in the domain of artificial intelligence. For a technically light-weight but sprawling pop-science introduction to this topic, see Hofstader (1979).

Linguistics beyond language Section 2.1 lists several examples where approaches from theoretical and/or computational linguistics made their ways into seemingly unrelated applications. All these approaches will be discussed more extensively in later chapters, but the curious reader may already take a gander at the following references: Chomsky (1957, 1959) and Chomsky & Schützenberger (1963) are seminal papers from the very beginning of what is now called formal language theory. One of the first papers on tree transducers is Rounds (1970), who explicitly names Chomsky's Transformational grammar as a major motivation for his work. Tree Adjoining Grammar (Joshi 1985) is a different, computationally minded grammar formalism. For applications of TAG in molecular biology see Uemura et al. (1999) and Matsui, Sato & Sakakibara (2005).

Other references The study on predicting the success of novels is Ashok, Feng & Choi (2013). The tri-level hypothesis is formulated in Marr & Poggio (1976) and Marr (1982). For recent findings on how how specific brain areas may be co-opted for different task, see Bola et al. (2017) and references therein. Jonas & Kording (2017) apply current techniques in neuroscience to the analysis of microchips to show that these methods provide insufficient insight into computation.

P Programming

P.1 Linear search and binary search

In Sec. 2.2, example 0.2, we saw that looking at computation in terms of hardware fails to illuminate even basic points such as the difference between linear search and binary search. This point becomes even more pronounced if we look at how these search algorithms can be implemented in a specific programming language.

Linear search is one of the simplest search algorithms and can be implemented in 4 lines. The code here implements it as a function that returns either `False` or an integer. `False` is returned iff the search item is not in the list. When an integer is returned, it indicates the leftmost position of the search item.

The docstring under the function name specifies the intended type of each argument. The type of the search item is set to `any`, which means that there are no restrictions on its type.

```
def linear_search(search_list, item):
    """Left-to-right search for position of item in search_list.

    Parameters
    ----------
    search_list : list
        list to be searched
    item : any
        item we are looking for

    Returns
    -------
```

```
    int or False

    Examples
    --------
    >>> linear_search([0,1,7,9], 7)
    2

    >>> linear_search([0,1,7,9], 8)
    False
    """
    # iterate over all positions in the list
    for i in range(len(search_list)):
        if item == search_list[i]:
            return i
    # if we made it this far,
    # then we haven't found anything
    return False
```

Here is a concrete example:

```
    >>> test_list = ['a', 'c', 'd', 'e', 'b', 'f']
    >>> linear_search(test_list, 'e')
    3
    >>> linear_search(test_list, 'g')
    False
```

Now let's contrast linear search against binary search. As you can see, the code is slightly different from the intuitive description in example 0.2. There we said that binary search picks the middle item of the list. If the middle it is not the search item, one discards one half of the list and runs binary search on the remainder. This is a recursive definition: an instance of binary search can spawn another instance of binary search. Recursive definitions are elegant and mathematically pleasing, but they are not always the best way of implementation.

Python in particular is not well-optimized for recursion and probably will never be because it would require some fundamental design changes with major sacrifices. Fortunately, binary search does not require recursion, that is just one of many different ways of describing the procedure. The implementation below uses a `while`-loop instead, which avoids Python's issues with recursion but looks very different from the intuitive description of binary search. With another programming language, e.g. Haskell, the latter might be the more natural and efficient implementation.

```
def binary_search(search_list, item):
    """Use binary search to find position of item in search_list.

    This algorithm is more efficient than linear search,
    but only works for sorted lists!

    Parameters
    ----------
    search_list : list
        list to be searched
```

```
    item : any
        item we are looking for

    Returns
    -------
    int or False

    Examples
    --------
    >>> binary_search([0,1,7,9], 7)
    2

    >>> binary_search([0,1,7,9], 8)
    False
    """
    start = 0
    end = len(search_list) - 1

    while start <= end:
        # pick middle element of list;
        # we use int() for rounding down
        middle = int(start + (end - start)/2)

        # Case 1: our item is to the left of the item at the midpoint,
        #         limit next search to left half
        if item < search_list[middle]:
            end = middle - 1
        # Case 2: our item is to the right of the item at the midpoint,
        #         limit next search to right half
        elif item > search_list[middle]:
            start = middle + 1
        # Case 3: we found our item, return its index
        elif item == search_list[middle]:
            return middle
    # Case 4: item not in list, while-loop aborted
    return False
```

```
>>> test_list = ['a', 'c', 'd', 'e', 'b', 'f']
>>> binary_search(sorted(test_list), 'e')
4
>>> binary_search(sorted(test_list), 'g')
False
>>> binary_search(test_list, 'c')
1
>>> binary_search(test_list, 'e')
False
```

This shows very clearly that one and the same idea can be implemented in many different ways. Which way is best often depends on very subtle and arcane details that have very little to do with the problem itself. In the case at hand, an idiosyncrasy of Python makes a `while`-loop preferable to recursion, but that has no bearing on binary search in general and how it compares to linear search. Such implementation details tend to obscure what really matters, and this is why we should always strive for a level of description that abstracts away from implementation-dependent differences.

P.2 Set intersection

Example 0.4 briefly raised the question how exactly set intersection could be implemented at the algorithmic level. Python actually provides an out-of-the-box solution in the form of a set data type that provides the method `intersection`.

```
>>> A = {1, 2, 3, 4}
>>> B = {2, 4}
>>> A.intersection(B)
{2, 4}
```

While useful and very fast, this solution hides all the interesting parts under the hood. Let's first look at a piece of code that only uses the most basic programming concepts and is also very intuitive. Unfortunately, it is also very inefficient.

```
def list_intersection(listA, listB):
    """Build a list that only contains elements of both lists.

    This algorithm is highly ineffecient
    because it loops over the second list multiple times!

    Parameters
    ----------
    listA : list
        first list of elements
    listB : list
        second list of elements

    Returns
    -------
    list

    Examples
    --------
    >>> list_intersection([3, 1, 2], [4, 7, 2, 1])
    [1, 2]

    >>> list_intersection([3, 1, 2], [])
    []

    >>> list_intersection([3, 1, 2], [1, 2, 3])
    [3, 1, 2]
    """
    # create empty intersection
    intersection = []

    # for each item a of listA
    for a in listA:
        # is any b of listB the same as a?
        for b in listB:
            if a == b:
                # found a, add it to intersection
                intersection.append(a)

    # all loops done, return intersection
    return intersection
```

Here we create an empty list and only add an item to it if said item occurs in both lists. Doing this requires two `for`-loops. Notice how we do a full iteration over the second list for each element in the first list. If the first list contains m items and the second one n, we perform a total of $m + (m \times n)$ lookups. So the number of lookups grows much faster than the combined number of elements in the lists.

The obvious problem is that we look at the second list over and over again, rather than just processing it once and memorizing what elements occur in it. In Python, such memorizing takes the form of converting a list to a dictionary (also known as a *hash map* or *hash table* in other programming languages). A dictionary uses keys as an addressing system for quickly looking up items.

```
>>> A = {'some_key': 'the_value',
         'key_2': [5, 3],
         10: 'integers can be keys, too'}
>>> A.get('some_key')
'the_value'
>>> A.get('key_2')
[5, 3]
>>> A.get(10)
'integers can be keys, too'
```

Instead of searching through the second list over and over again, we first convert it to a dictionary and then use that for quick lookup.

```
def list_intersection(listA, listB):
    """Build a list that only contains elements of all lists.

    This algorithm uses dictionaries for speed,
    but requires more memory.
    """
    # convert second list to dictionary
    dictB = {item: item for item in listB}

    # create empty intersection
    intersection = []

    # for each item a of listA
    for a in listA:
        # is a in listB?
        if dictB.get(a):
            # found a, add it to intersection
            intersection.append(a)

    # return intersection
    return intersection
```

With larger lists, this implementation will be much faster. And it is essentially what the very first solution with sets is doing, because Python's `set` data type is just a special case of Python dictionaries. For real-world applications, seemingly minor

differences like this can have major repercussions. But if the lists are always very small, memorizing them with dictionaries might not be worth it. There are no simple answers here, it all depends on the specific use case.

You already saw in the case of linear search versus binary search how implementation details can obscure the bigger picture. But here the issue wasn't even choosing between largely different algorithms, but just how exactly one wants to implement a (very simple) operation. The two pieces of code we came up with are almost exactly the same, yet they differ greatly in their runtime behavior.

E Exercises for Unit 0

E.1 Theory

Exercise 0.1 Consider once more the Turing machine from example 0.1. Show how this Turing machine operates on an input of the form 011110, with the read-write head starting on the rightmost 1 in state A. Since the tape of a Turing machine can be expanded without limits, you may pad it out with 0s as necessary. This does not require any special instruction for the machine.

Exercise 0.2 Specify a Turing machine that satisfies all the following requirements:

1. The machine uses only the following symbols on the tape: 0, 1, and ⋄ as a special symbol for cells that do not contain one of the two digits.
2. The Turing machine takes as its input a single string of 0s and 1s (e.g. 001101 or 1010110).
3. All other cells of the tape are filled with ⋄.
4. The machine starts on the leftmost symbol of the input.
5. The machine outputs a string that is almost exactly the same as the input, except that the first symbol is replaced by $1-n$, where n is the value of the last symbol. For instance, 11101 becomes 01101, whereas 01101 and 11100 stay the same.

Exercise 0.3 Continuing the previous exercise, modify your machine so that it works even if the read-write head can start on any random symbol of the input string.

Exercise 0.4* Specify a Turing machine that satisfies all the following requirements:

1. The machine uses only the following symbols on the tape: digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and ⋄ as a special symbol for cells that do not contain a digit.
2. The Turing machine takes as its input a single number n between (and including) 0 and 999.
3. The machine starts on the leftmost digit of the input.
4. The machine outputs $1000-n$ (leading 0s are allowed, e.g. 0001 instead of 1).

To save yourself some writing, you may conflate multiple lines by using variables. So if 7 is rewritten as 3 and 8 as 2, you can just have a line rewriting n as $10 - n$ and adding "n is 7 or 8". As long as there are only finitely many choices for n, this does not change anything about the machine, it is just a way to represent multiple instructions with a single line.

Exercise 0.5 Continuing the previous exercise, look at the specification of your Turing machine. Is there any individual part that captures the "essence" of substraction? Or does the whole come about only through the interaction of the individual parts?

Exercise 0.6 The practical argument for computation linguistics presupposes that man-made engineering solutions can be improved by learning from nature — in the case of NLP, human language use. But at the same time, airplanes aren't built like bumblebees, and submarines aren't fish. Doesn't this suggest that technology carves out its own way, with its own solutions? Formulate an extended argument for this position, and one against it. Which one do you find more convincing?

Exercise 0.7 Consider once more the sorted list A, C, D, F, G, X. Using the tabular format from example 0.2, show how linear search and binary search move through the list when looking for item C.

Exercise 0.8* Binary search is only guaranteed to work correctly with ordered lists. But this does not imply that binary search necessarily fails on unordered lists. Write down an unordered list with 5 elements A, B, C, D, E such that binary search will find A but not E.

E.2 Programming

Exercise 0.9 Given the Python definition of `linear_search` in this chapter, what is the output of `linear_search([1,1,1,1], 1)`?

Exercise 0.10 In Python, `0` is sometimes treated the same as `False`. This can result in some unexpected bugs, for instance with the piece of code below. What exactly is the problem? How would you fix it? Is this an issue that a high-level definition of linear search should address?

```
if linear_search([1,2,3,4], 1):
    print("Item in list!")
else:
    print("Item not in list!")
```

Exercise 0.11 Write a recursive implementation of binary search.

Exercise 0.12 What would a recursive implementation of linear search look like? Compared to binary search, do you find the recursive implementation of linear search simpler, more convoluted, or about the same in complexity?

Exercise 0.13* Consider the first version of the list intersection function in this chapter, which does not use dictionaries. Instead of the second `for`-loop, we could have used Python's built-in membership test for lists:

```
if a in listB:
    intersection.append(a)
```

Would this have solved the speed issues? Justify your answer. In order to do so, you might have to do some research on how exactly Python's `in`-operator works.

Exercise 0.14* The problem with multiple `for`-loops gets even worse when computing the intersection of multiple lists. For these cases, then, it's even more important to have an efficient solution.

Generalize the dictionary-based intersection function so that it can take an arbitrary number of lists as arguments. All lists except the first one should be converted to a dictionary.

Part I

Phonology

Unit 1

Implementing phonology as a list

Linguists distinguish many different aspects of language:

Phonetics the physical properties or speech, in particular the physiological production of sounds (*articulatory phonetics*) and their acoustic properties (*acoustic phonetics*)

Phonology the rules regulating the sounds of a language: which sounds are part of a given language's inventory, in which positions may they occur, how are sounds affected by the presence of other sounds, which syllables in a word are stressed, and so on

Morphology the rules regulating the structure of words, in particular how a words form can change depending on certain features such as person or number (*inflectional morphology*) and how new words can be built from other words (*derivational morphology*)

Syntax the rules regulating the structure of sentences, e.g. word order and morphosyntactic dependencies (person, gender, number agreement)

Semantics the formal study of the meaning of words (*lexical semantics*) and how the logical meaning of a sentence is derived from the meaning of its words (*compositional semantics*)

Pragmatics the study of how a sentence's intended meaning arises from the interaction of its logical meaning and the discourse context

All these areas have been looked at by computational linguists, but we will start with phonology, for several reasons. First, phonetics involves a lot of non-discrete math that is fairly demanding for the uninitiated. Similarly, syntax, semantics and pragmatics use fairly complicated linguistic formalisms. This leaves us with phonology and morphology, and since the two are very similar from a computational perspective (many computational models even handle them both at the same time), we pick the one that linguistics students usually have had greater exposure to: phonology.

1 A simple phonology problem

Word-final devoicing is a process that has been extensively studied by phonologists. As its name implies, final devoicing turns voiced consonants (/b/, /d/, /g/, /z/, ...) that

occur at the end of a word into their voiceless counterparts (/p/, /t/, /k/, /s/, ...). It usually applies only to a proper subclass of a given language's full inventory of voiced sounds. Final devoicing is attested in a rich variety of languages, from Indo-European ones like Catalan (Romance), German (Germanic), and Russian (Slavic) to Turkish (Turkic, Altaic) and Wolof (Senegambian, Niger-Congo).

Language	Voiced	Devoiced
Catalan	*grize* 'gray (F)'	*gris* 'gray (M)'
German	*räder* 'bikes'	*rat* 'bike'
Russian	*kniga* 'book (NOM.SG.)	*knik* 'book (GEN.PL.)'
Turkish	*sarabi* 'wine (ACC.SG.)'	*sarap* 'wine (NOM.SG.)'
Wolof	does anybody know	the data?

What kind of computational resources must a native speaker possess that has successfully learned this process for their language and can apply it correctly during speech? As innocent as this question may seem, it is actually very difficult to answer because it is unclear what kind of computational process underlies word-final devoicing.

The way it was described above — which is the standard view among phonologists — it is a process that takes a word as an input and returns an output where any word-final voiced consonants have been devoiced. That is a complex idea that involves three distinct components:

1. an input form,
2. an output form,
3. a process translating the former into the latter.

It is far from obvious that this is indeed what speakers are doing; all we can tell for sure is that speakers produce the correct output forms. So rather than jumping immediately into the deep waters of sophisticated phonological machinery, let's see what the **simplest empirically adequate** solution might be. It might well turn out that speakers are actually doing something more complicated, but at least we will have a better understanding of the problem and a computational baseline that we can compare speakers' behavior to.

Arguably the simplest conceivable solution is that there are no phonological processes at all, speakers simply memorize all the output forms.

Listedness model of phonology A speaker's knowledge of phonology is fully captured by a list of pronunciations.

This would reduce phonology to a long list of fully inflected words and speakers simply pick that item from the list that they want to pronounce. It explains speakers' ability to produce the correct output form purely via memorization, no computational machinery is required beyond a mechanism for storing and retrieving phonetic strings. Such a solution is certainly simple, and it has been used in NLP in the past. After all it only require a few people to go through a dictionary of English and write down the pronunciation for each word and all its fully inflected variants (e.g. *go*, *goes*, *went*, *gone*, *going*). Simple as it may be, though, the important question for us whether it is empirically adequate.

2 The Listedness model is inadequate

2.1 A failed argument about storage and retrieval

Your first gut reaction may be that the Listedness model cannot possibly be right because it would require storing an enormous amount of information. One can certainly construct a formal argument along those lines, but as we will see it is a weak one because if confuses specification and implementation.

Let's first try to fully spell out why storage might be a problem. Suppose that an English speaker knows around 50,000 words, each one of which has approximately 2 different pronunciation forms. For instance, the verb *go* require memorizing the pronunciations of *go*, *goes*, *went*, *gone*, and *going*, the noun *cow* also has a plural *cows*, and an adjective like *red* only has that one pronounced form. Let us assume furthermore that the average word has 8 sounds, and that there are 64 different sounds — both are vast overestimates. If there are 64 distinct sounds, a single sound takes 6 bits to store. Overall, then, our list would require at least $50,000 \times 2 \times 8 \times 6 = 4,800,000$ bits of storage, which is approximately 585 kilobytes.

Background Binary numbers and bits

In daily life we use the *decimal number system*. In this system, 53 represents a number that we can analyze as $50 + 3 = (5 \times 10) + (3 \times 1) = (5 \times 10^1) + (3 \times 10^0)$. So in the decimal system, the digit in the n-th position acts as a multiplier for 10^{n-1}, and we just sum the values encoded by each position of the number. That's why it is called the decimal system, 10 acts as the base with exponent $n-1$.

This system can of course be used with any other natural number as the base. The simplest case is the *binary number system*, where the base is 2. The number 5, for instance, has the binary representation 101 since $5 = (1 \times 2^2) + (0 \times 2^1) = (1 \times 2^0)$. And 53 is written 110101, whereas 1100001 is 97 (you do the math!). So even though both numbers have two digits in decimal, one has 6 digits in binary and the other 7 digits. A binary digit is also called a *bit*.

Binary numbers are tremendously useful because they are strings of bits, and each bit can have only two values. This means that each bit can be easily instantiated in a physical system, for instance via the on and off states of a switch, or the presence or absence of an electric current. That's exactly what computers do, and that's why computers "only know zeros and ones", as the tired old saying goes. It is also why a single bit represents the smallest unit of information: the most basic distinction one can make is that between being in a specific state on the one hand, and not being in that state on the other hand. This in turn also allows us to calculate memory usage: given a binary encoding of some piece of information, its memory usage equals the number of bits used for the encoding.

In our calculations above, we need 6 bits to encode the 64 phones because that is the smallest number of bits that allows us to make 64 distinctions. With 5 bits, there's only $2^5 = 32$ distinct binary representations, which isn't enough. If the language had 65 phones, we would need 7 bits per phone instead of 6. There are ways to optimize binary encodings so that very frequent phones would have short bit representations than rare ones, but we will not go into this here. We only want a rough approximation of memory usage, and the simple binary encoding explained here is sufficient for that.

Given our (rather pessimistic) assumptions, the Listedness model of English phonology requires barely more than half a megabyte. This isn't all that much in the grand scheme of things. We could even consider more extreme examples. The language Ubykh (extinct, Northwestern Caucasian) has 84 phonemes (≈ underlying sounds). Even if we assume that each phoneme has two pronunciation variants, this leaves us with at most 172 different sounds and hence 8 bits per sound. Keeping all assumptions as before, the Listedness model would consume around 780 kilobytes, which is still less than one megabyte. To put this number into perspective, the collected works of Shakespeare take up about 5 times as much (depending on how they are encoded), and they fit on 1500 densely printed pages. If we put these numbers in direct correlation, we may say that the Listedness model only works if humans can memorize 300 pages of text.

This may seem ludicrous, but human memory is capable of some surprising feats. The Vedic Sanskrit corpus, for instance, consists of over 1000 pages and despite being over 3000 years old, wasn't written down until the first century BC. For over a thousand years, it was preserved by a purely oral tradition that relied exclusively on memorization. That requires a lot of storage capacity. Of course this isn't a perfect comparison: oral traditions rely on elaborate memorization techniques, the memorization is done by adults and not by infants acquiring the language of their environment, and so on. But if memorizing 1000 pages is feasible, it is far from obvious that the Listedness model exceeds human storage capacities.

If you worked through section P.1 on linear and binary search in Chapter 0, then you might have another objection. The longer the list, the longer it should take to retrieve pronunciations, and as speakers with a large vocabulary do not take longer to retrieve pronunciations compared to those with a small vocabulary, the Listedness model is not a plausible model of human phonology. Once again there is a lot one could quibble with here, but let us get straight to the main problem with this argument as well as the previous one: the Listedness model is a claim about how the speaker's knowledge is specified, not how it is implemented.

The Listedness model does not claim that humans have a literal list of pronunciations in their head. There's many different ways a list can be implemented, be it a singly-linked list, a doubly-linked list, a hash table, or a prefix tree (see XXX). Some of them circumvent all the problems of storage and retrieval pointed out above. Even if they didn't, there might be some neural data structure that computer scientists are unaware of that allows humans to store very long lists in a very efficient manner. Criticizing the Listedness model because lists would be a suboptimal data structure is a category mistake. The model is meant as a high-level description of phonological knowledge, not how this knowledge is stored and queried in the human mind. If we want to argue against the Listedness model, we have to show that its notion of phonological knowledge is inadequate.

2.2 Phonological knowledge is more than memorization

The Listedness model claims that phonology is just a list of licit output forms that the speaker has memorized over the years. If this is correct, then several properties should hold of the phonological systems we find in natural languages:

- **Free typology**
  Since there are no restrictions on what items may occur in a list, any given list of output forms is a possible phonological system.