Algorithms and Data Structures II
Course Notes

Felipe Balbi

May 23, 2020

Contents

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:

```
function F($arrays$)
    for $1 \le j <$ LENGTH($s$) do
        $key \leftarrow s[j]$
        $i \leftarrow j - 1$
        while $i \ge 0 \wedge s[i] > key$ do
            $s[i+1] \leftarrow s[i]$
            $i \leftarrow i - 1$
        end while
        $s[i+1] \leftarrow key$
    end for
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.

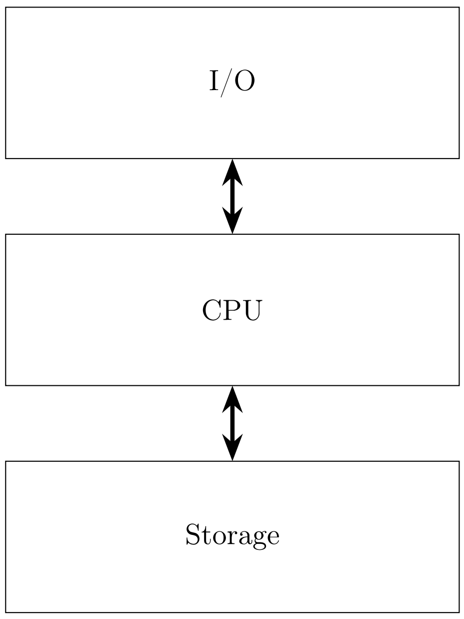

Figure 1: Random-access Machine Model

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.

```
function F1($a, b, c$)
    $max \leftarrow a$                ▷ 1 memory read, 1 memory write
    if $b > max$ then                 ▷ 1 conditional, 1 comparison, 2 memory reads
        $max \leftarrow b$            ▷ 1 memory read, 1 memory write
    end if
    if $c > max$ then                 ▷ 1 conditional, 1 comparison, 2 memory reads
        $max \leftarrow c$            ▷ 1 memory read, 1 memory write
    end if
    return $max$                      ▷ 1 memory read, 1 return
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 .

```
function F2($A, N, x$)
    for $0 \leq i < N$ do
        if $A[i] = x$ then        ▷ 1 cond., 1 array access, 1 comparison, 2 memory reads
            return $i$            ▷ 1 return, 1 memory read
        end if
    end for
    return $-1$                   ▷ 1 return
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:

```
$i \leftarrow 0$                  ▷ 1 memory write
if $i < N$ then                   ▷ 1 cond., 2 memory reads, 1 comparison
    <instructions>
    $i \leftarrow i + 1$          ▷ 1 memory write, 1 memory read, 1 addition
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.

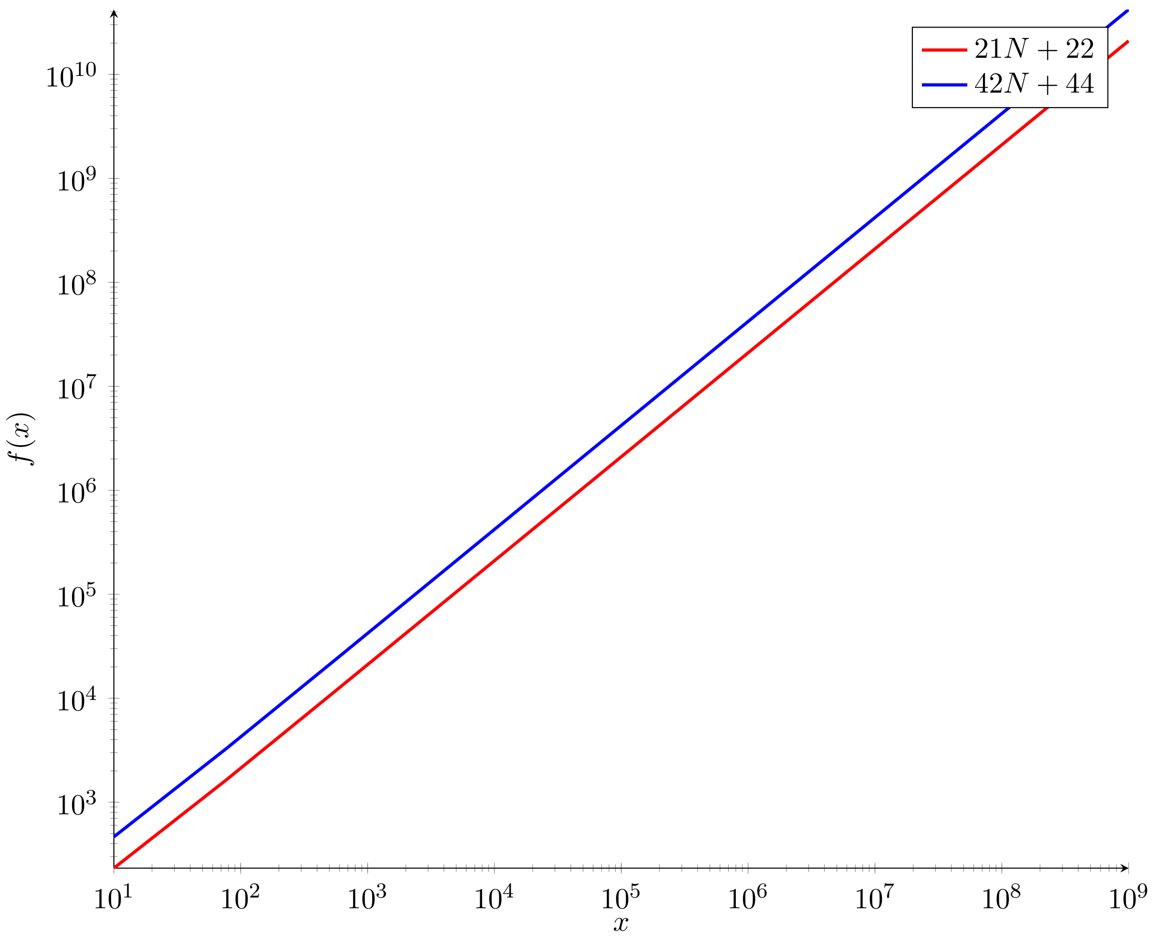

Figure 2: Running Time

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. $T(N) = C_1N + C_2$

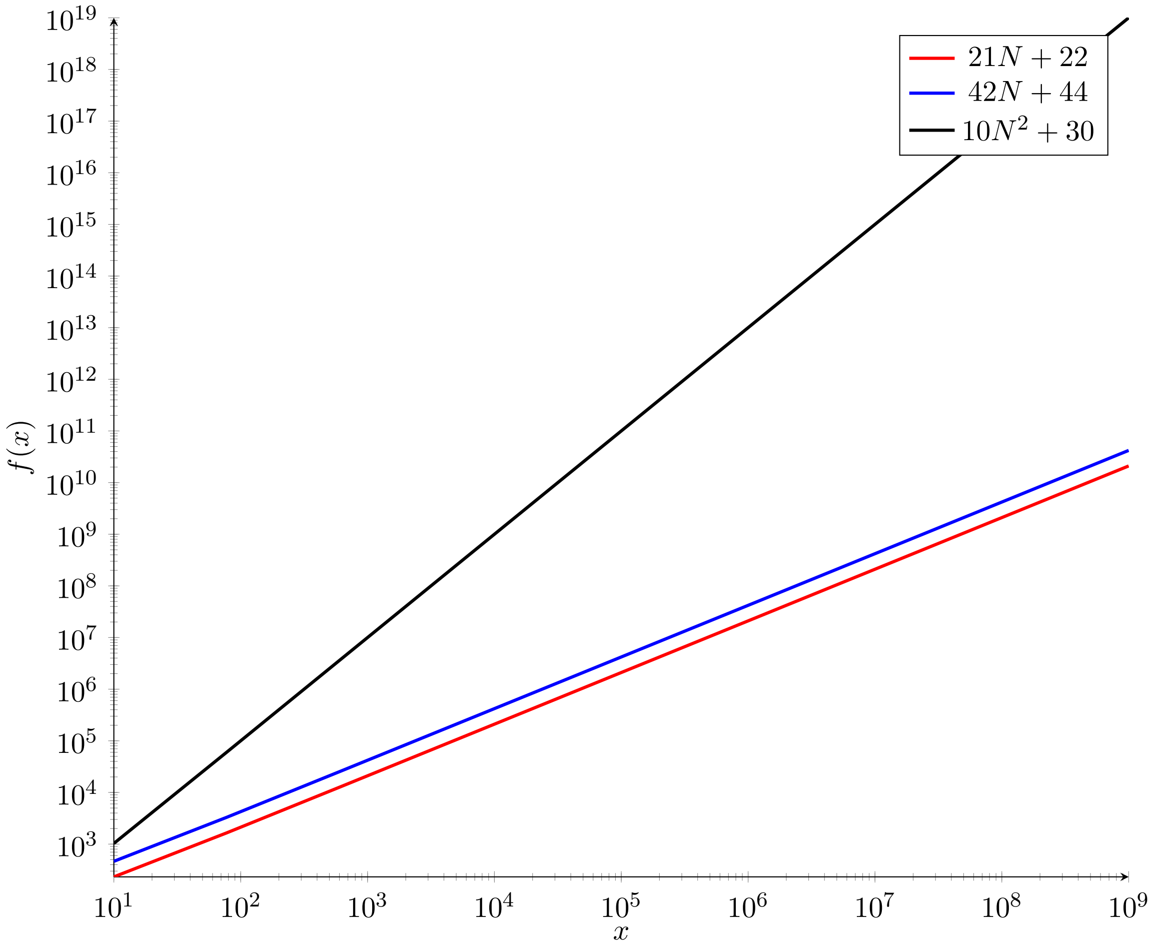

Figure 3: Running Time

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.

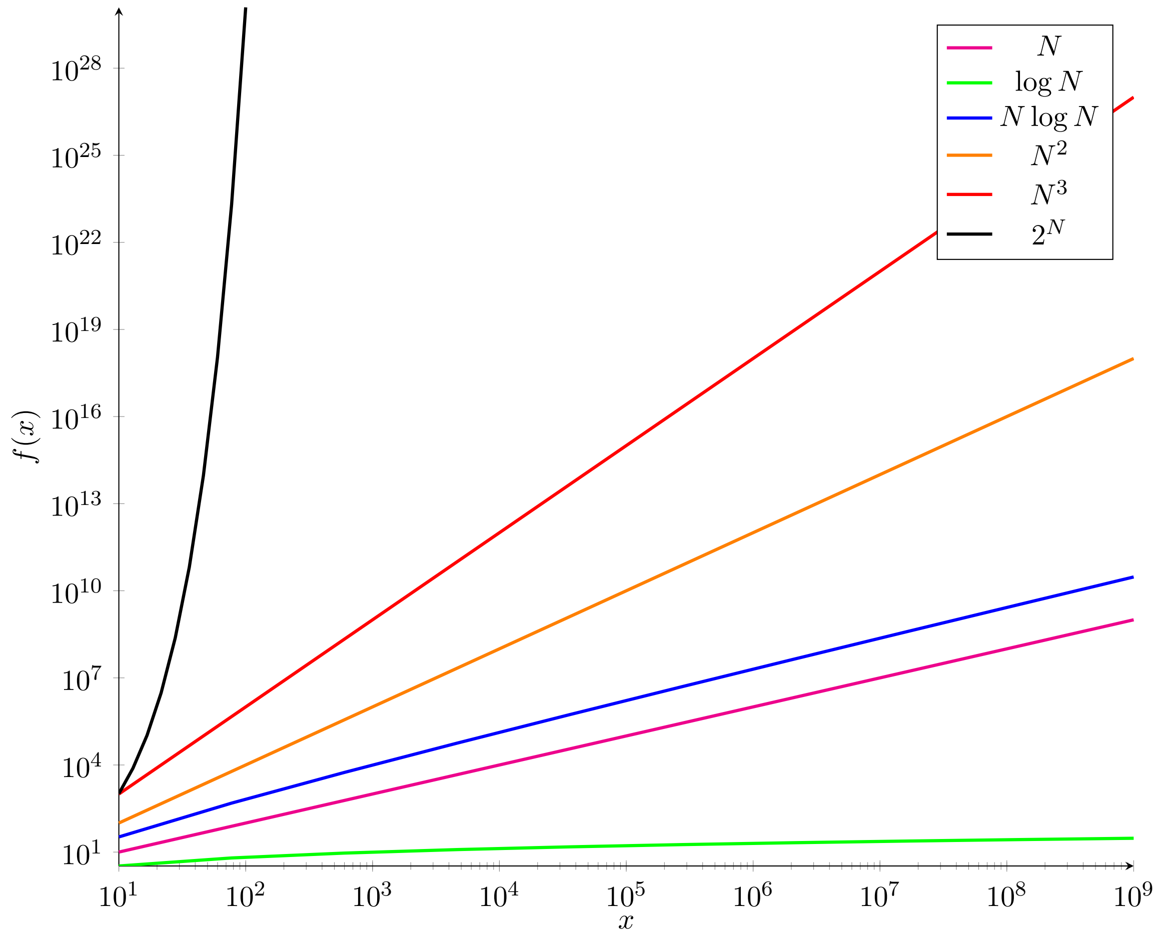

Figure 4: Running Time

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.

```
function SumDiag($A$)
    $sum \leftarrow 0$                              ▷ $C_0$
    $N \leftarrow$ Length($A[0]$)                   ▷ $C_1 N + C_2$
    for $0 \le i < N$ do                            ▷ $C_3 N + C_4$
        $sum \leftarrow sum + A[i, i]$              ▷ $C_5 N$
    end for
    return $sum$                                    ▷ $C_6$
end function
```

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

$$\begin{aligned} T(N) &= (C_1 + C_3 + C_5)N + (C_0 + C_2 + C_4 + C_6) \\ &= C_7 N + C_8 \\ &= N \end{aligned}$$

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 $25N^2$.

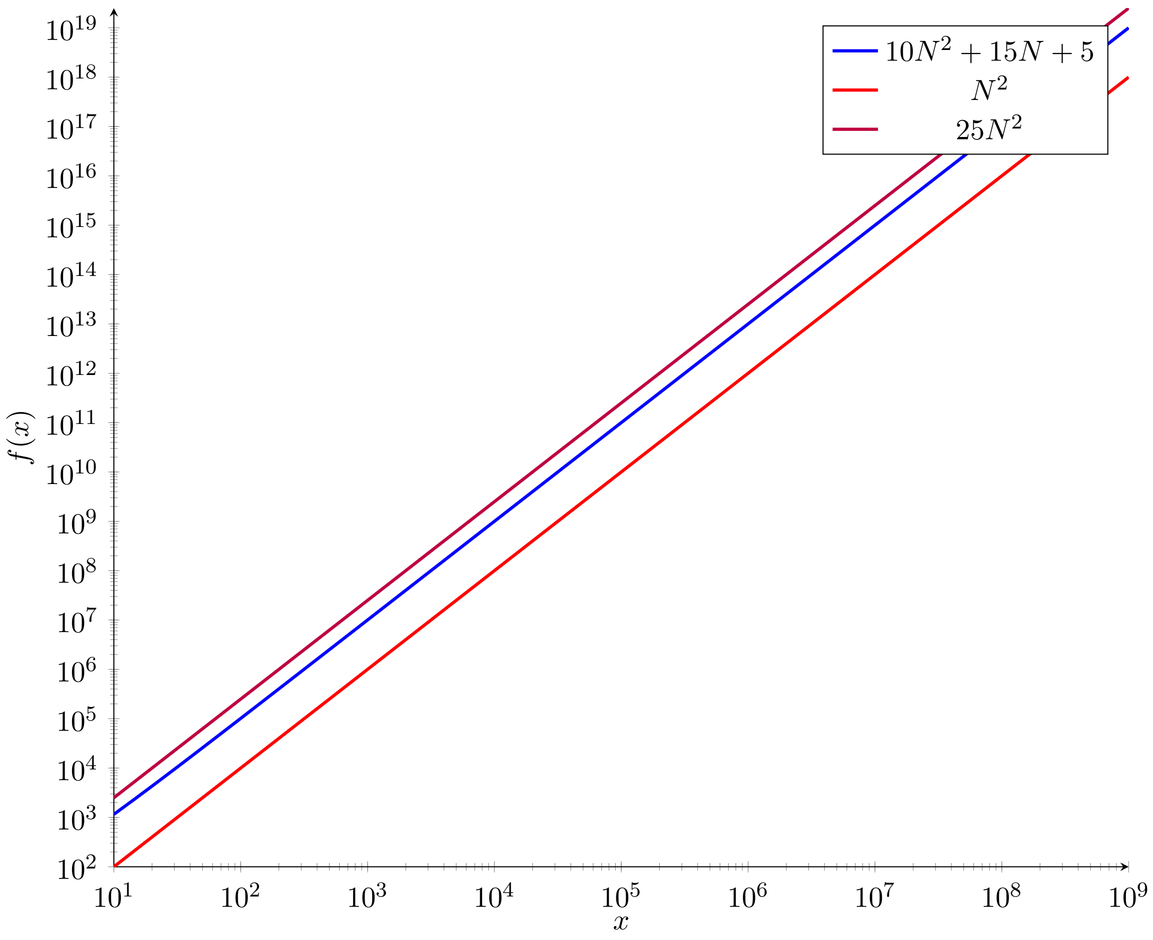

Figure 5: Big-$\mathcal{O}$

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

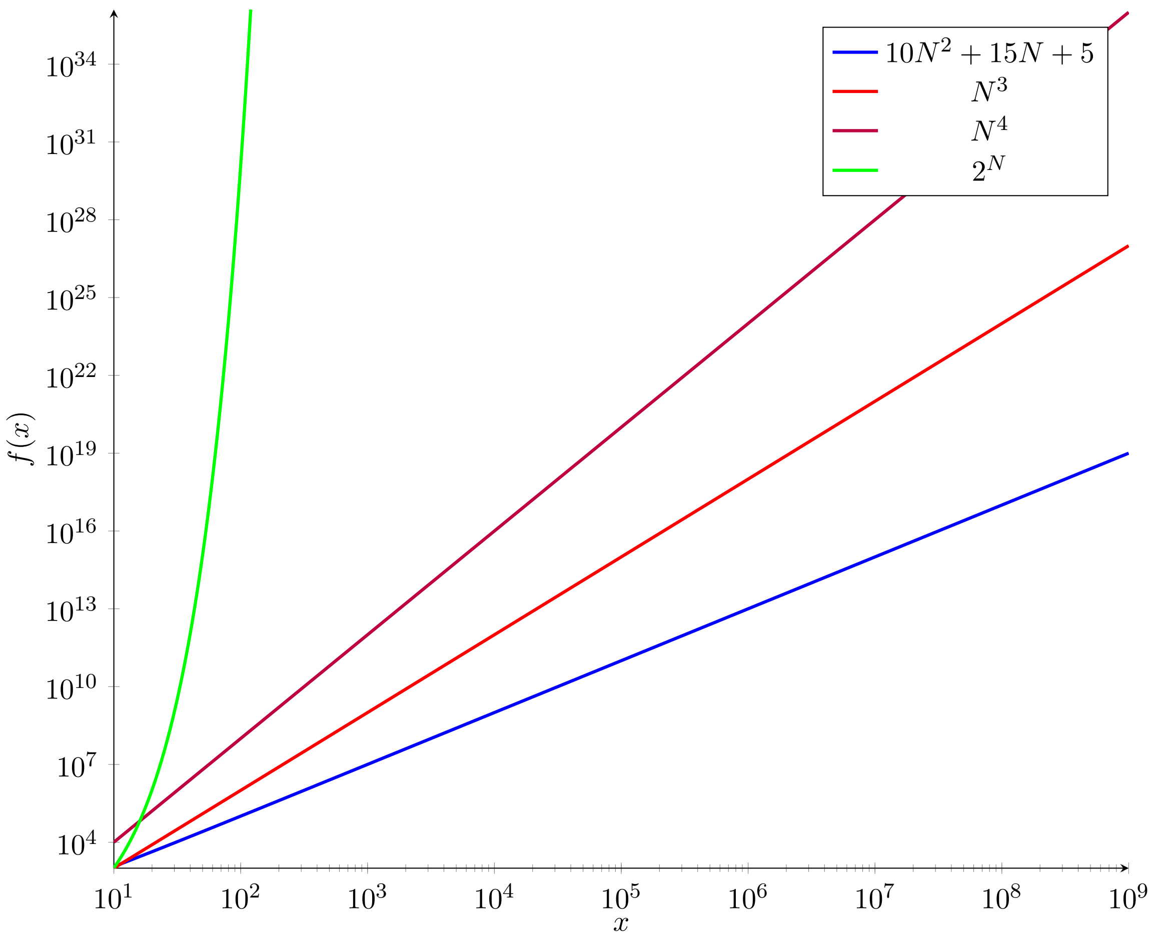

Figure 6: Big-$\mathcal{O}$: N^3, N^4, 2^N

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:

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

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

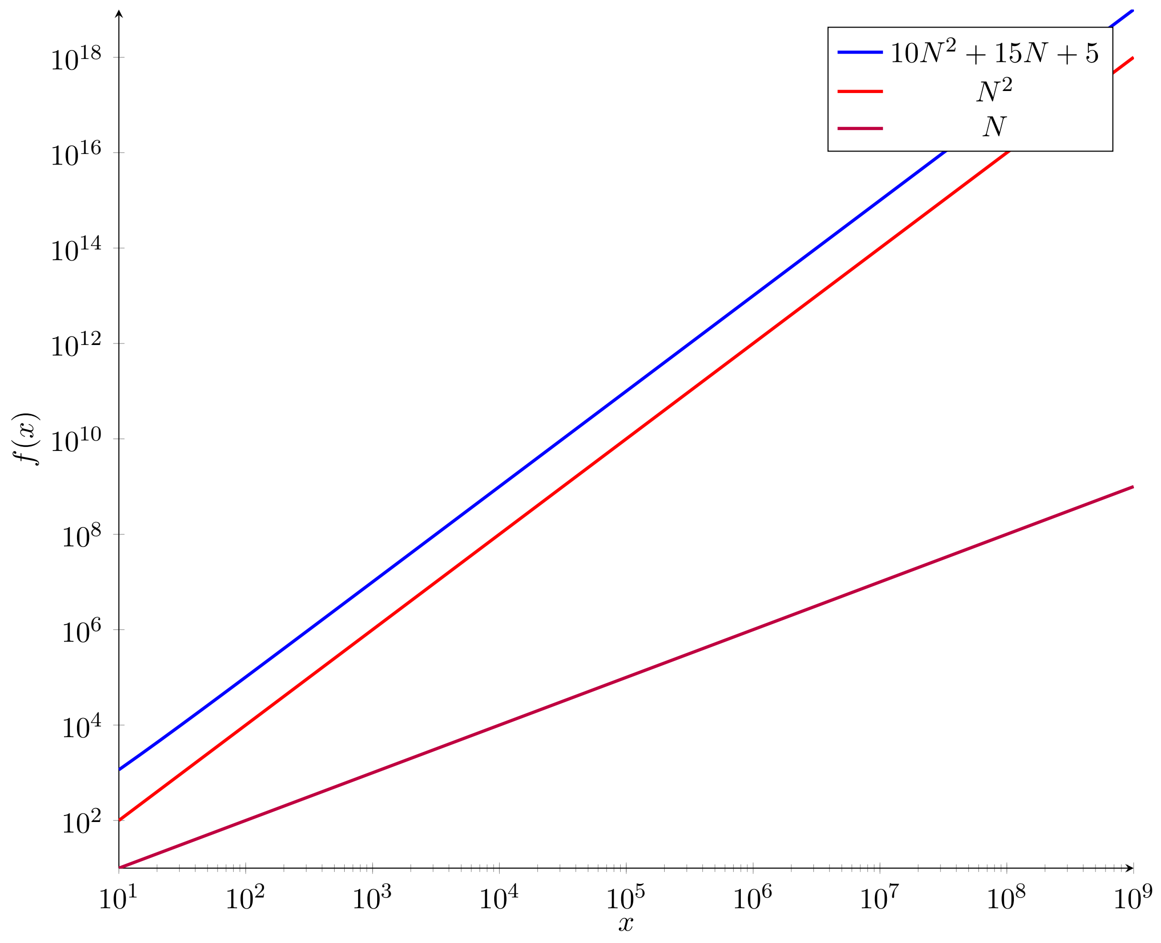

Figure 7: Big-Ω

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

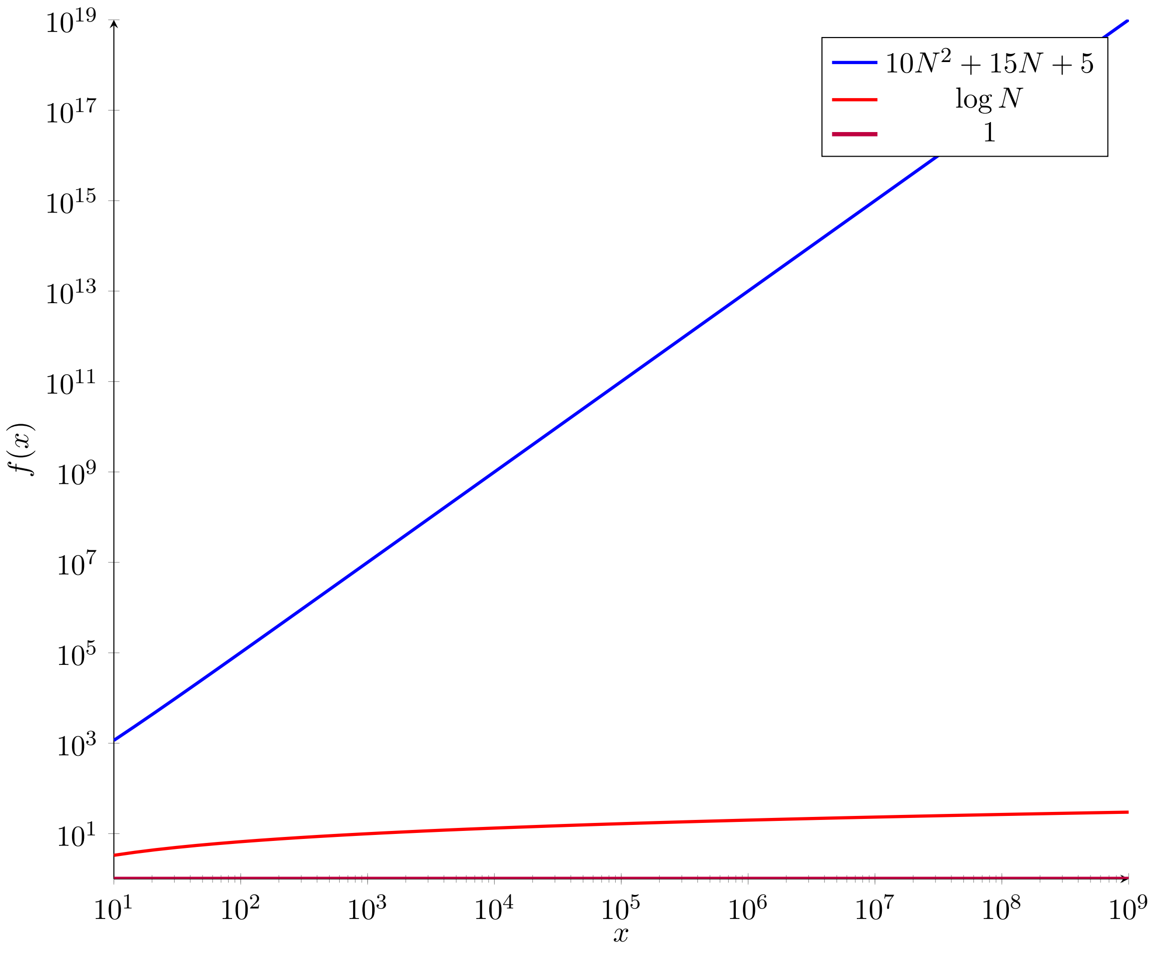

Figure 8: Big-Ω: $\Omega(\log N)$ and $\Omega(1)$

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:

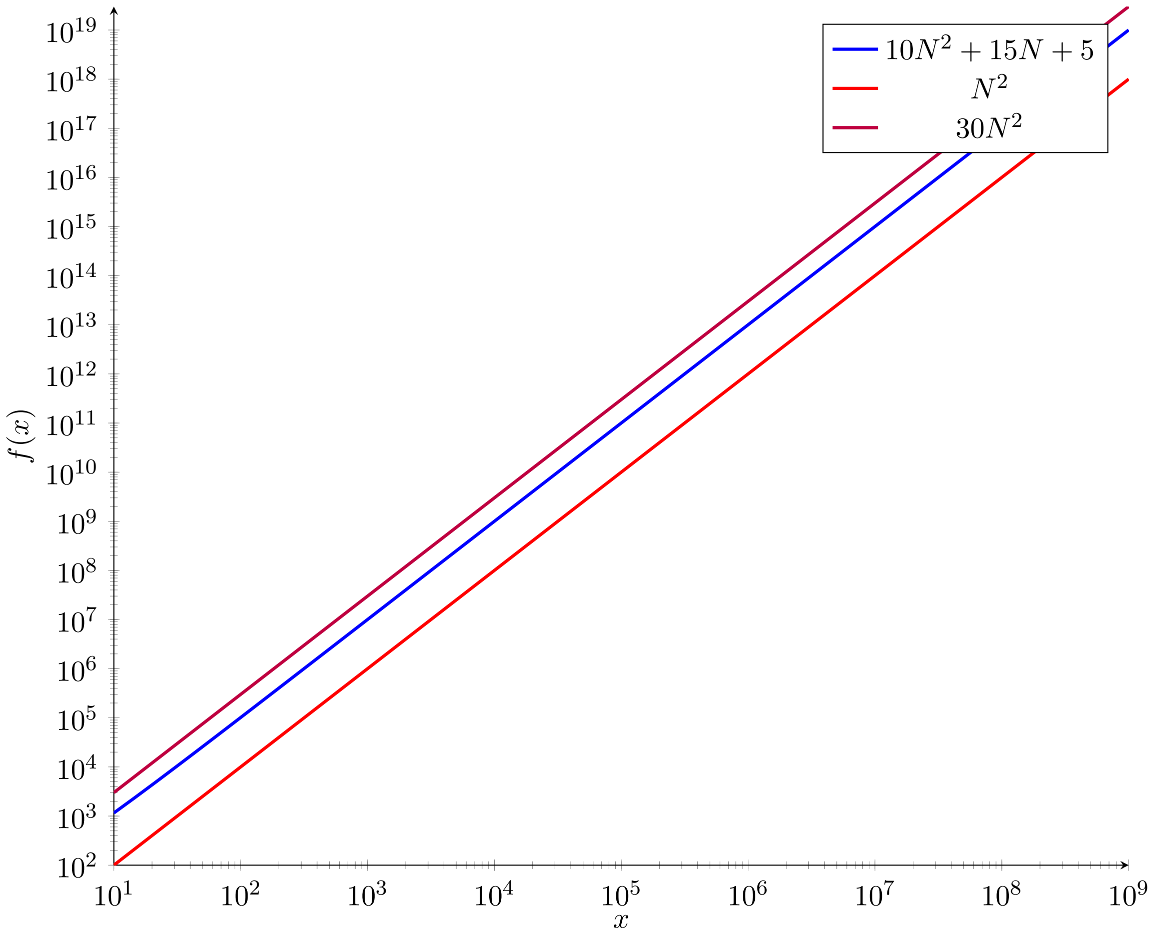

Figure 9: Big-Θ

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

```
function HELLO
    PRINT("hello")                    ▷ Print "hello" on the screen
    HELLO                             ▷ Recursive call
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

```
function HELLO($n$)
    if $n = 0$ then                              ▷ If $n = 0$...
        return                                   ▷ We're done
    end if
    PRINT("hello")                               ▷ Print "hello" on the screen
    HELLO($n - 1$)                               ▷ Recursive call approaching base case
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

```
function F($a, b$)
    if $b = 0$ then
        return $a$
    end if
    return F($a + 1, b - 1$)
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

```
function ITERCOUNTDOWN($n$)
    for $i \leftarrow n$ downto 0 do
        PRINT($n$)
    end for
end function
```

Algorithm 5 Recursive Count Down

```
function RECCOUNTDOWN($n$)
    if $n < 0$ then
        return
    end if
    PRINT($n$)
    RECCOUNTDOWN($n - 1$)
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

```
function FACTORIAL($n$)
    if $n \leq 1$ then
        return 1
    end if
    return $n\times$ FACTORIAL($n-1$)
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

```
function FACTORIAL($n$)                          ▷ $T(N)$
    if $n \leq 1$ then                           ▷ $C_0$
        return 1
    end if
    return $n\times$ FACTORIAL($n-1$)            ▷ $C_4 + T(N-1)$
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:

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

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

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

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

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:

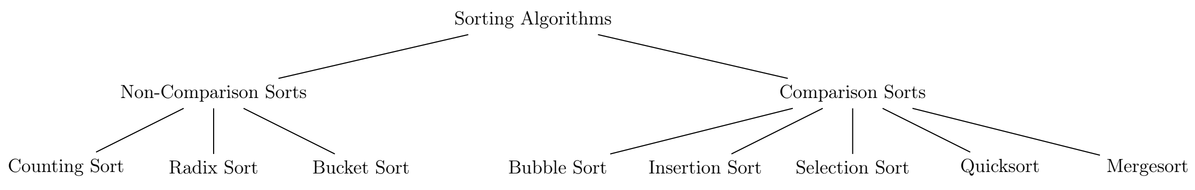

Figure 10: Sorting Algorithms

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	$\Theta(N^2)$	$Θ(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:     $swapped \leftarrow$ true
 3:     while $swapped$ do
 4:         $swapped \leftarrow$ 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:                 $swapped \leftarrow$ 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

```
function SelectionSort($A, N$)
    for $1 \le i < N - 1$ do
        $min \leftarrow$ PosMin($A, i, N - 1$)
        Swap($A[i], A[min]$)
    end for
    return $A$
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

```
function QUICKSORT($A, low, high$)
    if $low < high$ then
        $p \leftarrow$ PARTITION($A, low, high$)
        QUICKSORT($A, low, p - 1$)
        QUICKSORT($A, p + 1, high$)
    end if
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 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 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 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

```
function MERGE($A, l, mid, h$)
    $L \leftarrow A[l \dots mid]$
    $R \leftarrow A[mid + 1 \dots h]$
    $i \leftarrow 0$
    $j \leftarrow 0$
    $k \leftarrow l$
    while $i \leq mid \land j < (h - mid)$ do
        if $L[i] \leq R[j]$ then
            $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else
            $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
        end if
        $k \leftarrow k + 1$
    end while
    while $i \leq mid$ do
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
        $k \leftarrow k + 1$
    end while
    while $j < (h - mid$ do
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    end while
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 arragement 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 0^{th} element is smaller than the 1^{st} 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:

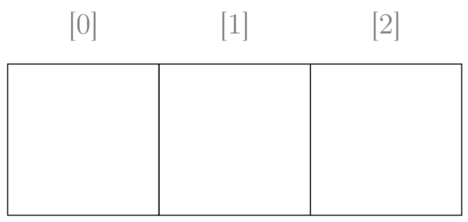

Figure 11: 3-element Array

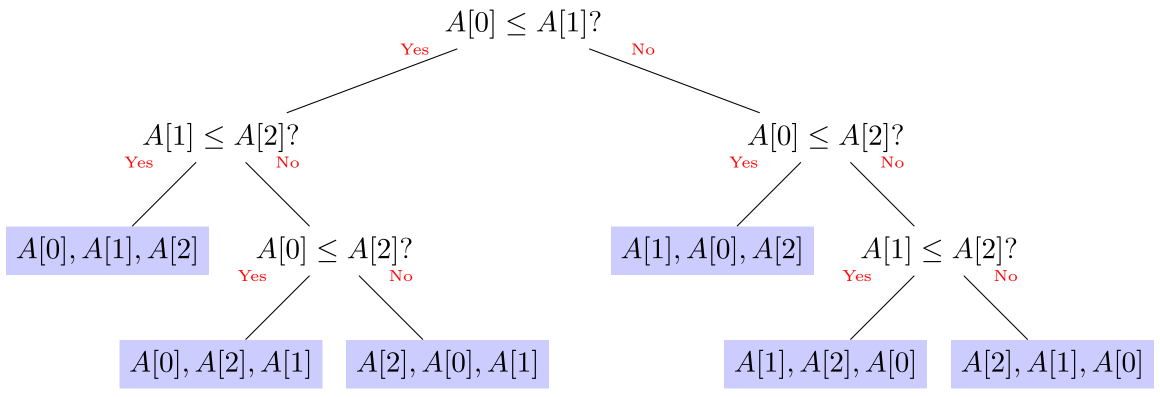

Figure 12: Decision Tree

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:

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

4.002 Lower bounds for commparison 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 nubmers, 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:

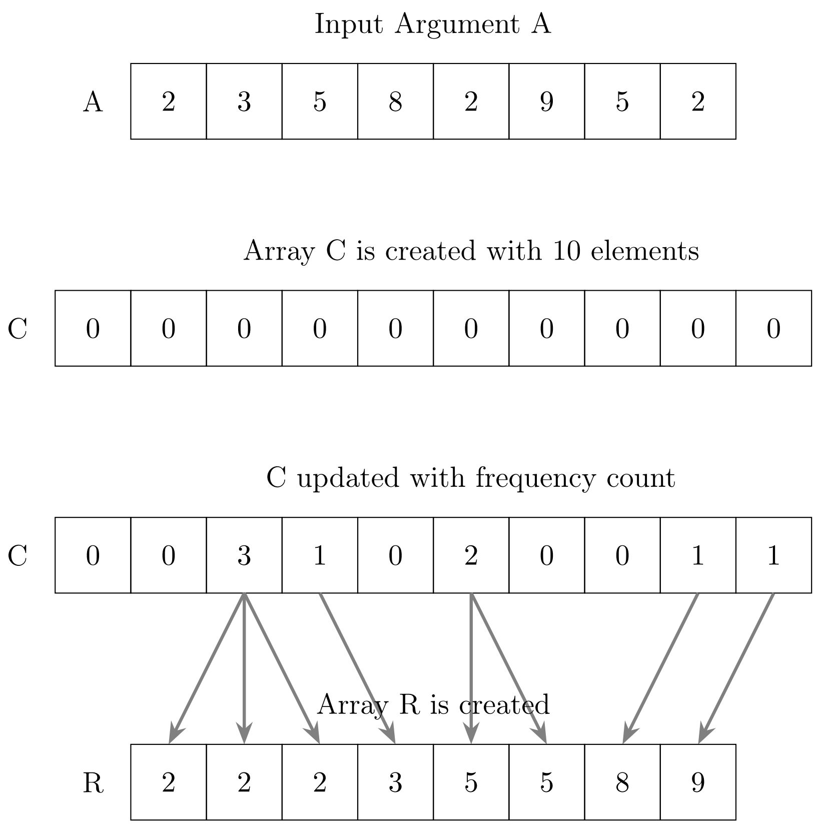

Figure 15: Counting Sort Execution

While the counting sort algorithm has a very desirable linear time complexity, there 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 $max + 1$ 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

```
function CountingSort($A, k$)
    $C \leftarrow$ new $Vector[k+1]$          ▷ We assume vector is zero-initialized
    $R \leftarrow$ new $Vector[$Length$(A)]$
    $pos \leftarrow 0$
    for $0 \leq j <$ Length$(A)$ do
        $C[A[j]] \leftarrow C[A[j]] + 1$
    end for
    for $0 \leq i < k + 1$ do
        for $pos \leq r < pos + C[i]$ do
            $R[r] \leftarrow i$
        end for
        $pos \leftarrow r$
    end for
    return $R$
end function
```

The algorithms 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.

Week 8

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

4.201 Radix sort

Radix sort employs a novel technique of sorting. It will sort its input numbers by splitting the number in digits and sort the digits in steps. First it will sort the least-significant digits, then then sort the next and the next until all digits are sorted.

One caveat of Radix sort is that the algorithm used to sort the digits **must** be a stable sort, meaning that once the least-significant digit is sorted, sorting the second-significant digit will preserve the original sorted order of the least-significant digits.

Counting sort, luckily, is a stable sort. We can use it for the digit sorting part of Radix sort.

Radix sort runs for as many iterations as there are digits in the numbers, i.e., if we're sorting 3-digit numbers, Radix sort will make three passes through numbers.

We can visualize Radix sort in figure 16.

Note that from one pass to another, the relative position of elements remain. In other words, after the first pass, 157 will always come before 457, even though we're running other passes of counting sort along the way.

Using counting sort to sort the digits results in Radix sort exhibiting the Time Complexity of $\Theta(d(N+k))$ where d is the number of digits, N is the number of numbers and k is the maximum value of digits. Moreover, $\Theta(N+k)$ is the time complexity of counting sort, therefore Radix sort has a time complexity of $\Theta(d \cdot g(N))$ where $g(n)$ is the time complexity of the sorting algorithm used to sort the digits.

One extra good aspect of Radix sort, is that it also works for sorting numbers containing decimal/fractional digits.

The basic minimal pseudocode from Radix sort is shown in listing 18.

4.203 Radix sort

Please read Section 8.3 (pp.197–9) from the guide book

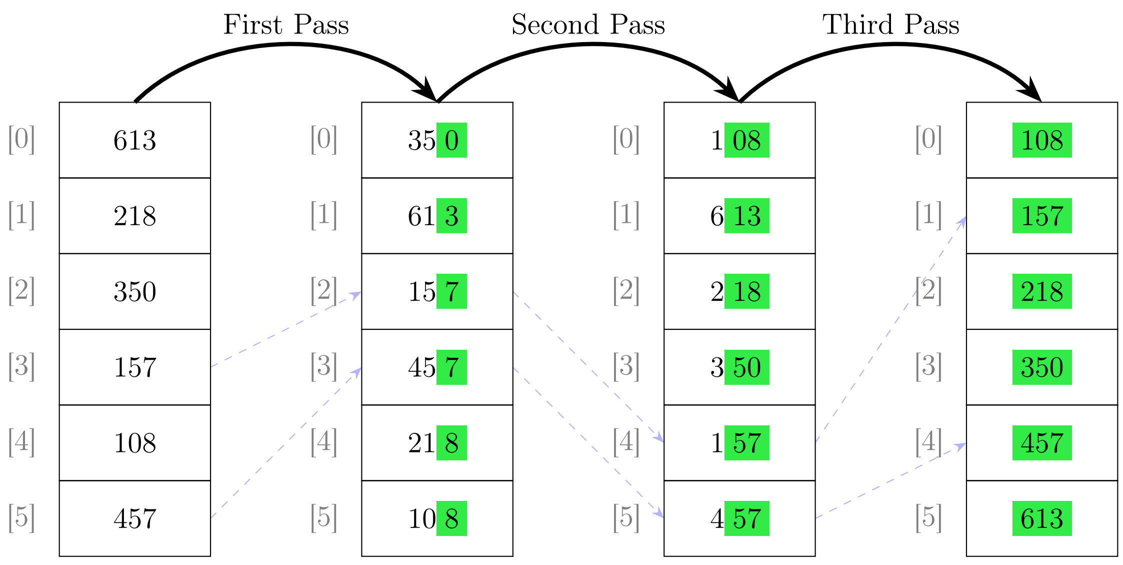

Figure 16: Radix Sort

Algorithm 18 Radix Sort

```
1: function RADIXSORT($A, d$)
2:     for $0 \leq j < d$ do
3:         COUNTINGSORT($A[j]$)                ▷ Sort A on digit j
4:     end for
5: end function
```

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.

4.301 Bucket sort

A simple analogy for Bucket sort is when we want to sort a pile of coins. Generally, we don't compare one coin against another to sort them, we would pick a random coin and place it in a stack according to its value. After doing that to every coin, we would be left with n stacks of coins, all sorted.

Bucket sort works in a similar fashion. In the best case, we will have one copy of each number which will result in each number going to a different bucket. The array will, therefore, be sorted after n operations.

Figure 17 shows a depiction of this idea.

To calculate the buckets in figure 17 we divide each by 100 and take the floor of that, e.g. $\left\lfloor \frac{137}{100} - 1 \right\rfloor = 0$.

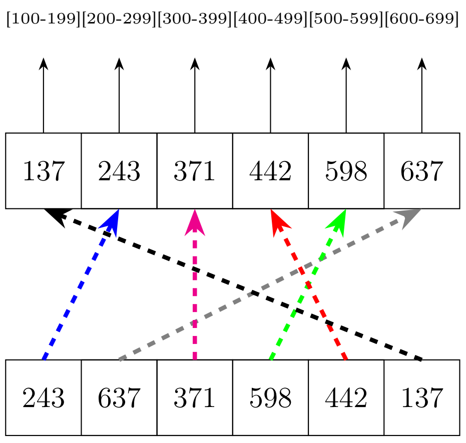

Figure 17: Bucket Sort

Bucket sort will behave well if the numbers in the input array are uniformly distributed. In case they aren't, we could fall into a bad case where all numbers fall into the same bucket.

In a normal situation where more than one number can fall into the same bucket, we must find a way to accomodate more than one number into the same position in the array. After that, we must sort the numbers inside every bucket before copying them back to the original array.

The first challenge can be solved with a linked list (future topic). The second challenge, i.e. sorting the elements in a linked list, can be solved with any other sorting algorithm. We could either use a comparison sort (insertion sort seems to be common) or a non-comparison sort, where we could apply counting sort, radix sort or recursively call bucket sort itself.

A simplified pseudocode of Bucket sort is shown in listing 19.

The time complexity of Bucket sort is $T(N) = C_4 \cdot N + T(\textit{Sorting linked lists})$.

4.303 Bucket sort

Please read Section 8.4 (pp.200–) 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 19 Bucket Sort

```
function BucketSort($A, N, max$)
    $Buckets \leftarrow$ new $Array[N]$                                ▷ New array of size N
    for $0 \leq i < N$ do
        $Buckets[i] \leftarrow$ empty linked list          ▷ New list in each element of Buckets
    end for
    for $0 \leq i < N$ do
        $Buckets\left[\left\lfloor \frac{A[i] \cdot N}{max+1} \right\rfloor\right] \leftarrow A[i]$    ▷ Add A[i] to correct list
    end for
    for $0 \leq i < N$ do
        Sort($Buckets$[i])                                     ▷ Sort each list
    end for
    for $0 \leq i < N$ do
        Copy($Buckets[i], A$)                   ▷ Copy list Buckets[i] back to A
    end for
end function
```

Week 9

Key Concepts

- Describe the different methods used to search for data
- Describe different collision resolution methods
- Implement a hash table with linear probing collision resolution.

5.001 What is hashing?

Hashing is the process of transforming a sequence of alphanumeric characters to a value. The algorithm responsible for the conversion process is referred to as a *Hash Function.*

This function receives a sequence of characters as input and returns a hash value. We can use any function that's able to transform the input into a value. For example, we can use the sum of the ASCII values of the input character:

$$\text{cat1} = 99 + 97 + 116 + 49 = 361$$

Hash functions are computationally cheap to apply, but computationally expensive to reverse given a hash value. Because of that, they are also referred to as *one-way functions.*

Hashing has important applications in security and information retrieval.

We can employ hashing on a password input to avoid transmitting the actual password outside of the users' computer. The example given above – i.e. that of summing the ASCII values of the input characters – is not robust enough for real-world applications. Different passwords containing the same characters in a different order will result in the same hash value. We want hash values to be unique. There are, however, much more secure hashing functions such as Secure Hash Algorithms (SHA).

Hashing functions can also be used for content verification. Assuming we will transmit sensitive information through the network, how can we guarantee that the data being sent has not been tampered with along the way?

While we can't stop a malicious agent from tampering with the data, we can provide means for detecting that the data hasn't been modified. We can achieve that by hashing the information before sending it and transmitting both the hash value and the information through the network. In case the information is modified somehow, the hash value won't match. More information about this application here.

Another important application of Hash functions is to enable *Fast Searching.* Say we want to verify if particular number is stored in an array. One could implement Linear

Search and check every position in the array; however, given the size of the input array, the worst case can take a long time.

Another option is to hash the numbers before inserting them into the array. The hash value will tell us where to store the number in the array. This means that when we want to search for the number, we can hash it again to get the position in the array where it should have been stored. This means that our search can be completed in $\mathcal{O}(1)$. This is the basic idea of a *Hash Table*.

5.003 Hash tables

To motivate the discussion of Hash Tables, we will define our searching problem as follows:

- the input data will be an array of numbers and a number to search for.
- the algorithm must give a result (true or false)

With this in mind, we will discuss three possible solution to the searching problem before introducing Hash Tables as a possible fourth solution.

The first solution to this problem is Linear Search. As shown in figure 18, this algorithm will visit every position of the array to check if the number we're looking for is there or not.

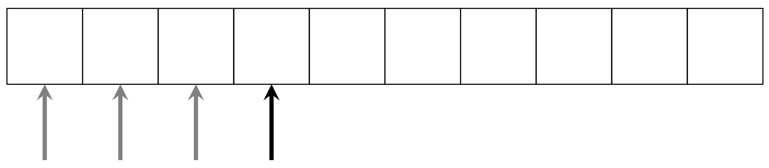

Figure 18: Linear Search

This algorithm will return either when we find the number we're looking for (in which case, it returns *true*) or we reach the end of the array (in which case, it returns *false*).

The pseudocode for this algorithm is shown in listing 20.

Algorithm 20 Linear Search

```
function LinearSearch($A, N, x$)
    for $0 \le i < N$ do
        if $A[i] = x$ then
            return true
        end if
    end for
    return false
end function
```

In the worst-case, this algorithm will run in $T(N) = \Theta(N)$. In the best-case, Linear Search is $T(N) = \Theta(1)$. In terms of memory, Linear Search is $S(N) = \Theta(1)$.

The second solution to this problem is Binary Search. Binary search requires the array to be sorted, but with that we can complete the search in $T(N) = \Theta(\log N)$ in worst-case and $T(N) = \Theta(1)$ in the best-case. In terms of memory, Binary Search has different behavior if it's a recursive version or iterative version.

In the recursive version we have $S(N) = \Theta(N)$ in the worst-case and $S(N) = \Theta(1)$ in the best-case. The iterative version is always $S(N) = \Theta(1)$.

The pseudocode for the recursive version of Binary Search is shown in listing 21.

Algorithm 21 Binary Search

```
function BinarySearch($A, low, high, x$)
    if $low > high$ then
        return $-1$
    end if
    $mid \leftarrow \left\lfloor \frac{low+(high-low)}{2} \right\rfloor$
    if $A[mid] = x$ then
        return $mid$
    end if
    if $A[mid] > x$ then
        return BinarySearch($A, low, mid-1, x$)
    end if
    if $A[mid] < x$ then
        return BinarySearch($A, mid+1, high, x$)
    end if
end function
```

The third solution for the search problem is called *Direct Addressing.* The idea is to use the index of the array to represent a number. When we want to search for the number, we check if the array at the number's index contains a 0 or a 1. The array of 1s and 0s created for this algorithm is referred to as *Bit Vector.*

The pseudocode for direct addressing is shown in listing 22.

Algorithm 22 Direct Addressing

```
function DirectAddrSearch($B, x$)
    return $B[x]$
end function
```

The time complexity of this solution is $T(N) = \Theta(1)$. Space complexity is $S(N) = \Theta(k)$ where k is the maximum value stored in the original array.

Finally, we reach to the *Hash Table* solution. Its behavior is similar to Direct Addressing, but requires far less memory. Similarly to Direct Addressing, we create another array to store the numbers. When the new array is uninitialized, it's filled with -1.

What we do, is that we transform each number into an index in the hash table using a hash function. To search for the number, the algorithm is similar to Direct Addressing, but before indexing the table with argument, we run the argument through the same hash function as shown in listing 23.

Algorithm 23 Hash Table Search

```
function HashSearch($H, x$)
    $i \leftarrow h(x)$
    if $H[i] = x$ then
        return true
    end if
    return false
end function
```

Much like Direct Addressing, Hash Table Search has a time complexity of $T(N) = \Theta(1)$ and space complexity of $S(N) = \Theta(<)$ where M is the number of elements in the hash table.

The table table below summarizes the information.

Algorithm	**Time Complexity**	**Space Complexity**
Linear Search (iterative)	$\Theta(N)$ (worst) $\Theta(1)$ (best)	$\Theta(1)$
Binary Search (iterative)	$\Theta(\log N)$ (worst) $\Theta(1)$ (best)	$\Theta(1)$
Binary Search (recursive)	$\Theta(\log N)$ (worst) $\Theta(1)$ (best)	$\Theta(1)$ (best) $\Theta(\log N)$ (worst)
Direct Addressing	$\Theta(1)$	$\Theta(k)$ (k is max value)
Hash Table (We know the numbers)	$\Theta(1)$	$\Theta(M)$ (M numbers in hash table)

5.101 Collisions in hash tables

What problems can arise when we don't know the numbers to be placed in the Hash Tables? One of the possible problems is that of collisions, which is happens when more than one input number map to the same location in the Hash Table.

Collisions can't be avoided, but there are ways to deal with them.

One possible method is known as *Extend And Re-hash*. In essence, we must enlarge the hash table then come up with a new hashing function to re-hash all elements.

This process consists of 3 steps:

1. Enlarge the Hash Table

   This is so it can fit more items. It corresponds to the "extend" part of the method's name.

2. Modify the reduction function

The part of the hash function which we modify during step 2 is known as the reduction function because it *reduces* the input values to the range of values permitted by the hash table. When we increase the number of buckets in the hash function, we must update the reduction function.

3. Re-hash numbers already stored in the hash table

   Now that the reduction function has been updated, all numbers must be re-hashed and moved to their new correct locations. It corresponds to the "re-hash" part of the method's name.

This collision resolution method can be applied in two ways:

Reactive Executed after a collision occurs

Proactive Executed after utilisation of the hash table reaches a threshold

The second resolution method is known as *Linear Probing.* It consists of searching for the next available bucket to store the number in collision.

The third and final collision resolution method is known as *Separate Chaining.* It works by creating a chain of buckets at each position of the hash table. As numbers are added to buckets, they each occupy a different position in the chain. The advantage of chains is that they can grow and shrink on demand.

When it comes to asymptotic analysis, we have:

Best-case No collisions happen. In this case we have INSERT $= \Theta(1)$, SEARCH $= \Theta(1)$ and DELETE $= \Theta(1)$.

Worst-case Everything collides. In the case of Separate Chaining we have INSERT $= \Theta(1)$, SEARCH $= \Theta(N)$ and DELETE $= \Theta(N)$.

5.103 Hashing

Please read Chapter 11, pp.253–285 (except sections 11.3.3 and 11.5) 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 10

Key Concepts

- Describe the different methods used to search for data
- Describe different collision resolution methods
- Implement a hash table with linear probing collision resolution.

5.301 End of Topic 5

We have reviewed one of the most common uses of Hashing: fast searching using a hash table.

During this week, we should be preparing our midterm assignment and nothing more.

Week 13

Key Concepts

- Describe linear data structures and its operations using pseudocode
- Understand array and linked list based implementations of stacks and queues
- Implement a sorted linked list.

7.001 Introduction to data structures

During the first half of the module, we focussed on the study of algorithms. During the second half, we focus on the study of data structures.

We will study the following data structures:

- Lists, Stacks, Queues
- Trees
- Heaps
- Graphs

A Data Structure is a container of data where data is organized in a specific way. As an example, lists are linear data structures because data is organized linearly, i.e. one element follows the other, like shown figure 19 below.

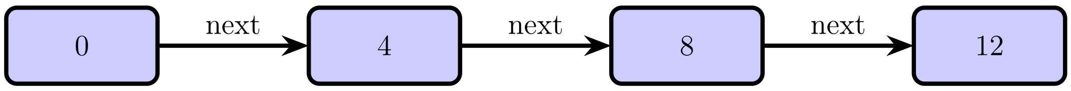

Figure 19: List

Trees and Heaps, on the other hand are organized in a hierarchical way, like the one shown in figure 20 below.

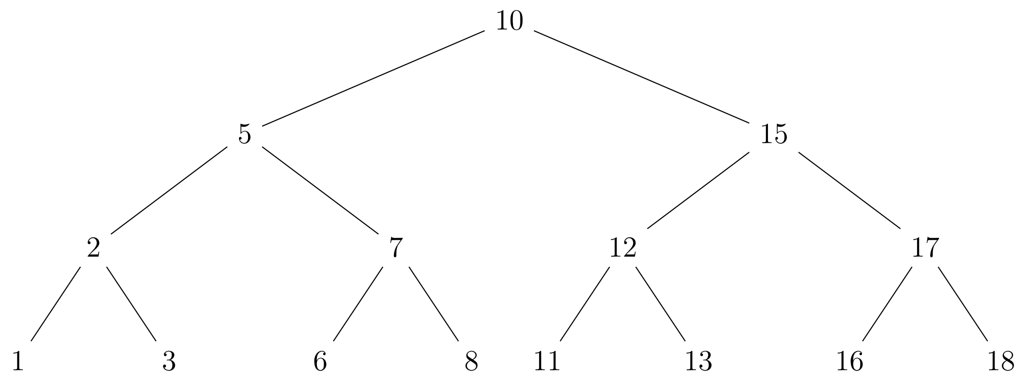

Figure 20: Tree

Every data structure has a set of operations associated with them which allows us to access and manipulate the data stored in them. As an example, Hash Tables have the operations *insert*, *search*, and *remove* associated with them.

7.003 Linked lists: Introduction

Much like a one-dimensional array, a Linked List is a linear data structure. However, unlike a one-dimensional array, a Linked List does not require a contiguous block of memory. Each element points to the next one using a pointer. This means that elements of a linked list can be located anywhere in memory so long as we update the *next* pointer of the previous element to point to the new one.

A *pointer* is, simply put, a memory address. Each element of the linked list must store, not only the data, but also a pointer to the next element, i.e. the memory address of the next element.

To access the next element we say that we *dereference* the pointer. This should be easy to understand, a memory address is a reference to a data, much like the index of a book is a reference the content we're looking for. *Dereferencing*, therefore, is accessing the memory address (or the page on the book) that contains the data we want to access.

Whenever we create a linked list, we must hold a reference to the first element (commonly referred to as the *head* of the list) otherwise we won't be able to recover any data. Moreover, the first time the list created, it contains nothing, therefore the *head* elements points to a special address known as *NULL*. The same *NULL* is used as a list terminator.

7.101 Linked lists: Insert operation

After our big overview of linked lists, we start studying its operations. The first operation we will study is insertion.

Before looking at the insertion pseudocode, let us define the representation of linkked lists in memory. Figure 19 is the simplest abstract representation of a list. It depicts the order of elements and arrows play the role of the pointer portion. Albeit being a very good representation for depicting the contents of the list, it lacks important details necessary to understand the pseudocode.

Therefore, figure 21 shows an improved representation of the list and its elements.

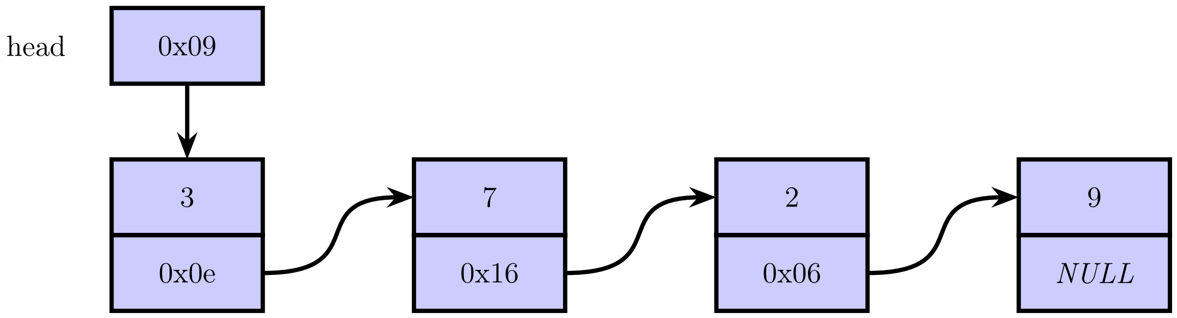

Figure 21: Improved List Representation

Inserting an element into a linked list involves traversing the linked list until we find the location in the list we want to insert the element.

As we can see, each element in a list is composed of two parts:

- A data part, shown as the top square
- A pointer part, shown as the bottom square

We will refer to each element in a list as a *Node*. In our pseudocode, when we refer to the data part we will write *node.data*, similarly the address of the next element will be written as *node.next*.[1]

We're ready to look at the pseudocode 24 of the *insert* function. The first step is to allocate a new Node to contain the new item. After the Node is initialized, we must link it to the list. This part changes depending on where we want to inser the element.

If we want to insert the element at the beginning of the list, first we must make the new Node point to the element currently pointed to by head. The final step would be change head to point to the new Node. Note that if we change the order of these two operations, i.e. changing head first, we would loost the reference to the node originally pointed to by head. In other words, we would loose the reference to number in figure 21.

Inserting to the beginning of the list has a time complexity $T(N) = \Theta(1)$ because we will always a constant number of operations to carry out the insertion.

There are two other methods of inserting into a list:

Inserting at the end In this case, we will always traverse the list until we find an node whose *next* pointer is *NULL*, this means we have found the end of the list, then we make this node's *next* point to the new Node.

[1]We're differing from the lecture, which uses the -> operator, simply because the . is more common in LaTeX. It's also a little easier to type.

Algorithm 24 Linked List Insert

```
function INSERT($head, x$)
    $newNode \leftarrow$ new $Node(x)$
    $newNode.next \leftarrow head$
    $head \leftarrow newNode$
end function
```

Inserting at an arbitrary position In this case, we must traverse the list until we find our arbitrary position containing node *pos*, modify *newNode.next* to point to *pos.next*, modify *pos.next* to point to *newNode*.

7.103 Linked lists: Delete operation

The second operation available in linked list is the *delete* operation. It gives us the ability to remove a node from a list. Essentially, we will reverse the steps done in listing 24.

The first thing we need to do is *bypass* the node to be removed, that is $prev.next = tmp.next$. It should look similar to figure 22 below:

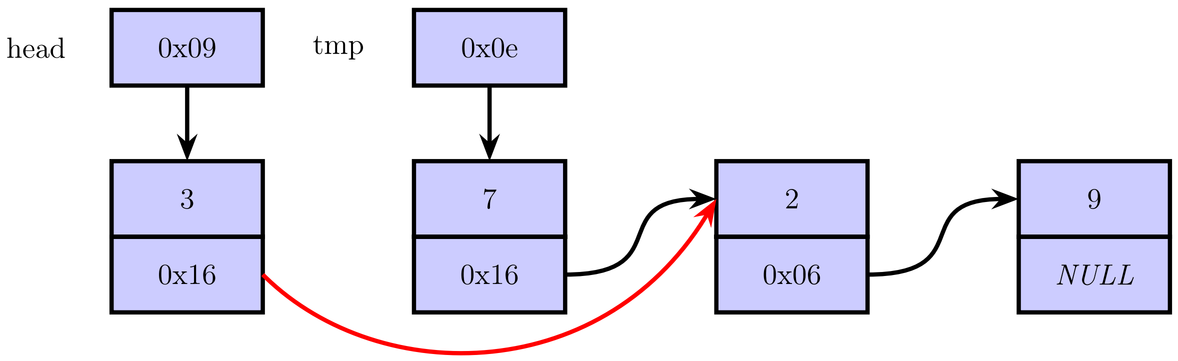

Figure 22: Linked List Delete

After the node to be removed is *bypassed* we can free the memory originally allocated for it. Listing 25 shows the pseudocode for deleting an item from the list.

7.105 Linked lists: Summary

The complexity of the main operations associated with a linked list, insert, delete, and search, is described below.

Starting with insert, its complexity depends on where we're inserting the new node. There are three cases, as below:

Algorithm 25 Linked List Delete

```
function DELETE($list, x$)
    $Nodetmp \leftarrow head$
    $Nodeprev \leftarrow NULL$
    if $tmp = NULL$ then
        return                                          ▷ Nothing to delete
    else
        if $tmp.data = x$ then
            $head \leftarrow tmp.next$
            return $list$
        else
            $prev \leftarrow tmp$
            $tmp \leftarrow tmp.next$
            while $tmp \neq NULL$ do
                if $tmp.data = x$ then
                    $prev.next \leftarrow tmp.next$
                    return $list$
                end if
                $prev \leftarrow tmp$
                $tmp \leftarrow tmp.next$
            end while
        end if                    ▷ If we get here, $x$ was not found in the list
    end if
end function
```

Beginning $T(N) = \Theta(1)$

End $T(N) = \Theta(N)$

Arbitrary Location There are two possibilities

- **Best case** $T(N) = \Theta(1)$
- **Worst case** $T(N) = \Theta(N)$

In the case of delete, we have to look at the best case and worst case. The best case happens when the node to be deleted is the first node in the list and the worst case happens when the node to be deleted is at the end of the list.

Best case $T(N) = \Theta(1)$

Worst case $T(N) = \Theta(N)$

Search will always have the same complexity as the Linear Search algorithm. This is because we must visit node n before we get the address of node $n + 1$. Therefore the time complexity of Searching a linked list is always the same as Linear Search algorithm, which is:

Best case $T(N) = \Theta(1)$

Worst case $T(N) = \Theta(N)$

Note that it's the same time complexity as deleting a node. The reason for this is that in order to delete a node, we must first search for it.

There are a few types of linked lists which we can build:

Doubly Linked List Each node points to the next and previous nodes.

Circular Linked List The last node points to the first node, instead of $NULL$.

7.108 Linked lists

Please read the definition of data structures on p.9 and then section 10.2 (pp.236–41) 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 14

Key Concepts

- Describe linear data structures and its operations using pseudocode
- Understand array and linked list based implementations of stacks and queues
- Implement a sorted linked list.

7.201 Stacks: Introduction

The *Stack* is another example of a linear data structure. It behaves much like a stack (e.g. of books) in the physical world. Objects can only be inserted at the top of the stack and removed from the top of the stack.

Because of this behavior, the insertion and removal operations have special names for them, *push* and *pop* respectively. Two other operations are *isEmpty()* (which returns **true** when the stack is empty) and *peek* (which returns the value of the element at the top).

Whenever we want to query the content of the stack, only the top of the stack is accessible. To get the next element, we must, first, *pop* the top element.

While they seem limited at first, stacks have very important applications in Computer Science. For example, Stacks are used to check for matching curly braces (*{}*) when parsing source code[1], implementation of Reverse Polish Notation calculators[2], procedure calls[3], and countless other applications.

7.203: Stacks: Implementation

Stacks are so common that they're part of the standard library of virtually every programming language.

Because a stack is a linear data structure, it can be implemented on top of other linear data structures, such as an array of a linked list. Arrays are peculiar because they have

[1] Whenever a *{* is found, an element is pushed onto the stack. Whenever the matching *}* is found, that element is popped. If we reach the end of the statement with a non-empty stack, we have an error.

[2] Whenever an operand is entered, push it onto the stack. Whenever an operator is entered, pop the correct amount of operands off the stack, execute the operation and push the result back onto the stack.

[3] Whenever a different procedure must be called, context of the calling procedure (i.e. the content of CPU registers) is pushed onto the stack. Upon returning from the called procedure, context is popped from the stack and restored onto respective registers.

a static size, therefore we either end up with unused memory (and that's wasteful) or we run out of space, in which case we could take one of three different paths:

1. Stop accepting new elements;
2. Allocate bigger array and copy elements from small to big array before inserting new element; or
3. Corrupt memory due to overflow of the stack space[4]

Whenever we want to use an array or a linked list to implement a stack, we must enforce the access rules of the stack. Using the example of arrays, a push would be implemented with the algorithm shown in listing 26 (note that *top* is initialized to -1 to signify an empty stack):

Algorithm 26 Stack: Push

```
function PUSH($x$)
    $top \leftarrow top + 1$
    $A[top] \leftarrow x$
end function
```

This doesn't take into consideration the fact that we can run out of space in the array. As mentioned before we can stop accepting new elements (see 27), or allocate a bigger array and move elements over (see 28) or do nothing (and introduce a possible bug).

Algorithm 27 Stack: Push with block

```
function PUSH($x$)
    if $top = \text{SIZE}(A) - 1$ then
        return
    end if
    $top \leftarrow top + 1$
    $A[top] \leftarrow x$
end function
```

Algorithm 28 Stack: Push with extend

```
function PUSH($x$)
    if $top = \text{SIZE}(A) - 1$ then
        EXTENDANDCOPY($A$)
    end if
    $top \leftarrow top + 1$
    $A[top] \leftarrow x$
end function
```

[4] `https://en.wikipedia.org/wiki/Stack_buffer_overflow`

In the case of 28, the time complexity of the *push* operation grows from $\Theta(1)$ to $\Theta(N)$ because we must copy all elements over to the new, bigger array before inserting a new element.

Moving on to the *pop* operation, its algorithm is shown in listing 29. All operations in *pop* take a constant time, therefore its time complexity is $\Theta(1)$.

Algorithm 29 Stack: Pop

```
function Pop
    if $top = -1$ then
        return
    end if
    $top \leftarrow top - 1$
end function
```

The next operation is *peek*, shown in listing 30. Similarly to *pop*, all operations in *peek* take a constant time, which makes its time complexity $\Theta(1)$.

Algorithm 30 Stack: Peek

```
function Peek
    if $top = -1$ then
        return $-1$
    end if
    return $A[top]$
end function
```

The last operation is *isEmpty*, shown in listing 31. Much like the previous two operations, all statements in *isEmpty* take a constant time and its time complexity is also $\Theta(1)$.

Algorithm 31 Stack: isEmpty

```
function IsEmpty
    if $top = -1$ then
        return true
    end if
    return false
end function
```

The linked list implementation of stacks is analogous to that of the array implementation. In listings 32, 33, 34, 35 we show linked list versions of *push*, *pop*, *peek*, and *isEmpty* respectively. Every operatio has time complexity of $\Theta(1)$.

Algorithm 32 Stack: Push (Linked List)

```
function PUSH($x$)
    $n \leftarrow$ new $Node$
    $n.data \leftarrow x$
    $n.next \leftarrow top$
    $top \leftarrow n$
end function
```

Algorithm 33 Stack: Pop (Linked List)

```
function POP
    if $top = NULL$ then
        return
    end if
    $top \leftarrow top.next$
end function
```

Algorithm 34 Stack: Peek (Linked List)

```
function PEEK
    if $top = NULL$ then
        return $-1$
    end if
    return $top.data$
end function
```

Algorithm 35 Stack: isEmpty (Linked List)

```
function ISEMPTY
    if $top = NULL$ then
        return true
    end if
    return false
end function
```

7.301 Queues: Introduction

The queue is the final linear data structure that we will study. A queue may seem like witchcraft at first , however it's far from it. It behaves exactly like a queue in real life: people queue by standing at the end of the queue and are served from the front of the queue. Once served, they are removed from the queue.

We say that queues behave in a **FIFO** (standing for *First In, First Out*) manner.

The operations associated with a queue are:

1. Enqueue
2. Dequeue
3. Peek
4. isEmpty

Much like stacks, queues have several applications. Many of which refer to processing requests in the order they come.

7.303 Queues: Array-based implementation

Similarly to a stack, a queue can be implemented with arrays or linked lists. Many of the concerns with array-based stacks, apply to array-based queues as well.

Considering array-based implementation for now, we initialize the *front* (sometimes referred to as *head*) and *tail* pointers to -1 to signify an empty queue. To enqueue an element, we move the *tail* ahead by 1 position. When the queue is initialy empty, *front* must also be moved by 1. We should the algorithm in listing 36.

Algorithm 36 Queue: Enqueue

```
function ENQUEUE($x$)
    if $(tail + 1) \bmod N = front$ then
        return $-1$
    end if
    if ISEMPTY then
        $front \leftarrow 0$
        $tail \leftarrow 0$
    else
        $tail \leftarrow (tail + 1) \bmod N$
    end if
    $A[tail] \leftarrow x$
end function
```

The dequeue operation is analogous to enqueue. It is shown in listing 37.

Algorithm 37 Queue: Dequeue

```
function DEQUEUE
    if ISEMPTY then
        return
    end if
    if $front = tail$ then
        $front \leftarrow -1$
        $tail \leftarrow -1$
    else
        $front \leftarrow (front + 1) \bmod N$
    end if
end function
```

Algorithm 38 Queue: Peek

```
function PEEK
    if $front = -1$ then
        return $-1$
    end if
    return $A[front]$
end function
```

Peek is, also, a very simple algorithm. All we have to do is return the value of the element at the *front* if the list is not empty. The algorithm is shown in listing 38.

The algorithm for *isEmpty* is the simplest of them all. We just need to return **true** or **false** depending if the list is empty or not. Listing 39 shows the algorithm.

Algorithm 39 Queue: isEmpty

```
function ISEMPTY
    if $front = -1$ then
        return true
    end if
    return false
end function
```

The time complexity of all operations is $\Theta(1)$.

7.305 Queues: List-based implementation

To implement a queue with a linked list, we need one extra pointer for the tail. Without it, one of the operations would have to traverse the entire list before executing its role (enqueueing or dequeueing).

Traversing the list has a time complexity of $\Theta(N)$. With the second pointer, we can guarantee operations in $\Theta(1)$.

Listings 40, 41, 42, and 43 show the operations *enqueue*, *dequeue*, *peek* and *isEmpty* respectively. All operations have a time complexity of $\Theta(1)$.

Algorithm 40 Queue: Enqueue (Linked List)

```
function ENQUEUE($x$)
    $n \leftarrow$ new $Node$
    $x.data \leftarrow x$
    if $front = NULL \wedge tail = NULL$ then
        $front \leftarrow n$
        $tail \leftarrow n$
    else
        $tail.next \leftarrow n$
        $tail \leftarrow n$
    end if
end function
```

Algorithm 41 Queue: Dequeue (Linked List)

```
function DEQUEUE
    if $front = NULL \wedge tail = NULL$ then
        return
    end if
    if $front = tail$ then
        $front \leftarrow NULL$
        $tail \leftarrow NULL$
    else
        $front \leftarrow front.next$
    end if
end function
```

7.307 Stacks and queues

Please read Section 10.1 (pp.232–5) 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 42 Queue: Peek (Linked List)

```
1: function Peek
2:     if $front = NULL \wedge tail = NULL$ then
3:         return $-1$
4:     else
5:         return $front.data$
6:     end if
7: end function
```

Algorithm 43 Queue: isEmpty (Linked List)

```
1: function IsEmpty
2:     if $front = NULL \wedge tail = NULL$ then
3:         return true
4:     end if
5:     return false
6: end function
```