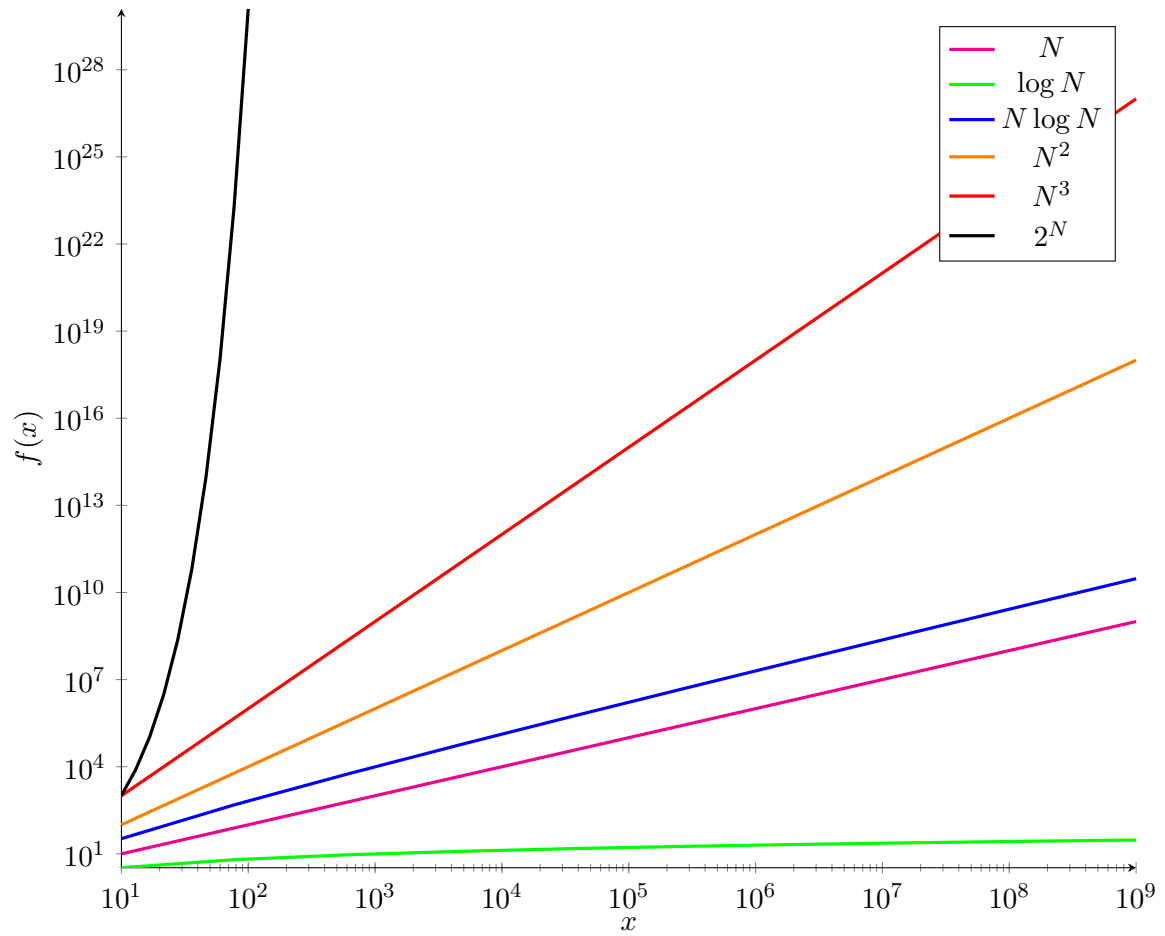


Figure 4: Running Time

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: $sum \leftarrow 0$	$\triangleright C_0$
3: $N \leftarrow \text{LENGTH}(A[0])$	$\triangleright C_1N + C_2$
4: for $0 \leq i < N$ do	$\triangleright C_3N + C_4$
5: $sum \leftarrow sum + A[i, i]$	$\triangleright C_5N$
6: end for	
7: return sum	$\triangleright C_6$
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:

Week 1

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 for 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 for is in the first position of the array is the *Best Case* scenario. Conversely, the case where the number we're looking for 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 an 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 $25N^2$.

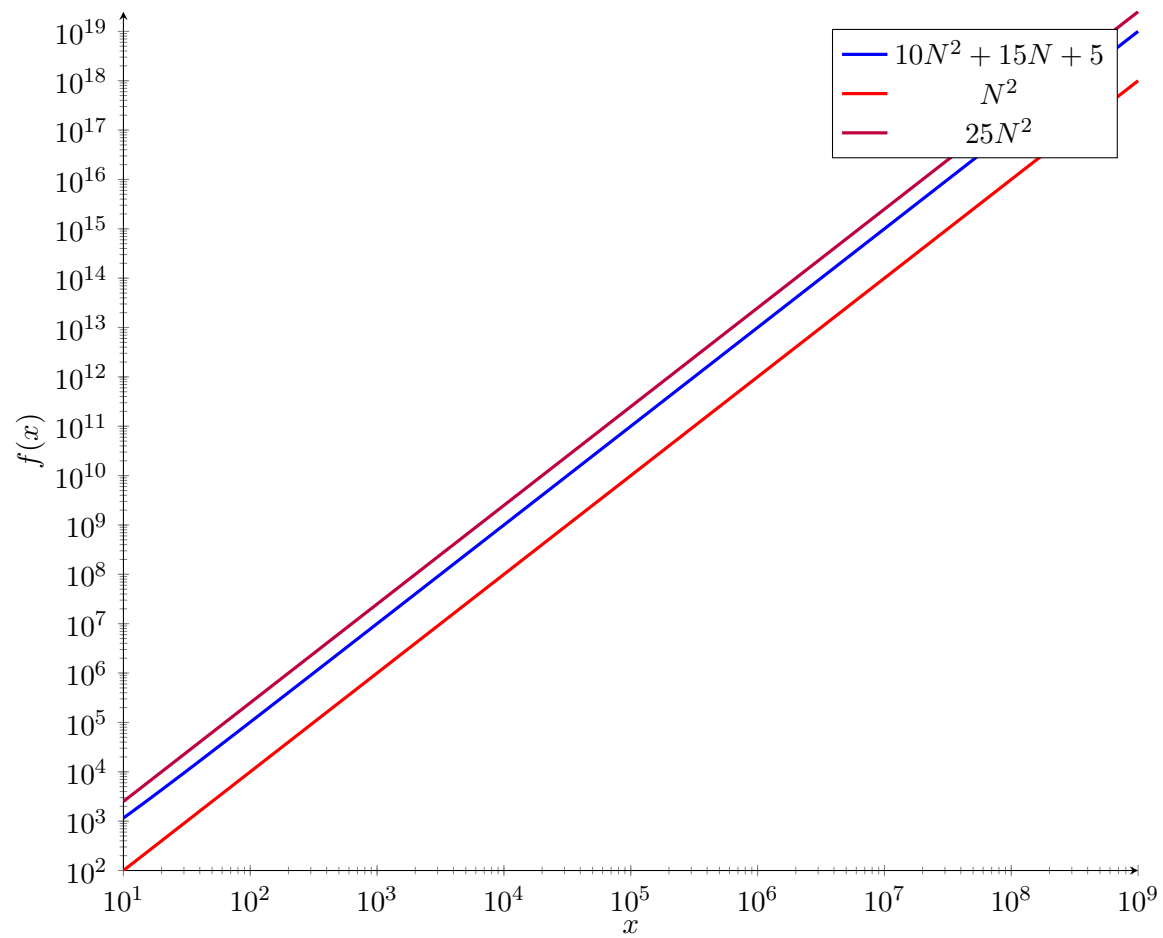


Figure 5: Big- \mathcal{O}

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

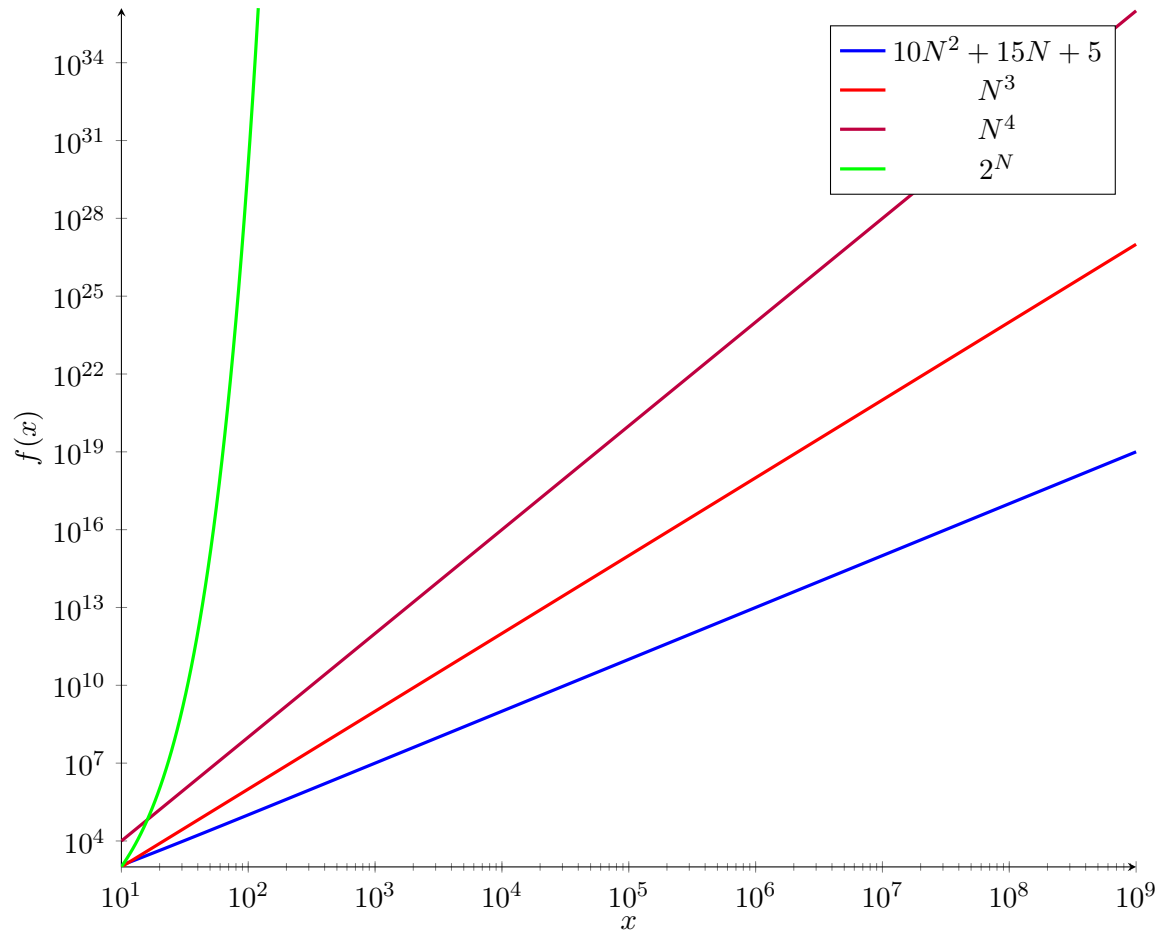


Figure 6: Big- \mathcal{O} : N^3 , N^4 , 2^N

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.

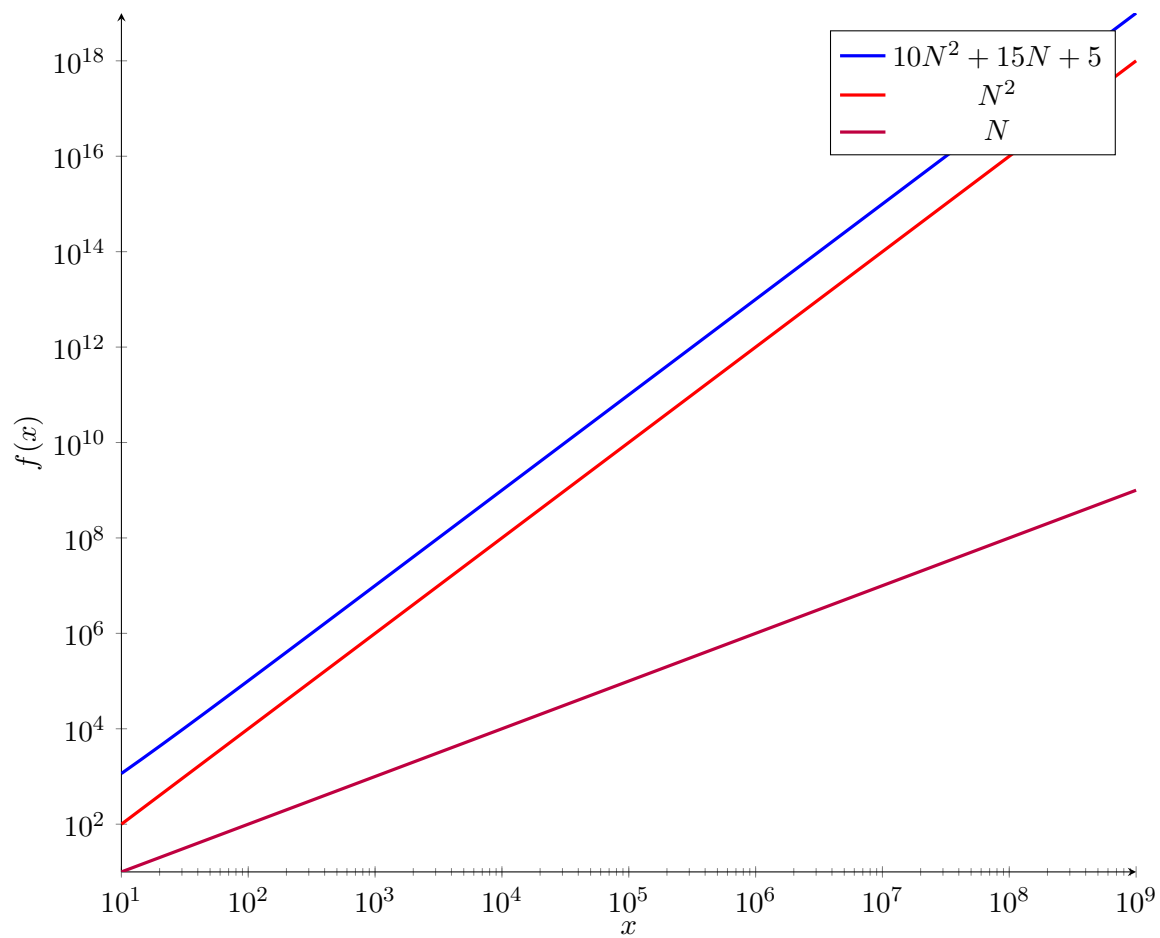
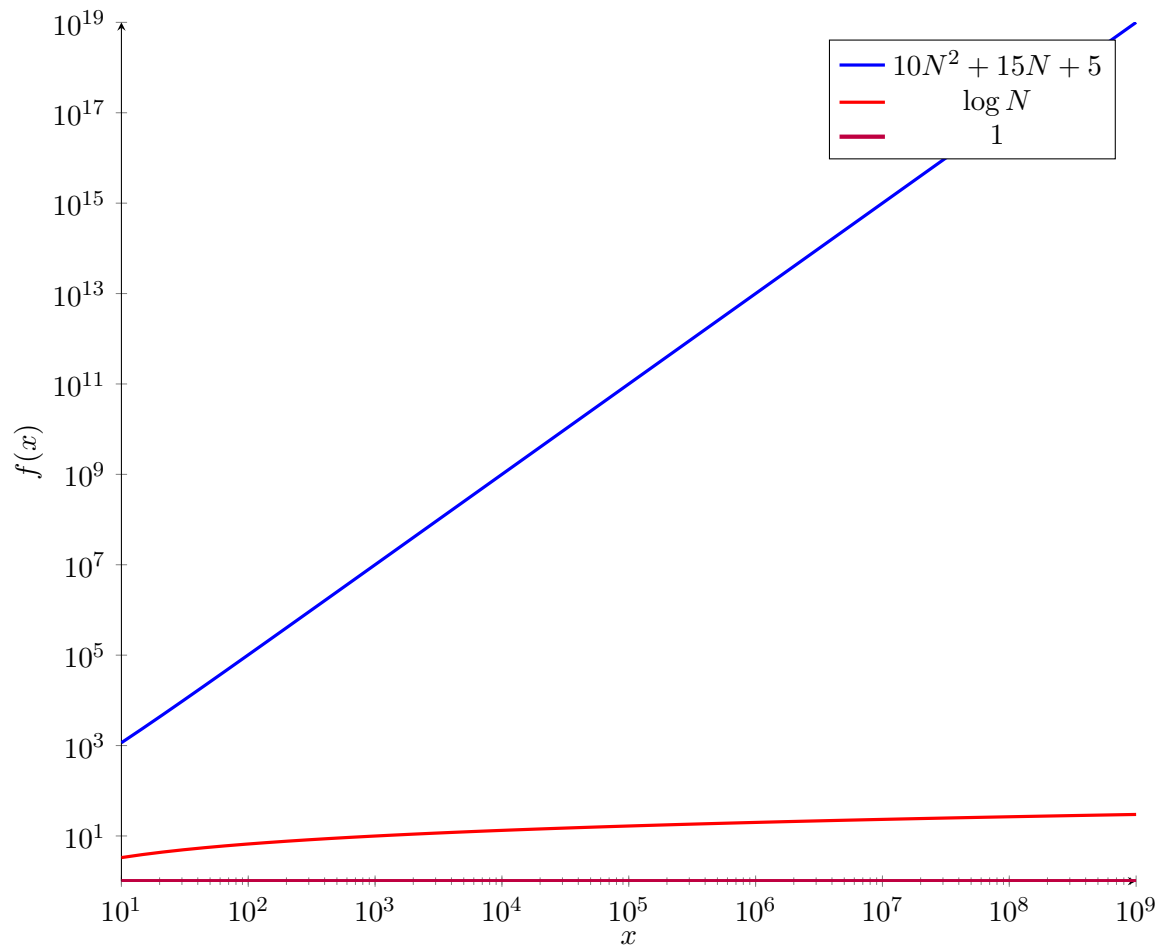


Figure 7: Big- Ω

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:

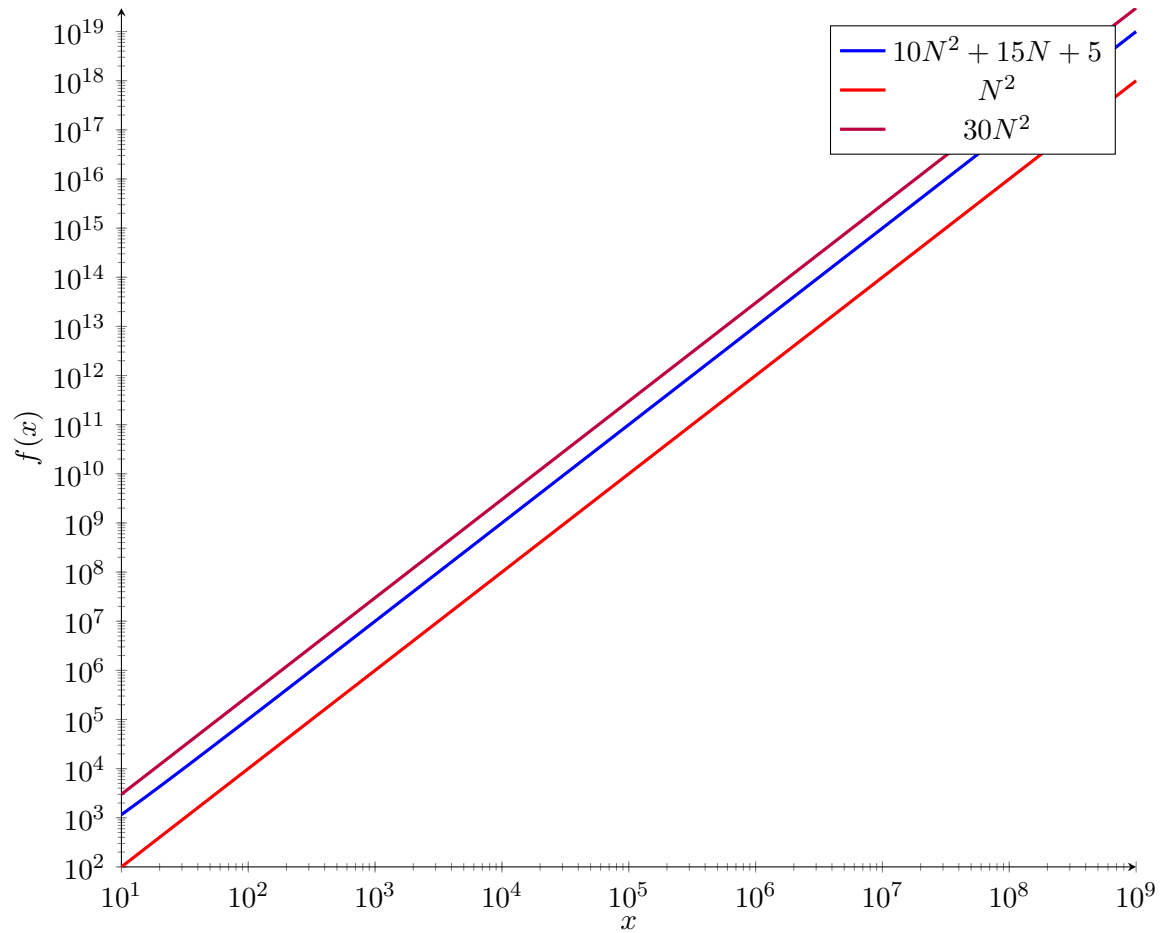


Figure 9: Big- Θ

What we can see in 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].

Week 2

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  $i \leftarrow n$  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.

Week 4

Key Concepts

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

2.201 Time complexity of recursive algorithms

The time complexity of an algorithm is the asymptotic number of simple operations executed by the algorithm. We can apply the same analysis to recursive algorithms.

As an example, we use the **Factorial** function whose pseudocode is shown in listing 7:

Algorithm 7 Factorial Function

```
1: function FACTORIAL( $n$ )
2:   if  $n \leq 1$  then
3:     return 1
4:   end if
5:   return  $n \times$  FACTORIAL( $n - 1$ )
6: end function
```

We can annotate this algorithm with the cost of each line, seen below in listing 8

Algorithm 8 Factorial Function Annotated With Cost

```
1: function FACTORIAL( $n$ )  $\triangleright T(N)$ 
2:   if  $n \leq 1$  then  $\triangleright C_0$ 
3:     return 1
4:   end if
5:   return  $n \times$  FACTORIAL( $n - 1$ )  $\triangleright C_4 + T(N - 1)$ 
6: end function
```

With that we can extract the expression:

$$T(N) = C_0 + C_4 + T(N - 1)$$

$$T(N) = C_5 + T(N - 1)$$

We can see that the running time of $T(N)$ depends on the running time of the $T(N-1)$, we refer to this type of equation as *Recurrence Equation*.

2.203 Solving recurrence equations

The main problem with a recurrence equation is that we don't have an explicit expression for the running time of an algorithm.

To solve a recurrence equation we follow a two-step process:

1. Find a value of N for which $T(N)$ is known. Usually, this can be achieved with the running time of the best-case scenario input.
2. Expand the right side of the recurrence equation until you can't replace the known value of $T(N)$ on it anymore.

For example, looking back at algorithm 7 we can see that the best case is achieved when the number 1 is our argument. In such a case, the conditional statement evaluates to true which causes the algorithm to immediately return. Both instructions, i.e the **if** and the **return** execute in constant time, therefore our best case runs in constant time. We conclude that $T(1) = C$. With that in mind, we can start to expand the right side of the expression:

$$\begin{aligned}
 T(N) &= C_5 + T(N-1) \\
 T(N) &= C_5 + C_5 + T(N-2) \\
 T(N) &= C_5 + C_5 + C_5 + T(N-3) \\
 T(N) &= C_5 + C_5 + C_5 + C_5 + T(N-4) \\
 T(N) &= C_5 + C_5 + C_5 + C_5 + \dots + T(1) \\
 T(N) &= C_5 + C_5 + C_5 + C_5 + \dots + C \\
 T(N) &= (N-1)C_5 + C
 \end{aligned}$$

Now that the recurrence equation is known, we can do an asymptotic analysis for $T(N)$:

Big- \mathcal{O} $\mathcal{O}(N), \mathcal{O}(N^2), \mathcal{O}(N^3), \dots$

Big- Ω $\Omega(N), \Omega(\log N), \Omega(1), \dots$

Big- Θ $\Theta(N)$

2.301 The master theorem

The Master Theorem is a simpler way of executing asymptotic analysis, however it can't be applied to every recurrence equation.

In order to apply the Master Theorem, the recurrence equation must be of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$.

When the Master Theorem can be applied, there are three cases to take into account:

1. $f(n) < n^{\log_b a}$

In this case, $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = n^{\log_b a}$

In this case, $T(n) = \Theta(n^{\log_b a} \log n)$

3. $f(n) > n^{\log_b a}$

For this case to be applicable, there is one extra requirement to be met: $a \cdot f(\frac{n}{b}) \leq c$, where $c < 1$ and n is large. In this case, $T(n) = \Theta(f(n))$

2.303 Recursive algorithms and their analysis

Please read:

- Section 2.3 (pp.29–37), only if you are familiar with Mergesort. If not, we will review this section again later
- Chapter 4, pp.65–113 (except section 4.6)

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 5

Key Concepts

- Identify the different approaches of different comparison sorting algorithms
- Implement different comparison sorting algorithms
- Calculate the time complexity of different comparison sorting algorithms

3.001 Comparison and non comparison sorting algorithms

Sorting algorithms can be split into two main categories: Comparison Sorts and Non-comparison Sorts.

We can quickly build a simple tree showing the main algorithms:

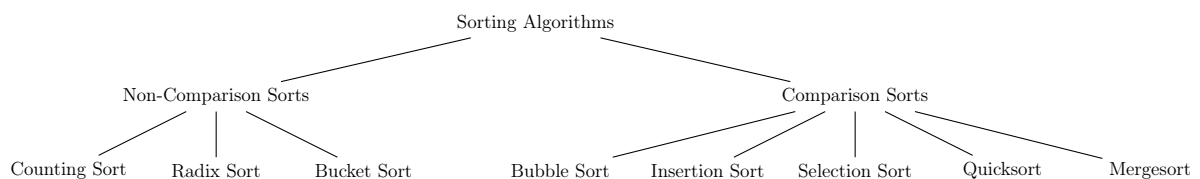


Figure 10: Sorting Algorithms

The difference between them is that comparison sorts will compare two elements to decide the order, while non-comparison sorts will not.

Comparison sorts have a limit on their worst-case time complexity; they can never be faster than $N \log N$ while non-comparison sorts do not suffer from this limitation.

Table 0.1 below provides a summary of worst- and best-case time complexity of the comparison sorts listed above.

3.004 Bubble sort: Pseudocode

We can see bubble sort pseudocode in algorithm 9.

3.102 Insertion sort: Pseudocode

We can see insertion sort pseudocode in algorithm 10.

Table 0.1: Comparison Sorts Complexity

Algorithm	Worst-case	Best-case
Bubble	$\Theta(N^2)$	$\Theta(N)$
Insertion	$\Theta(N^2)$	$\Theta(N)$
Selection	$\Theta(N^2)$	$\Theta(N^2)$
Quicksort	$\Theta(N^2)$	$\Theta(N \log N)$
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$

Algorithm 9 Bubble Sort

```

1: function BUBBLESORT( $A, N$ )
2:    $swapped \leftarrow \mathbf{true}$ 
3:   while  $swapped$  do
4:      $swapped \leftarrow \mathbf{false}$ 
5:     for  $0 \leq i < N - 1$  do
6:       if  $A[i] > A[i + 1]$  then
7:         SWAP( $A[i], A[i + 1]$ )
8:          $swapped \leftarrow \mathbf{true}$ 
9:       end if
10:    end for
11:     $N \leftarrow N - 1$ 
12:  end while
13:  return  $A$ 
14: end function

```

Algorithm 10 Insertion Sort

```

1: function INSERTIONSORT( $A, N$ )
2:   for  $1 \leq j < N - 1$  do
3:      $ins \leftarrow A[j]$ 
4:      $i \leftarrow j - 1$ 
5:     while  $i \geq 0 \wedge ins < A[i]$  do
6:        $A[i + 1] \leftarrow A[i]$ 
7:        $i \leftarrow i - 1$ 
8:     end while
9:      $A[i + 1] \leftarrow ins$ 
10:  end for
11:  return  $A$ 
12: end function

```

3.104 Insertion sort

Please read Sections 2.1 (pp.16–22) and 2.2 (pp.23–9) 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](#).

3.105 Selection sort: Pseudocode

We can see selection sort pseudocode in algorithm 11.

Algorithm 11 Selection Sort

```
1: function SELECTIONSORT( $A, N$ )  
2:   for  $1 \leq i < N - 1$  do  
3:      $min \leftarrow \text{POSMIN}(A, i, N - 1)$   
4:     SWAP( $A[i], A[min]$ )  
5:   end for  
6:   return  $A$   
7: end function
```

Week 6

Key Concepts

- Identify the different approaches of different comparison sorting algorithms
- Implement different comparison sorting algorithms
- Calculate the time complexity of different comparison sorting algorithms

3.202 Quicksort: Pseudocode

Quicksort is a comparison sorting algorithm that's very simple to implement if we use recursion.

Listing 12 shows the pseudocode for Quicksort.

Algorithm 12 Quicksort

```
1: function QUICKSORT( $A, low, high$ )
2:   if  $low < high$  then
3:      $p \leftarrow \text{PARTITION}(A, low, high)$ 
4:     QUICKSORT( $A, low, p - 1$ )
5:     QUICKSORT( $A, p + 1, high$ )
6:   end if
7: end function
```

We see that Quicksort calls itself twice during its execution. It does this by partitioning the input array into two smaller arrays of roughly half the size. The function **Partition** is responsible for selecting a *pivot*, moving all numbers lower than *pivot* to the left side of the array, and moving the *pivot* to its final position.

As any recursive algorithm, Quicksort requires a base case. In the pseudocode above, the base case is implicit in the **else** part of the **if** condition. Note that if $low \geq high$ the algorithm will stop executing.

We should write the pseudocode for the **Partition** function. The requirements are:

1. Select number in position **high** as the pivot
2. Move all numbers lower than pivot to the left of the array
3. Return the pivot

The pseudocode may look like the one shown in listing 13.

The **Swap** function is a simple helper to swap the i^{th} and j^{th} elements of the array A . Its pseudocode is shown in listing 14.

Algorithm 13 Partition

```

1: function PARTITION( $A, low, high$ )
2:    $p \leftarrow A[high]$ 
3:    $i \leftarrow low$ 
4:   for  $low \leq j < high$  do
5:     if  $A[j] \leq p$  then
6:       SWAP( $A, i, j$ )
7:        $i \leftarrow i + 1$ 
8:     end if
9:   end for
10:  SWAP( $A, i, high$ )
11:  return  $i$ 
12: end function

```

Algorithm 14 Swap

```

1: function SWAP( $A, i, j$ )
2:    $t \leftarrow A[i]$ 
3:    $A[i] \leftarrow A[j]$ 
4:    $A[j] \leftarrow t$ 
5: end function

```

3.204 Quicksort

Please read the introduction to Chapter 7, Section 7.1 (pp.170–3) and Section 7.2 (p.174–8) 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].

Available from [here](#).

3.302 Mergesort: Pseudocode

Mergesort is another comparison sorting algorithm that's very easy to implement if we use recursion. There are several possible implementations of Mergesort depending on:

1. The data structure used
2. The way the merge part works

During this section, we use the Array data structure and an out-of-place merge, which means that we will allocate extra memory during the merge operation. This gives us a better time complexity.

Listing 15 contains the pseudocode for Mergesort.

Much like Quicksort, the base case for Mergesort is implicit in the **if** condition. We can see that whenever $l \geq h$ the algorithm won't do anything and simply return.

Algorithm 15 Mergesort

```

1: function MERGESORT( $A, l, h$ )
2:   if  $l < h$  then                                ▷ Should continue?
3:      $mid \leftarrow \lfloor \frac{h+l}{2} \rfloor$                 ▷ Midpoint calculation
4:     MERGESORT( $A, l, mid$ )                          ▷ Sort left half
5:     MERGESORT( $A, mid + 1, h$ )                      ▷ Sort right half
6:     MERGE( $A, l, mid, h$ )                            ▷ Merge left and right halves
7:   end if
8: end function

```

The midpoint between l and h is calculated by line 3. Right after calculating the midpoint, we execute our first recursive call to Mergesort. This will try to sort the left half of the array, this can be seen in line 4. What follows is a recursive call to Mergesort to operate on the right side of the array, as seen in line 5. When this is complete, both halves of the array will be sorted. The only thing left to do is to merge both halves maintaining the order. This can be seen in line 6.

We must write the pseudocode for the **Merge** function. The requirements are:

1. Copy the already sorted elements between l and mid into a new array called L .
2. Copy the already sorted elements between $mid + 1$ and r into a new array called R .
3. Compare first elements of L and R , smallest goes back into A . Repeat until both L and R are empty.

A possible implementation of **Merge** is provided in listing 16.

We should also calculate the worst- and best-case time complexity of Mergesort.

3.305 Mergesort

Please read Sections 2.3.1 (pp.30–34) and 2.3.2 (pp.34–37) 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](#).

Algorithm 16 Merge

```

1: function MERGE( $A, l, mid, h$ )
2:    $L \leftarrow A[l \dots mid]$ 
3:    $R \leftarrow A[mid + 1 \dots h]$ 
4:    $i \leftarrow 0$ 
5:    $j \leftarrow 0$ 
6:    $k \leftarrow l$ 
7:   while  $i \leq mid \wedge j < (h - mid)$  do
8:     if  $L[i] \leq R[j]$  then
9:        $A[k] \leftarrow L[i]$ 
10:       $i \leftarrow i + 1$ 
11:    else
12:       $A[k] \leftarrow R[j]$ 
13:       $j \leftarrow j + 1$ 
14:    end if
15:     $k \leftarrow k + 1$ 
16:  end while
17:  while  $i \leq mid$  do
18:     $A[k] \leftarrow L[i]$ 
19:     $i \leftarrow i + 1$ 
20:     $k \leftarrow k + 1$ 
21:  end while
22:  while  $j < (h - mid)$  do
23:     $A[k] \leftarrow R[j]$ 
24:     $j \leftarrow j + 1$ 
25:     $k \leftarrow k + 1$ 
26:  end while
27: end function

```

Week 7

Key Concepts

- Identify the different approaches of different non-comparison sorting algorithms
- Implement different non-comparison sorting algorithms
- Calculate the time complexity of different non-comparison sorting algorithms

4.001 The limits of comparison sorts

We have, thus far, reviewed 5 comparison sorts:

Bubble Sort compares pairs of elements and swaps them if they are in the wrong order

Insertion Sort finds the correct position in which to insert the next unsorted element in the array

Select Sort selects the minimum value in the unsorted part of the array and stores it at the beginning of the unsorted part

Quicksort recursively selects a pivot and stores it in its final correct position

Merge Sort divides the array in halves until individual elements are left which are then merged back in sorted order

During analysis of the worst-case time complexity of these algorithms, we found out that they will never perform better than $\Theta(N \log N)$, where N is the number of elements in the array.

We can get an idea for why this is the case with a simple thought exercise. Let's assume that we have an array with N unsorted numbers. There are exactly $N!$ ways of arranging the numbers in the array. Among all the different arrangements, only 1 is the correct order. Now, the question we're asking is "what is the **maximum number of comparisons** a sorting algorithm must do to find the correct arrangement of numbers?"

Taking a 3-element array 11 as an example:

The first comparison happens between $A[0]$ and $A[1]$, essentially we're testing if the 0th element is smaller than the 1st element. If the answer is *yes*, the elements are already sorted. If the answer is *no*, then we need to put them in the correct order. This will continue until we process the entire array. We can build a decision tree 12 of all these cases:

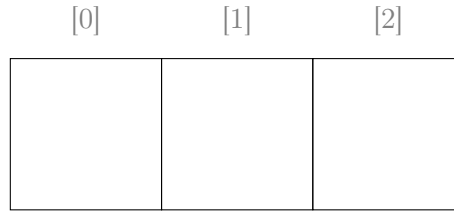


Figure 11: 3-element Array

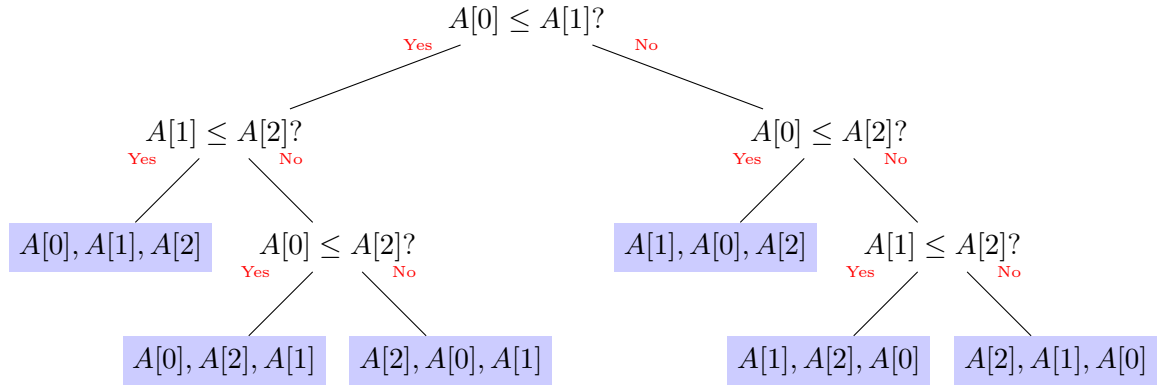


Figure 12: Decision Tree

From decision tree 12, we can conclude that:

1. there are exactly $N!$ leaves in the tree (the ones colored blue);
2. the maximum number of comparisons is the length of the longest path in the tree;
3. there are at most 2^L leaves in this tree.

We also know that actual number of leaves cannot be greater than the maximum number of possible leaves, therefore:

$N! \leq 2^L$ $\log N! \leq \log 2^L$ $\frac{\log N!}{\log 2} \leq L$ $L = \Omega(\log N!)$ $N! \approx N^N$ $L = \Omega(\log N^N)$ $L = \Omega(N \log N)$	$N! \text{ is at most } 2^L$ <p>Applying log to both sides</p> <p>Dividing by log 2</p> <p>Applying asymptotic notation</p> <p>Stirling's approximation</p> <p>Substituting $N!$ for N^N</p> <p>Logarithm rule</p>
--	--

4.002 Lower bounds for comparison sorts

Please read the introduction to Chapter 8 and Section 8.1 (pp.191–3) 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](#).

4.101 Counting sort: Introduction

Counting sort is a non-comparison sort with a linear worst-case running time.

Let's assume we're sorting numbers within the range 0 through 9, both inclusive. We can build a frequency array of ten items where the value at the index k is the number of times the number k appears in the set of numbers we're trying to sort.

Like shown in figure 13:

1	2	0	0	0	1	3	1	1	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 13: Counting Sort: Array C

Figure 13 tells us that the number 0 appears once in the set of numbers, the number two is not part of the set of numbers, and the number 6 appears three times.

Given array C, we can find out what the array of sorted numbers looks like, that's shown in figure 14 below:

0	1	1	5	6	6	6	1	1
---	---	---	---	---	---	---	---	---

Figure 14: Counting Sort: Array R

Array R is sorted and we never did a single comparison to produce it. All we did was visit every element in Array C and place as many copies as indicated in R.

Counting sort works in a very similar fashion. Given an input argument A (an unsorted array) we must:

1. Create array C
 - a) Find maximum value in array A
 - b) Create array C with $(k+1)$ elements, where k is the maximum value in A
 - c) Traverse array A and update frequencies in C

2. Create array R

- a) Create array R with the same length as A
- b) Traverse array C and copy corresponding elements as many times as required.

Figure 15 below shows a depiction of the process:

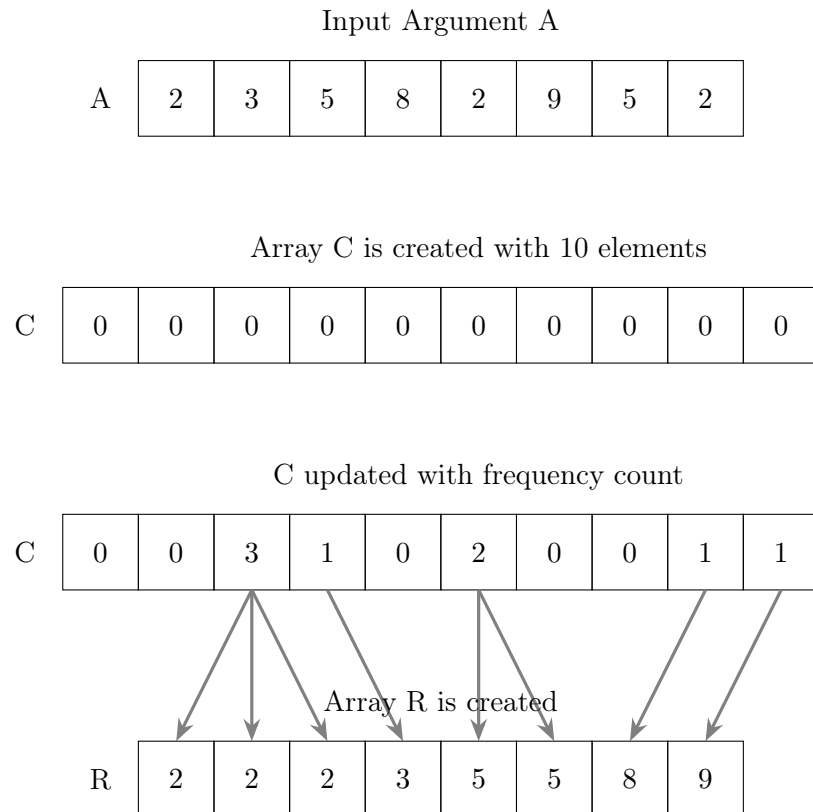


Figure 15: Counting Sort Execution

While the counting sort algorithm has a very desirable linear time complexity, there are two main drawbacks:

- It can only sort integer numbers** Because the algorithm uses the array indices to represent the numbers to sort and indices must be integer
- C must have as many elements as the $\text{max} + 1$** The extra memory used by array C can be significant if the maximum number in the set of numbers to represent is too big. For example, what happens if we have to sort an array with 10 elements where the maximum is 10^9 ?

4.102 Counting sort: Pseudocode

The pseudocode for Counting is as shown in listing 17.

Algorithm 17 Counting Sort Pseudocode

```

1: function COUNTINGSORT( $A, k$ )
2:    $C \leftarrow \text{new Vector}[k + 1]$  ▷ We assume vector is zero-initialized
3:    $R \leftarrow \text{new Vector}[\text{LENGTH}(A)]$ 
4:    $pos \leftarrow 0$ 
5:   for  $0 \leq j < \text{LENGTH}(A)$  do
6:      $C[A[j]] \leftarrow C[A[j]] + 1$ 
7:   end for
8:   for  $0 \leq i < k + 1$  do
9:     for  $pos \leq r < pos + C[i]$  do
10:       $R[r] \leftarrow i$ 
11:    end for
12:     $pos \leftarrow r$ 
13:  end for
14:  return  $R$ 
15: end function

```

The algorithm receives as input an unsorted array A and the maximum value stored in array A k . Arrays C and R are allocated and zero-initialized. A variable pos used to indicate the position in array R is also declared and initialized to zero.

The first *for* loop is responsible for counting frequencies of numbers in array A and storing them in array C . The second *for* loop is responsible for updating array R with the final sorted numbers.

4.104 Counting sort

Please read Section 8.2 (p.194–6) 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](#).