Algortihms and Data Structures: Reader*

Diederik M. Roijers & Jorn Bunk

October 2019

Contents

*This is a HU University of Applied Sciences Utrecht reader, intended for internal use for students and teachers at HBO-ICT at this university only.

Lecture	Sections	Appendices
1	2, 3, 4	A
2	5, 6, 7	
3	8, 9, 10 11, 12	
4		

Table 1: Caption

1 Introduction

Algorithms and data structures are cornerstones of computer science. In fact, having studied (technical) computer science for over a year, you have encountered many of them already and have probably developed a feeling for which algorithms and data structures are efficient in what situation. However, as the programs you will be writing become ever more complex, it helps to think about algorithms and data structures more formally, in order to make reasoned choices about which of them to use. Furthermore, when no readily available algorithms and/or data structures exist, this course will hand you the tools to analyse the efficiency (in terms of both runtime and memory) of the those you will invent yourself. This will help you write programs that are significantly – sometimes even orders of magnitude – faster, which is of course essential when computation power and memory are limited (e.g., on embedded hardware).

To design algorithms, it helps to analyse the problems these algorithms are meant to solve. Therefore, this course will introduce methods to analyse the complexity of such problems. This will lead down a path of ever higher complexity, all the way to problems that can no longer be solved exactly in any kind of reasonable time or with the memory available on the hardware that you are using. But, not to worry, this course will introduce heuristic algorithms that will even help you tackle those.

1.1 Reading manual

In this reader, some sections will be marked with an asterix *, and parts of sections with "**for the diehards**". This means that that part is additional information that will not be tested. We do however wholly recommend that you read this material, as it will go just a bit deeper into the subject matter, which may help increase your understanding and structure the rest of the information better.

The sections of this reader cover various topics. Multiple sections correspond to a single lecture. Furthermore, appendices that explain topics more thoroughly, and motivate choices for this reader may also be part of the lecture (and may be tested). In Table 1.1 you will find an overview of how the lectures correspond to the material in this reader.

2 The Better Algorithm

The question of why one algorithm is better than another is typically answered in terms of computational and memory efficiency. For example, let us compare the two algorithms, Algorithm 1 and Algorithm 2, for computing the maximum value in an array.[1]

Algorithm 1 max1

Input: an array a
Output: the maximum of a

```
1: currentMax ← −∞
2: for i ∈ 0...length(a)−1 do
3:     if a[i] > currentMax then
4:         currentMax ← a[i]
5:     end if
6: end for
7: return currentMax
```

Algorithm 2 max2

Input: an array a
Output: the maximum of a

```
1: for i ∈ 0...length(a)−1 do
2:     maxFound ← true
3:     for j ∈ 0...length(a)−1 do
4:         if a[j] > a[i] then
5:             maxFound ← false
6:             break inner loop
7:         end if
8:     end for
9:     if maxFound then
10:        return a[i]
11:    end if
12: end for
```

Both algorithms successfully return the maximum value in the array: Algorithm 1 by maintaining and updating a current maximum (initialised at negative infinity), and returning this maximum once it has looped over the entire array, and Algorithm 2 by checking for each element of the array whether there is another element that is larger, and when there is no larger element, returning it. Yet Algorithm 1 is clearly better than Algorithm 2. But how can we make a formal argument that this is indeed the case?

To claim that Algorithm 1 is better than Algorithm 2, we want to say something about their respective runtimes. Runtime is typically measured in milliseconds. However, for runtime it matters a lot on which type of hardware the actual code for an algorithm is run, in which programming language the actual code is implemented, and how it is implemented in practice. Furthermore, as hardware and compilers improve, the runtime of actual code can decrease, even when the actual code has not changed. Of course, this has no bearing on the quality of an algorithm. Therefore, measuring the actual runtime to say something about the quality of an algorithm is undesirable. The quality of an algorithm should not depend on external circumstances. Instead, we would like

[1] Note that we use *pseudocode* to denote the algorithms in this reader. Please refer to Appendix A for an explanation of the usage of pseudocode. The ← symbol is the assignment operator (typically denoted "=" in various programming languages), ∈ means "element of" or "in" (see also the `in` operator in Python), ∞ means infinity, and $0 \ldots X$ is the set/range of all numbers between 0 and X.

to use a metric that is *programming-language and hardware independent*, yet says something about the *expected* runtime when translated to actual code and run on actual hardware. To do so, we can count the number of computations that an algorithm has to perform.

So let us look at the number of computations for Algorithm 1 and 2. Let us suppose the most favourable case for both algorithms, i.e., the maximum value in a is the first element, $a[0]$. In this case, both algorithms loop over a once: Algorithm 1 assigns the maximum value to currentMax on its first iteration in its loop, and in subsequent iterations compares this value to the subsequent values in a (and finds that they are not larger than currentMax). Algorithm 2 goes into its outer loop, sets maxFound to `true`, and then goes into its inner loop. In the inner loop it compares all values in a to $a[0]$, and does not find a bigger value, so it does not break this inner loop, and does not change the value of maxFound. Then upon exiting this inner loop, maxFound is still `true`, so Algorithm 2 immediately returns $a[0]$ as the maximum of a. In other words, both algorithms perform either one or two assignments to a local variable and $n = \texttt{length}(a)$ comparisons, before returning the maximum value. So in the best case, both algorithms perform about equally well. However, we already know that Algorithm 1 is better than Algorithm 2, so we should probably not take the number of operations in the best case as a metric for algorithm quality.

Typically, the best case is not very informative for the quality of an algorithm. If you are not convinced by this, please consider Algorithm 3 for computing a max.

Algorithm 3 sillyMax

Input: an array a
Output: the maximum of a

```
1:  while true do
2:      x ← pick a random element of a
3:      maxFound ← true
4:      for j ∈ 0 ... length(a)−1 do
5:          if a[j] > x then
6:              maxFound ← false
7:              break inner loop
8:          end if
9:      end for
10:     if maxFound then
11:         return x
12:     end if
13: end while
```

In this algorithm, a random element of a is selected at each iteration, which makes no sense whatsoever: the same element could be selected in multiple iterations, and because of its randomness, it could run for a very long time indeed before it randomly picks the maximal element. Yet in the best case, x

just happens to be the maximum in the first iteration, and the algorithm still performs only n comparisons.

To measure the quality of algorithms the literature typically takes the worst-case performance rather than the best-case performance. This is of course a conservative thing to do, and we could also look at the average performance over all cases. However, this is of course much more difficult to analyse, and implies that we have an idea about what type of input to the algorithm we can expect. So, because it is easier to do – or maybe because we computer scientists are just a gloomy bunch – we take the worst-case performance as the standard for analysing the performance of algorithms.

When we look at the worst-case performance (the input is a sorted list with the maximum value as its last element), it is clear that Algorithm 3 is very silly indeed: in the worst case it will never terminate.[2] Algorithm 2 has two loops that each go through the array. Each time it goes through the array, it needs to perform one more comparison (averaging at $\frac{1}{2}n$ for each iteration of the outer loop), making the total number of comparisons $n \cdot \frac{1}{2}n = \frac{1}{2}n^2$ comparisons[3], plus $2n - 1$ assignments to the maxFound variable. And finally, Algorithm 1 just goes through the array once, performing a comparison and an assignment for iteration of its loop, i.e., at least $2n$ operations.

Note that of course, we have been a bit handy-wavy when counting the number of actual operations. When we would need to calculate the actual number of operations, we would need to inspect the Assembler code, which corresponds to the number of machine instructions that are executed. So if we do so for Algorithm 1, assuming we have an array of unsigned integers as input[4], the Assembler code for the Cortex-m0 instruction set might look somewhat like the code in Figure 1.

As in each loop there are 8 operations: two comparisons, a memory access, two branches (one for the `if` and one for the `for`), incrementing of the pointer to the next element in the array, and an assignment (storing the new current maximum in a register), at the beginning there are 4 operations (push counting as two), and at the end there are 3 operations, we could estimate our runtime as $8n + 7$. This is significantly higher than the $2n$ that we had estimated above.

2.1 Let us go less precise

When comparing algorithms, the exact number of operations is not actually all that interesting. Partially, this is because the number of operations necessary depends on the Assembler code written, or indeed, the Assembler code that the compiler produces (which might be a bit more convoluted than you would have expected – as you have seen in the CSPE1 course). More importantly however,

[2] Of course, that is not its expected behaviour, but it could theoretically happen.

[3] Please note that we are cheating a bit here of course; when implementing this algorithm in Assembler, we would need two memory accesses to retrieve the values of $a[i]$ and $a[j]$, store them in separate registers, perform the comparison, and then branch or not, but, as we will see later, cheating by a constant factor is in fact not a problem.

[4] Note that we had not previously specified this.

```
max1:
   push {r4,lr}
   ldr r2,=0 //currentMax
   ldr r3,=0 //i
loop:
   cmp r3, r1 //assuming r1 = length(a)
   bge done
   ldr r4, [r0,r3] //r0: pointer to the start of the array
   add r3, r3, #1
   cmp r4, r2
   ble loop
   mov r2, r4
   b   loop
done:
   mov r0,r2
   pop {r4,pc}
```

Figure 1: Assembler code implementing Algorithm 1

we expect that it is the largest term – also called the dominating term – that we expect to make the most difference, so we would care about the $8n$ but not about the extra 7. Furthermore, we typically do not care about constant factors all that much, because we are typically interested in how well algorithms *scale up* to larger inputs. So we care that it is something times n, but not about that it is in fact $8n$. This makes the analysis of algorithms significantly easier.

We typically use a so-called *asymptotic upper bound*[5] on the number of operations, as a function of the input to an algorithm, as a measure for its quality. This expresses an upper bound on how quickly the runtime of an algorithm grows as the input changes. We call this the *time complexity*. So, for our `max` algorithms, we can measure it as a function of n, the length of the inputted array. We are interested in the growth of the number of operations our `max` algorithms perform as n becomes larger, in the worst case. For this we will use the *order* or **big-Oh** notation.

The time complexity of Algorithm 1 is order n, or $O(n)$ ("big-Oh n"). That is to say, the number of operations that Algorithm 1 performs is no bigger than $c \cdot n$, for some constant c. That means that if we measure the actual number of operations for Algorithm 1 there is a constant c that makes cn bigger than this number of operations. So, taking our Assembler code for $max1$ above, setting $c = 15$ as the constant factor is sufficient to make the claim that for all non-

[5]The "upper bound" bound means it is a guaranteed overestimation. However, we should immediately note that as we are not interested in constant factors this will be interpreted rather loosely (see Definition 1). We are interested an as tight as possible upper bound. The word "asymptotic" means that we are interested in behaviour in the limit, i.e., even if it is not an upper bound for small inputs, as soon as the input size hits a certain value, it is an upper bound.

empty input arrays (i.e., $n \geq 1$), $8n + 7 \leq 15n$, and therefore Algorithm 1 is $O(n)$. Of course, for different implementations in Assembler, c might have to be bigger than 15, but ultimately, that does not matter for the big-Oh notation.

In general, if the exact number of operations is denoted $f(n)$, i.e., some function of n, then:

Definition 1 *The time complexity of an algorithm, $f(n)$ is order $O(g(n))$, when there exist positive constants, n_0 and c, such that for all values of $n > n_0$, $f(n) < cg(n)$.*

So, in the case of our `max` algorithms, where n is the length of the input array, we can say that the time complexity of Algorithm 1 is $O(n)$, because for all non-empty arrays ($n_0 = 1$) the number of operations is smaller or equal to $cn = 15n$.

When we compare Algorithm 1 to Algorithm 2, we see that there are more operations required in the worst case for Algorithm 2: at least something in the order of n^2 operations. In other words, the time complexity of Algorithm 2 is $O(n^2)$. If you would like, you could check this precisely by writing out the corresponding Assembler code for Algorithm 2, and determining the exact constants required to fill in for Definition 1. However, in practice it is enough to count the number of operations in the pseudo-code, as we did above, leading to $O(\frac{1}{2}n^2 + 2n - 1)$, ignoring the constant factors and constants, leading to $O(n^2 + n)$, and dropping everything but the largest term, leading to $O(n^2)$.[6]

Finally, as Algorithm 3 may never finish, its time complexity is $O(\infty)$. In other words, the conditions for Definition 1 cannot be fulfilled by any finite function in n, and constants n_0 and c.

So, in short, Algorithm 1 is better than Algorithm 2, because its time complexity of $O(n)$ is better than that of Algorithm 2, $O(n^2)$, or (heaven-forbid) that of Algorithm 3, $O(\infty)$.

2.2 Counting the loops

In algorithms that are structured using nested for-loops exclusively, determining the time complexity is often relatively straight-forward. For example, consider Algorithm 4, that creates a list of all groups (tuples) of three students:

[6] Please note that you are only allowed to drop terms that are always smaller than another term. So if you have two variables, e.g., $O(n^3 + m)$, where n and m are different properties of the input, you cannot drop m, unless you can *prove* that $m < n^3$ in all circumstances.

Algorithm 4 groupsOf3

Input: an array of students, of length l: arr
Output: an array of all tuples of three unique students

```
result ← an empty array of 3-tuples of length (l choose 3)        ▷ Choose 3 from l.
m ← 0
for i ∈ 0 ... l−1 do
    for j ∈ i+1 ... l−1 do
        for k ∈ j+1 ... l−1 do
            tup ← (arr[i], arr[j], arr[k])
            result[m] ← tup
            m++
        end for
    end for
end for
return result
```

This algorithm has three nested loops, all of which loop over (almost the entire) list. Hence this algorithm has a complexity of order $O(n^3)$, where n is the length of the list of students.

However, be ware that in the pseudo-code, functions may be used that may be straightforward to implement, but have a non-constant time complexity themselves. For example if we adjust Algorithm 4 to not only store the group of students, but also – for whatever strange reason – how their average hair length compares with the rest of the students in lst, we might get something like Algorithm 5:

Algorithm 5 hairyGroupsOf3

Input: an array of students, of length l: arr
Output: an array of all tuples of three unique students and their average hair length

```
result ← an empty array of 3-tuples of length (l choose 3)        ▷ Choose 3 from l.
m ← 0
for i ∈ 0 ... l−1 do
    for j ∈ i+1 ... l−1 do
        for k ∈ j+1 ... l−1 do
            tup ← (arr[i], arr[j], arr[k])
            relativeHair ← avgHairLength(tup) - avgHairLength(arr \ tup)
            result[m] ← (tup, relativeHair)
            m++
        end for
    end for
end for
return result
```

where $A \backslash B$ is the set A without the elements of B. We abuse this notation a bit:

we see use the array, and the tuple, as though they were sets. $\binom{l}{3}$ are binomial coefficients, please refer to `https://en.wikipedia.org/wiki/Combination` if you are unfamiliar with this concept. With respect to this new algorithm we should note that computing the average hair length (assuming that we have stored this for all students), would also require us to loop over all students of $lst \setminus tup$ every time, making the algorithm's complexity $O(n^4)$.

In general, we call algorithms that run in order $O(n^c)$, where c is a constant, polynomial-time algorithms.

Definition 2 *The time complexity of an algorithm is* **polynomial***, when its time complexity is $O(n^c)$ for a finite constant c.*

By the way, it is possible to make the time complexity of Algorithm 5 $O(n^3)$ again. Can you figure out how? During the seminar (werkcollege), this will be one of the exercises.

2.3 Problems and their complexity

The complexity of problems is typically expressed in terms of the time complexity of the algorithms that solve them. As an example, Table 2.3 contains common tasks that are used in many applications and/or more complex algorithms.

Task	**Complexity**	**Algorithm**
Looking up a value in a lookup table (as in CPSE1)	$O(1)$ *(constant)*	array access
Calculating the maximum value in a sorted array	$O(1)$	array access
Checking whether an item is in a sorted array	$O(\log n)$	binary search (see Section 6)
Calculating the maximum value in an unsorted array	$O(n)$ (linear)	loop over the array (Algorithm 1)
Sorting a list	$O(n \log n)$	e.g., Merge sort (Section 7)

Table 2: The complexity of well-known problems through associated algorithms. For a plot of these different functions (n, $n \log n$, etc.) see Figure 2.

Note that not all complexities are simply exponents of n. Logarithms appear in complexity results a lot as well. We will explain later why this is the case.

Furthermore, note that for some tasks the exponents might not be whole. For example, the best complexity result for the multiplication of two $n \times n$ matrices – which is a key subroutine in graphics, computer vision, machine learning, etc. – is $O(n^{2.3728639})$ using an algorithm invented in 2014 by Le Gall [3]. In fact, matrix multiplication is such a common operation, that in other papers that

use matrix multiplication as a subroutine, the complexity of this multiplication is written as $O(n^\omega)$, where the omega stands in for the exponent for the latest possible complexity results for matrix multiplication. It has been proven that ω is bigger than 2, but how low it can actually go is still the subject of ongoing research.

3 Recursive Algorithms

In the previous section, we have discussed that were structured around nested for-loops. However, there are also algorithms that are structured quite differently. For example, let us take the classic example of (a naive algorithm for) computing the Fibonacci numbers (Algorithm 6).

Algorithm 6 recFibonacci

Input: a non-negative integer n
Output: the n-th Fibonacci number

```
1: if n = 0 then
2:     return 0
3: end if
4: if n ≤ 2 then
5:     return 1
6: end if
7: return recFibonacci(n − 2) + recFibonacci(n − 1)
```

This algorithm is *recursive*, i.e., it calls itself. There are no loops, but clearly its runtime increases as n increases, and rather steeply at that. So how do we analyse the time complexity of this algorithm?

First, we observe that within one call to the algorithm there are only simple computations (two comparisons and an addition; it is only the function calls that make it expensive to execute. So, what we should count here is the number of function calls that Algorithm 6 does – recursively – as n increases.

Second, we observe that for each increase of n, the number of function calls roughly doubles. That is, when n is larger than 2, two new function calls are performed, specifically nrecFibonacci$(n-2)$ and nrecFibonacci$(n-1)$, which in turn (if $n-2$ and $n-1$ are still bigger than 2, will *both* execute two function calls. So the first call triggers 2 function calls, which will trigger 4 function calls, which will in turn trigger 8 function calls, and so on, until the base cases are reached. This leads us to the conclusion that this an instance of an exponential number of function calls with base 2. So the time complexity of Algorithm 6 is $O(2^n)$.

Note that this is significantly worse than anything polynomial. See for example Figure 2, where 2^n (the dashed orange line) is plotted against other common complexity functions. Note there is only one function worse than exponential: the factorial function ($n!$). Exponential time complexity is worse

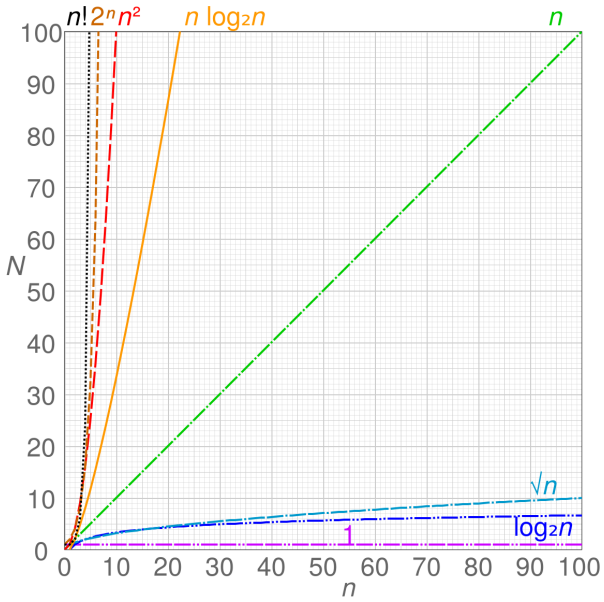

Figure 2: Common functions in complexity results as a function of input. Original figure by "Cmglee", taken from `https://commons.wikimedia.org/wiki/File:Comparison_computational_complexity.svg`.

than polynomial complexity. Even for really bad polynomials with high (but constant) exponents, the exponential function 2^n will eventually overtake it and grow significantly more steeply. For example, 2^n overtakes n^{42} at 356 (rounded) already. Therefore, in Algorithmics it is critical to avoid algorithms with exponential time complexity whenever possible. In the next section, we discuss two tricks that may help to avoid such exponential time complexity results, with a trick called "Memoïsation" (Section 4).

Of course, for some problems it is not possible – or at least highly unlikely – to get anything better than exponential time complexity. Furthermore, some problems are not solvable in finite time at all (not even exponential time). For more on this see Section 16.

3.1 A more precise runtime analysis*

This subsection is for the diehards, i.e., this will not be on the test, so do not worry if you do not get this, it is just a bit of additional information and fun.

After reading the explanation of the $O(2^n)$ complexity results, you might object along the following lines: *"Alorithm 6 does indeed make two function calls, but because in one of them n is decreased by 2 rather than by 1, the number of function calls does not quite double!"* and indeed you would be right.

While the time complexity of naive recursive Fibonacci is exponential, the base is in fact a bit better than 2. If we want to find out how much better, we need to perform a few mathematical tricks. First, let us state the runtime of the algorithm as an equation:

$$runtimeFibo(n) \approx runtimeFibo(n-2) + runtimeFibo(n-1).$$

Note that we are ignoring constants (e.g., the number of computations executed inside a function call without the subsequent function calls) here, but they do not really matter anyway for complexity results. Now, for *runtimeFibo* let us fill in an exponential function with an unknown base x,

$$x^n \approx x^{n-2} + x^{n-1},$$

divide by x^{n-2},

$$x^2 \approx 1 + x,$$

and then we can solve $x^2 \approx 1 + x$ as an equation (noting that we are only interested in positive solutions for this equation). Doing so – with the help of the ever faithful Wolfram Alpha on the internet[7] – yields:

$$x \approx \frac{1+\sqrt{5}}{2} \approx 1.62,$$

rounded upwards.[8] So the actual complexity of Algorithm 6 is approximately $O(1.62^n)$. This is still exponential – and therefore really bad indeed – but slightly better than $O(2^n)$.

3.2 Effective recursive algorithms and time complexity

The naive recursive Fibonacci example of Algorithm 6 makes recursive algorithms look bad. Especially as, as we will see in the next section, much more efficient algorithms are available. But why is it so bad?

The main reasons for Algorithm 6 being bad are the high *branching factor*, low *sub-problem size* reduction, and high *recursion depth*. That is, a recursive algorithm typically reduces the problem at hand (in this case computing the n-th Fibonacci number) by calling the same function of a smaller instances of the problem called *subproblems* (i.e., computing the Fibonacci numbers for $n-2$ and $n-1$). Then it combines the results for these subproblems by doing some additional computation (in this case, adding them up). This is what is known as a *divide-and-conquer* strategy.

The steps of a recursive divide-and-conquer algorithm are:

[7] `https://www.wolframalpha.com/input/?i=x%5E2+%3D%3D+x+%2B+1`

[8] Coincidentally, $\frac{1+\sqrt{5}}{2}$ is a rather special number that crops up a lot in mathematics and nature in general, known as the "golden ratio".

1. **The base cases** – If the problem is smaller than a certain size (e.g., Fibonacci for $n \leq 2$), solve directly. Typically the problem is cut down to such a small size that it takes very little time to solve.

2. **Identifying subproblems** – If the problem is bigger than the base cases then hack up the problem into smaller subproblems (e.g., $n-2$ and $n-1$ for Algorithm 6). Of course these smaller problems should be useful, in the sense that it is easier to solve the full problem if the results for the subproblems are known.

3. **Recursion** – Calling the function on the subproblems. The number of subproblems for which the function is called is the *branching factor*.

4. **Recombination** – using the results for the subproblems to more quickly find a solution to the full problem.

The *recursion depth* is the maximum of the *recursion depths* for the subproblems plus 1, with the recursion depth of a base-case being 1 – which is of course (as it should be) a recursive definition. The recursion depth of Algorithm 6 for an input n is n: the recursion depth of its largest subproblem (size $n-1$), plus 1, i.e., $\mathit{depth}(n) = \mathit{depth}(n-1) + 1$, so, $\mathit{depth}(n) = n$.

Effective recursive algorithms:

1. cut up the problem in much smaller subproblems (reducing the recursion depth), and

2. never compute a solution for the same subproblem twice (limiting the branching factor as much as possible).

Algorithm 6 fails on both accounts: the subproblems are too large, and the same subproblems are solved many times. In the next section, we will tackle problem 2, and show that this is in fact sufficient to make Algorithm 6 efficient.

For the diehards* The runtime $\mathcal{T}$ of a recursive algorithm, for an input size n, can be written as an equation:

$$\mathcal{T}(n) = \mathcal{T}_{\texttt{recombine}}(n) + \sum_{i \in \texttt{subproblems}} \mathcal{T}(n_i).$$

where $\mathcal{T}(n_i)$ is the time it takes to solve subproblem i assuming that the size for that subproblem is n_i, and $\mathcal{T}_{\texttt{recombine}}(n)$ is the time it takes to combine the results for the subproblems into the solution for the original problem of size n. For Fibonacci, we already worked out this equation in Section 3.1.

A great example of a recursive algorithm is `mergeSort` (which we will discuss in Section Section 7.4), which sorts an array of length n by recursively splitting the array in half (i.e., the subproblems are of size $\frac{n}{2}$, sorting both halfs, and then combining these two sorted lists in only n time. This makes the above equation $\mathcal{T}(n) = n + 2\mathcal{T}(\frac{n}{2})$, which when you solve it is just $O(n \log n)$ (just

trust us on that one). If you want to know more about this splendid algorithm, stay tuned for next lecture and Section 7.4.

You will not be asked to solve such equations on the test – we will provide you with the results if needed, and when you need them in practice you can use equation solvers (such as Wolfram Alpha) to find the answers for you. It is however illustrative and useful to know where these complexity results come from.

4 Memoisation

One of the reasons why carelessly written recursive algorithms (and the resulting code) can get out of hand in terms of complexity (and resulting actual runtime) is that an answer for the same subproblem is computed recursively multiple times. For example, in the naive Fibonacci algorithm (Algorithm 6), if we call the function with $n = 6$, the function call with $n = 3$ happens three times ($5, 4, 3$ $4, 3,\ 5, 3$), and then recurses in the exact same way. This is unnecessary: the Fibonacci numbers do not change, so the function call with $n = 3$ will return the same result every time.

An elegant way of preventing this is "memoisation". This term comes from the Latin word *memorāre*: to remind or bring to mind. Basically the idea is: the algorithm will store the answers to the subproblems, and will immediately return the answer the second (or third or forth...) time the function is called with the same arguments. For Fibonacci the resulting algorithm is Algorithm 7.

Algorithm 7 mFibonacci

Input: a non-negative integer n,
an array `answers` of at least length $n + 1$ (default value `null`)
Output: the n-th Fibonacci number

1: **if** `answers` $=$ `null` **then**
2: `answers` $\leftarrow$ a new array of length $n+1$ (at least length 2) filled with -1
3: `answers`$[0] \leftarrow 0$
4: `answers`$[1] \leftarrow 1$
5: **end if**
6: **if** `answers`$[n] = -1$ **then**
7: `answers`$[n] \leftarrow$ mFibonacci$(n-2,$ `answers`$)$ + mFibonacci$(n-1,$ `answers`$)$
8: **end if** ▷ *Make sure to pass* `answers` *by reference.*
9: ***return*** `answers`$[n]$

This algorithm is much more efficient than Algorithm 6; the recursion is only being done when the value is not known yet. Each value in the array is only computed once, and returned immediately thereafter. So, a function call to mFibonacci with a specific n might happen twice at most, but then the recursion stops. This means that the complexity is reduced from (about) $O(2^n)$ in

Algorithm 6 to $O(n)$ in Algorithm 7. From exponential to linear complexity is about the biggest complexity gain possible in algorithmics, and all we needed to do is just remember something!

In general, to apply memoisation, an array is created with the necessary number of fields to hold all the necessary results for the subproblems. Such an array - plus associated functions to set and get the right elements of this array - is called a *lookup table* in the context of memoisation. We used this name in Table 2.3, where we indicated that looking up a value in a lookup table is $O(1)$, i.e., constant, which is as cheap as it can get.

Please note that – unlike in the Fibonacci case – the subproblems might not just be integer values between 0 and n. Therefore, some translation might be necessary to find the right indices in the array. As long as this translation from the arguments of the function to the right array indices is cheap (i.e., $O(1)$), this is of course not a problem.

Furthermore, please note that the array that holds the results for the subproblems might have to be 2 or 3 dimensional. You will see an example of such a problem in the practical assignments.

4.1 At Runtime Memoisation and Compile-time Memoisation

At this moment you might recall that during CSPE1, you created a *lookup table* already. Specifically, in the assignement with the clock, you created a lookup table to hold the answer to various calls to a function that (helped) determined the positions of the hands of your clock. Again, this is an instance of only ever computing a value for the same arguments of a function call once. In other words, you already applied memoisation before. Albeit for an entirely different use case. (Is it not awesome how the same idea can help you with multiple problems?)

In CPSE1, the lookup table was created to not only ever compute the values in the lookup table once (though that is indeed welcome), but also – and in that case more importantly – to move the computation from runtime to compile time, so that the computation can be done when more computational resources are available (i.e., on your laptop rather than on the Arduino). This is a common feature of algorithmics for embedded software: you need to analyze the time complexity on the embedded system separately from the time complexity at compile time, or other parts of the system where more computational resources are available (e.g., imagine a robot that is connected to a computer via Wifi or Bluetooth that might perform intensive computation for it). Often, it will be worth it to move computation from the embedded platform to other hardware. In other words, keeping the time complexity of the algorithms that need to run on the microprocessor is key to making these systems run smoothly.

4.2 Memoisation is not free

"There ain't no such thing as a free lunch" as they say. This is a core principle in much of computer science – in optimisation there is even a formal "no free lunch theorem". Memoisation is often awesome, and can reduce the runtime of algorithms tremendously (as we have seen with Fibonacci). However, it does come at the price of having to use more memory.

This is worth reminding oneself of, as embedded systems typically not only have limited computational capacity, but also limited memory. In the case of the clock, the extra memory used by the lookup table was actually less than the amount of memory that it would have taken to load the functions required instead onto the Arduino, so there all was well. But of course this is not always the case. When the lookup table has a lot of fields (and 2-, 3-, 4-, etc. dimensional lookup tables quite quickly have quite a lot of fields), it is typically more memory-intensive to put a lookup table in memory, than to put the functions in memory that are needed to compute its values.

The other way around is often also possible: reducing the amount of memory used by doing more computation. In fact, data compression (which, incidentally, you also did during CPSE1 with the LZ-compressor and -decompressor assignment), is a great example of this.[9]

4.3 Memory complexity

In algorithmics, it is often possible to trade of memory for computation, and vice versa. Both are key resources, so we need to be careful about both. Therefore, we are often not only concerned with *time complexity* but also with *memory complexity*.

We can analyse the memory complexity of an algorithm in much the same way as we analyse the time complexity of algorithms. For example, Algorithm 7, holds a lookup table with $n + 1$ values, i.e., $O(n)$. An n by n lookup table would be order $O(n^2)$, an n by m lookup table $O(mn)$, and so forth.

Again, of course, it is not quite that straight forward. Sometimes, we use Data Structures (see Section 5) that may be dynamic in size, so we would have to analyse it is largest possible size during execution. Again, we are interested in the worst-case scenario.

Furthermore, not all memory is created equal. One might for example argue that the naive recursive Fibonacci implementation (Algorithm 6) also has a memory complexity of $O(n)$. Specifically, function calls require putting things

[9] For an even more extreme – and admittedly less-than-serious example, see πfs (the π file system, `https://github.com/philipl/pifs`). πfs achieves sheer-infinite compression, by using sequences of digits of the number π. This is in fact possible: all finite sequences of digits (i.e., bytes) exist in the digits of π, so you just need to remember the right starting point as it were to retrieve a file. Of course, there are no guarantees as to how long it will take you to find this starting point though... Sheer-infinite compression (very little memory) for sheer-infinite computation (as the github page puts it "Why is this thing so slow? It took me five minutes to store a 400 line text file!").

on the stack[10], specifically the value of the link register and all values kept in protected registers, so a function call typically requires $O(1)$ memory (unless e.g., arrays of order $O(n)$ or higher are copied with the function call). But recursion does a whole series of function calls, making the the stack grow. The worst case of the number of link registers (and other things) on the stack is exactly the *recursion depth* (that we defined Section 3.2), which for Algorithm 6 (and Algorithm 7 for that matter) is also n.

Finally, please keep in mind that different types of memory exist. Typically, compilers reserve a limited amount of memory for the stack, aside from the rest of the memory reserved for the program. This stack size is typically much smaller than what is reserved for the rest of the memory. It is therefore possible that a recursive algorithm might give a stack-overflow error, while an equivalent non-recursive algorithm with the same memory complexity may not give an out-of-memory error. It is usually possible to reserve a larger amount of memory for the stack via compiler arguments.

5 Data Structures

In the previous sections, we have discussed time and memory complexity, and seen that algorithms have a profound influence over both. However, algorithms specify the structure of the code, while this is not all that determines the runtime and the memory usage; the other key component of keeping runtime and memory usage in check is how to store data. This section is about exactly that: data structures.

5.1 Containers

A highly prevalent type of data structure is *containers*. Containers are objects whose sole purpose is to store and retrieve other objects efficiently. As such, containers are often templated in programming languages. For example, an array *of integers*, a vector *of floats*, a list *of students*, etc.

Containers may implement different methods, depending on their intended usage, but ultimately, they all have some basic methods/features:

- A way to *store* objects in the container
- A way to *access* objects in the container
- A way to *traverse* the objects in the container (i.e., systematically go over each individual object) in the container.

Object traversal is often referred to as *iterating* over the objects. As such, containers often implement an *iterable* interface in programming languages. The container then is able to return an *iterator* (possibly itself) which can be initialised, and implements a function to retrieve the "next" object.

[10] See Section 5.5 for a further explanation of stacks as a data structure.

This of course sounds very abstract, so let us look at different instances of, and types of containers.

5.2 Arrays

The most common container data structure is probably the array. The array stores objects on a given index ($a[i] = x$), and access is also done using this index ($x = a[i]$). Traversing an array is also straightforward: one can loop over all the indexes in the array, and because an array has a fixed size (e.g., N), the indices are always from 0 to this size minus one ($N - 1$).

Arrays are simple, easy to translate to assembler code, and implemented in almost every modern programming language. Of course, arrays also have downsides: they are fixed in size, the elements of the array need to be initialised somehow, and locating and replacing objects in the array may be less than efficient.

That being said, arrays often form the basis of more extensive data structures that have more capabilities and/or additional properties (desirably for their intended use). Note though, that unless special tricks are applied – see Section 8.1 – data structures implemented using arrays will have a fixed capacity.

5.3 Sets

An example of a container class that you have already implemented (in CPSE1) is the Set. A set is a container with a special requirement: it may not contain duplicates. Therefore, when adding an element to the set, it needs to checked whether the set already contains the given object. A result of this, is that a set requires objects that can test for equivalence (e.g., via the `==` operator).

When the set has a fixed (max) capacity, it is possible to implement a set on top of an array. Specifically, a set class would need:

- an array of length `maxCapacity`,
- a counter with how many elements are currently in the set, and
- an `insert` method that respects the set property (i.e., it needs to check whether an equivalent object is already in the set, and not insert the item if it is already in the set)
- in addition to methods to *remove* and *traverse* objects in the set, but those are relatively standard.

These methods are all little algorithms in their own right. It is key to keep the time complexity of these methods in check, as containers are typically used frequently by the programs. Furthermore, it is often the case that implementing the data structure in one way will lead to efficient insertion, may lead to slower removal or more memory usage. The best way to implement a container may thus heavily depend on which methods are being called most frequently (e.g., inside of another algorithm).

So, let us have a look at possible algorithms for *insert* (Algorithm 8) and remove (Algorithm 9 for the above-mentioned array-based implementation of a Set.

Algorithm 8 set.insert(x)

Object variables: an unsorted array `elems`, an integer `size`
Input: a new object to insert x
Output: boolean indicating whether x was successfully inserted

```
1: if size=length(elems) then
2:     return false          ▷ max cap.
3: end if
4: for i ∈ 0...size−1 do
5:     if elems[i] = x then
6:         return false
7:     end if                ▷ already in set
8: end for
9: elems[size] = x           ▷ add to set
10: size++
11: return true
```

Algorithm 9 set.remove(x)

Object variables: an unsorted array `elems`, an integer `size`

Input: an object to remove x

Output: boolean indicating whether x was successfully removed

```
1: for i ∈ 0...size−1 do
2:     if elems[i] = x then
3:         elems[i]=elems[size–1]
4:         size−−
5:         return true
6:     end if
7: end for
8: return false
```

For both *insert* and *remove* the time complexity is $O(\texttt{size})$, i.e., the number of elements currently in the set.`size` is in turn limited by the maximum capacity of the array. For remove, this would also be the case if it were not a set, as in order to remove an object from a collection implemented with an unsorted array, all elements that are currently in that collection could have to be compared against x to test whether x is in that collection. For *insert* however, the fact that it takes $O(\texttt{size})$ time is caused by the fact that it is a set, i.e., because of the condition that the set may not contain doubles, it must be checked whether x is already in the set, which takes $O(\texttt{size})$ time. Adding x without checking whether x is already in current elements would only take $O(1)$ in a collection implemented using an unsorted array. So, the set property makes insertion significantly more difficult.

So, could inserting into a set be made cheaper? For example, what if the array of elements were sorted? In that case, finding the element in the sorted array would be cheaper (as we will explain in Section 7). However, inserting an element into a sorted array could still take $O(\texttt{size})$, i.e., when the new object is the smallest, it should be inserted at index 0, which means that all the elements currently in the array would have to move one place to the right (that is, to their current index+1). As such sorting the array does help to make insertion cheaper. So, is there another way? Indeed there is, but not using an array. In the next lecture, we will have a look at a completely different data structure (i.e., trees) that could be used to make inserting and removing objects cheaper.

5.4 Array lists

Now, let us compare to a container data structure that does not require the set property, but is implemented using an array as its basis. This is called an ArrayList[11].

When such an ArrayList has a fixed (max) capacity, it is possible to implement a set on top of an array. Specifically, a set class would need:

- an array of length `maxCapacity`,
- a counter with how many elements are currently in the ArrayList, and
- an `insert` method, as well as a *remove* method.

The remove method is identical to that of set (Algorithm 9, in Section 5.3). However, the insert method becomes much simpler. Because no check needs to happen whether x is already in the set, inserting into an ArrayList takes only $O(1)$ (constant) time.

So, the ArrayList can have a variable number of objects contained. Note however, that because the array is always `maxCapacity` long, the memory requirements are always the same.

We further note that it is possible to get around the max capacity by a trick. That is, when trying to insert a new item to a "full" ArrayList, we could simply create a new array to contain the elements that is much longer, and copy all elements from the old array to the new array. Note however, that this takes $O(n)$ time, so this would every now and then seriously impact the runtime of insertion into an array list. Nonetheless, standard implementations of ArrayLists in various programming languages typically do this. We will explain why this is so in Section 8.1.2.

5.5 Stacks

A container type data structure that can be implemented highly efficiently using an array (if having a maximum capacity is okay), is a Stack. Stacks are highly prevalent in the literature, and are indeed one of two data structures typically implemented on chips.

A stack is a container a simple requirement:

- The insert and remove methods can only change the "top" op the stack, according to the last-in-first-out (LIFO) principle.

This explains the metaphorical name "stack". On a stack of objects (e.g., plates), only the top can be safely removed, and an object can only safely be added to the top. In a stack, the insert method is called *push*. The remove method is called *pop*. An illustration of a stack is given in Figure 3

[11] Though, abusively, the name `vector` is used for this in C++.

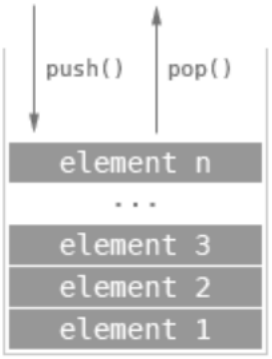

Figure 3: A graphical depiction of a stack, taken from `https://commons.wikimedia.org/wiki/File:Stack_data_structure.gif` (created by Hyperthermia).

Algorithm 10 stack.push(x)

Object variables: an array `elems`, an integer `n` (current number of elements in the Stack)
Input: a new object to insert x
Output: boolean indicating whether x was successfully pushed

1: **if** $n =$ `length(elems)` **then**
2: **return** `false` ▷ max cap.
3: **end if**
4: `elems`$[n] = x$ ▷ add to stack
5: $n{+}{+}$
6: **return** `true`

Algorithm 11 stack.pop()

Object variables: an array `elems`, an integer n (current number of elements)
Output: the last object added to the stack

1: **if** $n = 0$ **then**
2: **return** `null`
3: **end if**
4: $n{-}{-}$
5: **return** `elems`$[n]$

Due to these simple requirements both inserting (pushing, Algorithm 10) and removing (popping, Algorithm 11) objects on/from stacks is order $O(1)$ time.

5.6 Queues

A different data structure with slightly more complex requirements is the *queue*. Like a queue at the supermarket, the requirements are:

- The insert (queue) method can only change the "back" of the queue, according to the first-in-first-out (LIFO) principle.

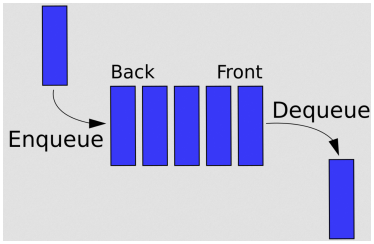

Figure 4: A graphical representation of the Queue data structure and its methods. This figure was taken from `https://commons.wikimedia.org/wiki/File:Data_Queue.svg`, and was created by User:Vegpuff.

- The remove (dequeue) method can only change the "front" of the queue, according to the first-in-first-out (LIFO) principle.

A graphical depiction of a queue is provided in Figure 4.

Because of the slightly more complex requirements, it is slightly more complex requirements the data structure must now keep track of where the front and the back of the queue is. However, if the queue is implemented via an array, it is still possible to have both insertion (queuing) and removal (dequeuing) in $O(1)$ time. This is possible because the queue can be implemented in a circular manner, so that the elements in the array still do not need to be shifted when elements are added or removed.

Algorithm 12 queue.queue(x)

Object variables: an array `elems`, three integers n (the current number of elements), and `front` and `back`
Input: a new object to insert x
Output: boolean indicating whether x was successfully queued

```
1: if n=length(elems) then
2:     return false        ▷ max cap.
3: end if
4: elems[back] = x ▷ add to queue
5: back++
6: if back = length(elems) then
7:     back ← 0
8: end if
9: n++
10: return true
```

Algorithm 13 queue.dequeue()

Object variables: an array `elems`, three integers n (the current number of elements), and `front` and `back`
Output: the oldest object added to the queue

```
1: if n = 0 then
2:     return null
3: end if
4: result ← elems[front]
5: front++
6: if front = length(elems) then
7:     front ← 0
8: end if
9: return result
```

Please note that, in the algorithms the front is the leftmost element, rather than

the rightmost, contrary to Figure 4.

5.7 Dictionaries

Note that the topic of dictionaries is not discussed during the lectures, but will be on the test. Therefore, please read this section carefully, and ask questions to your "klassedocent" if you have any.

Another common datatype is the dictionary. A dictionary is a container in which objects, called `value`s, in the container are identified using a `key`. For example, we could have a dictionary in which student objects are identified via a student number (an int) as a key. Dictionaries store (`key`, `value`)-tuples.

In order for keys to correctly identify the accompanying values, it is required that each key is unique. Dictionaries are thus highly similar to sets (Section 5.3), with the difference that the objects are stories in a (`key`, `value`)-tuple, and that it is the `key`s that are required to be unique, rather than the objects themselves (i.e., `value`s need not be unique).

Because of the similarity between sets and dictionaries, each algorithm (insertion, removal, search, and traversal), can be implemented in the same way for a dictionary as for a set, with the same time complexity.

5.8 Hashmaps

Note that the topic of HashMaps is not discussed during the lectures, but will be on the test. Therefore, please read this section carefully, and ask questions to your "klassedocent" if you have any.

A common way to trade off memory for computation, at least in expectation, is HashMaps. A HashMap is a specific case of a lookup table (see Section 4) where the index in the table of an object is its *hash*. This hash value is determined by a hash function, which does nothing other than to compute some (rather unstructured) value to function as the index (or key if you will) for insertion and search in the array.

A HashMap consists of the following components:

- A hash table (typically an array)
- A hash function (to determine the indices of objects in the table), and
- A collision strategy, i.e., what to do if – as misfortune might have it – two objects have the same hash.

For example, let us take a HashMap based on an table (i.e., an array) of length 7, and let us use $f(x) = x \mod 7$ as the hash function (mod being the modulo function, denoted % in many programming languages). If we now insert the values 14, 78, 34, 79, and 43, be would obtain the hastable in Figure 5. In this figure we see that at index 1, there are now two elements. This is called a *collision*. Of course, the reason for this collision occurring is that we have made the hash table much too small. In order for collisions to occur as little

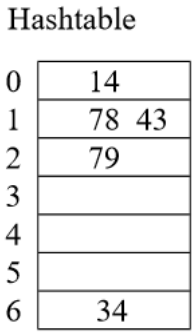

Figure 5: A example of a hashtable with a collision at index 1. Figure by Joop Kaldeway.

as possible, we want to make HashMaps that are sparsely populated, i.e., with much more space than actual objects in the table. However, it is never possible to completely rule out collisions. Therefore, we need a collision strategy.

For example, in a *linear hashing collision strategy*, we would simply try to insert the new value in the next index, and if that is full, in the subsequent index, and so on. However, this may lead to larger clusters of filled fields. This is undesirable, as it makes searching more expensive. That is, when we need to search for an object later, we need to search from the has value until the next empty spot, if we want to be certain that the object we are searching for is not in it. Slightly more sophisticated collision strategies therefore take bigger jumps in indices when collisions occur.

The collision strategy for example used by the JAVA programming language is to have another container (e.g., a list as we will define in Section 8.1) at each index, rather than inserting the objects directly into the hash table. Of course, these containers will take more time to search (see Section 6). But, as these containers will contain very few elements, this search time will typically be negligible in practice (as long as the HashMap is sufficiently sparsely populated).

6 Search

A key algorithm connected to container data structures that we have not yet discussed is search, i.e., *"How do find an object in a container"?* Sometimes, this is just to check whether we already have the object in the container, and sometimes, it is to retrieve the correct object (e.g., the correct value for a given `key` in a dictionary).

Searching for objects is a highly common operation, and often the most intensively used method for container-type data structures. We also note that any remove method needs to search for the object in the container as well.

If we have an unsorted array as the basis for storing our objects, we have

already seen that finding an object takes order $O(n)$ time. That is, we have to loop over the entire array in order to find the object in our container.

However, it can help if we impose the constraint that the objects in the array are stored in a sorted order. So, we will make sure that element 0 is smaller than element 1, element 1 smaller than 2, and so on.[12]

6.1 Logarithmic time complexity

In this subsection we assume that you know what a logarithm is. We will be using logarithms with base 2. If you are unfamiliar with logarithms, please check out one of the many internet tutorials on the matter. For example the one by Khan academy[13]*.*

Counting the number of nested for-loops is sufficient for determining the time complexity of many algorithms. However, for some tasks it is possible to be faster than polynomial. If we get a *sorted* array (e.g., of integers) as input, it is possible to test whether a given value is in that array in less than $O(n)$ time – $O(n)$ being the complexity of just looping through the entire array.

Algorithm 14 binarySearch

Input: a sorted array: a, and an item to find: x
Output: either the index of x in a

1: $l \leftarrow 0$
2: $r \leftarrow \texttt{length}(a) - 1$
3: **while** $l \leq r$ **do**
4: m = l + (r - l) / 2;
5: **if** $a[m] = x$ **then**
6: **return** m
7: **end if**
8: **if** $a[m] < x$ **then** ▷ If x is bigger than $a[m]$, ignore everything left of m
9: $l \leftarrow m + 1$
10: **else** ▷ If x is smaller than $a[m]$, ignore everything right of m
11: $r \leftarrow m - 1$
12: **end if**
13: **end while** ▷ If the algorithm gets here, x was not in a
14: **return** not found

Let us have a look at Algorithm 14, which is commonly known as binary search. At each iteration of its while-loop, it is able to cut the "search window" (the part of the array that may contain x) in half by inspecting the middle m of its current search window, i.e., everything between the indices l (left) and r (right). If the middle is smaller than x, x and only be to the right of index

[12] Of course, this means that we have to be able to sort the elements in our array. We will discuss sorting algorithms in Section 7.

[13] `https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:logs/x2ec2f6f830c9fb89:log-intro/a/intro-to-logarithms`

m. This comes from the fact that the array is sorted. Conversely, if m is larger than x, x can only be to the left of x in the array.

The number of iterations required to check whether x is in a sorted array (and return its index if found) is the number of times it takes to reduce the search window to 1 remaining element. For example, if the number of elements would be 16, iteration 1 reduces the search window to 8, iteration 2 to 4 elements, iteration 3 to two elements, and iteration 3 to one element, after which, in iteration 4, we need to check this final element. As you can see, we can check double the amount of elements for every extra iteration i, 2^i. E.g., for 4 iterations we can have $2^4 = 16$ elements. The reverse of this is the $^2\log$ (which we will write as simply log from now on). Therefore, Algorithm 14 has a time complexity of just $O(\log n)$.

7 Sorting

As we have seen in the previous section, finding items in a sorted array is much faster than in an unsorted array. When a lot of searches need to happen, it may thus be well worth it to keep the elements in a collection in a sorted array. Furthermore, for many other algorithms that take an unsorted set of objects as input, it helps to sort the objects beforehand.

In this section, we will first discuss how expensive it is to insert a new object into an already sorted array. Then we will proceed to various sorting algorithms that take an unsorted array as input, and produce a sorted array containing the same elements.

7.1 Inserting one element into a sorted array

Inserting a new object into an already sorted array takes $O(n)$ time. This is not because it is expensive to find the right place to insert the new object; we can use an algorithm highly similar to binary search (Algorithm 14) to return the right location. Specifically, at the end of the algorithm, after the while loop, the algorithm should check whether $a[m]$ is bigger than x, and if so x should be inserted at m, and else at $m + 1$. It is the insertion given the index which takes $O(n)$ time. This is because all elements at and after this index should be moved to the right (to their current index plus one).

Exercise: try writing the pseudo-code for the algorithm described in the text of this subsection, and show this to your "klassendocent".

7.2 Insertion sort

Now that we know that we can insert a new object into a sorted array in order $O(n)$, this begs the question whether we can use this as the basis for an efficient sorting algorithm. The basic idea of this algorithm – known in the literature as *insertion sort* – is shown in Figure 6. In each iteration of the algorithm the part of the array that is sorted increased by one element. I.e., at each iteration,

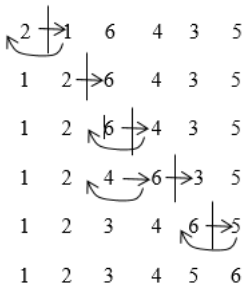

Figure 6: An example of the execution of insertion sort (made by Joop Kaldeway). At every timestep the current index (verticale bar) moves 1 place to the right, and the elements to the left of this index become sorted.

we "insert" the element at index i of the sorted part of the array: elements 0 to $i-1$.

Algorithm 15 insertion sort

Input: an unsorted array: a
Output: a, but now sorted

1: **for** $i \in 1...length(a) - 1$ **do**
2: $x \leftarrow a[i]$
3: $j \leftarrow$ find the place to insert x between index 0 and $i-1$ ▷ $O(\log i)$
4: Move all elements of a between j and $i-1$ one index to the right ▷ $O(i)$
5: $a[j] \leftarrow x$
6: **end for**
7: **return** a

The pseudocode for insertion sort is given in Algorithm 15. Please note that in this algorithm we describe what to do in English, rather than in code-like statements. This is to make the algorithm better understandable to the reader, without going too much into detail. This is possible because we know that the reader, i.e., you, would know how to implement these steps efficiently.

For insertion sort the worst-case input would be an array in reverse order. In that case, at each iteration, it would have to move exactly i elements, as at each iteration the index to insert the new element would be 0. If the length of the array is n, then it has to more $\frac{n}{2}$ per iteration of the outer loop on average. Because this outer loop is executed $n-1$ times, this leads to a time complexity of $O(n^2)$. Surely, we should be able to do better.

7.3 Quick sort

We have seen that the most straight-forward way of sorting, insertion sort (Algorithm 15), has a time complexity of $O(n^2)$. Now let us see whether we can

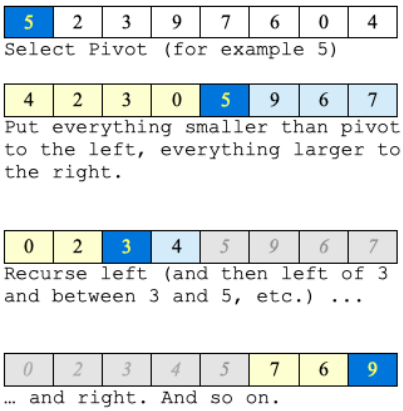

Figure 7: The central idea behind the quick sort algorithm (note, this example does not use a particular implementation, it just depicts the central idea).

do better.

Maybe we can use a similar trick to the one we have used to find an object in a sorted array. So, we would try to start somewhere in the middle, and divide up the elements in everything smaller and larger, having to sort two halves of the array separately – leading to a recursive scheme. This is the central idea behind an algorithm that is known as *Quick Sort*. The element we pick to divide the array up into two (everything smaller to the left, and everything larger to the right), is called the *pivot*. The operation of splitting the array in two parts (smaller and bigger than the pivot) is called *partitioning*. A graphical depiction of this idea is given in Figure 7.

There are now two problems in making this idea work:

1. how to implement the partitioning operation efficiently, and
2. how to pick the pivot such that the array is split in (roughly) equal parts.

For the first problem there is an elegant solution: we can use the swapping of elements in the array. Please take a moment to appreciate (and understand) how this works looking at Algorithm 17, using the partitioning scheme by Lomuto. Using this partioning scheme, QuickSort (Algorithm 16) becomes a straightforward recursive algorithm.

Algorithm 16 quicksort

Input: an unsorted array a, two indices, hi and lo
Output: a, which is now sorted between hi and lo

```
1: if lo ≥ hi then
2:     return a                ▷ base case
3: end if
4: p ← partition(a, lo, hi)
5: a ← quickSort(a, lo, p − 1)
6: a ← quickSort(a, p + 1, hi)
7: return a
```

Algorithm 17 partition

Input: an unsorted array a, two indices, hi and lo
Output: i, an index for where everything left of i is smaller in value, and everything right larger.

```
1: pivot ← a[hi]
2: i ← lo
3: for j ∈ lo ... hi−1 do
4:     if a[j] ≤ pivot then
5:         swap values of a[i] & a[j]
6:         i++
7:     end if
8: end for
9: Swap values of a[i] and a[hi]    ▷
   ... where the pivot is.
10: return i
```

The second problem however, we cannot actually solve effectively. That is because the best possible pivot would of course be the one that splits the elements in two equal parts. If this is successful we would only need to do $O(\log n)$ splits, making the entire algorithm $O(n \log n)$. To find this best pivot however, we would need to know what the middle element of the elements to sort is. But in order to do this, we would have to sort these elements, which defeats the purpose. On average, if we just select a random pivot from the remaiming elements, we will probably be fine. If for some reason however, we always pick either the largest or the smallest element, then we would have to recurse $O(n)$ times, and we would still end up with a $O(n^2)$ time complexity.

As the worst-case complexity is $O(n^2)$, our quest for a sorting algorithm with better guarantees continues. It is worth noting though that due its elegance, good average performance, and little additional memory requirements, it is still often used in practice.

7.4 Merge sort

The problem with quick sort is that there is no way to guarantee that we always choose a good pivot. Therefore, the question becomes, can we split the sorting problem up into subproblems, in a way that does not depend on choosing such a pivot?

Indeed, there is. As the basis for this strategy, we make use of the following observation:

- If we have two sorted arrays, of respectively length l and m as input,

- then merging them into one sorted array of length $l+m$ only takes $O(l+m)$ time.

This is because we can start at index 0 for both arrays, and keep inserting the next element from the array which happens to be smallest. This operation we call `merge`, shown in Algorithm 18.

Algorithm 18 merge

Input: two sorted arrays, a and b
Output: a sorted array containing all elements from a and b.

```
1:  i, j, k ← 0, 0, 0
2:  res ← a new array of length length(a) + length(b)
3:  while i < length(a) ∧ j < length(b) do
4:      if a[i] < b[j] then
5:          res[k] ← a[i]
6:          i++
7:      else
8:          res[k] ← b[j]
9:          j++
10:     end if
11:     k++
12: end while
13: if i < length(a) then
14:     insert the remainder of a to res   ▷ Using a loop from i to length(a)−1
15: else
16:     insert the remainder of b to res
17: end if
18: return res
```

Using this merge operation to combine the results of the subproblems, we can simple split our array down the (index-based) middle, sort both halves, and merge the result. An example of this idea is depicted in Figure 8.

The resulting algorithm is called *merge sort*. Because we can now guarantee that the subproblems are indeed half the size of the original problem (i.e., the entire array that we have to sort), the recursion depth will just be $\log n$. At each recursion though, we have to first put all n elements into smaller arrays, and then merge them together again. This copying and merging takes order $O(n)$ time, making the total complexity of our

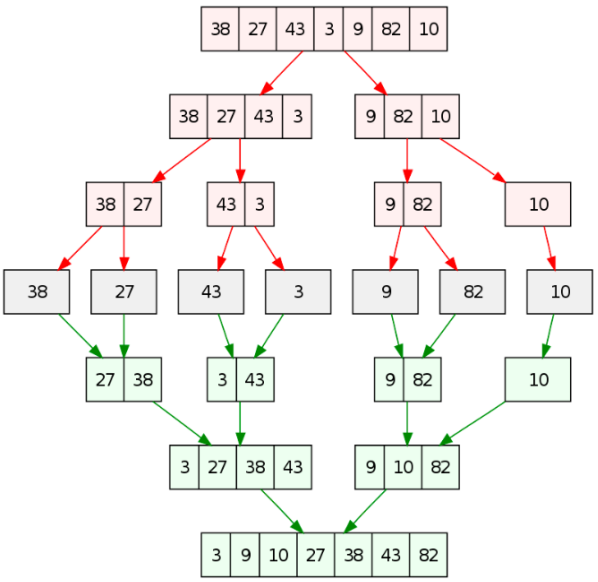

Figure 8: The idea behind the merge sort algorithm. (Image taken from `https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg`, own work by VineetKumar).

Algorithm 19 merge sort

Input: an unsorted array a
Output: a, now a sorted array.

1: **if** $length(a) = 1$ **then**
2: **return** a
3: **end if**
4: $m \leftarrow length(a)/2$ ▷ Cast to integer value
5: $l \leftarrow$ an array containing all elements of a between index 0 and $m - 1$
6: $r \leftarrow$ an array containing all elements of a between index m and $length(a) - 1$
7: $l \leftarrow$ `mergeSort`(l) ▷ This algorithm!
8: $r \leftarrow$ `mergeSort`(r) ▷ This algorithm!
9: $a \leftarrow$ `merge`(l, r) ▷ Algorithm 18
10: **return** a

8 Recursive Data Structures

In the previous sections of this reader – as well as the lectures – we have discussed recursive algorithms, and have seen that this can drastically reduce the time complexity for many common computational tasks. For example, linearly looping over an array in order to find an item is $O(n)$, but performing binary

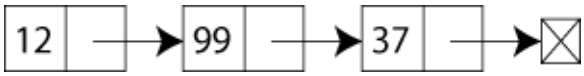

Figure 9: An example list of integers. The first "node" has a head: the integer 12, and a tail, a pointer to the next node in the list, i.e., the list starting from 99. Figure taken from `https://commons.wikimedia.org/wiki/File:Singly_linked_list.png`, own work by Derrick Coetzee.

search (Algorithm 14) on a sorted array only takes order $O(\log n)$[14]. Furthermore, to sort an array, the straightforward approach using two loops, i.e., insertion sort (Algorithm 15) takes $O(n^2)$ time, but using a clever recursive trick we can get to merge sort, which takes only $O(n \log n)$ time.

This of course begs the question, if we can do recursion in algorithms, can we not also use recursion in data structures? That is, if we define data structures in terms of themselves, does that work, and if so what advantages can that bring?

8.1 (Linked) Lists

A well-known recursive data structure is the list. The list is an alternative to an array, i.e., a linear way to store data. Roughly speaking a list consists of:

- A head: an element of the list (e.g., an integer), and
- a tail: a list (i.e., the rest of the list).

If the tail is empty (or `null`) the end of the list is reached.

For example, Figure 9 depicts a small list of integers. The list is stored as objects called *nodes*. One node contains an element, and a pointer to the rest of the list, i.e., the next node.

The main advantage of this data structure over the array is that it has an "infinite" capacity, and is flexible in size. Another advantage is that while iterating over the nodes, it is simple to take one node out, i.e., if we want to remove the node with element 99 from the list, we need to set the pointer to the next node of the node with element 12 to the node with element 37 (and of course destruct the node with element 99).

The main disadvantage of lists with respect to arrays is that it is not indexed. This means that we cannot directly access the third node in the list, but we have to start at the beginning of the list, and follow the pointers to the next node, and the next node again. This means that accessing the x-th node in the list as a time complexity of $O(x)$.

While the list is uncommon in most imperative programming languages (like C++, JAVA and Python) – for reasons we will explain in Section 8.1.2 – it is one of the cornerstone data structures in many functional programming languages, such as Haskell. If you take functional programming as an elective subject later, you can therefore expect to see a lot more of them.

[14] Please remember that in this reader, we mean $^2\log n$ (the logarithm base-2) when writing $\log n$, as is common in the computer science literature.

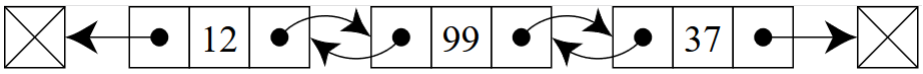

Figure 10: An example doubly linked list of integers. The first "node" has a head: the integer 12, and a tail, a pointer to the next node in the list, i.e., the list starting from 99, as well as a pointer to the previous node in the list, which is `null` in the case of the first node. Figure taken from `https://commons.wikimedia.org/wiki/File:Doubly-linked-list.svg`, own work by Lasindi.

8.1.1 Doubly Linked Lists

Another disadvantage of the singly linked lists of Figure 9, is that is easy to iterate through a list in a forwards way, but it is difficult to go back. Therefore, aside from singly-linked lists, there are also doubly linked lists, in which each node has a pointer to the previous as well as the next node. See Figure 10 for an example.

8.1.2 A key side-note to linked lists

On most PC/laptop processors, using arrays is significantly faster than using linked lists. This is because in a linked list, the memory addresses of the nodes are unstructured and scattered around the (heap) memory. Therefore, object traversal in a linked list is slower, as the processor is accessing relatively random memory locations. In contrast, a contiguous memory structure such as a C-style array has a continuous memory block. Most PC and other "fancy" CPUs will automatically cache contiguous chunks of memory while executing programs. And, as the access time for this cache is orders of magnitude lower than that of RAM, iterating over an array much faster in practice than iterating over a linked list. When using linked lists, this hardware-based advantage will not be available. Therefore, even when using unlimited capacity data structures, the standard implementation in many programming languages (such as C++) is typically still array-based.

When implementing a Stack or Queue with an unlimited capacity via an array, the trick is to just create a new array with double the capacity whenever an element is added to a currently full array. Of course this requires copying all the elements from the previous array to the new array costing $O(n)$ time. In practice, on PCs and other "fancy" CPUs however, the average runtime gain of caching is worth this expense. On microprocessors on the other hand, the caching advantage is typically not there. Therefore, on microprocessors is can still be worth it to use Linked Lists to implement container data structures that have unlimited (or at least uncertain max) capacity.

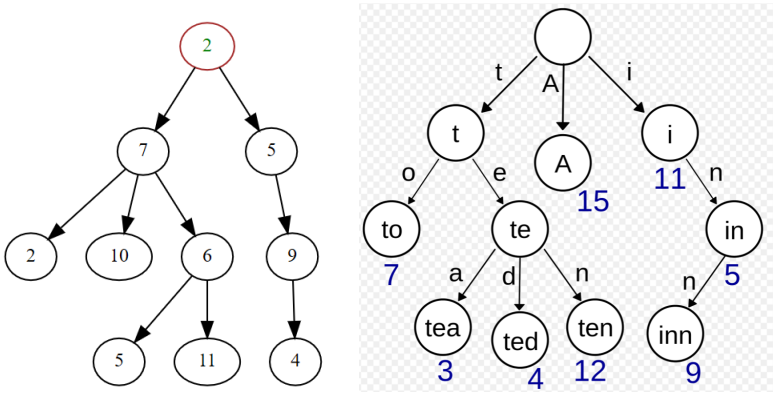

Figure 11: (left) an example of a tree of integers (right) an example prefix tree.

8.2 Trees

Another highly common data structure is the Tree[15]. A tree has a *root* node (i.e., the starting point of the tree). Each node has child nodes. When each node has at most two children it is a *binary tree*. Each child is itself the root of a *subtree*. A node that has no more children is called a *leaf* node.

Binary trees are highly common, but it is of course possible for each note to have more than two child nodes.[16] For example, Figure 11 (left) shows an example tree where the root node (the green 2 in the red circle) has two child nodes. The subtree rooted by the left child node (7) has a total of 6 nodes. The left child node itself has three child nodes (2, 10, and 6), two of which are leaf nodes (2 and 10). In this tree each node contains an integer, in addition to its list of child nodes.

Another example of a tree, this time to index words, is the prefix tree in Figure 11 (right). In this tree, going down the tree means adding to the prefix of a word. So the word "tea" can be found in a leaf node that descends from "te", "t", and "" (the empty root node). Such prefix trees are for example very useful for quickly finding dictionary entries.

Trees are very useful data structures, whose content can be adapted for usage in all kinds of algorithms. In this reader we will scratch the surface in two areas: trees as efficient containers optimised for search as well as well-performing for inserting and removing (Sections 8.3–**??**), which are called search-trees. And trees that are created as part of so-called *tree search algorithms* for optimisation (Sections 11, 12, and 17.2).

[15] Want bomen zijn relaxt.

[16] We could also say that linked lists are a special case of trees: a tree where each node has exactly one child node.

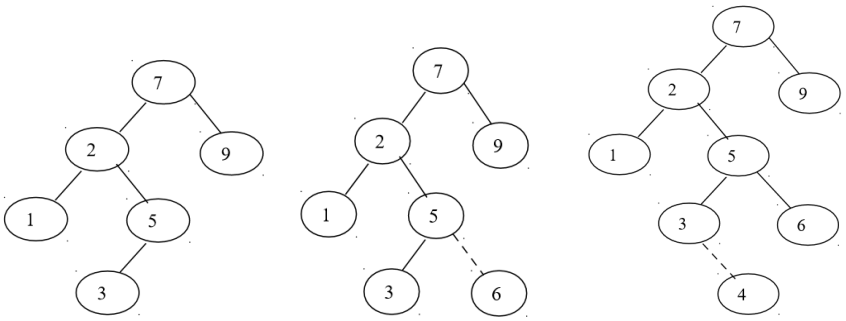

Figure 12: An example of a binary search tree: (left) original, (middle) after inserting 6, and (right) after inserting 4. Figures by Joop Kaldeway.

8.3 Search Trees

In Section 6, in Algorithm 14 we have seen that having an array sorted can reduce the time complexity of search to $O(\log n)$. This is because we can always ignore half of the array after a comparison. However, inserting into a sorted array is $O(n)$, as is removing an object from a sorted array. This is because, to keep the array sorted, we might have to shift (almost) all values in the array to the left (i.e., one index lower) or right (i.e., one index higher). This is unavoidable when using arrays.

So, what if we change the data structure from an array to something more resembling the binary search scheme, for which inserting and removing becomes cheaper. This is called a *binary search tree (BST)*. A BST has the following conditions:

- If n is a node in a BST, with an element of value v, then,
- all left descendants of n (i.e., its left child node, both its child nodes, their child nodes and so on...), n_d have values v_d that are smaller than v, i.e., $\forall(d \in \texttt{left\_desc(n)}) : v_d < v$, and,
- all right descendants of n have values larger than v, i.e., $\forall(d \in \texttt{right\_desc(n)}) \ : \ v_d > v$.

If double values are allowed, then equal values should be place on either the left or the right *consequently*. An example of a BST can be found in Figure 12 (left).

Searching in a BST follows the same scheme as binary search for arrays. However, rather than a search window (two indices) in the array, now a current node in the tree is maintained; the window being it and all its descendants. For example, if we want to look for 5 in Figure 12 (left), we start at the root: 7. $5 < 7$ so we go to its left child, 2. $5 > 2$, so we go to its right child, which brings us to 5, and we can return the node containing 5 (or just `true` depending on what we need).

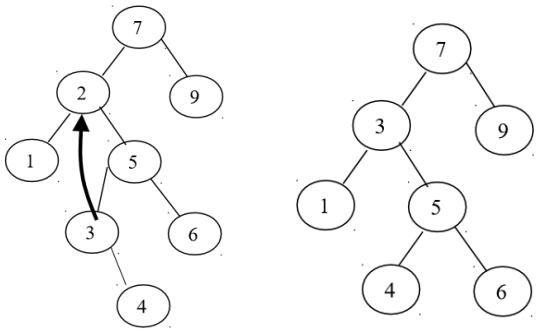

Figure 13: An example of a removal binary search tree: (left) we want to remove 2. (right) after removing 2. Figures by Joop Kaldeway.

Inserting in BSTs is straightforward too. We follow the tree down the same way as in search, and insert a new node as soon as we do not find an empty spot. For example, when inserting 6, we would follow the tree down the same path as we did in searching 5. Then, because $6 > 5$ we would want to go right, but 5 has no right child yet. This means that a new node with 6 can be inserted exactly at this spot, leading to the situation in Figure 12 (middle).

Removing from a BST is a little more complex, as we might want to remove a node that is right in the middle of the tree "orphaning" its children. For example, let us assume we want to remove 2 from the BST in Figure 12. This would orphan the 1 and the 5 node. It is not possible to simply bring up one of them, as this may violate the BST properties. This can however be fixed by:

- bringing up the left-most of the right descendants of 2, i.e., the smallest value in the subtree rooted by the right child of 2 (i.e., 3).
- Of course this may orphan the right child of this value (3 still has 4 as a right child), but because it is the smallest value of its subtree, it cannot have a left child.
- Therefore, this right child can simply move up to replace the place of this smallest value (i.e., 4 replaces 3).

The removal of 2 is illustrated in Figure 13. Note that for removal, it is useful to keep a `parent` pointer in each node, as will as pointers to the left and right child nodes.

When considering the complexity of *search*, *insertion* and *removal* from BSTs, we see that we always have to take a path down the tree, and possibly do some (constant time, i.e., $O(1)$) shuffling of pointers. Therefore, the dominating factor in the complexity of these algorithms is the number of comparisons we need to do, i.e., the length of the path we need to take down the tree. This is limited by the *depth* of the tree: i.e., the longest possible path from the root node down to a leaf plus one. For example, in Figure 13 (right)

the longest path from the root (7) to a leaf (1, 4, 6 or 9) is of length 4, i.e., $7-3-5-4$ or $7-3-5-6$.

Because the complexity of the operations insert, remove, and search in a BST, $\mathcal{T}$, are all $0\,(\texttt{depth}(\mathcal{T}))$, we need to be careful about controlling this depth. For example, from the values in the tree in Figure 13 (right), i.e., $1, 3, 4, 5, 6, 7, 9$, we could also build a BST with depth 3 rather than for (can you see how?). We can also build an extremely unbalanced tree by inserting all the nodes in ascending order; leading to 1 at the root, 3 as its right child, 4 as its right child, and so on. The depth of the tree thus depends on (the quite possibly rather random) order in which the values were inserted into the BST. On average, this will often more-or-less balance out the BST in practice. However, of course there are no guarantees. If we do want guarantees, we are going to have to algorithmically ensure balance during insertion and removal.

8.4 Well-balanced Trees

If we want a runtime guarantee in terms of the number of items in a BST for search, insertion and removal, we need to ensure balance in the tree. As the time complexity of these methods for BSTs is $0\,(\texttt{depth}(\mathcal{T}))$, and the depth of a well-balanced tree, $\mathcal{T}$, is $\texttt{depth}(\mathcal{T}) = \log n$,[17] we can obtain a time complexity of $O(\log n)$ for all these three key methods. Provided of course that we can do the rebalancing operations in either $O(1)$ or $O(\log n)$ time.

First, let us define what it means to be balanced:

- For all nodes in the tree, the difference in depth of the subtrees rooted by its left and its right child, is at most 1.

This means that we have to count the depth of all subtrees rooted by any node in the tree. The depth of a subtree $\mathcal{T}_{\mathcal{N}}$ rooted by a node, $\mathcal{N}$ is:

$$\texttt{depth}(\mathcal{T}_{\mathcal{N}}) = 1 + \max\{\texttt{depth}(\mathcal{T}_{\texttt{leftChild}(\mathcal{N})}), \texttt{depth}(\mathcal{T}_{\texttt{rightChild}(\mathcal{N})})\},$$

where $\texttt{leftChild}(\mathcal{N})$ is the left child node of $\mathcal{N}$, and $\texttt{righttChild}(\mathcal{N})$ is the left child node of $\mathcal{N}$. When $\mathcal{N}$ has no child nodes (i.e., is a leaf), then its depth is 1.

A tree can become inbalanced both by insertion (i.e., its deepest branch becomes 1 deeper) and by removal (i.e., its shallowest branch becomes one shorter). This results in one of two situations, depicted on the left of Figures 14 and 15.

There are two operations necessary for re-balancing a BST when it becomes unbalanced due to an insertion or removal. These were invented by Adelson-Velsky and Landis in 1962 [1]. As such, BSTs that implement automatic rebalancing are now known in the literature as AVL trees.

The first operation, i.e., the *single rotation*, is required when the deepest part of the tree, is on the same side of the child node which causes the imbalence. For example, in Figure 14 (left), node n_1 is unbalanced, as its left subtree (rooted

[17] Where n is the number of nodes/values in the BST.

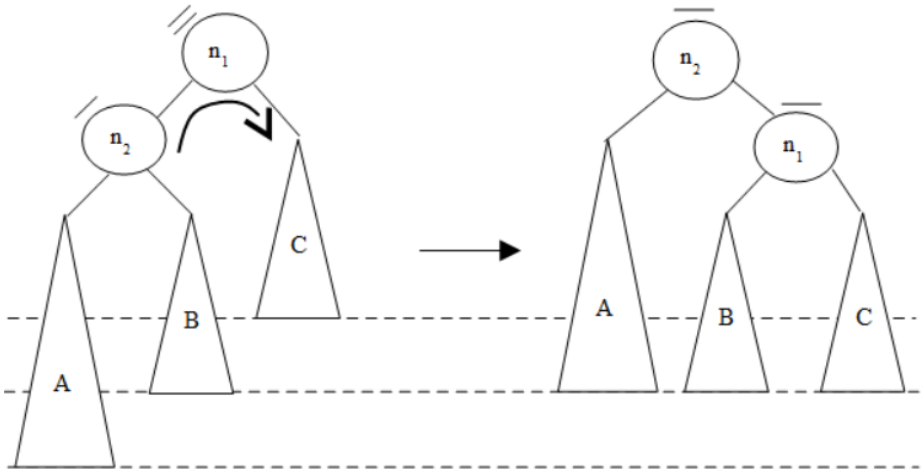

Figure 14: An unbalance in an AVL tree requiring a single rotation. A, B, and C represent subtrees of multiple nodes (not explicitly shown). Figure by Joop Kaldeway.

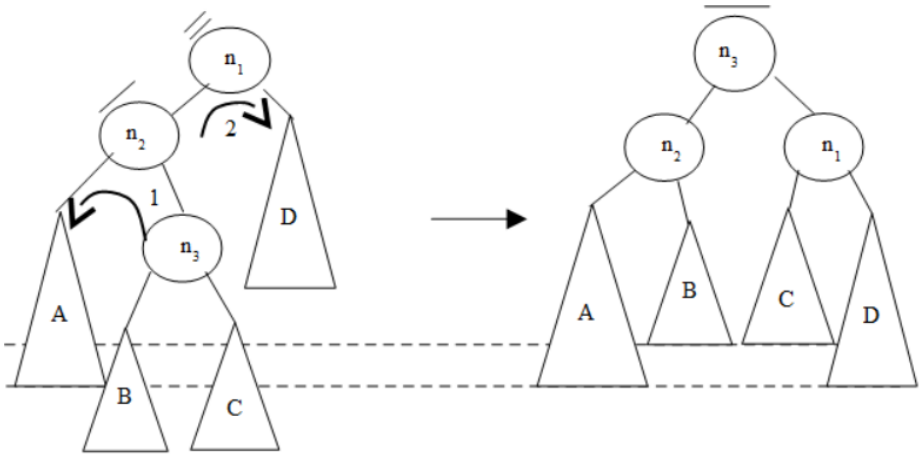

Figure 15: An unbalance in an AVL tree requiring a double rotation. A, B, C, and D represent subtrees of multiple nodes (not explicitly shown). Figure by Joop Kaldeway.

by n_2) is 2 deeper than its right subtree C. Furthermore, the deepest subtree under n_2 is also to the left. In this case the tree can be re-balanced by making n_2 the root, with its left subtree, A, to the left, and n_1 to the right, which now receives B as its left subtree instead of n_2. The results of this *single rotation* is shown in Figure 14 (right).

The situation becomes more difficult when the imbalance is not on the same side as the child node causing the imbalance. For example, in Figure 15, the tree is imbalanced because the left subtree of n_1, rooted by n_2 is deeper than n_1's right subtree (D). However, the deeper subtree under n_2 is now to the right of n_2, i.e., the subtree rooted by n_3. In this case, the tree can be rebalanced by first rotating n_3 to the left (and up), and then again to the right (and up again), resulting in Figure 15 (right). Of course, it is not necessary to implement this as two rotations, one can simply do all the administration of the pointers necessary in one go.

We observe that both the single and the double rotation require a reshuffling of a constant number of pointers. Therefore, the time complexity of these operations is $O(1)$. This may thus be a bit of extra work while inserting or removing, but will not negatively impact the time complexity of either operation. Therefore, in AVL trees the time complexity of search, insertion and removal, are all $O(\log n)$, where n is the number of nodes/values in the tree.

As an exercise, try writing out the pseudocode for insertion, removal, and search in AVL trees. You can show this to your klassendocent to check your answers.[18]

9 Randomised Algorithms

Now let us take a step back from data structures, and move back to algorithmics. As you may have noticed, the previous sections contain a lot of relatively elegant algorithms that have good expected runtimes, but bad worse-case runtimes.

9.1 Randomisation as a stochastic fix

Such worst-cases often depend on a very specific structure in the input. For example, inserting a list of already sorted elements into a BST. Such worst cases tend to not occur out of the blue. For example, humans are a typical source of structure; e.g., they might sort things while inputting the data in the system.

An obvious and often-used strategy to make worst-case input unlikely is to randomise. For example, in the case of adding lists of elements into a BST, we might add these elements in a random order. This would make the worst-case occurring naturally increasingly unlikely (but of course not impossible). This type of randomisation is often applied as a quick fix for worst-case occurrence in production code, and when you encounter it in practice, worst-cases occurring a lot during operation is probably the reason for it.

[18] Pro tip: we might ask something similar on the exam.

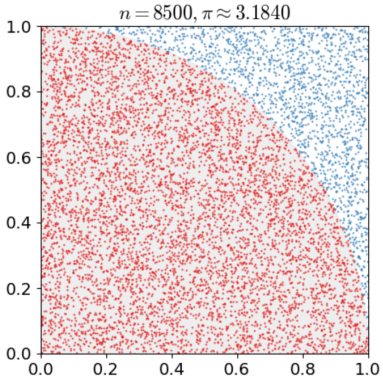

Figure 16: A randomised approximator for ($\frac{1}{4}$ of) π, by drawing random positions between $(0, 0)$ and $(1, 1)$. With probability $\frac{\pi}{4}$ these points (red) will be inside of the quarter circle, and with probability $1 - \frac{\pi}{4}$ outside of it (blue). Figure created by nicoguaro, and taken from `https://commons.wikimedia.org/wiki/File:Pi_30K.gif`.

9.2 Approximation

Another very useful trick using random numbers is approximation for unknown quantities. For example, we can approximate π by drawing tuples of two random numbers, x and y between 0 and 1. These two numbers would correspond to a position on a square with a height and width of 1 (see Figure 16). The probability that $\sqrt{x^2 + y^2} \leq 1$, i.e., that the point is inside the quarter circle shown in the figure (with radius 1), should be $\frac{\pi}{4}$. To estimate $\frac{\pi}{4}$, one therefore has to simply count the number of random positions drawn that are inside this quarter circle N_{in}, and divide this by the total of positions randomly drawn $(N_{in} + N_{out})$.

The above procedure, which uses random number generation to estimate something that is actually deterministic, is an example of Monte-Carlo method.[19] Monte-Carlo methods are used widely to estimate all kinds of quantities of static objects as well as processes and decision problems. For example, they have even been applied in theoretical physics [4]. In this reader, we will be looking at its application – together with trees (Section 11) – to optimise policies/strategies for automated game-playing in Section 12.

[19] Yes, indeed, this is named after the Monte Carlo casino in Monaco.

10 Intermezzo: Automated Game Play → Optimisation

We have seen multiple classic algorithms and data structures that have been applied in almost every piece of software (and hardware for that matter) since the beginning of computer science, i.e., inserting, sorting, removing and searching, in different kinds of container data structures. These are essential bits of knowledge that will serve you well in designing many an application for a variety of platforms, as long as you remember to always think which operations are key to the performance of the system, and reduce the time complexity of those operations accordingly.

For the classic problem types above, we do not expect much more improvements to follow in the future above the old faithful algorithms already described. What we want to do in the rest of this course is give you an insight into modern algorithmics, on problems where research is still ongoing. A – par chance – highly suitable domain to study these algorithms is of course *gaming*. This is because games are things we as humans have invented for ourselves to challenge our own intellect (among other things of course). As such these games provide highly challenging domains for automated play.

Before we go into *automated game play* however, it should be noted that while automated game play may seem highly specific, it is itself an instance of *sequential decision making*. Sequential decision making is of course a much more general problem, that applies to planning under uncertainty for train parking and maintenance [5] or mapping a program onto a (dedicated) chip [6]. Furthermore, sequential decision making itself is an instance of *optimisation*. Optimisation is a highly general problem which says: we have to input a number of variable values, i.e., an array (or vector) of inputs, $\mathbf{x}$, and given the optimisation problem, there will be some evaluation of this to a reward, $f(\mathbf{x})$. An optimisation task is then simply $\max_{\mathbf{x}} f(\mathbf{x})$.[20] That is, find such inputs that the reward is maximised. This can of course be very complex: in automated game play, $\mathbf{x}$ is an entire specification of how we want our program to play the game, and $f(\mathbf{x})$ can be our expected probability of winning the game.

The algorithms we will present in this can the next section are general sequential decision making algorithms, but can (with a bit of adaptation) also be used in wider optimisation tasks.

11 Tree Search Algorithms

Please note that this section is written specific to games. Of course, a general sequential decision making description would not be much different.[21] *A general*

[20] Or, if the rewards depend on a stochastic process (i.e., have a degree of randomness), it should be written as the expected reward given the input $\mathbf{x}$, i.e., $\max_{\mathbf{x}} \mathbb{E}(f(\mathbf{x}))$.

[21] The term *game state* become *(environment) state*, *players* would become *agents* and *plies/moves* would become *actions* or *decisions*.

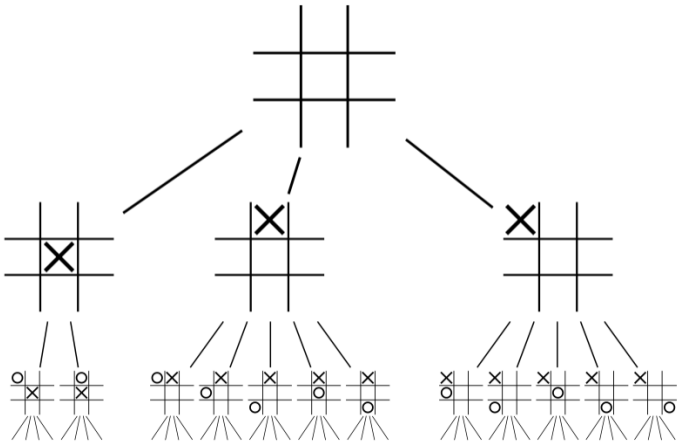

Figure 17: Game tree for tic-tac-toe. Please note that only unique positions after symmetry-based reduction have been inserted into this tree. Figure taken from `https://commons.wikimedia.org/wiki/File:Tic-tac-toe-game-tree.svg`, own work by en:User:Gdr.

optimisation description would be a bit different. Please see Section 17.2 for the description of a similar algorithm written from an optimisation perspective.

What does it mean to play a game? Basically, a *player* gets a state of the game, i.e., a *game state* as input, and then makes a *move*. In other words, the player needs a *policy*, i.e., a specification of what move to make in any valid game state that the player might encounter. Automated game play therefore means optimising such a policy as to maximise the probability of winning.[22] There may also be another player, who might be fully adversarial to your player. For example, if you play naughts in tic-tac-toe (boter-kaas-en-eieren), the crosses player will try to minimise your probability of winning (and thus win himself instead). In such games, a move by a single player is also called a *ply*.

A natural way to model a policy in a game is to build a game tree (Figure 17). The root of this tree represents the begin (or current) state. From this game state, other game states can be reached by doing moves. If the game is deterministic (i.e., the next state is always the same after playing a given move in a given game state and does not depend on any randomness), the number of children of a node in the game tree is thus equal to the number of valid moves the player can do.

The best move is the move that maximises the score (i.e., do we win?). Of course, this can only be determined by going down the tree: I maximise my score by taking the best move for the current position, then my opponent will take the worst possible move for me, then I maximise again, and so on until the

[22] Or to maximise the score, or some other metric of successful game play. This is a design choice.

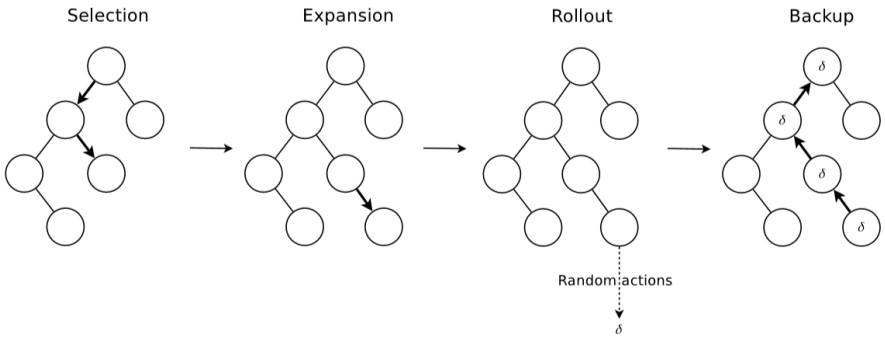

Figure 18: The basic steps in one iteration of Monte-Carlo Tree Search. Image taken from De Waard, Roijers and Bakkes (2016) [2].

end of the game. While building the game tree, it is most efficient to start at the leafs, and then *backup* the value of these leafs to their parents. That is, the parent node will take the maximising child as its own value (or minimising if it is the opponent's turn).

On possible issue in building a full game-tree is the size of this tree. For example, if we take the game of tic-tac-toe, there are 9 valid moves on an empty field, leading to 9 different game-states. Then, the subsequent player can do 8 different moves, and so on. This would mean that the full game-tree would consist of $9! = 362880$ nodes. This is of course still very doable for a computer. Note however that in more complex games, e.g., *go*, which is also turn-based but has 19×19 fields, this will most likely no longer be doable. In this case, limited gain be achieved by exploiting the fact that in many games (and other decision problems), some states are actually the same after mirroring or rotation, and this should not matter for their value. For example, in tic-tac-toe, when thinking about it in a structured manner, the three options for the first move are actually: the middle, one of the sides, or a corner, and it does not actually which. This leads to a reduced game tree of which the first two layers are displayed in Figure 17. For most games however, we need to accept that we will never be able to systematically search the entire game tree.

12 Monte-Carlo Tree Search

Many games have too many (reachable) game states that we cannot possibly generate an entire game tree to find out what the best move to play currently is. What we can do instead of exploring this entire game tree, is exploring different parts of the game tree in a well-reasoned, but fundamentally heuristic manner. This has led to one of the most well-known algorithms in game playing: *Monte-Carlo Tree Search (MCTS)*.

Instead of building the entire game tree, MCTS builds a partial tree (left).

Of course this means that we cannot know exactly what the value (i.e., whether we win or lose) for each game state is, as we are no longer able to maximise over our moves and minimise over the opponent's moves until the game ends. Therefore, we need some kind of stand-in. For this, randomisation (Section 9) helps.

In this partial tree, the leaves correspond to game states that may not be terminal (win or lose). To estimate the value of such game states, MCTS performs a lot of random games (Monte-Carlo roll-outs) using a random (or better heuristic) policy for both players from that game state onward. Playing such a random game until termination (i.e., one of the players wins, or there is a draw), is called a Monte-Carlo *roll-out*.

Now that we have a way to evaluate leaf nodes (game states) that do not yet correspond to finished games, MCTS can build its partial tree iteratively. That is, it starts at the current (or begin) game state, and add child nodes (representing subsequent game states after a single move), as in standard full tree search. However, when we add a child node, we now estimate its value by a lot of random plays. Adding a new child node is called *expansion* of the game tree.

After *expanding* and evaluating the leaf node through *roll-outs*, the new leaf node gets a value. This value needs to be processed back up the tree. This is called *backup*. After the values have been back-upped, we can select a new place to extend the game tree with a new node. For such *selection*, we need to balance exploration, i.e., looking at previously un(der)explored possible moves, and exploitation, i.e., further investigating moves that we already highly suspect that they are good. Together these steps form the steps during a single iteration of the main loop of MCTS: 1) select a place to expand the tree, 2) create the new node, 3) perform Monte-Carlo rollouts to determine the value of the node, and 4) back up this value to the ancestors of the new node (i.e., its parent, its parent's parent, and so on).

Dear student,

You have reached the end of the current version of the reader. Well done! We will append this reader as the course progresses. Stay tuned!

Many thanks,
your friendly neighbourhood course designers,
Jorn and Diederik.

13 Graphs

14 Shortest Path

14.1 Dijkstra's algorithm

14.2 Dynamic programming for finding shortest paths

15 Dynamic Programming (in general)

16 Complexity Classes and Really Hard Problems

17 Coordination graphs

17.1 Variable Elimination

17.2 AND/OR Tree Search

18 An Introduction to Heuristic Search Algorithms

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

[2] M. de Waard, D. M. Roijers, and S. C. Bakkes. Monte Carlo tree search with options for general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[3] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014.

[4] M. Newman and G. Barkema. *Monte Carlo methods in statistical physics*. Oxford University Press: New York, USA, 1999.

[5] E. Peer, V. Menkovski, Y. Zhang, and W.-J. Lee. Shunting trains with deep reinforcement learning. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3063–3068. IEEE, 2018.

[6] R. Piscitelli et al. *Pruning techniques for multi-objective system-level design space exploration*. PhD thesis, Universiteit van Amsterdam, 2014.

A On the Usage of Pseudocode

B Exercises

B.1 week 1

B.1.1 Max

In Section 2 Algorithm 1 describes a max function. Implement this function in Python.

B.1.2 getNumbers

Write the function *getNumbers(s)*. The function should returns a list with all numbers that were in the input string, *s*.
Example: given *text = 'een123zin45 6met-632meerdere+7777getallen'* then *getNumbers(text)* returns : [123,45,6,632,7777]

B.1.3 Between

Consider Algorithm 20. Describe (in either Dutch or English) in one or two sentences what this algorithm does.

Algorithm 20 pseudoLoops

```
Input: an array of integers: a,
           an integer: max
  Output: an array of integers
1: result ← an empty array of integers with length a.length
2: for i ∈ 0 ... a.length/2 do
3:     j ← a.length-i-1
4:     sum ← 0
5:     for k ∈ i ... j do
6:         sum ← sum + a[k]
7:     end for
8:     if sum > max then
9:         sum ← max
10:    end if
11:    result.append(sum)
12: end for
13: return result
```

B.1.4 Order!

Assume that each of the expressions below gives the processing time $T(n)$ spent by an algorithm for solving a problem of size n. Select the dominant term(s) having the steepest increase in n and specify the lowest Big-Oh complexity of each algorithm. The second row is an example.

Expression	Dominant term(s)	O(...)
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n \cdot log(n)$		
$0.3n + 5n^{1.5} + 2.5 \cdot n1.75$		
$n^2log(n) + n(log(n))^2$		
$nlog(n) + nlog(n)$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		

B.1.5 Double looping

Give the time complexity of Algorithms 21, 22, 23, 24, 25 and 26. Watch out: Algorithm 25 has a different result than the other algorithms.

Algorithm 21 addSum1

Input: an array of integers: a
Output: an array of integers

```
1: sum ← 0
2: result ← an empty array of integers
3: for i ∈ 0...a.length−1 do
4:     sum ← sum + a.get(i)
5: end for
6: for i ∈ 0...a.length−1 do
7:     tmp ← a.get(i) + sum
8:     result.append(tmp)
9: end for
10: return result
```

Algorithm 22 addSum2

Input: an array of integers: a
Output: an array of integers

```
1: result ← an empty array of integers
2: for i ∈ 0...a.length−1 do
3:     sum ← 0
4:     for j ∈ 0...a.length−1 do
5:         sum ← sum + a.get(j)
6:     end for
7:     tmp ← a.get(i) + sum
8:     result.append(tmp)
9: end for
10: return result
```

Algorithm 23 addSum3

Input: an array of integers: a
Output: an array of integers

```
1: result ← an empty array of integers
2: for i ∈ 0 . . . a.length−1 do
3:     tmp ← a.get(i)
4:     for j ∈ 0 . . . i−1 do
5:         tmp ← tmp + a.get(j)
6:     end for
7:     for k ∈ i . . . a.length−1 do
8:         tmp ← tmp + a.get(k)
9:     end for
10:    result.append(tmp)
11: end for
12: return result
```

Algorithm 24 addSum4

Input: an array of integers: a
Output: an array of integers

```
1: result ← an empty array of integers
2: for i ∈ 0 . . . a.length−1 do
3:     sum ← 0
4:     for j ∈ 0 . . . a.length−1 do
5:         sum ← sum + a.get(j)
6:         for k ∈ 0 . . . a.length−1 do
7:             rand ← getRandomInt()
8:         end for
9:     end for
10:    tmp ← a.get(i) + sum
11:    result.append(tmp)
12: end for
13: return result
```

Algorithm 25 addSum5

Input: an array of integers: a
Output: an array of integers

```
1: result ← an empty array of integers
2: if a.length ≤ 0 then
3:     return result
4: end if
5: sum ← 0
6: for i ∈ 0 . . . a.length−1 do
7:     sum ← sum + a.get(i)
8: end for
9: tmp ← a.get(0) + sum
10: result.append(tmp)
11: result.extend(addSum5(a[1:]))
12: return result
```

Algorithm 26 addSum6

Input: an array of integers: a
Output: an array of integers

```
1: result ← an empty array of integers with size a.length
2: sum ← sum(a)
3: for i ∈ 0 . . . a.length−1 do
4:     tmp ← a.get(i) + sum
5:     result.append(tmp)
6: end for
7: return result
```

B.1.6 Hairy Exercise

In Section 2 Algorithm 5 describes a function to make an array with all possible combinations of three students and the difference between their average hair length and the average of the hair length of all other students.

1. Translate the pseudo-code of Algorithm 5 to Python-code. Assume the array of students contain for each student their name and their hair length.

2. What is the time complexity of the algorithm?

Nested loops [23] increase the time complexity. Removing a nested loop or placing it outside the loop will increase the performance (decrease the time complexity) of the program.

4. The time complexity of Algorithm 5 can be reduced by calculating the average hair length outside of the three loops and using it inside the loop.

[23] Loops inside (an)other loop(s)

Write the pseudo-code of an algorithm with the same functionality, but with a lower time complexity.

5. Translate your pseudo-code to Python code.

6. Make use of the Python library *timeit*[24] to measure the time difference between the two implementations. Give the measurements and answer the following questions:

   - Why should you measure multiple times?
   - Why do we use the Big-Oh notation instead of just timing algorithms?

B.1.7 Find x

Write the recursive function *find(lst, x)* that returns the index of x in lst. If lst does not contain x, then the function returns -1.

B.1.8 Euclid's algorithm

Write a function that calculates the Greatest Common Divisor (GCD) of the two integers p and q using Euclid's algorithm and recursion. The GCD of two integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the GCD of 8 and 12 is 4. The GCD can be calculated quite easily using the following characteristics:

- if $p > q$, then the GCD of p and q is equal to the GCD of q and $p\%q$
- if $p == q$, then the GCD is equal to p

B.1.9 Prime numbers

A prime number is a number that is greater than 1 that cannot be formed by multiplying two smaller numbers. Examples of prime numbers are: 2, 3, 5, 7, 11.

1. Describe in two sentences (or give the pseudo-code / python-code) how you now, without any research, would write a function that returns all prime numbers under 1000.

2. What is the time complexity of your solution?

3. Write a function that returns all prime numbers under 1000. Use the Sieve of Eratosthenes[25].

4. What is the time complexity of the Sieve of Eratosthenes? Figure it out on your own (hard but great practice!) or use your favorite search engine.

[24] `https://docs.python.org/3/library/timeit.html`
[25] see for more information: `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

B.1.10 Money

Write a program that calculates for a given amount of money in how many different distinct combinations of coins and bank notes it can be paid. Make use of memoisation and recursion in your program. Work in your program with amounts in eurocents. The following coins and bank notes are available (in eurocents): 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 and 10000.
Example: When the amount is 7 the following six combinations are possible:

- $7 = 1 + 1 + 1 + 1 + 1 + 1 + 1$
- $7 = 1 + 1 + 1 + 1 + 1 + 2$
- $7 = 1 + 1 + 1 + 2 + 2$
- $7 = 1 + 2 + 2 + 2$
- $7 = 1 + 1 + 5$
- $7 = 2 + 5$

The following strategy can be used. Consider m as the list with all possible coins and banknotes and matrix A, where $A[i, j]$ is the number of combinations you can pay amount j with the coins and banknotes $m[0]$ up to and including $m[i]$. Then the following rules can be applied:

- $A[i, 0] = 1$ for all i
- $A[0, j] = 1$ for all j
- if $j \geq m[i]$ then $A[i, j] = A[i - 1, j] + A[i, j - m[i]]$
- if $j < m[i]$ then $A[i, j] = A[i - 1, j]$

Make sure that your program only calculates the values that it needs. In other words, the program does not fill the complete matrix A, but leaves blank spaces for amounts of money that it does not require.

m		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	**0**	1	1	1	1	1	1	1	1
2	**1**	1	1	2	2	3	3	4	4
5	**2**	1	1	2	2	3	4	5	6

Figure 19: Matrix A. Note that for calculating the example A[2,7] you do not need to calculate all values listed here.

B.2 Week 2

B.2.1 myStack

Write the class *myStack*, using the standard Python list datatype as a base data structure. The class has the following methods:

- *push()* (add an item to the "top"[26] of the list).
- *pop()* (return and remove the item on top of the list).
- *peek()* (return the item on top of the list without removing it).
- *isEmpty()* (returns whether myStack contains any items).

B.2.2 Brackets-problem

We have a language build out of the brackets: <, >, [,], (and). The rules of this language are as follows:

- Every opening bracket has a corresponding closing bracket.
- The opening bracket is always before its corresponding closing bracket.
- Between a opening bracket and its corresponding closing bracket, their can be sets of opening and corresponding closing brackters, but no "loose" opening or closing brackets.

Examples of correct expressions:

- ((<>)())
- (<>)[]( )(( )( ))
- ((<>))

Examples of incorrect expressions:

- ([)]
- (((<)>))
- [(])
- (()

Write a program that for a given string evaluates if it is a valid expression. Make use of your stack created in exercise B.2.1. Place every opening bracket on the stack. Removing the opening bracket on top of the stack when a corresponding closing bracket is read. If the program end with an empty stack, then the expression is valid. If a read closing bracket is not corresponding to the opening bracket on the top of the stack or if the program ends with still items on the stack, then the expression is invalid.

[26] You may choose which side of the list is the top.

B.2.3 Birthday problem

Write a simulation to approximate the chance that two students in one class have their birthday on the same day[27]. Do this as follows:

- Generate a list random numbers between 1 and 356, one for each person in the class.
- Repeat this 100 times.
- Return the estimated probability of two students have the same birthday, i.e., the number of times the list contained multiple times the same number divided by 100.

B.2.4 Quick sort

The running time of Quick sort is dependent on the value of the pivot. In this exercises we take a look at what happens when we are unlucky, lucky or let put our faith in the hands of lady Luck.

1. Write a function for sorting a given list using quick sort. (See Algorithm 16)
2. Modify your function such that it also returns the number of times two elements are compared with each other.
3. Write a second function for sorting a given list using quick sort, but change line 1 of algorithm 17 into: *pivot* $\leftarrow$ *min(a)*. Make also sure that the function returns the number of times two elements are compared (do not count the comparisons in the *min()* function).
4. Write a simulation (run your functions a lot of times on different lists) to make an approximation of their time complexity.
5. Run your simulation. Are the approximations close to the values you expected? Explain why.
6. Write a second function for sorting a given list using quick sort, but change line 1 of Algorithm 17 so that the pivot has the value of the median[28] of the list. Make also sure that the function returns the number of times two elements are compared (do not count the comparisons in the algorithm for finding the median).
7. Also run a simulation on the quick sort function with the median pivot. Is the approximation close to what you expected? Explain why.

[27] see for more information on the birthday paradox: `https://en.wikipedia.org/wiki/Birthday_problem`

[28] The median is the value separating the higher half from the lower half of a list. It may be thought of as the "middle" value. For example, in list [1, 7, 3, 8, 3, 9, 6] the median is 6. In python 3 you can use the *median()* function of the *statistics* module.

B.2.5 Linked list

In this exercise you are going to write an linked list data structure. For more information see Section 8. Write the class linked list. The class has the member variables:

- *value*; which contains a value of the list
- *tail*; which points to the next linked list. If their is no *tail* then *tail* is equal to *None*.

The class contains the following functions:

- *append(value)*; which adds a value to the end of the linked list.
- *delete(value)*; which removes all elements with that value from the list.
- *min()*; which finds the minimum value of the linked list.

All of the functions need to be implemented with "recursion". For example the *min()* function calls the *min()* function of its tail. Make sure you test your functions thoroughly.

B.2.6 Priority queue

A priority queue is an abstract data type which is like a regular queue, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. A priority queue can be implemented using different data structures[29], each with its own pros and cons. In this exercise you are going to write your own priority queue and compare its performance with a different implementation.

1. Research which data structures you can use for building a priority queue (use resources outside the reader).
2. Choose a data structure and explain why you chose this data structure.
3. Find a practicum-partner that is going to use a different data structure.
4. Build a priority queue using your chosen data structure. Make sure the priority queue has the following functions:
    - *queue(v, p)*; adds value v with priority p to the queue.
    - *dequeue()*; returns the value with the highest priority and removes the element from the queue.
    - *contains(v)*l returns *True* if and only if the queue contains value v.
    - *remove(e)*; removes all elements with value v from the queue.

[29] it is not limited to a "remix" of the basic queue data structure

5. Give the time complexity of the functions of your priority queue implementation.

6. Compare the runtimes between your queue implementation and that of your practicum-partner. Describe and explain the differences in performance.

B.3 Week 3

B.3.1 8-queen-problem

The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. This means that if their is a queen on (x_0, y_0) and their is a queen on (x_1, y_1) the following holds true:

- $x_0 \neq x_1$; not on the same row.
- $y_0 \neq y_1$; not on the same column.
- $y_0 - x_0 \neq y_1 - x$; not on the same diagonal.
- $y_0 + x_0 \neq y_1 + x_1$; not on the same diagonal.

1. Although a chessboard can be represented in python by an 8 by 8 list of lists. When searching for a solution of the 8-queen-problem it is suffice to represent the board with a list with a size of 8. Why and how is the chessboard packed inside this list[30]?

2. Write a function that given a chessboard returns whether two or more queens threaten each other.

3. Write a recursive function that searches for and returns a solution to the 8-queen-problem. The function has as parameter the list *chessboard*.

B.3.2 Binairy Search Tree

Implement a binary search tree (BST) (see section 8.2). Structurally, a BST looks a lot like a linked list, but instead of having a pointer to one other linked list it has two pointers to another BST called *childs*. Make sure that the BST has the following recursive functions:

- *Add(v)*; add value v to the BST.

[30] **[Spoiler Alert!]** It is good to think about these kind of decisions, but to make sure you don't get stuck on this question we will give you the answer: the list contains the column positions of the queens. The index of the list is the row on the chessboard. This way we don't have to check if two queens are on the same row, because that is not possible. For example: [5, None, None, 2, None, None, None, None], means that there are queens on (1,5) and (4,2). Furthermore, Dumbledore dies, Neo is the one and the answer is 42. **[/Spoiler Alert!]**

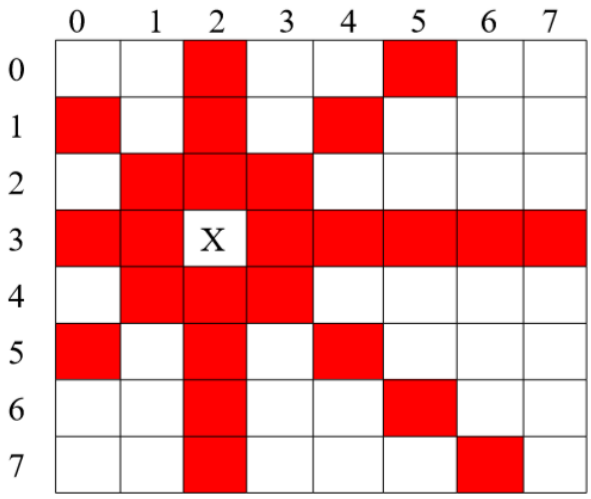

Figure 20: The reach of a queen on position (2,3).

- *Remove(v)*; remove all values v from the BST.
- *Max()*; return the max value in the BST.
- *Contains()*; returns true if and only if the BST contains value v.

Make sure you test your functions thoroughly.

B.4 Extra exercises

B.4.1 Look-and-say sequence

The look-and-say sequences[31] is a sequence of natural numbers that are not based on a mathematical expression, but on a linguistic description. To generate a member of the sequence from the previous member, read off the digits of the previous member, counting the number of digits in groups of the same digit. For example:
1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ...

- 1 is read off as "one 1" or 11.
- 11 is read off as "two 1s" or 21.
- 21 is read off as "one 2, then one 1" or 1211.
- 1211 is read off as "one 1, one 2, then two 1s" or 111221.
- 111221 is read off as "three 1s, two 2s, then one 1" or 312211.

[31] see for more information: `https://en.wikipedia.org/wiki/Look-and-say_sequence`.

A look-and-say sequence can also start with a different number than 1. Write an function that given the previous member will generate the next member of the sequence[32].

B.4.2 Dictionary vs. Tree

Write a program that counts all words in a text file. Do this with two different methods.
Method 1: **dictionary**

1. Write a function that counts the words from a textfile. The frequency table is returned as a (python) dictionary with the words as key and the frequencies as value.
2. Write a function that write the frequency table to a file.

Method 2: **Tree**

1. Write the class TreeNode with the following attributes:
   - n : the count
   - d : a dictionary with:
     - keys : next character
     - values: TrieNodes corresponding with the next character.

   See for an example the prefix tree in figure 11.
2. Write a function that counts the words from a textfile. The frequency table is returned as a Tree.
3. Write a function that write the Tree frequency table to a file.

B.4.3 HashMap

Create your own HashMap. The class contains at least the following functionalities: search, insert, delete and rehash. The filling rate (= number of elements in Hashmap / HashMap.size) of the HashMap should not be higher that 75%. When this is exceeded the HashMap needs to be made, with the use of rehash, twice as large. Use *hash(value) % HashMap.size* as the hash function. Make a informed decision which collision strategy you use. Test your code.

[32] Hint: you can get the last digit of a integer with: *ldigit = num % 10*